Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Injeção de DLL: Um estudo de caso aplicado à jogos

Pedro Yan Ornelas

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. João José Costa Gondim

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Injeção de DLL: Um estudo de caso aplicado à jogos

Pedro Yan Ornelas

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. João José Costa Gondim (Orientador)
CIC/UnB

Prof.a Dr.a Carla Denise Castanho
CIC/UnB

Prof. Dr. Robson de Oliveira Albuquerque
ENE/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 13 de outubro de 2019

Dedicatória

Dedico esse trabalho para todos que estiveram do meu lado durante a graduação e a minha criança interior que realizou o seu sonho de entender os processos místicos que os "deuses da programação" da época utilizavam para crackear e modificar jogos como bem entendessem.

Agradecimentos

Agradeço ao Professor Dr. Gondim por ter despertado em mim a paixão pela área de *Infosec* e pela disponibilidade para me auxiliar na confecção deste trabalho. Agradeço à Professora Dra. Carla Castanho pelas aulas mais desafiadoras e bem estruturadas que tive o prazer de presenciar na faculdade e pela sua forma humanizada, revigorante e amigável de tratar os alunos. Agradeço ao meu inseparável irmão de faculdade e futuro parceiro de negócios, Pietro Bertarini, por ter me dado a força que eu precisava para completar o curso e pelos memoráveis momentos nesta jornada. Agradeço à minha namorada, Camila Taunay, por sempre estar do meu lado nas horas mais difíceis e pelos indispensáveis *insights* sobre como estruturar textos acadêmicos. Agradeço à minha família, em especial os meus pais, por toda a base que foi me dada para chegar até aqui.

Resumo

Injeção de DLLs (Dynamic Link Libraries) é uma técnica que permite um processo externo modificar a lógica de um processo alvo em tempo de execução e acessar recursos protegidos com os mesmos privilégios que o hospedeiro. Quando usada para fins maliciosos, os resultados podem ser catastróficos, como é o caso do *Stuxnet*, um software malicioso responsável por destruir usinas de enriquecimento de urânio no Irã, fazendo-se necessário que a técnica seja profundamente compreendida para que sistemas de segurança sejam aprimorados contra esta ameaça. Sendo assim, este trabalho tem como objetivo fazer um aprofundamento na técnica por meio da implementação de uma ferramenta injetora que busca evadir soluções especializadas em impedir ou detectar a injeção, abordando conceitos teóricos e práticos de seu uso. Neste trabalho também será implementada uma DLL que modificará o funcionamento do jogo Assault Cube de formas complexas para demonstrar o poder da técnica, mostrando na prática o alto grau de conhecimento necessário sobre o alvo para que ele possa ser devidamente explorado.

Palavras-chave: Injeção de código, DLL, *malware*, segurança da informação, Windows Internals

Abstract

DLL (Dynamic Link Library) injection is a code injection technique that allows external processes to modify the execution logic of a target process and access all of the resources within its reach. When used for malicious ends like in Stuxnet, a malicious software that damaged uranium enriching facilities in Iran, the results can be catastrophic, thus, obtaining a profound understanding of the subject becomes crucial to develop proper security countermeasures against it. This study takes a deep dive into the technique by going over the theory, the practice and then implement a stealthy DLL injection tool. In order to demonstrate the power of the technique and the reverse engineering efforts needed to make it useful, the study will also implement a DLL that modifies the target process in complex ways.

Keywords: Code injection, DLL, *malware*, information security, Windows Internals

Sumário

Lista de Figuras

Lista de Abreviaturas e Siglas

API Application Programming Interface.

DLL Dynamic-link library.

ESP Extra Sensorial Perception.

FPS Frames Per Second.

IAT Import Address Table.

PE Portable Executable.

PID Process Id.

RAM Random-access memory.

RVA Relative Virtual Address.

SCADA Supervisory Control and Data Acquisition.

TID Thread Id.

TLS Thread Local Storage.

Capítulo 1

Introdução

Desde a invenção do transistor, a tecnologia vem crescendo em ritmo exponencial [1], estando cada vez mais próxima da figura onipotente, onipresente, que na idade média só poderia ser descrita como um Deus. Contando com processos automatizados em sistemas cada vez mais sofisticados, a sociedade avança a passos largos em direção à eficiência máxima, sem questionar os zeros e uns que tornaram isso possível.

Transações - que anteriormente requeriam a presença física de uma ou duas partes em um mesmo ambiente - agora podem ser realizadas no conforto de casa com somente alguns cliques ou toques e em questão de minutos. Mensagens - que anteriormente demoravam dias ou até meses, dependendo do seu trajeto - agora chegam ao seu destino em questão de milissegundos. Processos industriais - que antes haviam consideráveis perdas - foram aperfeiçoados por meio de SCADAs a patamares anteriormente considerados utópicos.

Porém, no verão de 2010, foi descoberta nas usinas de enriquecimento de urânio do Irã a presença de um dos malwares (softwares maliciosos) mais sofisticados da história da computação: O *Stuxnet*. Até então, achava-se que malwares eram somente uma inconveniência contida na esfera digital com a finalidade de roubar informações bancárias, derrubar serviços ou simplesmente perturbar usuários. No caso do *Stuxnet*, o principal objetivo era debilitar a capacidade de operação dessas usinas por meio da destruição das centrífugas de enriquecimento [2], sendo esta a primeira vez que um vírus obteve impactos diretos em equipamentos físicos de uma infraestrutura crítica do mundo real [3].

A principal técnica utilizada pelo malware para executar os primeiros estágios cruciais da infecção é chamada de *DLL Injection*, uma técnica abordada por este trabalho que permite um atacante inserir código num programa alvo e alterar o seu fluxo de execução da forma arbitrária. Quando utilizada para fins maliciosos, os resultados de seu uso podem ser catastróficos.

1.1 Motivação e Justificativa

Como a injeção de DLLs permite um atacante executar código arbitrário com a mesma autoridade que o processo alvo, o grau de liberdade conferido a ele dentro do processo e dos recursos administrados pelo mesmo é ilimitado, dando ao atacante a chance de acessar e manipular informações sigilosas e causar disrupção severa nos serviços prestados em máquinas hospedeiras ou conectadas. Devido ao potencial destrutivo da técnica evidenciado no mundo real pelo *Stuxnet*, que resultou na destruição de aproximadamente 1000 centrífugas de enriquecimento de urânio [4] no Irã, é necessário entendê-la para desta forma implementar mecanismos de segurança mais robustos capazes de detectá-la.

O programa escolhido como alvo das técnicas elaboradas neste trabalho foi um jogo pois:

1. Jogos são geralmente protegidos por soluções dedicadas e customizadas. Tais softwares, chamados de ***Anti-Cheats***, são capazes de detectar métodos ingênuos de injeção e anomalias no processo, fazendo-se necessário o uso de técnicas de injeção elaboradas para evitar detecção, o que faz de um jogo um alvo mais difícil de ser manipulado do que um processo comum;

2. Conforme observado na Figura 1.1, a industria de jogos é a maior no ramo de entretenimento do mundo, superando o cinema e a televisão [5], onde a garantia de condições equitativas entre jogadores é essencial para a qualidade do serviço, principalmente no caso de jogos competitivos. Por isso, segurança vem tomando cada vez mais uma posição de destaque entre os diferentes módulos que compõem o produto;

3. Jogos possuem lógicas internas razoavelmente complexas, o que resulta num interessante desafio de engenharia reversa. Tal característica também servirá para exemplificar que não basta somente o atacante injetar um código dentro do processo. É necessário bastante conhecimento sobre o funcionamento interno do alvo para explorá-lo;

4. Exigem que o código injetado seja eficiente. Como jogos são programas computacionalmente custosos devido aos diversos cálculos envolvidos com física, lógica e renderização, caso a DLL não seja bem implementada a queda de performance no jogo irá contrabalancear significantemente qualquer vantagem competitiva obtida.

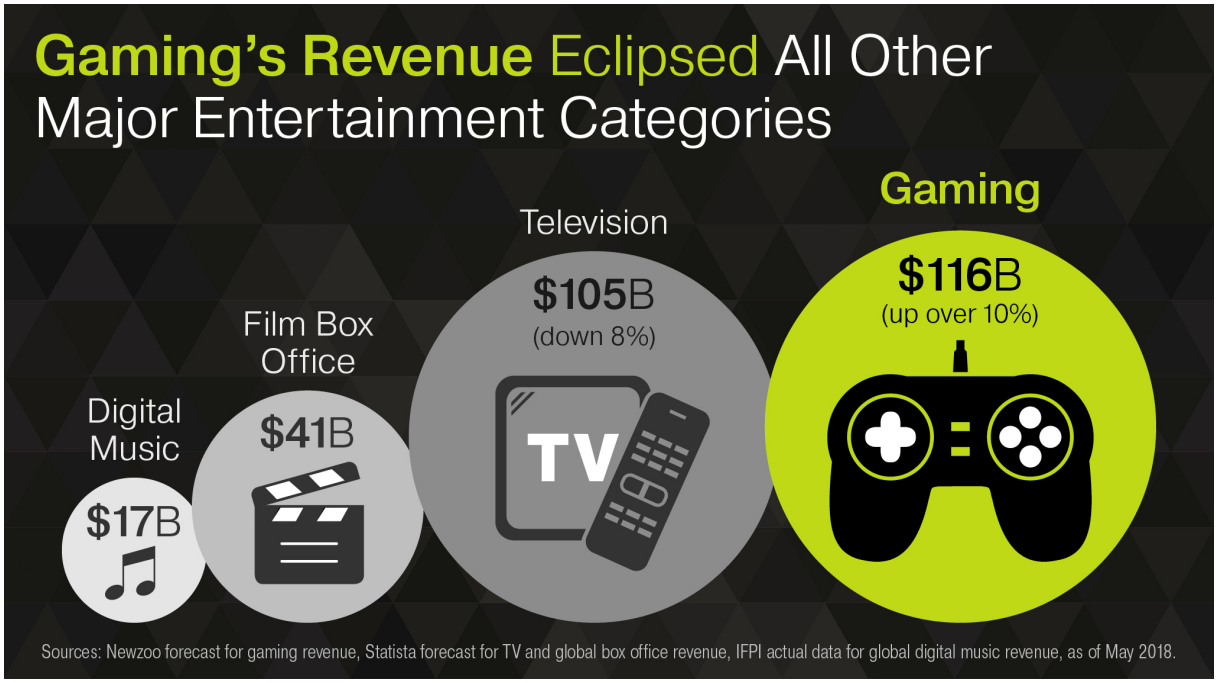

Figura 1.1: Receita gerada pelos diferentes segmentos da indústria de entretenimento (Fonte: [5]).

1.2 Objetivos

1.2.1 Objetivos Gerais

O objetivo desta monografia é demonstrar como funciona o processo de injeção e estudar possíveis mecanismos de evasão de detecção por meio da implementação um injetor de DLLs de uso geral para o sistema operacional Windows, comprovando a sua eficácia por meio de testes funcionais, que verificam se o comportamento do jogo foi alterado, indicando que a DLL foi injetada com sucesso, e per meio de análises do sistema, que visam identificar se a injeção de fato evadiu os mecanismos de detecção citados.

Este trabalho também tem como objetivo demonstrar, por meio da implementação de uma DLL, como esta pode modificar seu hospedeiro e os esforços de engenharia reversa necessários para que tais modificações forneçam alguma vantagem para um atacante. A eficácia da implementação será comprovada por meio de testes funcionais, que verificam se as funcionalidades listadas ocorrem de forma correta dentro do jogo.

1.2.2 Objetivos específicos

Objetivos do Injetor

O injetor deverá ser capaz de:

- Criar *handles* diretas para o processo alvo, que serão usadas para injeções legítimas;
- Realizar injeções sem a criação de novas handles, visando evitar detecções;
- Realizar injeções legítimas por meio de APIs (Application Programming Interfaces) do Windows que registram explicitamente a DLL na lista de módulos do processo;
- Realizar injeções furtivas, onde o injetor irá inserir a DLL manualmente no processo sem listá-la na lista de módulos;
- Executar a DLL injetada por meio de uma nova *thread* criada pelo processo injetor;
- Executar a DLL injetada por meio do sequestro de uma *thread* existente do processo alvo em execução, com a finalidade de evitar detecções.

Objetivos da DLL

A DLL desenvolvida para ser injetada no jogo Assault Cube, detalhado no Capítulo 4, deverá fornecer as seguintes vantagens competitivas para o jogador:

- Munição infinita;
- Tiro Rápido;
- Tiros sem recuo;
- Todas as armas automáticas;
- Vida infinita;
- Percepção Extra Sensorial (ESP), que permite o jogador ver a posição de todos os outros jogares por trás de qualquer obstáculo;
- Mira automática na cabeça de jogadores adversários (*Aimbot*).

1.3 Contribuição Original

A técnica de obtenção furtiva de handles chamada de **Pyjack**, detalhada no Capítulo 3, é a principal contribuição original deste trabalho. Apesar da breve referência como um conceito de altíssimo nível em fóruns de *hacking* a respeito de uma técnica de injeção a nível de usuário que não cria handles novas para seus alvos, o **Pyjack** trouxe essa ideia para o mundo real, comprovando a sua eficácia e descrevendo as etapas necessárias para sua realização.

1.4 Disclaimer

Vale ressaltar este trabalho foi feito apenas com fins acadêmicos, com o autor e seu orientador não se responsabilizando por qualquer uso inapropriado das técnicas e exemplos contidos nesta monografia.

O jogo Assault Cube, detalhado no Capítulo 4 e alvo das técnicas de exploração descritas neste trabalho, é um software de licença *freeware*, o que torna a sua modificação e exploração um ato lícito perante leis brasileiras e internacionais.

1.5 Organização do texto

Esta monografia foi estruturada da seguinte maneira:

- O Capítulo 2 aborda toda a fundamentação teórica necessária para o entendimento do trabalho;
- O Capítulo 3 apresenta em detalhes a implementação do injetor de DLLs;
- O Capítulo 4 apresenta em detalhes a implementação da DLL injetada, bem como os processos de engenharia reversa utilizados para a sua elaboração;
- O Capítulo 5 apresenta a conclusão e as propostas para trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo serão abordados os conceitos fundamentais necessários para entender o processo de injeção e a DLL injetada, fornecendo para o leitor os principais conceitos necessários para o entendimento desta monografia.

2.1 DLL

Dynamic-link library, ou DLL, é a implementação do sistema Windows para o conceito de ***Shared Library***. Visando um maior entendimento desta definição, primeiramente será necessário abordar os conceitos de **Biblioteca**, ***Linker***, **Biblioteca estática** e, finalmente, ***Shared Library***.

2.1.1 Biblioteca

No âmbito da computação, uma biblioteca é um conjunto de recursos não-voláteis que podem ser consumidos por programas de computadores [6]. Tal conjunto pode incluir dados além de sub-rotinas e código executável, como configurações, templates, classes, tipos e até imagens [7].

Apesar da ampla gama de dados que bibliotecas podem disponibilizar, este artefato computacional é majoritariamente utilizado como uma forma de compartilhamento de código, provendo interfaces bem definidas que permitem programas externos a consumir os serviços prestados. Devido a tal caraterística, as funcionalidades de uma biblioteca costumam a ter a sua implementação projetada para o reuso em programas distintos.

Bibliotecas também são utilizadas para encapsular implementações extensas ou de baixo nível por meio de APIs intuitivas, facilitando o uso das funcionalidades providas e consequentemente reduzindo a complexidade dos programas que as utilizam.

É possível até mesmo re-encapsular os serviços prestados por diversas outras bibliotecas em uma única API de nível ainda mais alto de abstração, aproximando-os ainda mais a uma linguagem natural. Um exemplo disto é da biblioteca *libvorbisfile* ilustrada pela Figura 2.1. Por meio de APIs de alto nível, a biblioteca provê métodos para a decodificação de arquivos *.ogg* sem expor a complexidade envolvida na série de chamadas para outras bibliotecas, expondo somente o comportamento de alto nível desejado (decodificação).

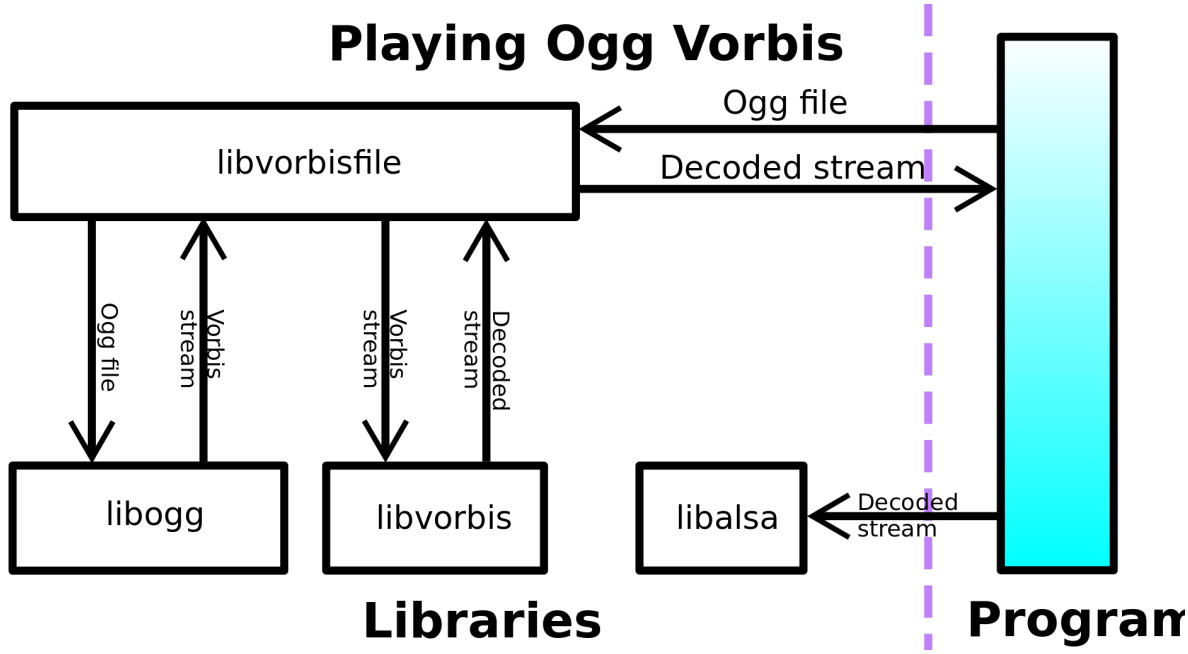

Figura 2.1: Chamadas executadas pela biblioteca *libvorbisfile* para decodificar um arquivo *.ogg* (Fonte: [8]).

2.1.2 Linker

O ***Linker*** é um programa que combina diversos arquivos objetos e bibliotecas num único artefato final por meio da resolução de símbolos externos indefinidos contidos em cada objeto, podendo o artefato gerado ser um executável ou biblioteca. Toda vez que um arquivo objeto referência itens externos ao seu escopo, um símbolo indefinido é criado e toda vez que o arquivo objeto declara um item que pode ser consumido por outros módulos, um símbolo publico é criado.

Durante o processo de *Linkagem*, o programa faz a resolução de símbolos indefinidos por meio da substituição destes pelos endereços dos símbolos públicos correspondentes que foram definidos em outros módulos [9], conforme ilustrado pela Figura 2.2.

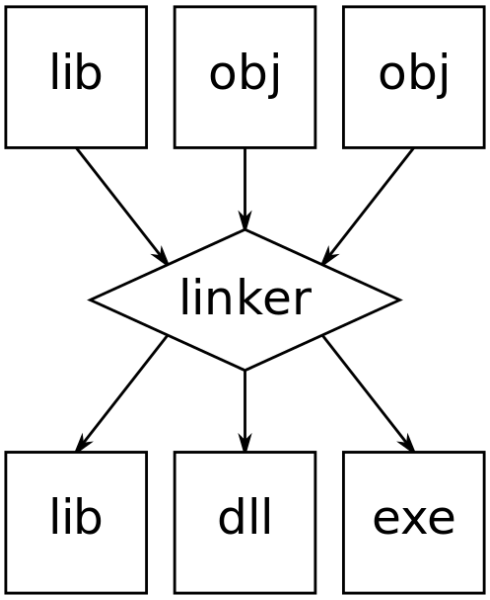

Figura 2.2: Diagrama de entradas e saídas de um *linker* (Fonte: [10]).

2.1.3 Biblioteca estática

Quando os símbolos públicos de um biblioteca são utilizados para a resolução conduzida por um **Linker** durante o processo de criação do artefato, a mesma pode ser considerada uma **Biblioteca estática**, sendo esta resolução chamada de **Linkagem estática**.

Como o conteúdo das bibliotecas estáticas é copiado para dentro do artefato final, o artefato gerado é completo e auto-contido, ou seja, uma vez criado, o artefato pode ser executado ou referenciado sem nenhuma dependência externa. Vale ressaltar que o ciclo de vida do artefato final está intrinsecamente ligado ao ciclo de vida das bibliotecas estáticas: caso haja alguma mudança nas bibliotecas, o artefato final precisa passar novamente pelo processo de ligação para manter-se atualizado.

2.1.4 Shared Library

Shared Libraries são bibliotecas que, ao contrário de bibliotecas estáticas, podem ser carregadas na memória somente quando necessário, em tempo de execução e uma única vez, podendo ser compartilhada por vários processos conforme ilustrado pela figura Figura 2.3.

Uma vez que a biblioteca é carregada, os símbolos indefinidos que a referenciam serão resolvidos para o endereço de sistema em que ela foi alocada, independente do processo onde o símbolo reside. Esta resolução, chamada de *Linkagem dinâmica*, evita carregamentos repetitivos da biblioteca e reduz o tamanho dos executáveis finais, otimizando o uso de disco e de memória RAM. A função **prinftf()**, por exemplo, foi implementada

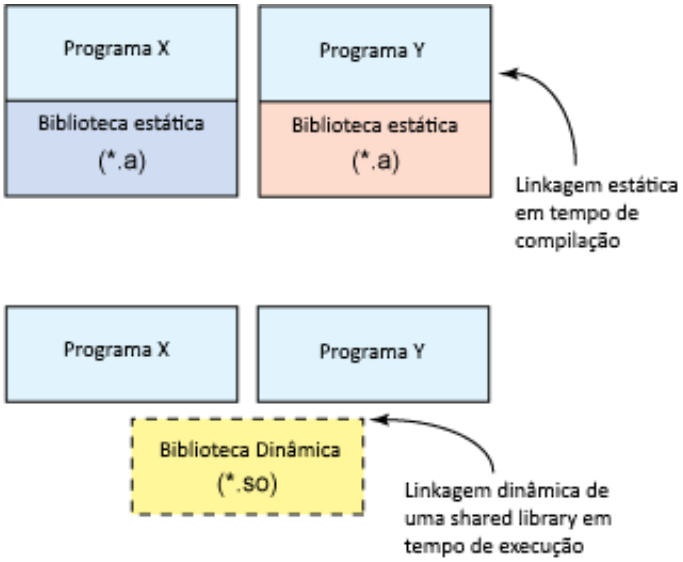

Figura 2.3: Linkagem estática vs Linkagem dinâmica (Adaptada de: [11]).

por meio de uma ***Shared Library*** tanto no Windows quanto no Linux justamente para otimizar o uso de memória.

Vale ressaltar que, ao contrário de bibliotecas estáticas, ***Shared Libraries*** possuem um ciclo de vida independente do executável consumidor, uma vez que não são incluídas no executável final em tempo de compilação. Com isso, caso haja alguma atualização na ***Shared Library***, tal modificação terá efeito imediato em todos os programas que a consomem, sem necessidade de recompilação dos mesmos.

Tal desacoplamento entre os ciclos de vida do executável e da biblioteca, apesar de vantajoso sobre a ótica de versionamento, pode trazer problemas comumente chamados de *DLL Hell* [12]. Caso uma ***Shared Library*** seja atualizada de tal forma em que o seu funcionamento interno não possua retrocompatibilidade com a versão anterior, qualquer aplicação que dependia do funcionamento antigo não ira funcionar corretamente, decorrendo em resultados inesperados e crashes. O problema se torna ainda pior caso a biblioteca seja um elemento central de uma cadeia de dependência entre ***Shared Libraries***, pois todos os nós que dependem diretamente ou indiretamente da biblioteca incompatível se tornam corrompidos. Nos dias de hoje, isso raramente é um problema pois DLLs são implementadas e gerenciadas para evitar estes conflitos por design, por meio de versionamento e gerenciamento inteligente de versões [13] ou o empacotamento da aplicação junto com a DLL específica.

2.2 Dynamic Loading

Dynamic Loading é o mecanismo que permite um processo em tempo de execução carregar uma ***Shared Library*** arbitrária na memória, acessar o seu conteúdo e desalocá-la. Com isso, o processo consegue inicializar sem a presença da biblioteca, podendo a busca e o carregamento da mesma serem realizados durante a execução do programa, permitindo o ganho de funcionalidade adicional de forma dinâmica [14]. Um uso comum de tal mecanismo é em aplicações que fornecem suporte a plugins.

O exemplo em C++ apresentado na Listagem 2.1 demonstra na prática como funciona o carregamento dinâmico de uma DLL no Windows. Neste exemplo, a função é tratada como uma variável que precisa ser inicializada antes de ser consumida. Tal função não pode ser utilizada da mesma forma que funções importadas por meio de ligação dinâmica ou estática, onde basta incluir um arquivo .h, configurar a ligação e chamar a função diretamente.

2.3 Handle

Uma **handle** nada mais é do que uma referência abstrata para um recurso no computador. Este recurso pode ser um arquivo, um processo, um socket ou outros. Ao contrário de ponteiros, que referenciam endereços, as handles são uma abstração para um recurso que é gerenciado externamente ao programa. Essa camada de abstração permite que o sistema ou sub-sistema gerenciador do recurso possua um maior controle sobre as operações feitas por outros programas no mesmo, como é o caso de *File descriptors* e Sockets, que são gerenciados pelo sistema operacional.

2.4 Inline Hooking

Inline hooking é uma metodologia para interceptação de funções em nível de assembly que permite um atacante executar código arbitrário, chamado de **callback**, sempre que uma determinada função no código original for chamada. O técnica consiste em sobrescrever 5 bytes do assembly original com uma instrução de JMP **(OpCode 0xE9)** para que a mesma desvie o fluxo de execução para o **callback**. Visando permitir que o **callback** seja executado sem afetar o restante do fluxo do processo, o uso de um **trampolim** se faz necessário, conforme ilustrado pela Figura 2.4

Ao realizar um **inline hook** deve-se idealmente procurar um conjunto de n instruções que somadas ocupam exatamente 5 bytes em memória. Por exemplo, caso uma instrução ou conjunto de instruções possua 6 bytes, como somente 5 serão usados para a instrução

de JMP, sobrará 1 byte não-sobrescrito chamado de *wild byte*, que resultaria num comportamento indeterminado do programa. Para contornar este problema, o **hooking engine** implementado neste trabalho e apresentado na Listagem 2.2 sobrescreve os *wild bytes* com NOPs (instruções que não fazem nada), fornecendo ao usuário uma maior liberdade de onde colocar o hook.

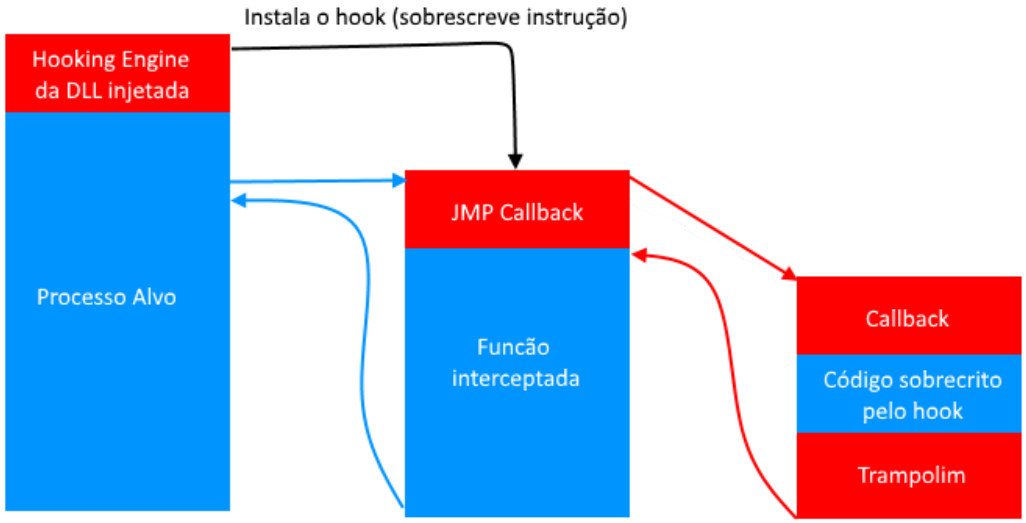

Figura 2.4: Fluxo de execução de um hook com trampolim.

2.5 Formato PE

O formato PE é um formato de arquivo para executáveis, arquivos objetos e DLLs nas versões 32-bits e 64-bits do sistema operacional Windows. A estrutura de dados contida em tal formato tem como o principal objetivo fornecer as informações necessárias para para que o *loader* do sistema operacional, programa responsável por carrega e gerenciar programas e bibliotecas na memória, consiga realizar suas funções. [15]

O layout do cabeçalho PE consiste em vários campos que descrevem para o **dynamic loader** do sistema operacional todas as informações necessárias para efetuar o carregamento do arquivo. O cabeçalho define informações de dependências externas (que devem ser carregadas caso não já não tenham sido) e como o arquivo deve ser mapeado em memória, especificando localização de cada bloco de memória (chamado de seção) alocado e com qual permissão deve ser mapeado [15]. Por exemplo, no caso de uma seção de código executável, a entrada correspondente no cabeçalho de seções (ou tabela de seções, conforme ilustrado pela Figura 2.6) irá constar que o segmento de memória alocado deverá

ser mapeado com permissões para execução e leitura mas sem permissão para escrita, sendo responsabilidade do **dynamic loader** do sistema garantir que essas especificações sejam respeitadas.

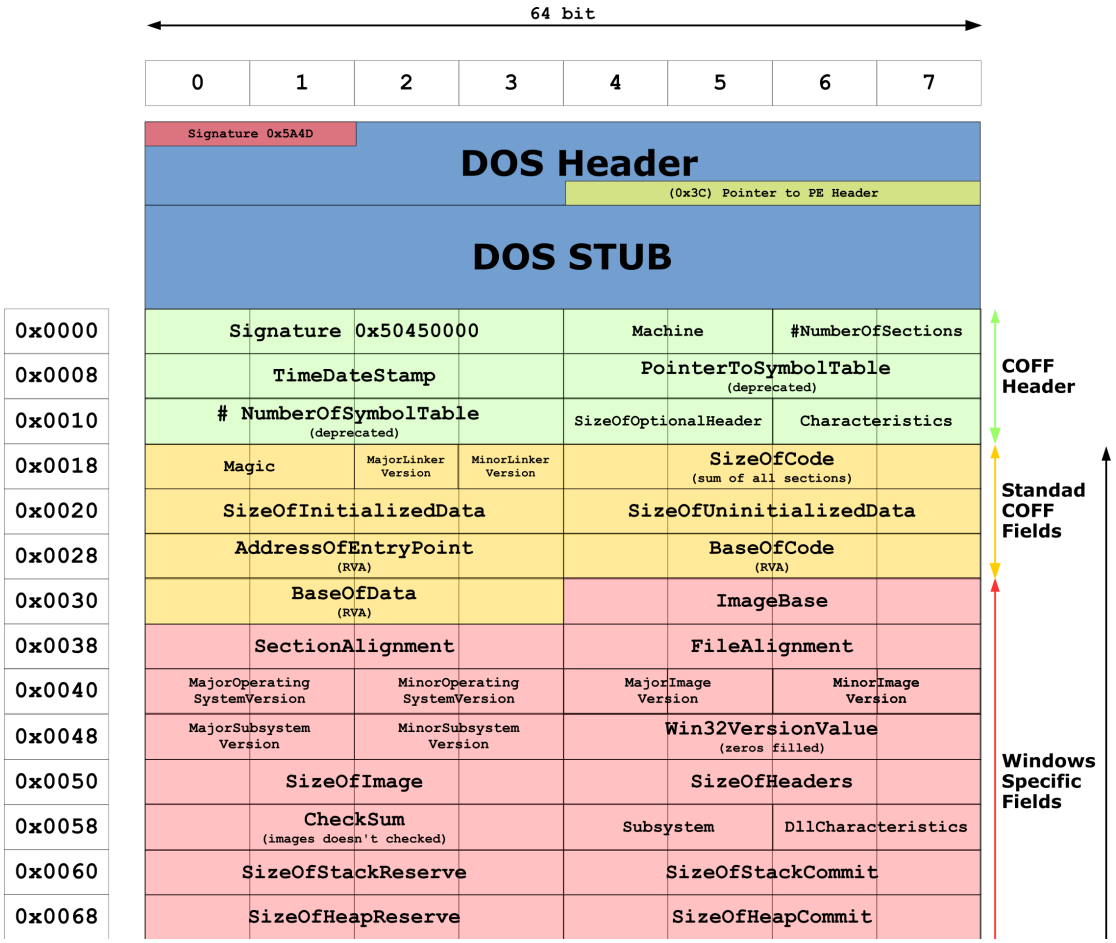

Figura 2.5: Parte 1 da estrutura do cabeçalho PE (Fonte: [16]).

Além de descrever as informações citadas anteriormente, o cabeçalho também armazena símbolos indefinidos que não foram resolvidos em tempo de compilação num diretório de dados chamado de **ImportAddressTable** (demostrado na Figura 2.6), que é, em sua essência, uma *lookup table* utilizada pelo código intra-modular para referenciar funções extra-modulares. Ao carregar as dependências externas em memória, o **dynamic loader** deverá realizar a resolução da tabela, preenchendo os slots dos símbolos indefinidos com o endereço de memória das funções correspondentes na biblioteca recém-carregada dinamicamente.

Vale ressaltar que o código contido dentro de arquivos PE não são independentes de posição, ou seja, para serem executados sem nenhuma alteração o arquivo PE precisa ser mapeado para o exato endereço base especificado pelo campo **ImageBase** (visível

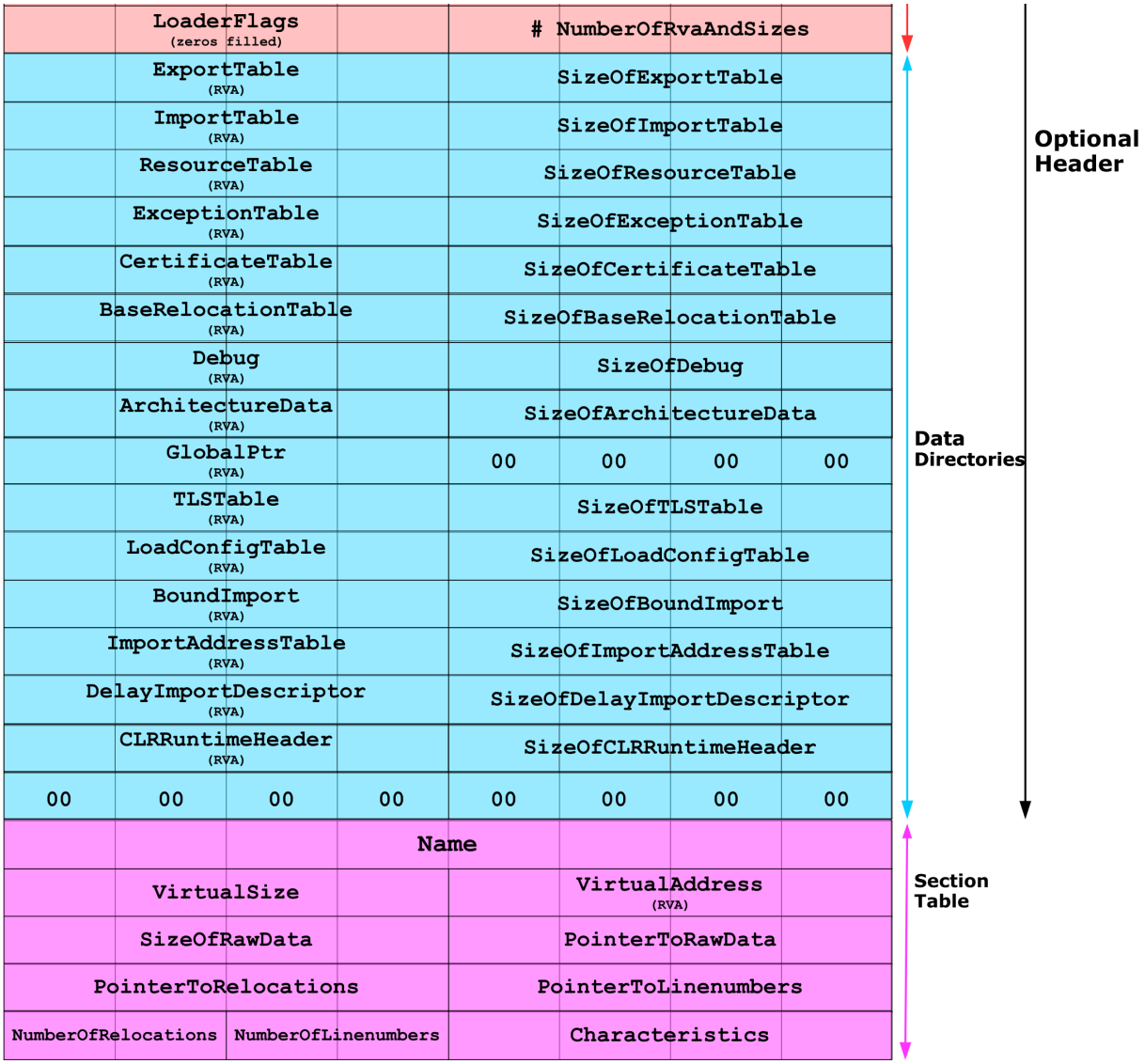

Figura 2.6: Parte 2 da estrutura do cabeçalho PE (Fonte: [16]).

na Figura 2.5). Caso o endereço base preferível não esteja disponível, é necessário realizar o processo de **relocação**. Neste processo, o **dynamic loader** aloca o arquivo em um endereço base qualquer e reescreve todas as referências absolutas intra-modulares, compensando-as por um *delta* calculado pela diferença entre o endereço alocado e o endereço esperado, conforme a Equação 2.1, sendo o código resultante único ao processo e não-compartilhável.

$$\Delta Addr = AllocatedAddr - ImageBase \tag{2.1}$$

Sempre que possível, o **dynamic loader** deverá alocar o módulo no endereço preferencial, pois assim possibilita o compartilhamento da biblioteca por diversos processos

e a alocação é feita de forma menos custosa. Antigamente a arquitetura dos módulos internos do Windows fazia uso extensivo da alocação preferencial, com todos os endereços preferíveis de cada DLL interna selecionados de tal forma que todos os módulos podiam ser carregados sem colisão e consequentemente sem relocação, otimizando a utilização de memória. Porém com o barateamento de chips de memória RAM e melhorias no poder de processamento de computadores atuais os custos com relocação se tornaram insignificantes, com muitas DLLs de sistema preferindo utilizar uma técnica de mapeamento seguro chamada de ASLR, que mapeia os módulos em regiões imprevisíveis [17].

Nota-se na Figura 2.5 e Figura 2.6 que todas as referências para blocos de memória dentro do arquivo são denotadas pela sigla RVA, que significa **Relative Virtual Address**. Num arquivo imagem (que é o caso de DLLs e executáveis), o RVA é o endereço de um item carregado em memória subtraído do endereço base do módulo. Em termos mais simples, o RVA representa o endereço relativo de alguma coisa dentro da DLL tomando o endereço base como ponto de referência.

2.6 DLL Injection

DLL Injection é uma técnica que permite forçar um processo alvo a carregar uma DLL escolhida. Caso a injeção seja bem-sucedida, o código contido na DLL será executado com as mesmas permissões que o processo original, podendo o código injetado realizar qualquer ação que o hospedeiro poderia.

A técnica é amplamente utilizada na engenharia reversa e no desenvolvimento de malwares pois que permite o atacante alterar o fluxo de execução de um programa alvo conforme desejar. Em razão disso, a técnica possibilita um maior leque de opções para um malware cumprir o seu objetivo, e, no escopo da engenharia reversa, fornece maior poder de observação visto que o analista pode introduzir logs que o ajudam a compreender o funcionamento interno do programa. Também existem usos legítimos da técnica, como é o caso do FRAPS [18] e da NVidia GeForce Experience [19] , que utilizam a técnica para renderizar overlays informativos em aplicações fullscreen.

Em alto nível o procedimento realizado por um software injetor, ilustrado pela Figura 2.7, consiste em 4 etapas: **Conexão** com o processo alvo, **alocação** da memória necessária, **carregamento** da DLL no espaço alocado e **execução** da DLL [20].

2.6.1 Conexão com o processo alvo

Nesta etapa, o processo injetor precisa obter uma **handle** para o processo alvo, que será utilizada para a manipulação do mesmo. Uma handle pode ser facilmente obtida utilizando-se a função **OpenProcess()**, que recebe o id do processo alvo e o nível de

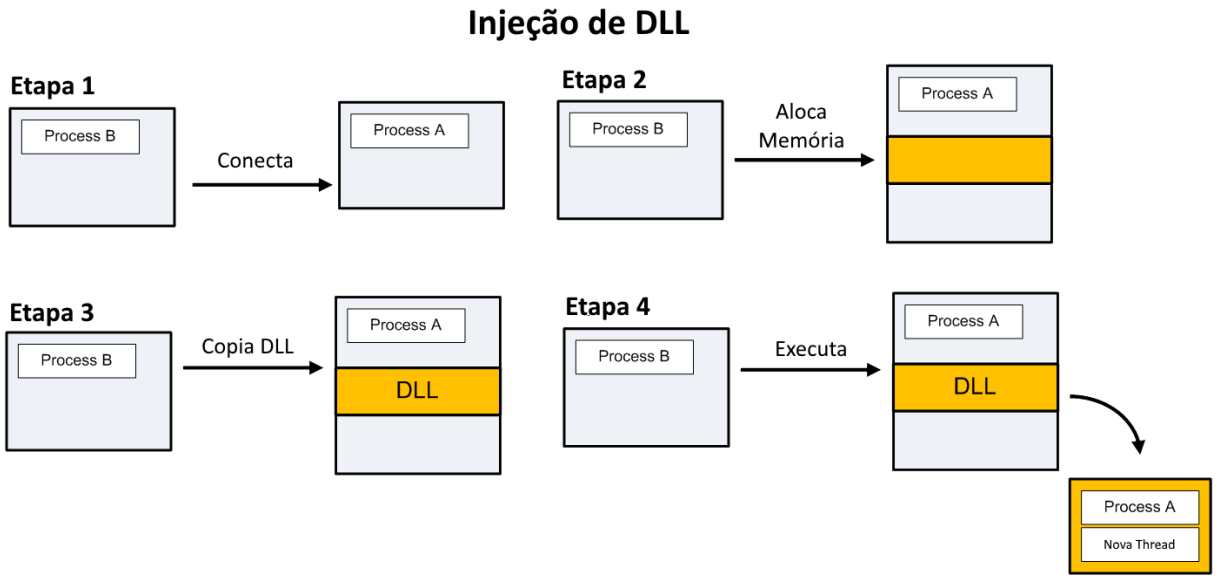

Figura 2.7: Etapas no processo de injeção de DLL (Adaptada de: [20]).

acesso requisitado para criar a nova handle. Vale ressaltar que tal método de obtenção de handles pode ser detectado por softwares Anti-cheat, que inutilizam a handle criada e marcam o usuário como um hacker em potencial, o que limita o uso da função a softwares legítimos autorizados pelo jogo. No Capítulo 3 será apresentado um método alternativo para a obtenção da handle chamado de **Pyjacking**, que dificulta a detecção desta etapa.

2.6.2 Alocação de memória para a DLL

Na etapa de **alocação**, o programa injetor utiliza a handle obtida para alocar memória dentro do processo alvo. O tamanho do bloco de memória alocado nesta etapa depende de como o **carregamento** da DLL será realizado, conforme ilustrado pela Figura 2.8. Caso seja feito utilizando a API do Windows **LoadLibraryA()**, somente será necessário alocar espaço para a string contendo o path da DLL, delegando responsabilidade de alocação da memória para a DLL em si ao sistema operacional. Porém, caso o carregamento for feito **manualmente**, é preciso alocar espaço para a DLL inteira.

2.6.3 Carregamento da DLL

LoadLibraryA

O carregamento via API do Windows é o processo mais **simples e direto**, pois a responsabilidade de efetuar a lógica de carregamento é delegada para o sistema operacional. Porém, tal método pode ser facilmente detectado por uma aplicação minimamente pro-

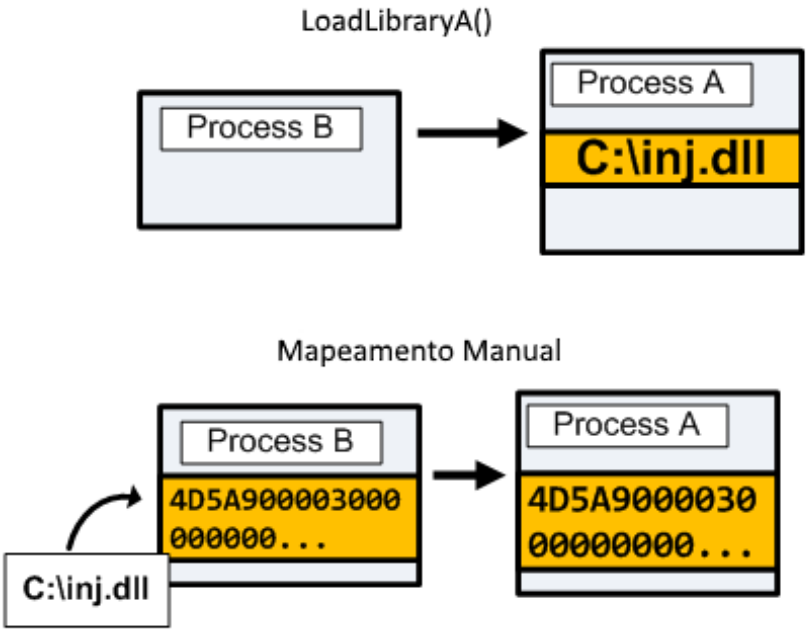

Figura 2.8: Tipos de alocação de memória para a DLL (Adaptada de: [20]).

tegida contra este tipo de ataque. Ao injetar uma DLL via **LoadLibraryA()**, o sistema operacional registra a DLL carregada na lista de módulos do processo, que pode ser prontamente enumerada por um software Anti-Cheat [21], facilitando uma possível detecção da injeção maliciosa.

Apesar da função ser facilmente detectada por soluções dedicadas de proteção, ela não deixa de ser válida devido à sua simplicidade e ao fato de que muitos processos não possuem nenhum tipo de defesa contra injeções inesperadas. A sua utilização pode ser observada em malwares injetores, como é o caso do *Rebhip* [22], um *Worm* classificado como severo pela Microsoft Security Intelligence [23] devido à sua capacidade de roubar informações sigilosas. O malware utiliza esta técnica de carregamento com o intuito infectar processos chaves, como o Windows Explorer e o Internet Explorer (*explorer.exe* ou *iexplore.exe*), para se espalhar para outros dispositivos e posteriormente obter dados sensíveis [23].

Mapeamento manual

Nesta forma de carregamento, o programa injetor precisa assumir o papel do sistema operacional de carregar adequadamente a DLL no processo alvo fazendo o uso do cabeçalho PE. O processo é complexo e envolve várias etapas de baixo nível intrinsecamente relacionadas ao sistema operacional e a arquitetura de processador do alvo do ataque. Apesar de ser um técnica complexa, o seu uso é justificado para fins maliciosos pois evita

mecanismos comuns de detecção, tais como hooks instalados por Anti-Cheas na função **LoadLibraryA()** e enumeração dos módulos carregados. Ambos mecanismos podem ser evitados pois no primeiro a função **LoadLibraryA()** não é chamada com o diretório da DLL suspeita e no segundo a DLL não aparece na lista de módulos.

Ao contrário da função **LoadLibraryA()**, esta forma de carregamento não requer que a DLL esteja salva em disco. Por isso, inúmeras possibilidades para o seu provisionamento são disponibilizadas, o que dificulta a detecção da DLL, já que a mesma pode ser provisionada de forma encriptada de um servidor.

Para o sistema Windows, o mapeamento manual ocorre, em alto nível, da seguinte forma:

1. Carregamento e validação do payload binário da DLL no processo injetor;
2. Alocação do espaço necessário para a DLL no processo alvo;
3. Mapeamento das Sections (unidade básica de código ou dados) no processo alvo;
4. Cópia do cabeçalho para o processso alvo e injeção do Shellcode de resolução interna;
5. Resolução da tabela de realocações;
6. Realização da ligação dinâmica;
7. Execução dos TLS callbacks;
8. Chamada da DLLMain;
9. Limpeza dos recursos alocados.

Cada etapa será abordada detalhadamente no Capítulo 3

2.6.4 Execução da DLL

A última etapa do processo de injeção é a execução da DLL carregada. A forma mais simples de iniciar esta etapa é por meio da criação remota de uma thread no processo alvo, conforme ilustrado pela Figura 2.9. Isto pode ser feito meio da função **CreateRemoteThread()**, que respeita a separação de sessões do Windows, um mecanismo de segurança implementado a partir do Windows Vista [20, 24].

O mecanismo de separação de sessões faz com que todos os serviços essenciais do sistema sejam executados dentro da *sessão 0* enquanto todos os processos de usuário executam numa sessão diferente. Em consequência desta implementação, processos em sessão de usuário ficaram impossibilitados de realizar injeção de DLLs em processos do sistema pois não é possível a criação de threads via **CreateRemoteThread()** entre processos em

Figura 2.9: Exemplo de thread criada para executar o código da DLL (Adaptada de: [22]).

sessões diferentes. Neste contexto, o uso da função não-documentada **NtCreateThreadEx()** se justifica, uma vez que esta função interna do Windows não respeita a separação de sessões [24], permitindo que um processo em sessão de usuário com privilégios suficientes crie threads em processos de *sessão 0*. Por se tratar de uma função não-documentada e voltada somente para uso interno do sistema operacional, a **NtCreateThreadEx()** pode ser alterada ou removida sem qualquer aviso prévio pela Microsoft, o que forçaria injetores que dependem desta metodologia de execução a se adequarem às atualizações. Vale ressaltar que, como não há documentação oficial para esta função, toda informação a seu respeito disponibilizada ao público é fruto de engenharia reversa [25, 26].

Independente de qual função seja utilizada para a de criação de uma thread, produtos de segurança como anti-vírus e anti-cheats rastreiam o seu uso por se tratarem de APIs suspeitas comumente utilizadas para fins maliciosos [22, 27]. Visando evitar detecção, o injetor implementado neste trabalho irá além dessas funções e oferecerá suporte para um método de execução chamado de **Thread Hijacking**, que não cria nenhuma thread adicional no processo alvo para iniciar a execução da DLL injetada.

Na execução por **Thread Hijacking**, o programa injetor suspende a execução de uma thread já existente no programa, altera os registradores de instrução para apontarem para uma instrução desejada, geralmente contida na DLL, e retoma a execução da thread parada. A implementação da técnica será abordada em detalhes no Capítulo 3.

Vale destacar que a DLL implementada (apresentada no Capítulo 4) foi arquitetada

para suportar qualquer tipo de execução com o máximo de performance possível. Por meio de um **Hook** no pipeline de execução do jogo é possível evitar que a sua lógica execute dentro de um loop infinito, o que resultaria na degradação do desempenho, caso uma thread fosse criada exclusivamente para a lógica da DLL, ou na inutilização do jogo, caso a thread principal fosse sequestrada e ficasse executando o loop da DLL ao invés do loop do jogo.

```
typedef int (__stdcall *my_func)();
int main()
{
  //Carrega a DLL em memoria
  HINSTANCE hGetProcIDDLL = LoadLibrary("test.dll");

  //Verifica se o carregamento foi bem sucedido
  if (!hGetProcIDDLL) {
    std::cout << "Nao foi possivel carregar a biblioteca
    dinamica" << std::endl;
    exit(EXIT_FAILURE);
  }

  //Resolucao do endere o da funcao contida na DLL
  my_func myFunc = (my_func)GetProcAddress(hGetProcIDDLL,
    "myFunc");
  if (!myFunc) {
    std::cout << "Nao foi possivel encontrar a fun ao
        desejada" << std::endl;
    exit(EXIT_FAILURE);
  }

  //Utilizacao da funcao
  std::cout << "myFunc() retornou " << myFunc()
    << std::endl;

  //Liberacao da memoria onde a biblioteca foi
  //carregada dinamicamente
  FreeLibrary(hGetProcIDDLL);
```

```
  return EXIT_SUCCESS;
}
```

Listing 2.1: Exemplo de *Dynamic Loading*

```
bool InlineHook(void* toHook, void* newFunc, int len)
{
    //toHook: Address of the instruction(s) being hooked
    //newFunc: Address to the detour function.
    //len: Length of the instruction block being hooked

    //The jump being placed is 5 bytes in size. If the spot
    //being overwritten is less than that, then we can't
    //hook that address
    if (len < 5)
        return false;

    //NOP the entire instruction being hooked to prevent
    //undefined behavior (ex: when the instruction(s)
    //size is 7 bytes 5 will be used for the JMP and
    //the remaining  2 bytes will be NOPed for safety)
    DWORD oldProtection;
    VirtualProtect(toHook, len, PAGE_EXECUTE_READWRITE,
        &oldProtection);
    memset(toHook, 0x90, len);

    //In this hook we are doing a near jump, which uses
    //a relative  address to indicate where
    //the destination address of the jump should be.
    /* 0x010    ADD edx, ecx  <= Jump points here
     * ...
     * 0x110    JMP -0x0105
     * 0x115    ...
     */
```

```
//The JMP is relative to the address of the
//instruction AFTER the JMP, and because of
//this we need to subtract 5 (size of JMP)
//from the offset between the current
//address being  hooked and our new function.
const DWORD relativeAddress =
    ((DWORD)newFunc - (DWORD)toHook) - 5;

//Set the first byte of the instruction being hooked as
//the JMP OpCode
*(BYTE*)toHook = 0xE9;

//Increase the toHook address by one to not override
//the JMP instruction and set the relative address
//parameter of the instruction
*(DWORD*)((DWORD)toHook + 1) = relativeAddress;

DWORD temp;
VirtualProtect(toHook, len, oldProtection, &temp);
return true;
```

Listing 2.2: Exemplo de *Função de Inline Hooking*

Capítulo 3

Programa Injetor

O injetor implementado para este trabalho visa atender as necessidades tanto de usuários legítimos, que buscam injetar DLLs por intermédio do sistema operacional, quanto de usuários maliciosos, que visam uma injeção furtiva que deixe o mínimo de rastros possíveis. Neste capítulo, será detalhada a arquitetura geral, o funcionamento e os princípios de design de software por trás desta ferramenta altamente configurável de injeção implementada em C++.

O injetor consiste num pacote composto por um **executável** e uma **DLL**, sendo o executável uma interface em linha de comando para o usuário interagir com a biblioteca de injeção implementada.

3.1 Requisitos

O injetor deverá:

- Possibilitar a configuração cada etapa do processo de injeção;
  - Obtenção de Handle;
  - Modo de injeção;
  - Modo de execução.
- Fornecer suporte a formas de injeção furtivas;
- Fornecer suporte a formas de injeção genuínas;
- Aderir boas práticas de desenvolvimento de software;
- Ser fácil de usar;
- Ser robusto;

- Ser extensível;
- Exibir claramente o motivo de falhas.

3.2 Princípios SOLID

Visando a alta qualidade do software desenvolvido, a arquitetura do injetor foi projetada com forte embasamento nos princípios **SOLID** [28] de design de software.

O ***S**ingle responsiblity principle*, talvez o princípio mais importante dos cinco, estabelece que módulos distintos devem tratar responsabilidades **únicas**, **distintas** e **bem definidas** e pode ser observado na solução em sua arquitetura modular de alta coesão e baixo acoplamento.

O ***O**pen-closed principle* estabelece que o software deve ser facilmente extensível, porém dificilmente modificado. Observa-se uma forte influência deste princípio na facilidade com a qual a solução pode ser estendida. Bastando apenas criar implementações concretas para as abstrações do programa, é possível adicionar novas técnicas de injeção, execução e conexão de forma isolada e quase que *plug-and-play*.

O ***L**iskov substitution principle* estabelece que um tipo pode ser substituído pelo seu subtipo sem alterar o bom funcionamento do programa. Constata-se a existência deste princípio ao observamos que as implementações concretas das abstrações funcionam corretamente.

Os princípios de ***I**nterface segregation* e ***D**ependency Inversion* podem ser observados na forma em como os módulos interagem entre si: Ao invés de módulos consumirem instâncias concretas de outros (salvo um caso em particular que será detalhado mais para frente), estes consomem abstrações com interfaces bem definidas, sendo a instância concreta injetada em tempo de execução. Tais princípios foram essenciais para o tornar o código modular e altamente configurável pelo usuário sem a necessidade da introdução de clausulas **if(...) else (...)** por todo código.

3.3 Módulos e suas responsabilidades

A solução injetora desenvolvida pode ser separada em quatro principais abstrações: **fábrica de serviços**, **serviços de handles**, **injetores** e **motores de execução**.

3.3.1 Fábrica de serviços

A **fábrica de serviços** é o módulo responsável pela tradução dos parâmetros introduzidos pelo usuário em funcionalidade por meio da criação de instâncias concretas para as

abstrações dos demais módulos na aplicação, conforme ilustrado pela Figura 3.1. Inspirado no padrão de design *Service Locator*, o módulo consegue encapsular de forma robusta a responsabilidade de definir como a injeção irá ocorrer. Ao invés de vários **if(...) else (...)** espalhados por todo o código checando os parâmetros introduzidos pelo usuário e definindo comportamento, toda essa responsabilidade está delegada **somente** a fábrica, o que resultou numa significativa redução da complexidade ciclomática [29] do código e melhora na legibilidade.

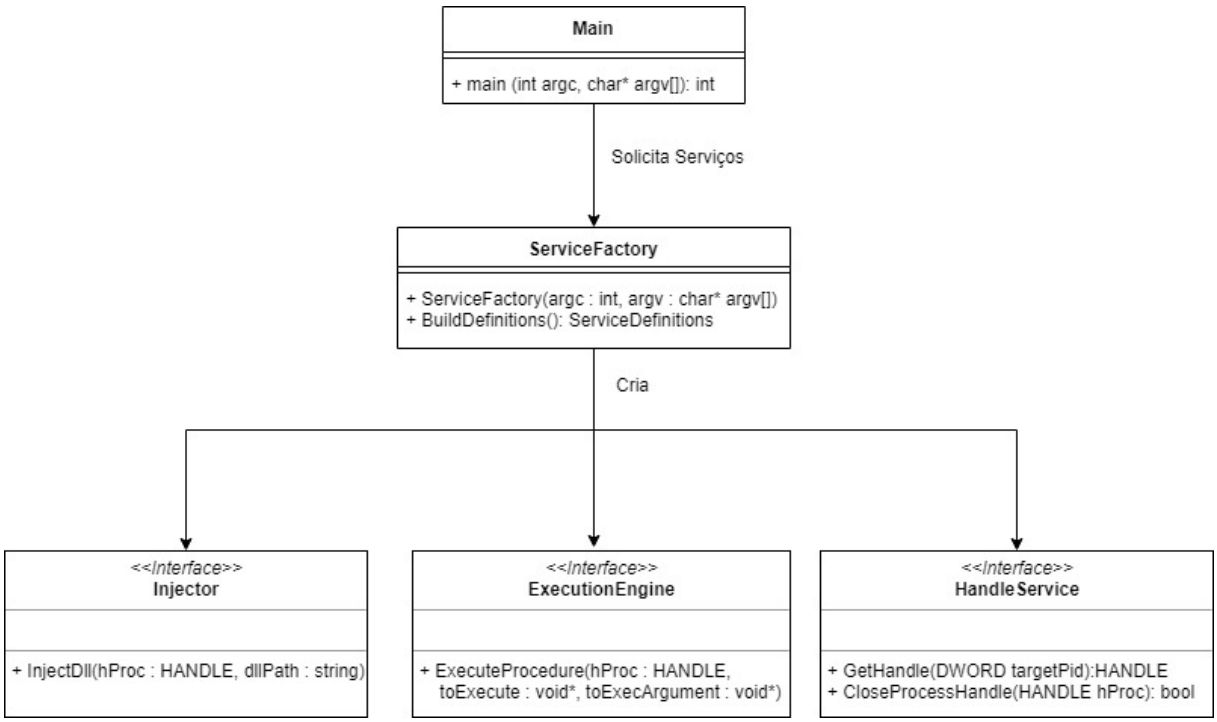

Figura 3.1: Diagrama UML da fábrica de serviços.

3.3.2 Serviços de handles

Serviços de handles são as abstrações responsáveis pela obtenção e gerenciamento de handles para o processo alvo, fornecendo à aplicação injetora uma forma de manipulá-lo por meio da handle obtida. As implementações concretas deste módulo são o **HandleCreatorService** e o **PyJackService**, conforme ilustrado pela Figura 3.2

HandleCreatorService

O **HandleCreatorService** é um serviço concreto trivial que cria uma handle diretamente para o processo alvo e pode ser utilizado quando o usuário não está preocupado com detecção pois soluções de segurança a nível de kernel, em especial anti-cheats, conseguem

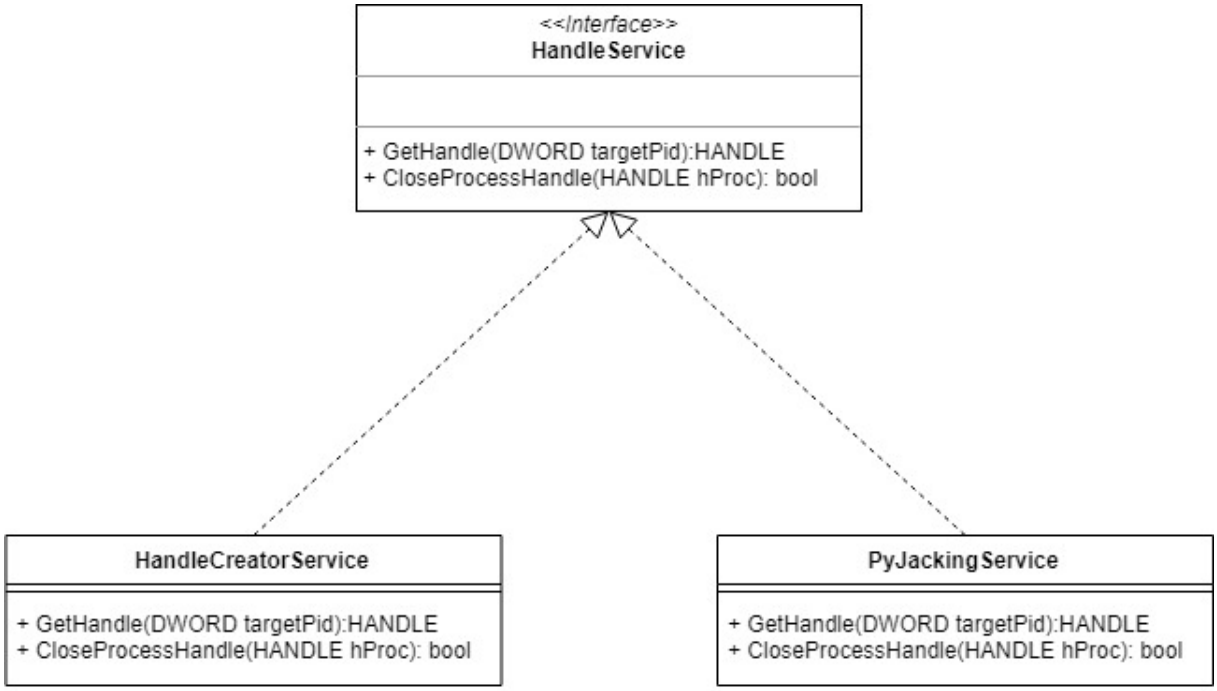

Figura 3.2: Diagrama UML das implementações concretas de serviços de handles.

interceptar chamadas para o sistema operacional, remover privilégios de handles criadas e sinalizar o processo que solicitou a handle como um atacante em potencial.

PyJackingService

O **PyJackService** foi projetado como uma alternativa furtiva ao **HandleCreatorService** que não cria novas handles para o processo alvo. Ao invés de criá-las, o serviço procura por handles para o alvo já existentes em outros processos e as utiliza para dar prosseguimento à injeção. O processo do qual este **HandleService** faz parte é chamado de **Pyjack** e será descrito com mais detalhes nas próximas seções deste capítulo.

3.3.3 Injetores

Os **injetores** são os módulos responsáveis por realizar a inserção da DLL dentro do processo alvo por meio de operações na memória, alocando espaço e copiando a DLL para dentro do hospedeiro. Após a injeção, este módulo faz um pedido para o **motor de execução** executar o código injetado. As implementações concretas deste módulo são o **LoadLibraryInjector**, **ManualMappingInjector** e o **Pyjector**, conforme ilustrado pela Figura 3.3.

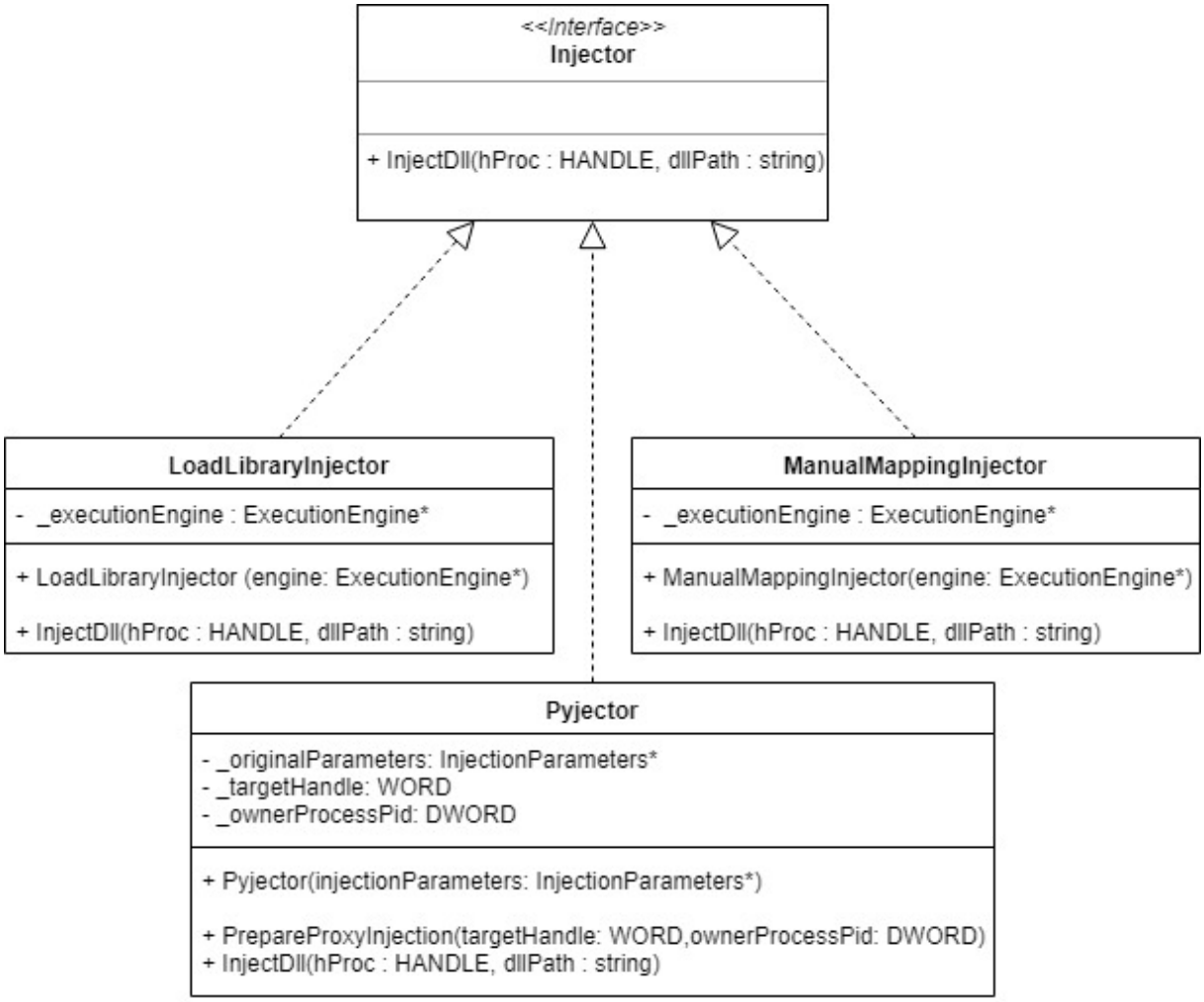

Figura 3.3: Diagrama UML das implementações concretas de injetores.

LoadLibraryInjector

O **LoadLibraryInjector** é um injetor trivial que realiza uma injeção no processo alvo por intermédio do sistema operacional. Conforme mencionado anteriormente, por se tratar de uma injeção intermediada pelo sistema operacional, este injetor deve somente ser utilizado quando o usuário não está preocupado com detecção, pois a injeção, além ser facilmente interceptada por soluções de segurança a nível de kernel, pode ser detectada por soluções a nível de usuário por meio de operações simples de enumeração dos módulos do processo [21].

Pyjector

O **Pyjector** é outro módulo que faz parte do **Pyjack** e é utilizado em conjunto com o **PyJackService** para realizar a etapa de injeção proxy do processo. O processo detalhado

do **Pyjack** será descrito mais adiante neste capítulo.

ManualMappingInjector

O **ManualMappingInjector** é a implementação concreta de um injetor que realiza o papel do kernel do sistema operacional de carregar a biblioteca no processo alvo manualmente. A grande vantagem deste injetor é que a DLL não fica registrada na lista de módulos e o processo de injeção não passa por uma função no kernel com possíveis hooks instalados por softwares de segurança por se tratar de uma função com forte potencial de uso malicioso. Portanto, este injetor é ideal para usuários que pretendem realizar injeções de forma silenciosa.

Tal forma de injeção também permite que um atacante não precise salvar a DLL em disco para conduzir a injeção, bastando apenas os bytes do payload em memória, que podem vir de um centro de centro de comando e controle, de um arquivo encriptado ou simplesmente num formato **hardcoded** no código fonte. Tais técnicas de provisionamento dos bytes da DLL dificultam significativamente o seu escaneamento por soluções de segurança, uma vez que não há um caminho para ela no disco ou, caso haja, ela não estará num formato aparente. Vale ressaltar que nenhuma dessas formas de provisionamento foram implementadas neste trabalho pois fogem do escopo do processo de injeção em si.

3.3.4 Motores de execução

Os **Motores de execução** são os módulos responsáveis pela execução do código injetado, recebendo como parâmetros um endereço de memória executável e um argumento que será utilizado na execução das instruções informadas. As implementações concretas deste módulo são os motores **CreateRemoteThreadEngine** e **ThreadHijackingEngine**, conforme ilustrado pela Figura 3.4

CreateRemoteThreadEngine

O **CreateRemoteThreadEngine** é um motor de execução que cria uma thread no processo alvo via a API pública do Windows **CreateRemoteThread()** para executar as instruções solicitadas. Por utilizar uma API pública do Windows com alto risco de detecção, tal motor deve ser utilizado somente em casos o usuário não esteja preocupado com furtividade na injeção.

ThreadHijackingEngine

O **ThreadHijackingEngine** já é um motor de execução mais sofisticado, que encontra uma thread ativa dentro do processo alvo e redireciona o seu fluxo de execução para a

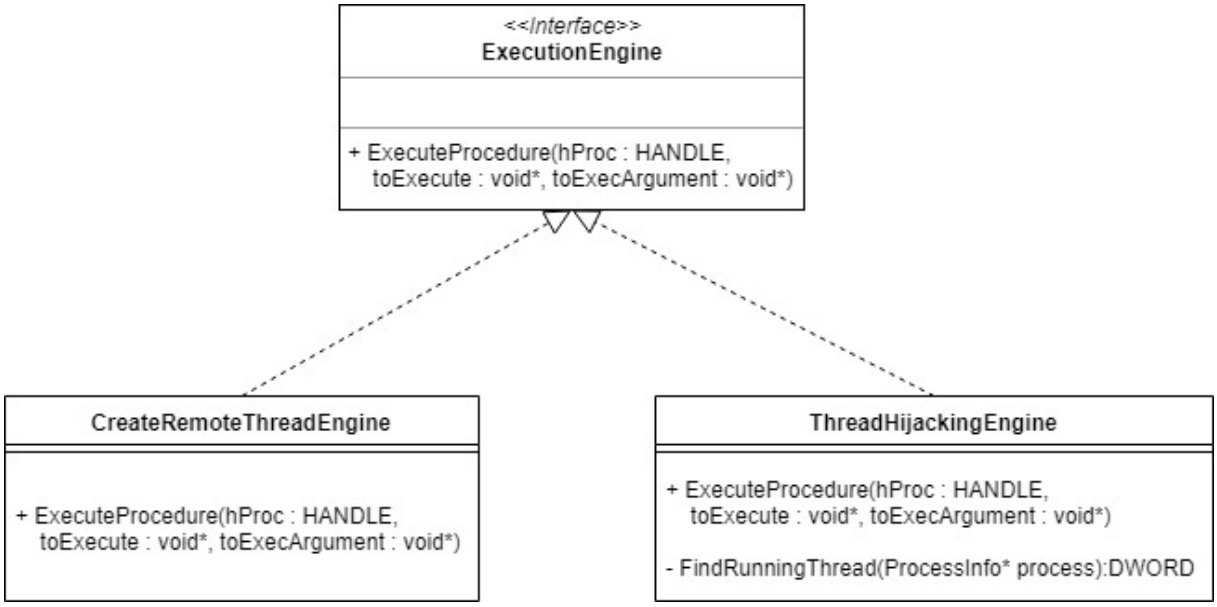

Figura 3.4: Diagrama UML das implementações concretas de motores de execução.

instrução fornecida como parâmetro, retomando o fluxo de instruções original do processo após completar a execução do código parasita.

3.3.5 Interação entre os módulos

Conforme demonstrado pelo diagrama de comunicação na Figura 3.5, temos que a execução do programa ocorre, em alto nível, da seguinte maneira:

1. Os parâmetros informados pelo usuário são passados para a **fábrica de serviços** que irá criar instâncias concretas de cada interface conforme especificado pelo usuário;
2. Utilizando-se do **serviço de handles**, o programa injetor obtém uma handle para o processo alvo;
3. A handle obtida é passada, conjuntamente com o caminho para a DLL a ser injetada, ao **módulo de injeção**, que fica encarregado de realizar as alocações necessárias e preparar a DLL para execução dentro do processo alvo;
4. Uma vez que a DLL foi alocada, o **módulo de injeção** informa um endereço de memória executável ao **motor de execução**, que ficará responsável por executar aquele código.

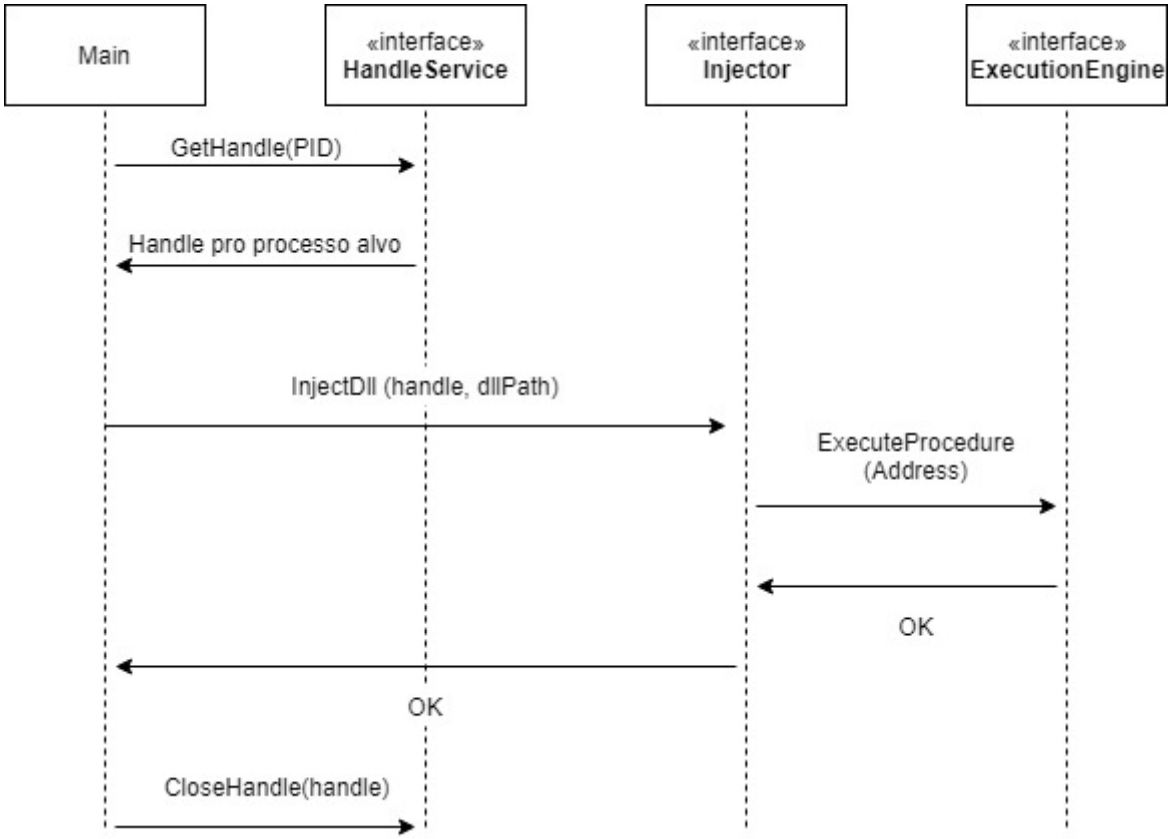

Figura 3.5: Diagrama de comunicação entre os módulos.

3.4 Mapeamento Manual

Conforme já apresentado no Capítulo 2, o processo de injeção por mapeamento manual implementado pelo **ManualMappingInjector** pode ser divido em uma série de etapas.

Carregamento e validação

Na primeira etapa da injeção, os bytes da DLL são copiados para a memória do injetor e validados por meio da verificação do Magic Number e da arquitetura da DLL. O Magic Number são 2 bytes fixos presentes no início de todas DLLs cuja a representação em hexadecimal é 0x5A4D e podem ser observados na Figura 2.5 no início do cabeçalho. Caso o número não esteja presente no payload, ele é dado como inválido. Já a verificação de arquitetura é necessária pois uma DLL compilada em x86 não pode ser injetada num processo x64 e vice-versa.

Alocação da memória no processo alvo

Na segunda etapa o injetor tenta alocar um bloco de memória no processo alvo para a DLL em seu endereço base preferencial por meio da API **VirtualAllocEx()** [30]. Como a probabilidade do processo alvo conter o endereço preferido pela DLL no sistema e este endereço estar disponível é muito baixa, o injetor muito provavelmente realizará a alocação fora deste endereço, necessitando realizar o processo de **relocação** mais adiante.

Mapeamento das seções de memória

Na terceira etapa o injetor mapeia todas as seções na memória do processo alvo, copiando-as por meio da API **WriteProcessMemory()** para o espaço alocado na etapa anterior. Para realizar o mapeamento, o injetor precisa utilizar vários campos da tabela de sessões exibida na Figura 2.6. O campo **VirtualAddress** é usado para definir onde a memória vai ser alocada no processo alvo e os campos **PointerToRawData** e **SizeOfRawData** são utilizados para definir o segmento de memória que será copiado para o endereço alvo.

Cópia do cabeçalho para o processo alvo

Após o mapeamento da seções em memória, o injetor copia o cabeçalho PE para dentro do processo alvo, preparando-o para a etapa de resolução interna das tabelas restantes via um *shellcode* que também será injetado no processo. Vale ressaltar que é seguro copiar o cabeçalho para dentro do processo alvo após o mapeamento das seções sem arriscar sobrescrevê-las pois a região na qual o cabeçalho é mapeado não intersecciona a região em que as seções foram mapeadas. Isso ocorre pois apesar do cabeçalho PE ser pequeno, o formato define que cada seção mapeada em memória não pode ser menor do que o tamanho mínimo de uma página de memória [31], que na arquitetura x86-64 representa 4096 bytes [32]. Por conta desta restrição o cabeçalho PE será sempre mapeado na página 0 de RVA 0x0000 seguido pela a primeira seção do arquivo que é mapeada na página 1 de RVA 0x1000.

Alocação, injeção e execução do Shellcode de resolução interna

Após mapear as seções e copiar o cabeçalho para o processo alvo, o restante da injeção precisa ser orquestrado internamente ao processo alvo. Para isso, o injetor aloca memória no alvo para um *shellcode* que será injetado e em seguida executado para a conclusão do mapeamento. O *shellcode* ficará responsável por:

- Resolver a tabela de relocações (caso necessário);
- Resolver a tabela de imports;

- Executar os TLS Callbacks;
- Executar a DLL Main.

Uma vez que o *shellcode* esteja alocado dentro do processo alvo, o mapeador manual irá fornecer o seu endereço para que o **motor de execução** selecionado pelo usuário possa executá-lo.

Resolução da tabela de relocações

Caso seja verificado que o módulo não foi alocado em seu endereço preferencial, o que é o caso da grande maioria das injeções por mapeamento manual, o injetor realizará o processo de **relocação**, que consiste em reajustar todas as entradas na tabela de relocação pelo delta resultante da Equação 2.1.

Após o calculo do delta, é preciso iterar todos os blocos e suas as entradas internas de relocação (Figura 3.7), verificando pelo campo **type** (4 bytes) se uma relocação será de fato necessária para o **offset** (12 bytes) correspondente. A estrutura de uma entrada pode ser observada na Figura 3.7.

```
//@[comment("MVI_tracked")]
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
//  WORD    TypeOffset[1];
} IMAGE_BASE_RELOCATION;
typedef IMAGE_BASE_RELOCATION UNALIGNED * PIMAGE_BASE_RELOCATION;
```

Figura 3.6: Estrutura de dados definida no header winnt.h que descreve um bloco de relocação.

A iteração é feita utilizando o RVA do bloco inicial, dado pelo campo **BaseRelocationTable** no cabeçalho PE (Figura 2.6), e o tamanho de cada bloco, dado pelo campo **SizeOfBlock** (Figura 3.6 e Figura 3.7). O endereço inicial do iterador é definido como o RVA do primeiro bloco e o endereço do próximo bloco será definido pela soma do endereço do bloco atual $*Bl_n$ com o tamanho do bloco atual Bl_{Sn}, conforme a Equação 3.1, seguindo a iteração até que o campo **VirtualAddress** do bloco atual seja *nullptr*.

$$* Bl_{(n+1)} = *Bl_n + Bl_{Sn} \tag{3.1}$$

Já a iteração das entradas de relocação não é tão simples. Como a estrutura que define o cabeçalho do bloco (Figura 3.6) não fornece acesso tipado às suas entradas internas de relocação, foi necessário utilizar aritmética de ponteiros para referenciá-las.

O primeiro passo para a iteração é definir quantas entradas o bloco possui por meio da Equação 3.2:

$$N = \frac{Bl_S - H_S}{W_S} \tag{3.2}$$

onde N é o número de entradas de relocação num bloco, Bl_S é o tamanho total do bloco dado pelo campo **SizeOfBlock**, H_S é o tamanho do cabeçalho do bloco dado for **SizeOf(_IMAGE_BASE_RELOCATION)** e W_S é o tamanho de uma entrada de relocação dado por **SizeOf(WORD)**.

Uma vez calculado o número de blocos, o *shellcode* inicializa um iterador do tipo *WORD** com o endereço da primeira entrada de relocação, que fica logo em seguida do cabeçalho do bloco sendo iterado (Figura 3.6) e passa por cada entrada analisando se o seu tipo necessita de relocação, sendo necessária a sua realização caso type = 3 em x86 ou type = 10 em x64 [33]. Uma vez constatada que a relocação é necessária, o *shellcode* a realiza, ajustando pelo delta calculado na Equação 2.1 o valor contido no endereço $*P = *B + *R + \Delta O$, onde $*B$ é o endereço base do módulo alocado, $*R$ é o RVA do bloco de relocação e ΔO é o offset contido na entrada de relocação.

Realização da ligação dinâmica

Nesta etapa, o *shellcode* importa DLLs externas consumidas pelo módulo mapeado por meio função **LoadLibraryA()** e preenche a **Import Address Table** (IAT) com o endereço real das dependências carregadas. Para realizar o carregamento das funções externas, o *shellcode* injetado percorre a **Import Table**, que lista todas as dependências externas, e importa cada uma.

Apesar desta etapa utilizar a função **LoadLibraryA()**, vale ressaltar que em nenhum momento a DLL suspeita é importada pela função e sim a suas dependências. De forma geral, importar DLLs adicionais que sejam legítimas no processo alvo não é um problema pois grande parte das soluções de segurança confiam em DLLs assinadas pela Microsoft ou outros fabricantes confiáveis, mas caso um atacante queira deixar o mínimo de rastro possível, é preferível consumir na DLL injetada somente DLLs que são utilizadas pelo processo alvo, pois dessa forma nenhuma DLL de sistema adicional será incluida na lista de módulos.

Após a importação dos módulos externos via **LoadLibraryA()**, o endereço real das funções presentes em tais módulos pode ser obtido pela API **GetProcAddress()**, que consegue realizar a resolução do endereço tanto pelo nome da função quanto o seu ordinal, que nada mais é do que um identificador numérico para as funções exportadas de um

<table>
<tr><td rowspan="7">Block[1]</td><td colspan="4">VirtualAddress</td></tr>
<tr><td colspan="4">SizeOfBlock</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>00</td><td>00</td></tr>
<tr><td rowspan="7">Block[2]</td><td colspan="4">VirtualAddress</td></tr>
<tr><td colspan="4">SizeOfBlock</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>00</td><td>00</td></tr>
<tr><td>...</td><td colspan="4">...</td></tr>
<tr><td rowspan="7">Block[n]</td><td colspan="4">VirtualAddress</td></tr>
<tr><td colspan="4">SizeOfBlock</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>type:4</td><td>offset:12</td></tr>
<tr><td>...</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>type:4</td><td>offset:12</td><td>00</td><td>00</td></tr>
</table>

Figura 3.7: Estrutura da cadeia de blocos de relocação em memória.

módulo. Tais endereços são então utilizados para preencher a IAT com os endereços reais das funções carregadas.

Execução dos TLS callbacks

Após a conclusão da ligação dinâmica, o *shellcode* executa uma série de funções registradas no cabeçalho PE chamadas de **Thread Local Storage** callbacks. Tais funções são executadas antes mesmo da main da DLL e tem como principal objetivo a inicialização de variáveis globais comuns a uma thread. Por se tratar de um trecho de código executável que é chamado antes da Main, existem malwares como o *Nadnadzzz* que executam a sua lógica da infecção dentro desses callbacks, possibilitando a infeção de máquinas utilizadas para realizar engenharia reversa no programa. Mesmo que um analista tenha sido cuidadoso de colocar um breakpoint na main, os TLS callbacks serão executados de qualquer forma, a não ser que o analista também inclua breakpoints nestas funções [34].

Chamada da DLLMain

Nesta etapa o *shellcode* executa o ponto de entrada da DLL, que tem o seu endereço definido no cabeçalho PE pela variável **AddressOfEntryPoint** (Figura 2.5), dando início à execução da lógica principal do módulo injetado.

Limpeza dos recursos alocados

Uma vez que o programa injetor verifica que o *shellcode* foi executado com sucesso, é realizada uma limpeza da memória alocada pelo injetor, reduzindo os rastros da injeção na memória do processo alvo.

3.5 Pyjack

O Pyjack é uma técnica de obtenção de handles para um processo alvo sem a necessidade de que uma handle direta seja criada, dificultando a detecção da injeção. A técnica consiste em sequestrar de um processo qualquer, chamado de **handle owner**, uma handle legítima para o processo alvo com privilégios suficientes para realizar a injeção. O processo pode ser dividido em 4 etapas, sendo a primeira realizada pelo **PyJackingService** e as demais pelo **PyJector**:

1. Busca por uma handle com privilégios suficientes para realizar uma injeção no processo alvo;
2. Carregamento do módulo de injeção no processo handle owner;
3. Injeção do payload via o processo handle owner;
4. Remoção dos vestígios da injeção.

Busca por uma handle com privilégios para o processo alvo

Nesta primeira etapa o programa injetor utiliza o **PyJackingService** para iterar todas as handles do sistema por meio de APIs e estruturas de dados não-documentadas em busca da handle ideal, ou seja, que aponte para o processo alvo e possua privilégios suficientes para conduzir o processo de injeção, conforme ilustrado pela Figura 3.8. Por não possuírem documentação oficial da Microsoft, as estruturas de dados utilizadas no injetor foram adaptadas de soluções de terceiros, que publicaram as suas definições para as tais estruturas [35] [36].

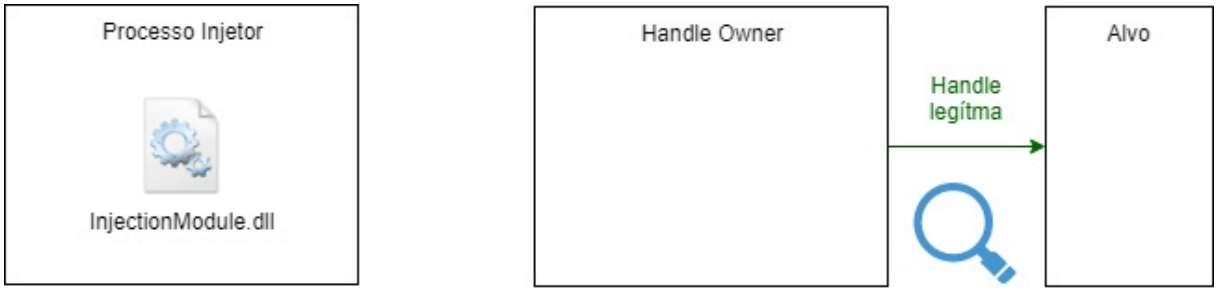

Figura 3.8: Primeira etapa - busca de handles para o alvo.

Para enumerar as handles, o injetor carrega dinamicamente a biblioteca *Ntdll.dll*, que funciona como uma ponte entre o Kernel e user-mode, para subsequentemente chamar a função **NtQuerySystemInformation()** [37]. A documentação da função não especifica nenhuma funcionalidade de enumerar handles, porém passando os parâmetros corretos e interpretando a resposta da forma apropriada, é possível realizar tal enumeração.

O injetor busca na coleção de handles retornada possíveis candidatas para a injeção, verificando se as mesmas possuem as permissões necessárias. Uma vez que a handle é encontrada, o injetor armazena o seu identificador e o PID do processo alvo para iniciar a próxima etapa.

Carregamento do módulo de injeção no processo handle owner

Para realizar a injeção pela handle encontrada, o programa injetor carrega a DLL de injeção no processo handle owner encontrado na etapa anterior, conforme ilustrado pela Figura 3.9, e o utiliza como hospedeiro para conduzir a injeção final por meio de sua handle já existente para o processo alvo.

A injeção no handle owner se inicia com uma busca pelo diretório da DLL de injeção. Para isso é utilizada a API **CreateToolhelp32Snapshot()** para obter um snapshot do processo injetor e o macro **Module32Next()** para iterar os módulos carregados no momento do snapshot em busca da DLL de injeção e o seu diretório. De posse do dire-

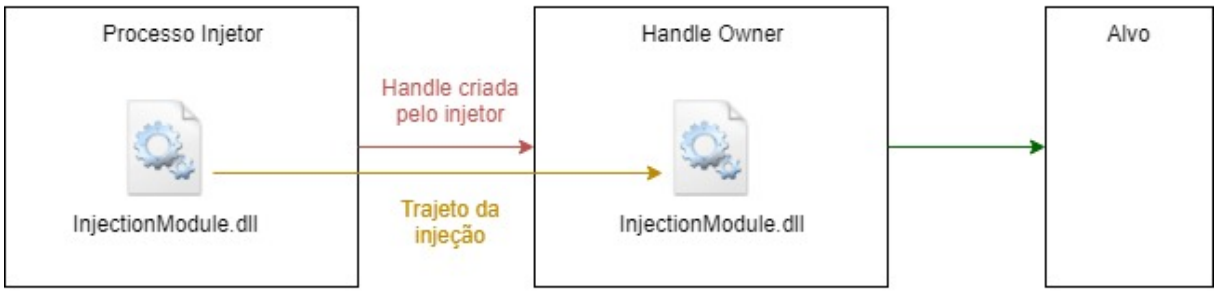

Figura 3.9: Segunda etapa - Injeção do módulo no handle owner.

tório da biblioteca, o processo injetor chama recursivamente a função de injeção, porém parametrizando uma injeção no processo handle owner ao invés do processo alvo.

Nesta chamada recursiva, o injetor cria uma handle para o handle owner e inicia o carregamento da biblioteca de injeção. Como o injetor precisará orquestrar externamente chamadas para a DLL por meio de APIs do Windows, a implementação do módulo injetor utilizado nesta etapa é o **LoadLibraryInjector**, que carrega a DLL por meio do sistema operacional e facilita a chamada remota posteriormente.

Já que a injeção não ocorre diretamente no processo protegido, o uso de APIs convenientes do Windows para carregamento de bibliotecas e criação de handles dificilmente será marcada como suspeita por softwares de segurança protegendo o processo alvo final, o que justifica o seu uso.

Concluída a injeção, o injetor verifica se a biblioteca foi devidamente carregada no handle owner por meio da enumeração de seu snapshot em busca da biblioteca, de forma análoga à como foi encontrada o caminho da DLL de injeção carregada no processo injetor.

Injeção do payload via o processo handle owner

Com a confirmação de que a DLL de injeção foi inserida no handle owner, o injetor inicia a injeção final conduzido-a de dentro do processo handle owner, conforme ilustrado pela Figura 3.10.

Como a função injetada no handle owner é executada de forma remota pelo processo injetor por meio da API **CreateRemoteThread()**, faz-se necessário encontrar manualmente o seu endereço na memória do hospedeiro. Para isso, o processo injetor verifica o endereço relativo da função injetora $*f_{rel}$ no cabeçalho PE do *injectionModule.dll* e a ajusta pelo endereço base da biblioteca de injeção dentro do processo hospedeiro $*B$, obtendo-se o endereço $*f_h$ da função dentro do hospedeiro, conforme explicitado pela Equação 3.3.

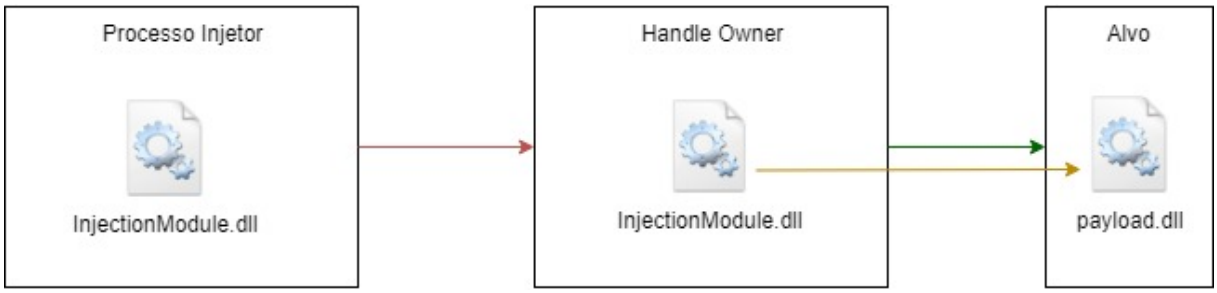

Figura 3.10: Terceira etapa - Carregamento do payload por meio do hospedeiro.

$$* f_h = *f_{rel} + *B \tag{3.3}$$

Sabendo-se o endereço $*f_h$ da função dentro do handle owner, o processo injetor aloca memória no mesmo para os argumentos da segunda injeção, preenchendo-os com parâmetros definidos pelo usuário e a handle a ser utilizada, e a inicia passando o endereço $*f_h$ da função injetora com argumentos para o motor **CreateRemoteThreadEngine**. Assim que a execução é concluída, o processo injetor limpa os rastros que deixou na memória.

Remoção dos vestígios da injeção

Nesta etapa o injetor libera a seção de memória alocada no handle owner para os parâmetros, remove a DLL de injeção, encerra as handles para o handle owner e encerra a sua própria execução.

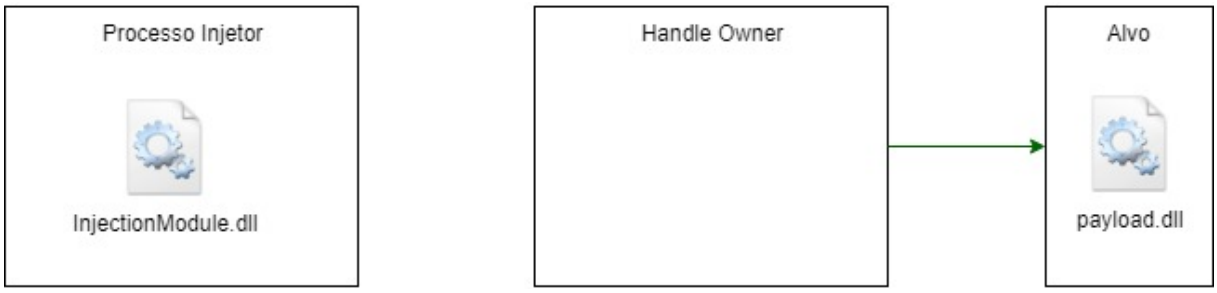

Figura 3.11: Quarta etapa - Remoção dos vestígios da injeção.

Para realizar a remoção da DLL de injeção na memória do handle owner, o programa injetor procura no processo a função **FreeLibrary** da biblioteca *ntdll.dll* utilizando a Equação 3.3 e a executa em uma thread nova. Vale ressaltar que como a maior parte dos programas interagem com o Kernel indiretamente ou diretamente, é aceitável assumir que a biblioteca *ntdll.dll* esteja carregada no hospedeiro. Caso a biblioteca não esteja disponível, o injetor não conseguirá remover o módulo que foi carregado via **LoadLibraryA**.

3.5.1 Limitações

A técnica **Pyjack** necessita que ambos os processos envolvidos sejam da mesma arquitetura, impossibilitando que a uma handle de um processo x64 para um processo x86 (ou vice-versa) seja sequestrada e utilizada para injeção, o que reduz o leque de possibilidades em seu uso. Como os processos detentores de handles válidas para os alvos são geralmente de sistema e nativos à arquitetura do computador, como o é caso do *svchost.exe*, o uso da técnica seria restrita em processos alvos x86 dentro de computadores x64, pois os processos que normalmente detêm handles válidas não poderiam ser utilizados. Caso um outro processo x86 detenha uma handle para o alvo de mesma arquitetura, a técnica poderia ser aplicada normalmente, porém este cenário não é comum. A condução da técnica em processos alvos x64 dentro computadores x64 não apresenta nenhuma limitação.

3.6 Thread Hijacking

A execução por **Thread Hijacking**, implementada pelo **ThreadHijackingEngine**, consiste em pausar um thread já existente no processo alvo e desviar o seu fluxo de execução para um endereço desejado, retomando o fluxo original ao concluir a execução do código desejado.

Apesar de um desvio por **Thread Hijacking** apresentar semelhanças conceituais com o Inline Hooking, as duas técnicas são bastante diferentes, pois enquanto o Inline Hooking desvia o fluxo código por meio da sobrescrita na memória executável do processo alvo, o **Thread Hijacking** atinge o mesmo resultado operando diretamente sobre o registrador de endereços do processador ao nível de assembly, o que o torna um método de desvio muito mais versátil. Para realizar um Inline Hooking é necessário conhecimento interno do alvo para determinar onde instalar o hook e recuperar o código sobrescrito, com **Thread Hijacking** é possível desviar código de forma arbitrária sem a necessidade de conhecer os detalhes internos do assembly do alvo.

Por se tratar de uma forma de executar código arbitrário parametrizado em qualquer processo **sem criar nenhuma thread**, o **ThreadHijackingEngine** é o motor de execução ideal para injeções que desejam deixar o mínimo de rastros possíveis. O motor de execução realiza o procedimento com as seguintes etapas:

1. Busca por uma thread em execução dentro do processo alvo;
2. Suspensão da execução na thread encontrada;
3. Injeção do assembly de desvio do fluxo de instruções;

4. Sobrescrita do registrador de instruções com o endereço do assembly injetado e retomada da execução da thread;

5. Limpeza dos recursos alocados.

Busca por uma thread em execução dentro do processo alvo

Para encontrar uma thread em execução dentro do processo alvo, o motor de execução utiliza a API **NtQuerySystemInformation()** [37] para buscar informações sobre o processo. Ao contrário da busca por informações de handles, a utilização da API para consultar informações de processos é parcialmente documentada pela Microsoft, que omite o significado de alguns campos retornados por essa API de uso interno do Windows. Conforme explicitado na Figura 3.12, o motor de execução reinterpreta a estrutura do Windows Internals como uma estrutura que explicita o significado de cada campo retornado, sendo a definição enriquecida fruto de uma adaptação de trabalhos de engenharia reversa do Kernel [38] [39] para esta ferramenta.

```
typedef struct _SYSTEM_PROCESS_INFORMATION
{
    ULONG           NextEntryOffset;
    ULONG           NumberOfThreads;
    LARGE_INTEGER   WorkingSetPrivateSize;
    ULONG           HardFaultCount;
    ULONG           NumberOfThreadsHighWatermark;
    ULONGLONG       CycleTime;
    LARGE_INTEGER   CreateTime;
    LARGE_INTEGER   UserTime;
    LARGE_INTEGER   KernelTime;
    UNICODE_STRING  ImageName;
    KPRIORITY       BasePriority;
    HANDLE          UniqueProcessId;
    HANDLE          InheritedFromUniqueProcessId;
    ULONG           HandleCount;
    ULONG           SessionId;
    ULONG_PTR       UniqueProcessKey;
    SIZE_T          PeakVirtualSize;
    SIZE_T          VirtualSize;
    ULONG           PageFaultCount;
    SIZE_T          PeakWorkingSetSize;
    SIZE_T          WorkingSetSize;
    SIZE_T          QuotaPeakPagedPoolUsage;
    SIZE_T          QuotaPagedPoolUsage;
    SIZE_T          QuotaPeakNonPagedPoolUsage;
    SIZE_T          QuotaNonPagedPoolUsage;
    SIZE_T          PagefileUsage;
    SIZE_T          PeakPagefileUsage;
    SIZE_T          PrivatePageCount;
    LARGE_INTEGER   ReadOperationCount;
    LARGE_INTEGER   WriteOperationCount;
    LARGE_INTEGER   OtherOperationCount;
    LARGE_INTEGER   ReadTransferCount;
    LARGE_INTEGER   WriteTransferCount;
    LARGE_INTEGER   OtherTransferCount;
    SYSTEM_THREAD_INFORMATION Threads[1];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    PVOID Reserved2;
    ULONG HandleCount;
    ULONG SessionId;
    PVOID Reserved3;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG Reserved4;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    PVOID Reserved5;
    SIZE_T QuotaPagedPoolUsage;
    PVOID Reserved6;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PagefileUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrivatePageCount;
    LARGE_INTEGER Reserved7[6];
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

Figura 3.12: Comparação entre a definição da estrutura utilizada pelo motor de execução (esquerda) e a estrutura definida no cabeçalho winternl.h pela Microsoft (direita).

A lista de informações sobre processos retornada pela chamada ao Kernel precisa ser iterada em busca do processo alvo por meio da comparação do campo **UniqueProcessId** da estrutura descrita na Figura 3.12 com o PID do processo alvo desejado. A iteração dos elementos ocorre conforme a Equação 3.4.

$$* P_{(n+1)} = *P_n + \Delta N \tag{3.4}$$

onde $*P_{(n+1)}$ é o endereço do próximo bloco de informações, $*P_n$ é o endereço do bloco de informações atual e ΔN é o offset para a próxima entrada, dado pelo campo **NextEntryOffset**.

Uma vez que a estrutura de informações sobre processo alvo é encontrada, inicia-se a busca por uma thread em execução, utilizando o campo não-documentado **Threads**, contido na estrutura enriquecida do Kernel na Figura 3.12. O motor de execução então itera as threads presentes no processo e procura por uma que esteja em execução.

Suspensão da execução da thread encontrada

Após encontrar uma thread em execução dentro do processo alvo, o processo injetor abre uma handle para a thread com uma mascara de acesso amplo via a API **OpenThread()** [40] e suspende a sua execução chamando função **SuspendThread()** [41]. Antes de realizar qualquer alocação ou modificação, o motor de execução consulta o contexto da thread por meio da API **GetThreadContext()** [42] e armazena o endereço da instrução original do contexto no qual a thread foi suspendida numa variável auxiliar.

Injeção do assembly de desvio do fluxo de instruções

Nesta etapa o motor de execução aloca memória no processo alvo para um shellcode binário de assembly que será injetado por meio da API **VirtualAllocEx()** [30]. Em C++ a estrutura utilizada para representar tal código foi um **BYTE[ ]**, onde o assembly foi escrito byte por byte dentro do vetor. O shellcode desenvolvido tem como objetivo:

1. Salvar o endereço da instrução original que foi desviada na stack;
2. Salvar o contexto de todos os registradores antes de executar o desvio;
3. Realizar o desvio para a função desejada;
4. Restaurar o contexto de registradores;
5. Sinalizar que a execução do código desviado foi bem-sucedida;
6. Retornar para a instrução original utilizando a variável salva na stack.

Pelo fato do shellcode lidar com parâmetros que só serão resolvidos em tempo de execução e presentes somente no escopo do código em C++, como por exemplo o endereço de execução original armazenado na variável auxiliar, o endereço para onde o desvio irá ocorrer e o argumento passado para a função de desvio, foi necessária uma estratégia para construir o shellcode binário de forma dinâmica. Para isso, foram criadas lacunas de tamanho e localização fixos dentro do vetor de bytes, que serão preenchidas com os valores adequados antes de injetar o código. Desta forma foi possível estabelecer um meio de transmitir dados do contexto do C++ para o contexto de assembly em binário.

Uma vez que o shellcode construído dinamicamente esteja completo com todos os parâmetros necessários, a injeção do payload executável no processo alvo se inicia por meio da API **WriteProcessMemory()** [43].

Sobrescrita do registrador de instruções com o endereço do assembly injetado e retomada da execução da thread

Após a conclusão da injeção do shellcode no processo alvo, o motor de execução sobrescreve o ponteiro de instruções da thread com o endereço do código assembly injetado e resume a sua execução a partir deste novo endereço por meio da API **ResumeThread** [44]. Assim que a execução for iniciada, o motor de execução encerra a handle para a thread, já que nenhuma nova operação será realizada na mesma.

Limpeza dos recursos alocados

Para evitar a remoção de recursos que ainda estão em uso, o motor de execução precisa se certificar que o código executando no outro processo foi concluído com sucesso. Para isso, um mecânismo de sincronização entre dois processos foi implementado: Quando o código passado para o motor de execução termina de executar, o shellcode sinaliza este fato por meio de uma flag na memória alocada para si. O motor de execução, por sua vez, entra num loop de espera ocupada que realiza sucessivas buscas pela flag na memória do processo alvo por meio da API **ReadProcessMemory()** [45] até que o assembly sinalize a conclusão da execução. Confirmada a conclusão, o motor de execução libera a memória do shellcode e sinaliza para o módulo solicitante que a execução do endereço informado **toExecute** (veja Figura 3.4) foi bem-sucedida.

3.7 Interface de uso

Ao chamar o injetor pela linha de comando sem nenhum parâmetro ou passando o parâmetro **-h** , é exibido um manual de como utilizar a ferramenta, conforme demonstrado

na Listagem 3.1. Caso algum parâmetro não seja reconhecido, a aplicação irá mostrar para o usuário somente os possíveis valores para aquele parâmetro, conforme exibido na Listagem 3.2.

3.8 Testes e Resultados

3.8.1 Ferramentas utilizadas

Para verificar a eficácia do injetor desenvolvido, foi utilizada uma ferramenta de monitoramento em tempo real do sistema operacional chamada de **Process Monitor** [46], que permite ver em detalhes a atividade do sistema operacional ou de um processo.

Foi também utilizada uma ferramenta de monitoramento de processos e detecção manual de malwares, chamada de **Process Hacker** [47], para listar os módulos injetados dentro de um processo e verificar o estado interno de threads.

3.8.2 Naive Injection

Na primeira sequência de testes, cuja a saída do console está exibida na Listagem 3.3, foi realizada uma injeção simples, despreocupada com furtividade e quase que completamente intermediada pelo sistema operacional. Tal injeção utiliza o **LoadLibraryInjector** para o injetor, **HandleCreatorService** para o serviço de handles e **CreateRemoteThread** para o motor de execução. Tal formato de injeção possui utilidade para softwares legítimos, como **Fraps**[18], **MSI Afterburner** [18] e **NVidia GeForce Experience** [19], que se injetam em aplicações *fullscreen* para exibir detalhes em tempo real de utilização de recursos do sistema.

Após a conclusão da injeção pelo processo injetor, foi confirmado por meio de testes funcionais no processo alvo que a injeção foi bem sucedida, verificando-se que todas as funcionalidades providas pela DLL injetada estavam disponíveis para o usuário. Ao verificar a captura dos eventos de sistema feita pela ferramenta Process Monitor percebe-se que realizar uma injeção com esses parâmetros deixa uma série rastros reveladores no sistema respectivamente enumerados na Figura 3.13:

1. Uma thread é criada remotamente no processo alvo por um processo externo, que neste caso é o injetor;
2. O carregamento da DLL é evidente e revela o path da DLL carregada;
3. Uma nova handle para o processo alvo é criada para conduzir a injeção. Tal comportamento não pode ser observado em nenhuma das imagens pois o ciclo de vida

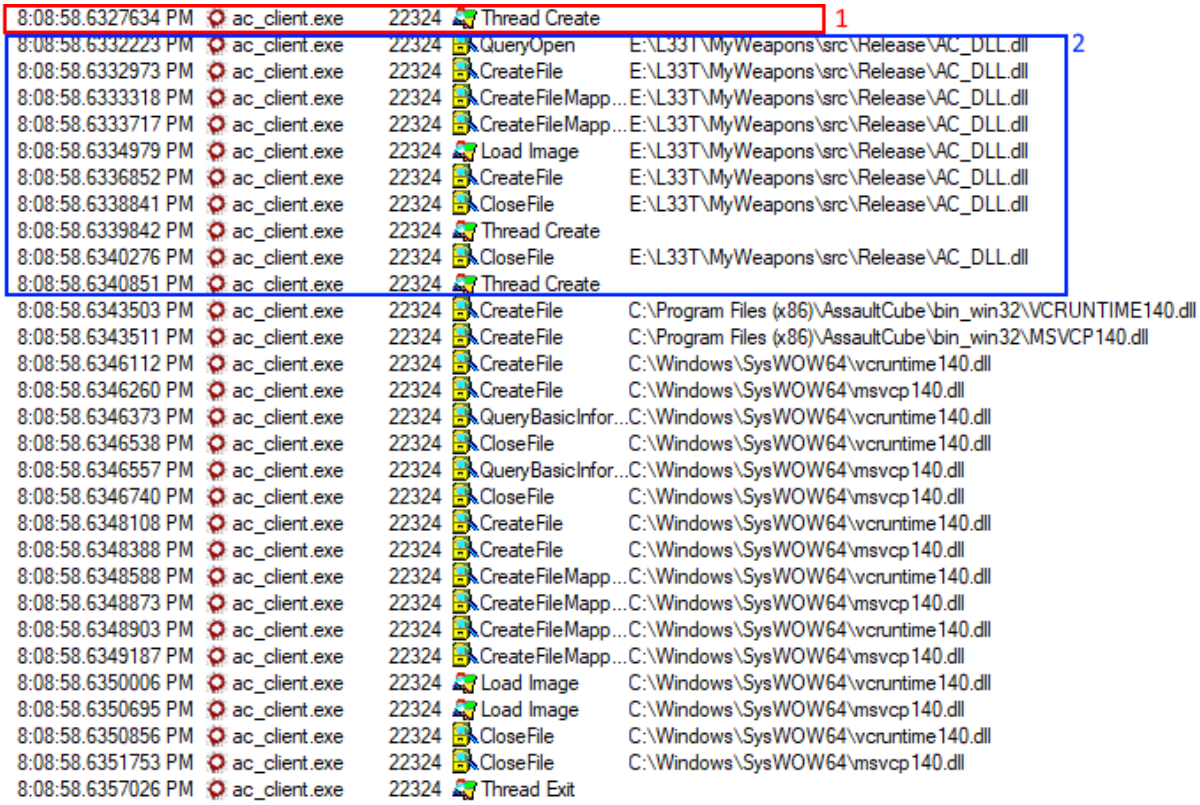

Figura 3.13: Captura de eventos do sistema pela ferramenta Process Monitor durante a injeção.

da handle é muito curto, porém uma solução de segurança atenta às criações pode detectá-las.

Também é possível notar que a DLL injetada aparece na lista de módulos do processo, conforme ilustrado pela imagem Figura 3.14

3.8.3 Stealth Injection

Nesta segunda sequência de testes, cuja a saída do console está exibida na Listagem 3.4, foram utilizados os componentes furtivos do programa injetor para conduzir a injeção, sendo eles o **ManualMappingInjector** para o injetor, **Pyjacking Service** para o serviço de handle e **ThreadHijackingEngine** para o motor de execução.

Para a realização deste teste, um programa auxiliar que cria uma handle para o processo alvo foi executado, visando a garantia de que haveria um processo handle owner da mesma arquitetura com uma handle privilegiada o suficiente para realizar a injeção.

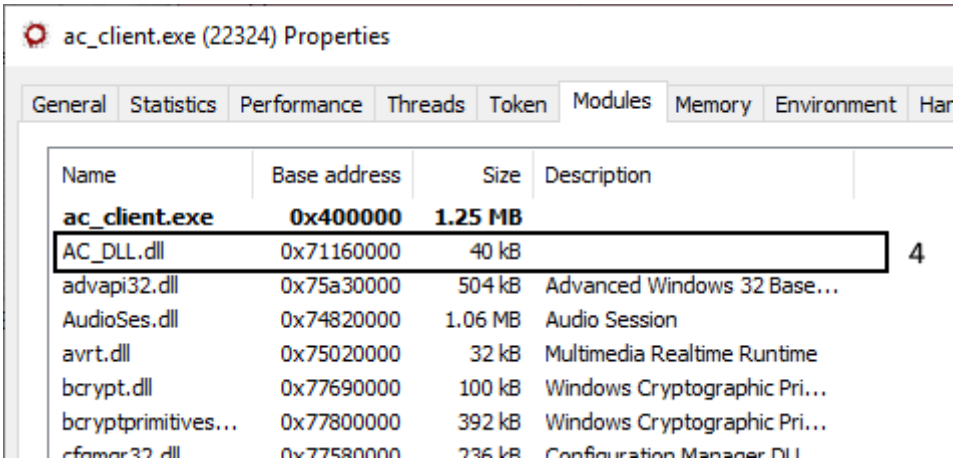

Figura 3.14: Módulos carregados no processo alvo da injeção.

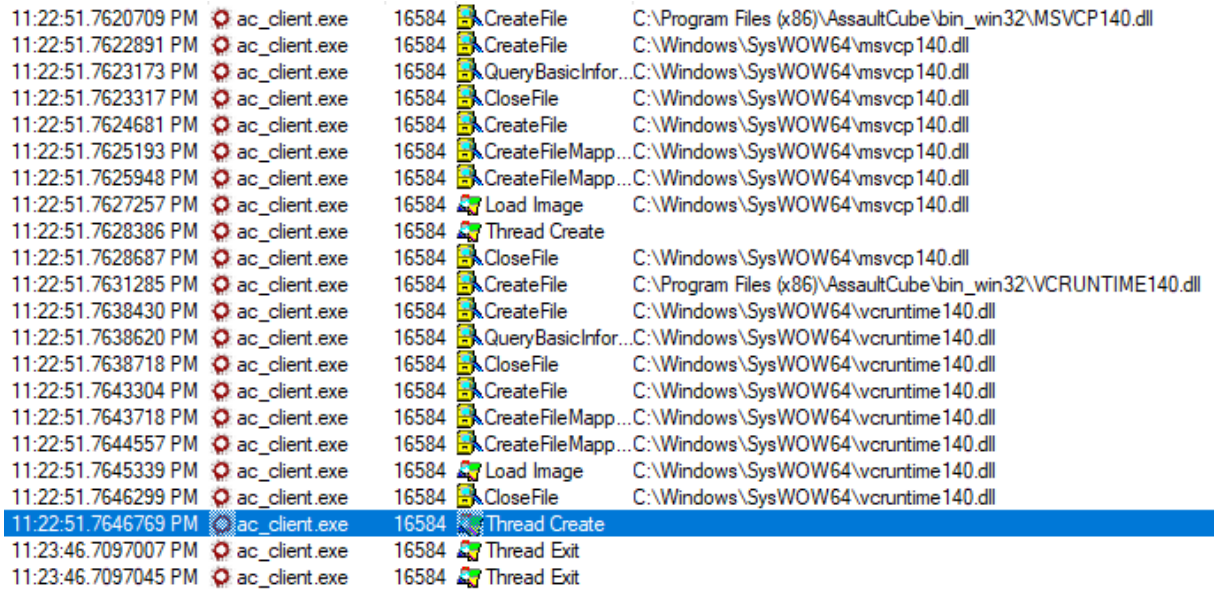

Figura 3.15: Captura de eventos do sistema pela ferramenta Process Monitor durante a injeção stealth.

Novamente foi possível observar que a injeção ocorreu de forma bem sucedida no processo alvo, porém, conforme ilustrado pela Figura 3.15, desta vez a injeção deixou bem menos rastros:

1. Não há rastros no sistema que citem explicitamente o path da DLL injetada;
2. Nenhuma thread remota foi criada para orquestrar a injeção;
3. Nenhuma nova handle foi utiliza para conduzir a injeção;

4. A DLL não consta na lista de módulos do processo, conforme ilustrado pela Figura 3.16.

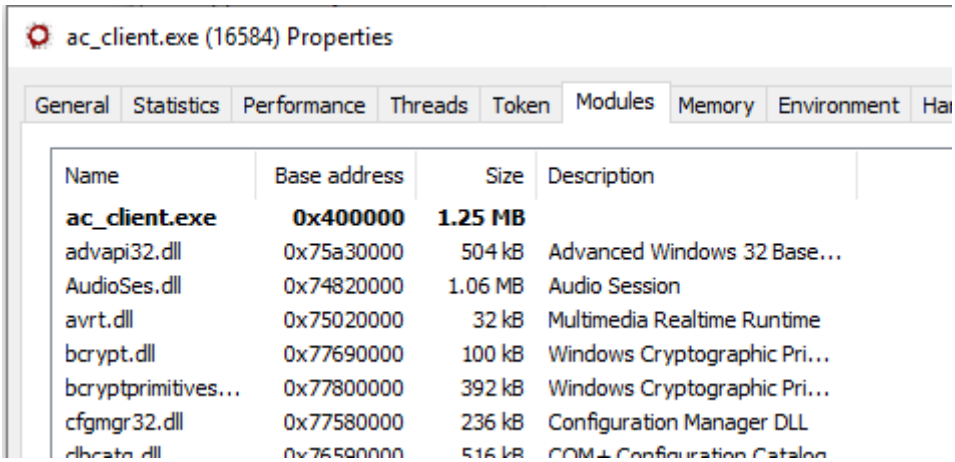

Figura 3.16: Módulos carregados no processo alvo da injeção stealth.

As novas threads que aparecem no Process Monitor como **Thread Create** foram criadas pelo próprio processo alvo durante a importação dos módulos externos pelo mapeador manual, que importa tais módulos via LoadLibrary. Conforme explicado anteriormente neste capítulo, o uso desta API não deve apresentar nenhum aumento no risco de detecção pois as DLLs **vcruntime140.dll** e **msvcp140.dll** importadas na Figura 3.15 são assinadas pela Microsoft e portanto consideradas confiáveis.

Por se tratar de criações de thread orquestradas pelo próprio processo, elas não são dadas como suspeitas por softwares de segurança. Observe que é possível confirmar a origem da criação da thread ao explorar o stack de chamadas que a originou, ilustrado pela pela Figura 3.17.

Ao analisarmos o log do injetor, escrito num arquivo .txt e exibido na Listagem 3.5, é possível perceber que:

1. A handle do handle owner de PID 9752, ilustrado pela Figura 3.18, foi de fato utilizada na injeção;

2. A thread principal do jogo de TID 19940, ilustrada pela Figura 3.19, foi de fato utilizada para executar o mapeamento manual.

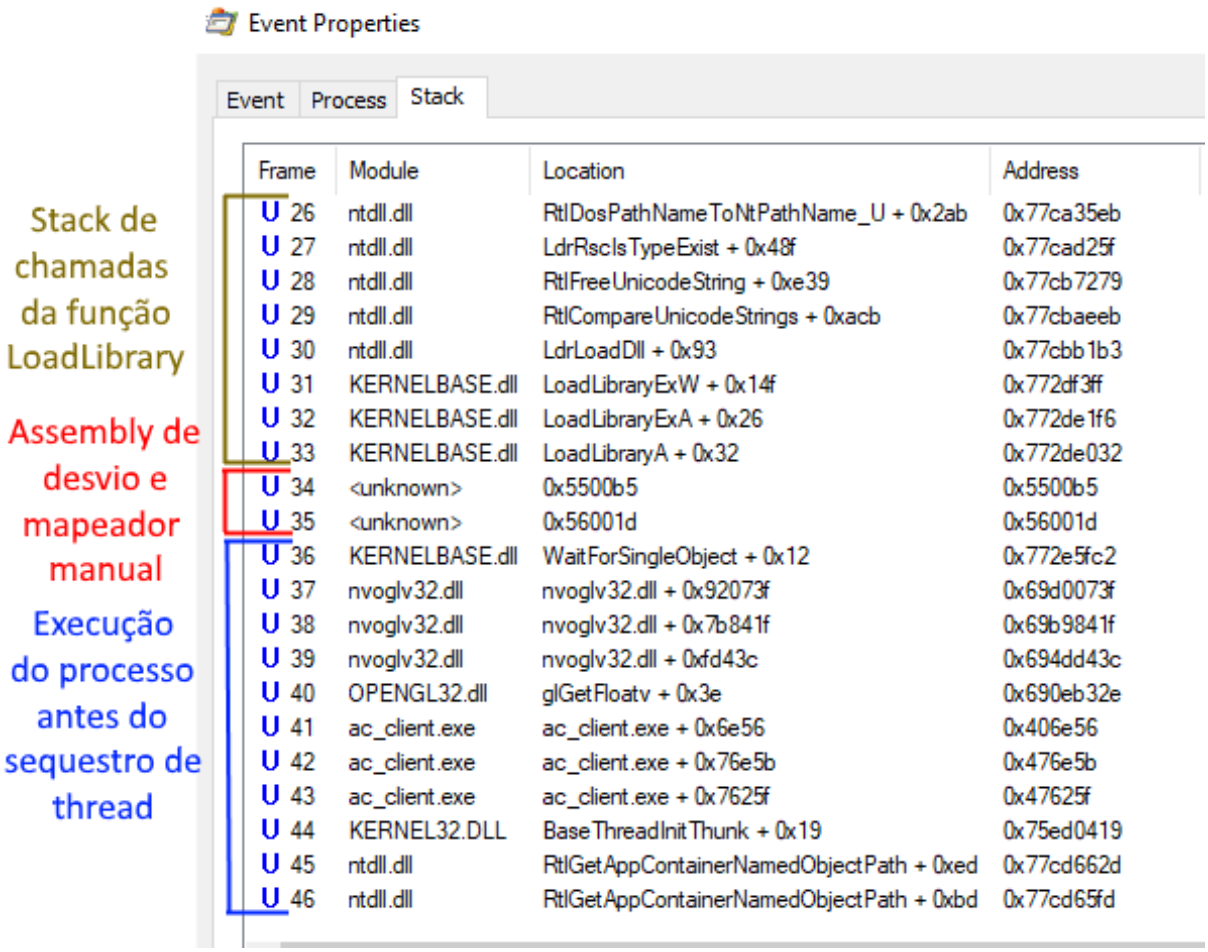

Figura 3.17: Call stack das funções que originaram a criação da thread.

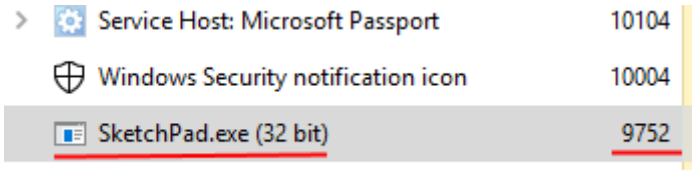

Figura 3.18: Processo handle owner auxiliar.

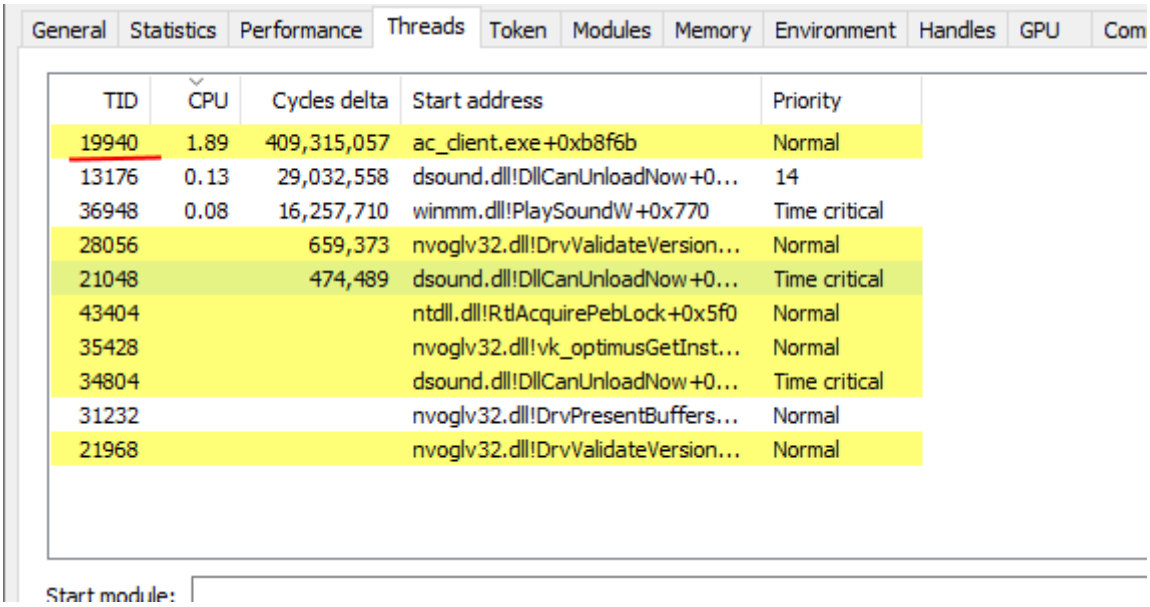

Figura 3.19: Threads to processo alvo.

```
.\DLLInjector.exe -h

Usage: DLLInjector <TargetProcessName> <DLLPath> <Injector>
<HandleService> <ExecutionEngine>

where <Injector> can be one of:

         LLI - Load Library Injector
         MMI - Manual Mapping Injector

<HandleService> can be one of:

         HC - Handle Creator Service
         PYJ - Pyjacking Service

<ExecutionEngine> can be one of:

         CRT - Create Remote Thread Engine
         THE - Thread Hijacking Engine
```

Listing 3.1: Output do console quando a função de ajuda é executada

```
.\DLLInjector.exe "ac_client.exe"
"E:\L33T\MyWeapons\src\Release\AC_DLL.dll" LLI LOL CRT

Selected target process: ac_client.exe
Selected injector: LoadLibraryInjector

Invalid handle strategy LOL
<HandleService> can be one of:

         HC - Handle Creator Service
         PYJ - Pyjacking Service
```

```
Selected execution engine: CreateRemoteThreadEngine
```

Listing 3.2: Output do console quando um parâmetro não é reconhecido

```
.\DLLInjector.exe "ac_client.exe"
"E:\L33T\MyWeapons\src\Release\AC_DLL.dll" LLI HC CRT

Selected target process: ac_client.exe
Selected injector: LoadLibraryInjector
Selected Handle service: HandleCreatorService
Selected execution engine: CreateRemoteThreadEngine
DLL E:\L33T\MyWeapons\src\Release\AC_DLL.dll Successfully
injected!

Closing injector in 4 seconds
```

Listing 3.3: Output do console para a injeção ingênua

```
.\DLLInjector.exe "ac_client.exe"
"E:\L33T\MyWeapons\src\Release\AC_DLL.dll" MMI PYJ THE

Selected target process: ac_client.exe
Selected injector: ManualMappingInjector
Selected Handle service: Pyjacking Service
Selected execution engine: ThreadHijackingEngine

DLL E:\L33T\MyWeapons\src\Release\AC_DLL.dll Successfully
injected!
```

Listing 3.4: Output do console para a injeção silenciosa

```
Starting Injection
Process found!
```

```
Found a valid handle owner for proxied injection. PID: 9752
-------------- Commencing proxy injection. ----------------


Starting Injection
Thread Hijacker found running thread. Thread Id: 19940
Thread Hijacker completed

Closing Handle (if needed)
```

Listing 3.5: Arquivo de log do injetor

Capítulo 4

DLL Desenvolvida

A DLL que será injetada no jogo Assault Cube visa demostrar de forma prática o quanto um atacante precisa conhecer sobre o seu alvo para que algumas de suas características possam ser exploradas.

4.1 O Assault Cube

O jogo Assault Cube, criado em 10 de novembro de 2013, é um *First-person shooter* 3D multijogador baseado na *CUBE engine* no qual jogadores competem em 8 modos diferentes pela vitória. Assim como a maioria dos jogos do seu gêneros, cada jogador possui uma quantidade definida de vida e é equipado com uma arma de sua escolha com uma quantidade de munição pré-determinada, conforme exibido no canto inferior esquerdo da Figura 4.1. Jogadores também podem encontrar itens espalhados pelo mapa que os auxiliam na partida, como pacotes que recuperam vida, munição, coletes que reduzem o dano recebido e granadas.

Sempre que um jogador acerta outro com um disparo ou granada, a vida do jogador atingido é reduzida pelo dano que a arma utilizada causa. Sempre que a vida de um jogador atinge o valor 0, o jogador é dado como morto e precisa esperar alguns segundos para renascer. Apesar de diferentes modos considerarem diferentes critérios para vitória (como capturar uma bandeira ou matar um certo número de inimigos), o *core gameplay* do jogo envolve matar outros jogadores para atingir ou facilitar a vitória.

4.2 Funcionalidades da DLL

Em termos funcionais, a DLL deve garantir ao jogador vantagens competitivas que nenhum outro jogador honesto poderia obter para este jogo. Elas são:

Figura 4.1: Perspectiva da um jogador durante uma partida do jogo Assault Cube.

- Munição infinita - Ativada pela tecla F1;
- Tiro Rápido - Ativada pela tecla F2;
- Tiros sem recuo - Ativada pela tecla F3;
- Todas as armas automáticas - Ativada pela tecla F4;
- Vida infinita - Ativada pela tecla F5;
- Percepção Extra Sensorial (ESP), que permita o jogador ver a posição de todos os outros jogares por trás de qualquer obstáculo - Ativada pela tecla F6;
- Mira automática na cabeça de jogadores adversários (Aimbot) - Ativada pelo botão direito do mouse.

O código implementado em C++ e assembly realiza operações de leitura e escrita tanto na memória executável do processo alvo, permitindo a modificação arbitrária do

código, quanto na memória comum, que permite a modificação e leitura direta de valores que eram a priori controlados pelo jogo.

4.3 Ferramentas utilizadas

Para a construção deste hack (DLL) foi essencial o uso de duas ferramentas de engenharia reversa para a obtenção das vantagens ilícitas implementadas. As ferramentas utilizadas foram:

- Um analisador de código binário chamado de **OllyDbg** [48];
- Um analisador de memória dinâmica chamado de **Cheat Engine** [49].

4.3.1 OllyDbg

OllyDbg [48] é uma ferramenta de análise de código assembly que o exibe num formato mais amigável ao usuário, exibindo a sintaxe legível do assembly x86 ao lado das instruções binárias e o seu endereço, conforme a Figura 4.2.

Além do suporte para análise estática, a ferramenta também suporta análise dinâmica, permitindo que usuários analisem o seu funcionamento em tempo de execução por meio de *breakpoints* e janelas auxiliares que exibem os valores contidos nos registradores.

Uma funcionalidade diferencial do programa é que ele possibilita a alteração da memória executável de um processo durante a sua execução, o que facilita o entendimento do assembly na prática. Por exemplo, caso um usuário suspeite que uma determinada linha de assembly é responsável por verificar a licença de uso de um programa pago, ele pode comprovar a sua teoria substituindo a chamada para a função de verificação por uma linha de assembly que sempre considere a verificação como válida, implementando desta forma um *license crack*.

É difícil derivar o propósito de alto nível de cada linha baseado somente numa análise do assembly. Para isso, faz-se uso de outras ferramentas, como o **Cheat Engine**, que complementam a análise de código.

4.3.2 Cheat Engine

O **Cheat Engine** [49] é um escaneador de memória, hex editor e debugger que permite o usuário não só visualizar e editar o código assembly de um determinado processo mas também possibilita a visualização e busca de conteúdo em memória.

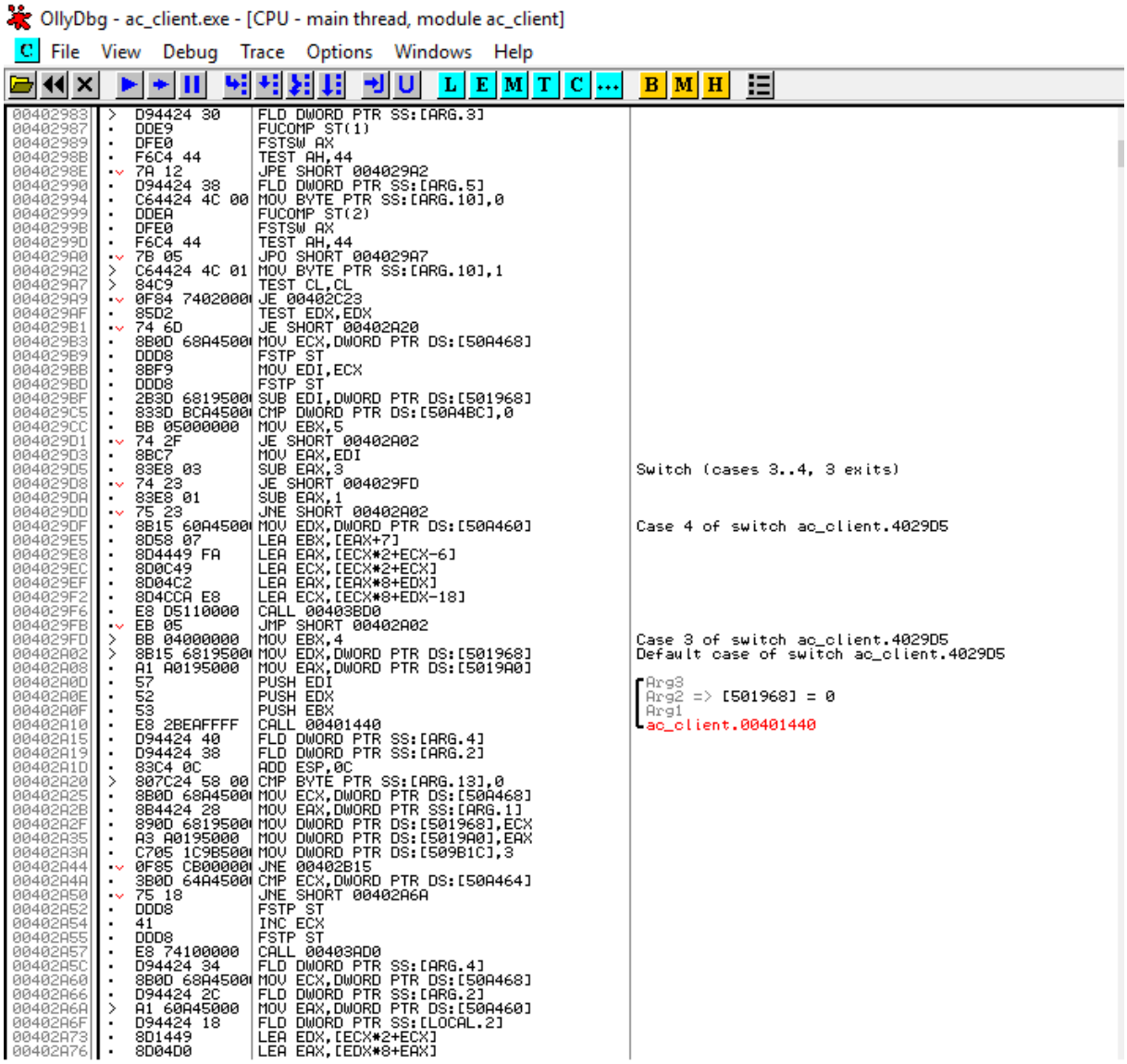

Figura 4.2: Interface de usuário da ferramenta OllyDbg.

Busca por endereços

A principal funcionalidade do programa é a de busca por endereços. Para utilizá-la, o usuário insere um valor na ferramenta e ela procura por aquele valor em todos os endereços de memória do processo alvo. Geralmente uma única busca não é o suficiente para determinar o endereço que referencia o valor procurado, pois é possível que vários endereços diferentes armazenem o mesmo valor. Visando o refinamento da busca, o usuário então deve modificar o valor dentro do processo alvo e realiza uma nova iteração de busca, procurando, dentro dos endereços já encontrados na primeira iteração, os valores que foram alterados para o novo valor procurado. O processo se repete sucessivamente até

que o endereço alvo seja encontrado.

Para exemplificar a funcionalidade descrita acima, suponha que um usuário queira encontrar o endereço de memória que armazena o valor da vida de seu personagem em um jogo. Ao observar a tela do jogo, o usuário percebe que a sua vida é de 100 pontos e realiza um busca em toda a memória do jogo por este valor, conforme a demonstrado na Figura 4.3.

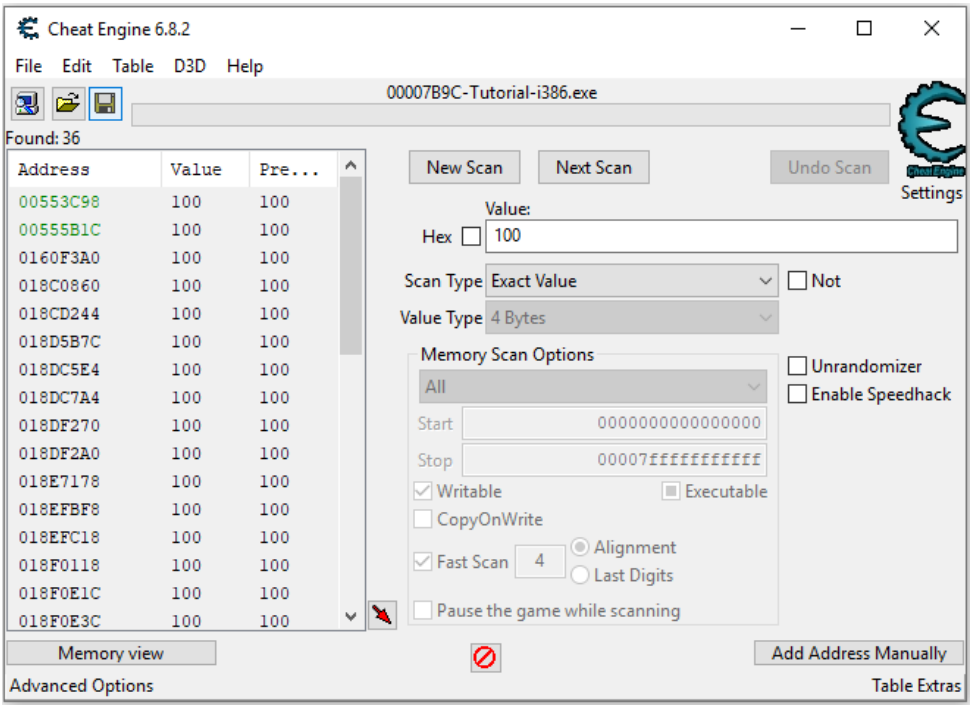

Figura 4.3: Primeira busca pelo endereço da vida.

Analisando o resultado da busca, o usuário percebe que diversos endereços continham aquele valor e que precisa determinar qual deles é o endereço correto. Para isso, o usuário se deixou ser atacado no jogo, reduzindo a sua vida para 97 pontos, e realizou novamente uma busca pelo valor 97 dentro dos endereços encontrados na primeira iteração, resultando em um único endereço encontrado, conforme exemplificado na Figura 4.4. O usuário então confirma que o endereço armazena a vida de seu personagem por meio da manipulação do valor naquele endereço utilizando o **Cheat Engine** e verificando que a sua vida no jogo foi modificada de acordo com a mudança.

Detecção de leitura e escrita a endereços de memória

Esta funcionalidade permite um usuário verificar qual instrução acessa ou modifica um determinado endereço de memória, oque auxilia na determinação do papel de cada linha do assembly. Além de dizer qual o endereço da instrução que acessou ou sobrescreveu a

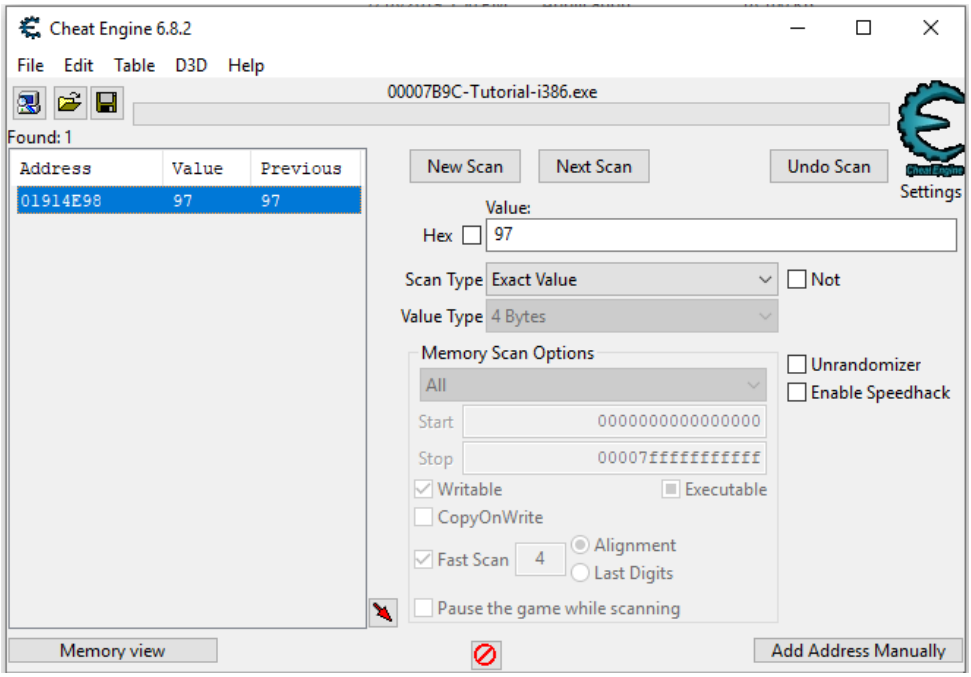

Figura 4.4: Segunda busca pelo endereço da vida.

variável observada e o seu respectivo *OpCode*, a ferramenta também mostra o código assembly correspondente da instrução no seu disassembler, oque facilita a leitura do binário pelo usuário.

Suponha que um usuário deseja saber qual instrução é responsável pelo disparo da arma no jogo e já encontrou o endereço que armazena o valor da sua munição. O usuário então realiza um disparo no jogo e verifica no **cheat engine** qual instrução atualizou o valor da munição, encontrando a região de memória executável responsável pela lógica de disparo.

Escaneamento de ponteiros

A cada vez que o processo alvo reinicia, raramente os valores relevantes encontrados em escaneamentos passados estarão no mesmo lugar pois são alocados e gerenciados dinamicamente pelo programa. Caso um hack queria alterá-los ou lê-los em execuções diferentes do processo é necessário uso de uma estratégia automatizada para encontrar o endereço dinâmico do valor por meio de alguma informação estática sobre a sua localização. Neste caso, faz se o uso de uma cadeia de ponteiros fixa descrita por um endereço base estático e um conjunto de offsets fixos, que, uma vez percorrida, o resultado final será o endereço dinâmico procurado, conforme ilustrado na Figura 4.5.

Para encontrar tal cadeia, a ferramenta disponibiliza um escaneador de ponteiros que traça todos as possíveis cadeias entre endereços bases e o endereço alvo. Assim como

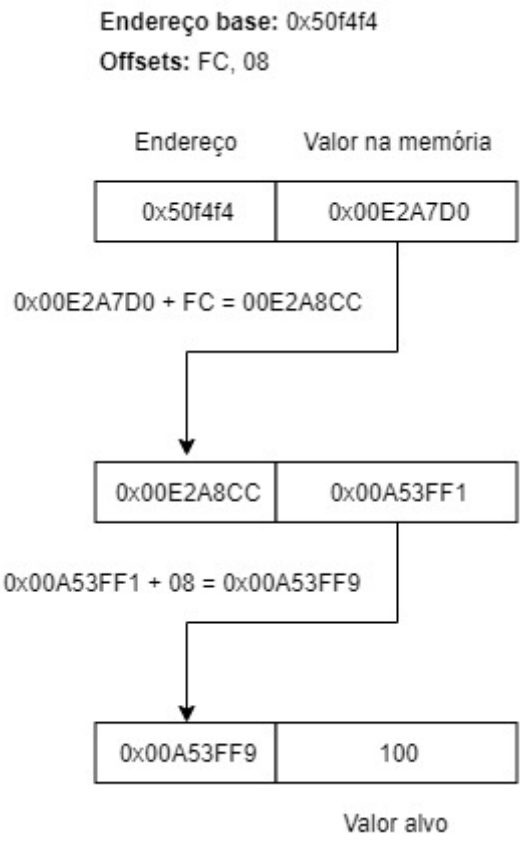

Figura 4.5: Exemplo de uma cadeia de ponteiros para um endereço dinâmico alvo.

na busca por endereços de memória, a busca por cadeia de ponteiros também precisa ser refinada através de sucessivos escaneamentos de cadeias em diferentes instâncias do programa alvo, visando encontrar uma cadeia que seja persistente em múltiplas execuções.

4.4 O Ciclo de vida um jogo

Um jogo eletrônico é um programa executado num loop condicional (*main game loop*) até que o usuário solicite o encerramento da aplicação, sendo que em cada iteração do loop, chamada de **tick** ou **frame**, a informação renderizada na tela para o jogador é calculadas a partir do estado interno do programa após modificações, podendo estas serem originadas do usuário ou da lógica do jogo. Por exemplo, no jogo *Space Invaders*, demostrado pela Figura 4.6, a movimentação dos **invaders** constitui uma mudança no estado interno do jogo promovida pelo próprio algoritmo enquanto a movimentação e os disparos da **nave** são mudanças promovidas pelo jogador.

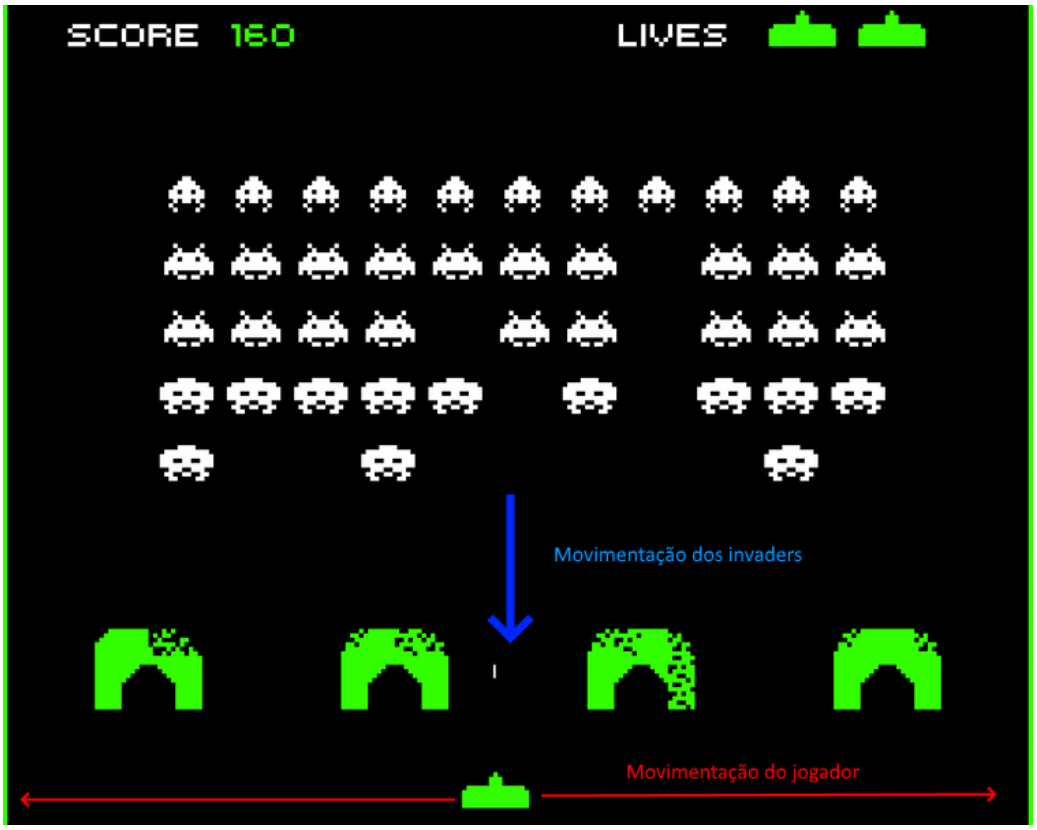

Figura 4.6: Screenshot do jogo *Space Invaders*.

De forma geral, o fluxo de execução básico de um jogo eletrônico pode ser dividido em três etapas, conforme ilustrado pela Figura 4.7:

1. Processamento da **entrada** do jogador;
2. **Atualização** do estado interno do jogo de acordo com uma lógica;
3. **Renderização** do estado atual na tela.

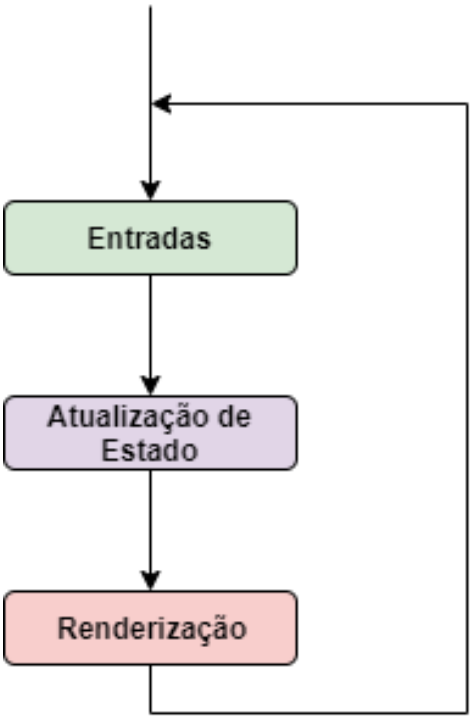

Figura 4.7: Fluxo de execução do game loop.

4.5 O Ciclo de vida do hack implementado

Por se tratar de um programa que opera sobre um estado de memória continuamente atualizado, o hack também é executado num loop contínuo, chamado de **hack loop**, até o encerramento do programa hospedeiro.

Hack loop simples

Nesta implementação, o **hack loop** é composto por uma cláusula **while(true)** que é executada em paralelo ao jogo, conforme explicitado pela Figura 4.8

Apesar de ser simples, esta implementação necessita que a lógica seja executada numa thread dedicada criada pela DLL, o que não aumenta o risco de detecção pois a sua criação não seria remota. Porém, tal abordagem traz uma série de problemas de consistência e de performance.

Como nesta implementação não há nenhum mecanismo de sincronização entre as threads, é possível que a informação que o hack esteja operando seja inválida para aquele frame. Caso o hack realize uma operação num frame antes da etapa de atualização, os dados utilizados não condirão com o estado real do jogo para aquele frame. Como o *frame rate* minimamente viável para um jogo é de 20 FPS [50], a defasagem entre valores operados pelo hack e os reais geralmente não é percebida pelo usuário, pois na pior da hipóteses

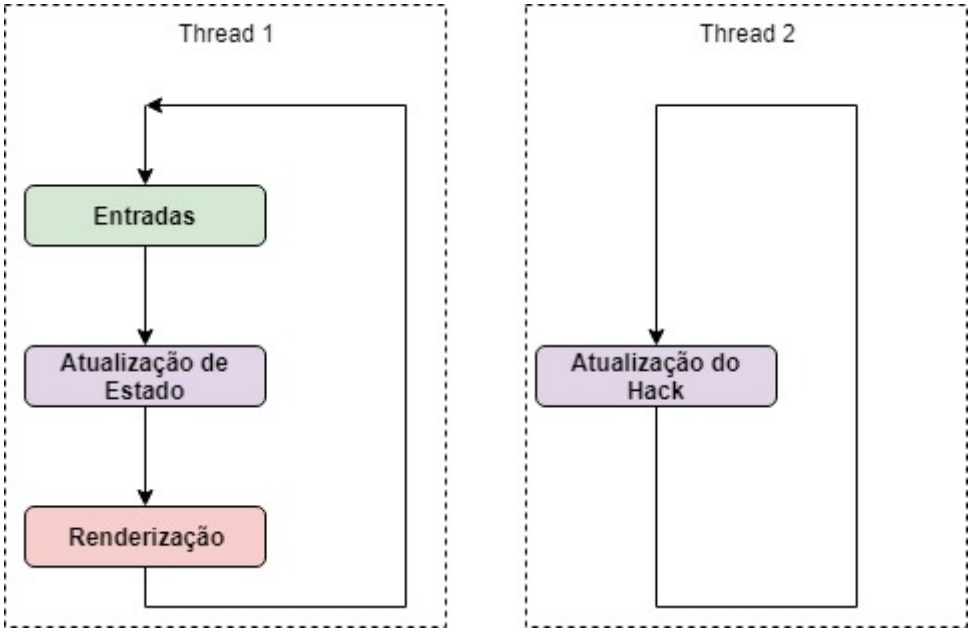

Figura 4.8: Fluxo de execução do hack loop simples.

as vantagens fornecidas pelo hack estariam apenas $\frac{1}{20}$ segundos (1 frame) atrasadas em relação ao estado real do jogo.

Porém, como jogos são aplicações computacionalmente caras, a nova thread utilizada pelo hack pode afetar negativamente o desempenho do jogo, especialmente quando múltiplos *ticks* do hack são executados em um único frame do jogo, degradando a experiência do jogador por meio da redução nas taxas de FPS. Certamente jogos com gráficos pouco realistas sendo executados em computadores potentes não sofreriam com este problema, porém a utilização do hack necessitaria de um hardware mais potente do que os requisitos mínimos do jogo, o que não é ideal. Caso um computador consiga executar o jogo, o mesmo computador deve poder executar o hack.

Hooked Hack Loop

Para solucionar os problemas de consistência e velocidade citados na seção anterior, o hack implementado neste trabalho faz uso de **inline hooking** para executar a sua lógica de forma estratégica no pipeline de execução de um frame do jogo, conforme a demonstrado na Figura 4.9.

Observe que com esta implementação não há mais problemas de consistência pois o estado que será operado pelo hack já é o estado ideal calculado pelo jogo na fase de **atualização** do frame. Consequentemente, não há necessidade de mecanismos de sincronização de threads caso a consistência seja requisito para o bom funcionamento do hack. O problema de performance também é resolvido pois, como a atualização do hack

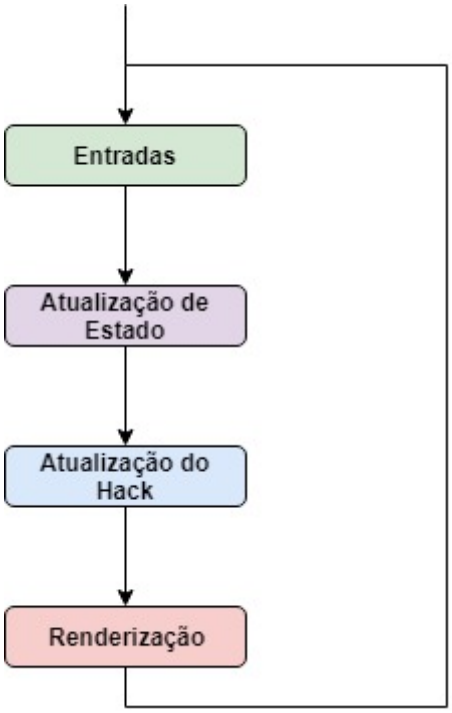

Figura 4.9: Atualização do estado do hack inserido no pipeline de execução de um frame.

ocorre uma única vez por frame, não haverão atualizações repetidas e desnecessárias para o mesmo frame.

Apesar de ser uma solução mais eficiente e elegante para a execução do hack, a sua implementação exige que o desenvolvedor encontre, por meio de engenharia reversa no programa alvo, o local para instalar o hook, o que exige profundo conhecimento do jogo e de suas bibliotecas. No caso deste trabalho, foi utilizada a ferramenta de análise de assembly **OllyDbg** [48] (Figura 4.10) para encontrar o ponto de instalação do hook, localizado no endereço **0x0040C375**, que corresponde ao começo da etapa de renderização do alvo.

4.6 Arquitetura

Ao ser injetado, o hack realiza a instalação do hook antes da etapa de renderização do jogo, garantindo que irá operar sobre o estado ideal do frame. Após a instalação do hook, a DLL cria um vetor de **Hacks** (classe apresentada na Figura 4.11) para armazenar as funcionalidades implementadas em instâncias separadas e aguarda a execução de um tick para ativar/desativar funcionalidades com base na entrada do usuário e atualizar o estado de cada funcionalidade com base nos dados mais recentes do jogo.

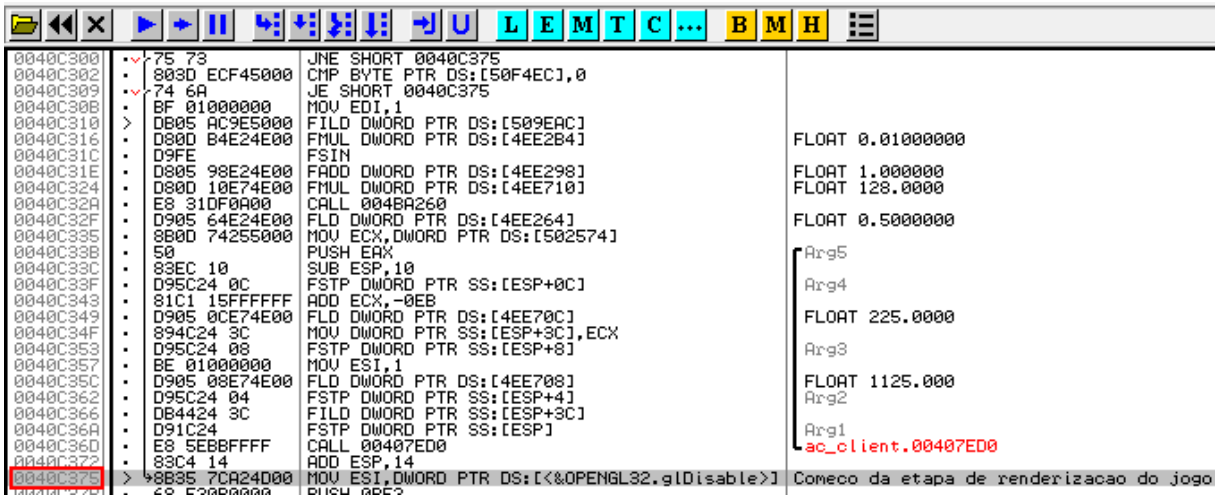

Figura 4.10: Entrada da etapa de renderização do jogo Assault Cube exibida na ferramenta OllyDgb.

4.7 Hacks de memória

Hacks de memória são aqueles que utilizam somente sobrescritas na memória para fornecer alguma vantagem competitiva, sem realizar nenhum cálculo adicional. Tais funcionalidades são implementadas na DLL por meio das classes **SimpleMemoryHacks** e **AssemblyMemoryHacks**.

4.7.1 SimpleMemoryHacks

SimpleMemoryHacks são hacks que sobrescrevem o valor contido num endereço dinâmico por um valor fixo a cada ciclo de atualização do hack, possibilitando que este seja forçado e congelado no endereço alvo. Por exemplo, para a implementação da funcionalidade de vida infinita foi utilizado um **SimpleMemoryHack** que sobrescreve a vida do jogador com o valor 1000 antes da renderização de cada frame. Vale ressaltar que, por se tratar de uma classe com tipagem genérica, a DLL possui acesso tipado à variável, o que garante uma maior estabilidade do hack e facilidade no desenvolvimento, uma vez que o programador estará operando diretamente com tipos estáticos ao invés de bytes.

Conforme ilustrado anteriormente pela Figura 4.5, para encontrar o endereço final contendo o valor a ser sobrescrito é preciso percorrer uma cadeia de ponteiros. Para isso, foi utilizado na aplicação o algoritmo descrito na Listagem 4.1.

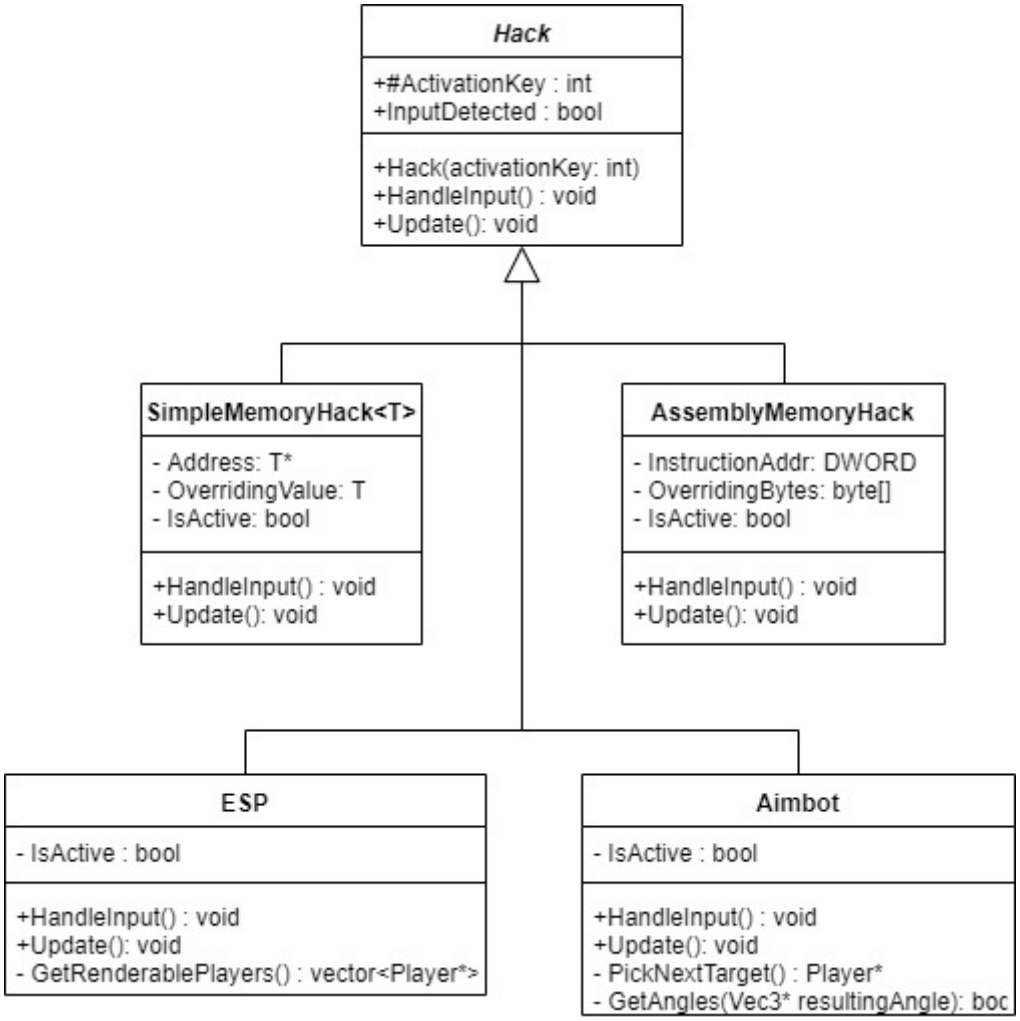

Figura 4.11: Diagrama de classes do hack injetado.

4.7.2 AssemblyMemoryHacks

AssemblyMemoryHacks são aqueles responsáveis por alterar a lógica de execução do jogo por meio de patches na memória executável, ampliando significativamente o leque de possibilidades para as vantagens implementadas. Ao contrário de um **SimpleMemoryHack** que sobrescreve a memória em cada ciclo de atualização, este tipo de hack somente sobrescreve a memória somente uma vez quando é ativado, restaurando a instrução original quando desativado. Utilizando este tipo de hack, foram implementadas as seguintes funcionalidades:

- Munição infinita, implementada com a sobrescrita do código de subtração da munição atual com NOPs (instruções que não fazem nada);
- Tiro rápido, implementado com a substituição da checagem de cadência de tiro por NOPs;

- Tiros sem recuo, implementado com a sobrescrita do código que simula o recuo da arma por NOPs;
- Todas as armas automáticas, implementada pela substituição do código original por um código que sempre classifica qualquer arma como automática.

4.7.3 Limitações

Apesar de fornecerem fortes vantagens ao jogador, tais hacks somente funcionam numa sessão de jogo local contra jogadores artificiais, onde o estado do jogo é computado completamente na máquina do hospedeiro. Em sessões *multiplayer*, onde vários jogadores se conectam numa máquina externa para jogarem juntos, tais funcionalidades não fornecem nenhuma vantagem ao jogador, e podem inclusive prejudicá-lo. Como nessas sessões o estado real do jogo é calculado numa máquina que o jogador não tem controle, não é possível alterar a memória e nem o código executando no servidor, o que inutiliza as vantagens obtidas por hacks de memória. Por exemplo, caso um jogador ative o hack de vida infinita, sua tela irá mostrar que o valor da vida está travado em 1000 independente de qualquer dano que o jogador possa ter recebido, pois o endereço de memória local, utilizado na renderização da tela, é sobrescrito pelo hack. Porém como o estado lógico do jogo no servidor é autoritativo em relação ao cliente, o jogador irá morrer de qualquer forma por decisão do servidor, que verifica em seu estado interno que a vida do jogador chegou em 0 devido ao dano recebido. Desta forma, as funcionalidades implementadas prejudicam o jogador ao invés de auxiliá-lo, pois distorcem o estado de seu jogo com o estado real calculado no servidor. Tais limitações no entanto não se aplicam a funcionalidades mais complexas como o **Aimbot** e **ESP**, descritos nas seções seguintes.

4.8 World to screen

Para melhor entendimento das funcionalidades mais complexas da DLL, é necessária a compreensão da função básica utilizada em suas implementações, chamada de **world to screen**, que converte a coordenada do espaço euclidiano de um objeto 3D no mundo do jogo para uma posição na tela por meio de uma série de transformadas baseadas em multiplicação de matrizes. Esta transformada é utilizada frequentemente por bibliotecas de renderização para projetar objetos do espaço 3D numa tela, como é o caso da **OpenGL** [51] [52], a biblioteca utilizada pelo processo alvo para a renderização do estado do jogo.

4.8.1 Matriz de modelo

O **espaço local** de um objeto em 3D considera que o centro deste objeto é a origem do sistema de coordenadas, sendo bastante utilizado durante a sua concepção pois facilita a definição do modelo de forma isolada, já que um designer não precisa se preocupar com o seu posicionamento num mundo virtual.

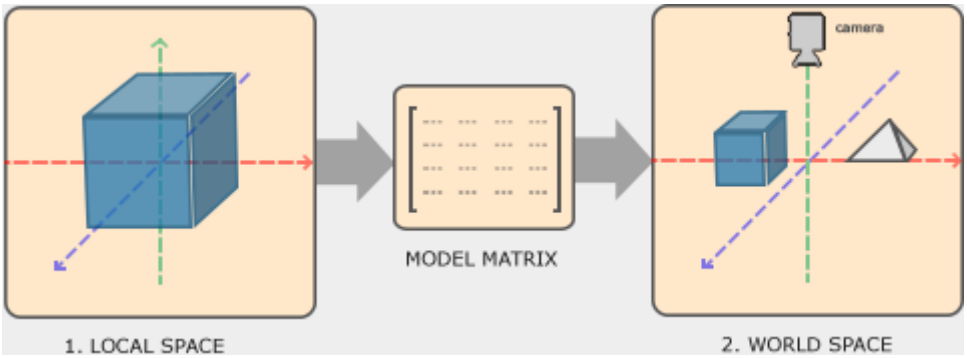

Figura 4.12: Transformada do espaço local para o espaço no mundo por meio da matriz de modelo (Fonte: [51]).

Quando utilizado num mundo virtual, o objeto 3D tem a sua posição definida neste mundo por meio da multiplicação de suas coordenadas locais pela matriz de modelo, explicitada na Figura 4.12, que define a translação, rotação e escala do objeto no mundo e pode ser calculada conforme a Equação 4.1.

$$M = (SR)T \tag{4.1}$$

onde M é a matriz de modelo, S é a matriz de escala, R é a matriz de rotação e T é a matriz de translação. Vale ressaltar que como a multiplicação de matrizes não é uma operação comutativa, ou seja, a ordem dos fatores altera o resultando final. Portanto, para obter-se uma matriz modelo válida é necessário preservar a ordem da multiplicação definida na equação.

4.8.2 Matriz de viewing

A matriz de *viewing* transforma as coordenadas no espaço do mundo do jogo, que tem como referência o centro deste mundo, para o espaço de câmera, que tem como referência a câmera observando o mundo 3D, conforme ilustrado pela Figura 4.13

Movimentar a câmera 1 metro para trás é equivalente a movimentar o mundo virtual inteiro 1 metro para frente e é exatamente esta transformada que a matriz de *viewing*

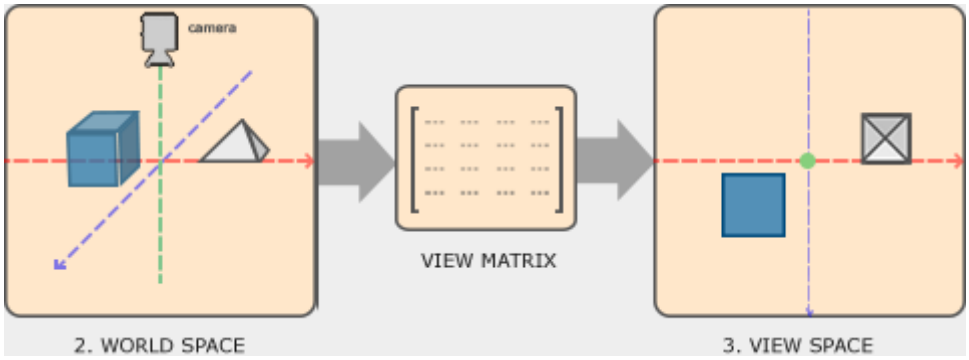

Figura 4.13: Transformada do espaço no mundo para o espaço em câmera por meio da matriz de *viewing* (Fonte: [51]).

realiza nos objetos do mundo: Ela movimenta o ambiente virtual inteiro de forma inversa ao movimento desejado da câmera.

4.8.3 Matriz de projeção

A *matriz de projeção* transforma as coordenadas no espaço de câmera para coordenadas no plano de visão, que define a posição final dos objetos que serão renderizados conforme ilustrado pela Figura 4.14. Objetos que se encontram fora do plano de visão são descartados, podendo este plano ser definido por uma **projeção ortográfica** ou uma **projeção perspectiva**. Ambas projeções são delimitadas por um volume de um tronco de bases paralelas.

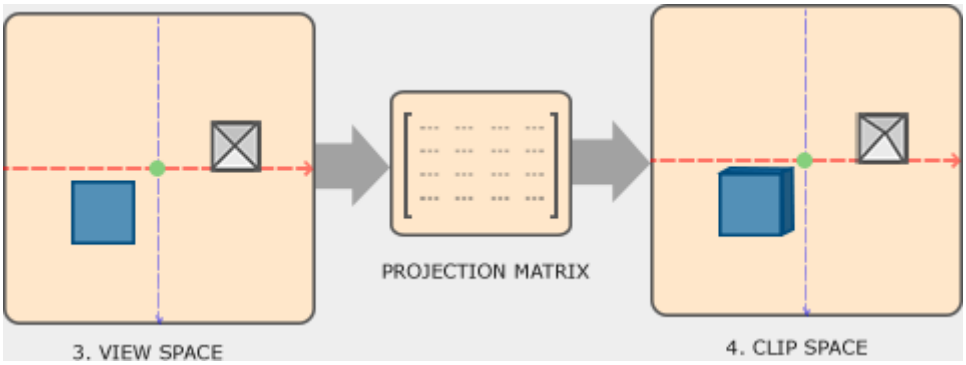

Figura 4.14: Transformada do espaço em câmera para o plano de visão (Fonte: [51]).

Projeção ortográfica

Numa projeção ortográfica, o plano de visão é definido por um tronco de bases paralelas no formato de um cubo, onde objetos dentro deste cubo são renderizados e objetos fora deste

cubo são descartados, conforme ilustrado pela Figura 4.15. Vale ressaltar que a projeção não é limitada somente pelas arestas laterais mas também pelas bases paralelas, sendo que qualquer objeto na frente da **base dianteira** ou atrás da **base traseira** também serão descartados do plano.

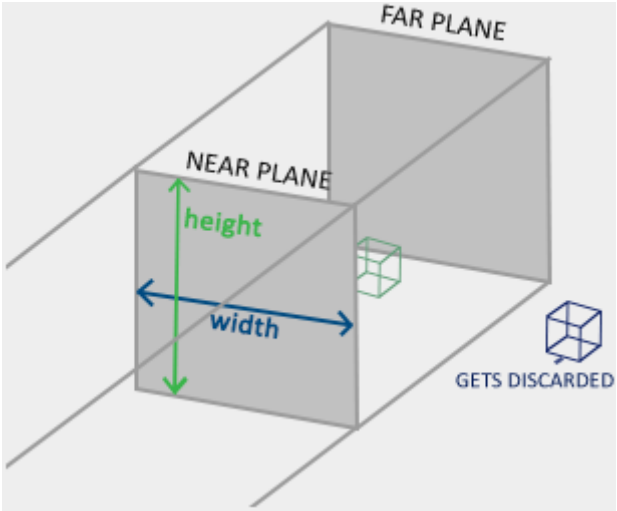

Figura 4.15: Plano de visão de uma projeção ortográfica (Fonte: [51]).

A projeção ortográfica é comumente utilizada em aplicações 2D pois mapeia diretamente as coordenadas projetadas na tela, que também é um plano 2D, sem nenhuma transformação. Porém para aplicações 3D, que é o caso do processo alvo utilizado neste trabalho, o resultado desta projeção acaba não sendo realista pois descarta efeitos de perspectiva.

Projeção perspectiva

Na projeção perspectiva, o plano de visão é definido por um tronco piramidal de bases paralelas onde objetos dentro do plano são renderizados e objetos fora do plano são descartados, conforme a Figura 4.16. Esta projeção é comumente utilizada em aplicações 3D pois nela são aplicados efeitos de perspectiva para aumentar a fidelidade da cena renderizada.

O efeito de perspectiva aplicado por essa projeção consiste em ajustar o tamanho de objetos em função da sua distância com a câmera, reduzindo o tamanho de suas arestas conforme a distância aumenta por meio de um processo chamado de **divisão de perspectiva**, demonstrado na Equação 4.2, resultando em coordenadas normalizada entre [-1,1] que posteriormente serão ajustadas para o tamanho da tela:

$$\overrightarrow{N_S} = \frac{C_x}{w}\hat{i} + \frac{C_y}{w}\hat{j} + \frac{C_z}{w}\hat{k} \tag{4.2}$$

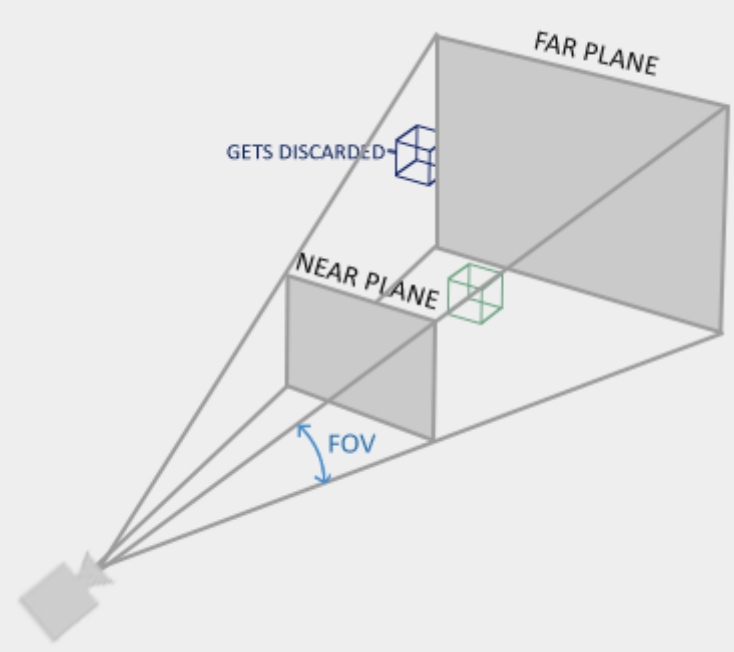

Figura 4.16: Plano de visão de uma projeção perspectiva (Fonte: [51]).

Na Equação 4.2 acima, $\overrightarrow{N_S}$ são as coordenadas de tela normalizadas, $\overrightarrow{C}$ são as coordenadas no plano de visão e w o divisor de perspectiva obtido na multiplicação das coordenadas de *viewing* pela matriz de projeção. O efeito desta divisão de perspectiva fica evidente na Figura 4.17, que compara uma projeção perspectiva com uma projeção ortográfica utilizado as mesmas coordenadas de câmera.

4.8.4 Transformada para coordenadas de tela

Nesta etapa as coordenadas obtdidas na etapa anterior, normalizadas em [-1,1], são portadas para coordenadas de tela, que variam de [0,W] no eixo X, onde W é a largura da tela, e [0,H] no eixo Y, onde H é a altura da tela, conforme ilustrado pela Figura 4.18.

4.8.5 Implementação da função

Para a implementação da função **WorldToScreen()**, que converte coordenadas no mundo 3D para coordenadas na tela, foi feito o uso da matriz *ViewProjection* calculada pela biblioteca OpenGL por meio da multiplicação da matriz de *viewing* pela matriz de projeção. Esta matriz, cujo o endereço foi encontrado por engenharia reversa no alvo, condensa as matrizes de *viewing* e projeção numa única matriz, de forma análoga à matriz modelo citada anteriormente, que condensa as transformadas de translação, rotação e escala.

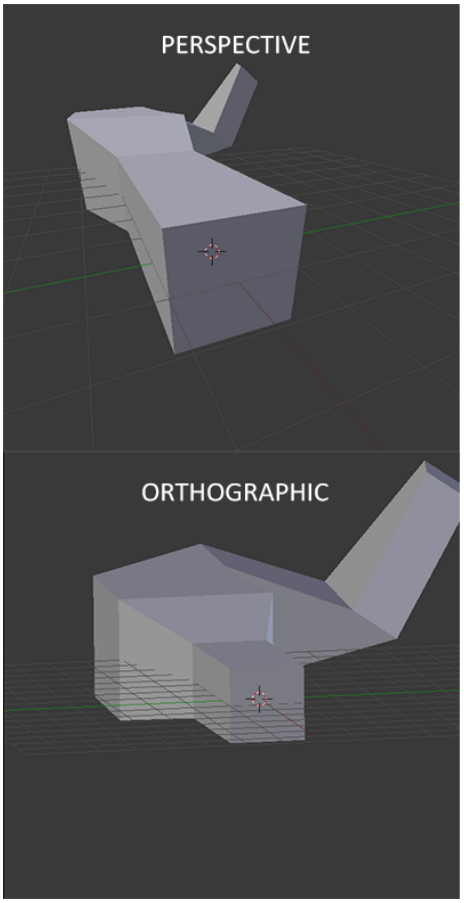

Figura 4.17: Comparação entre uma projeção ortográfica e uma projeção perspectiva (Fonte: [51]).

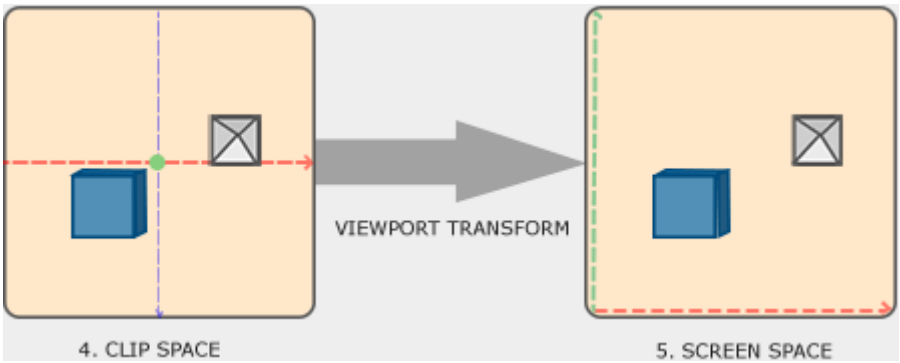

Figura 4.18: Transformada de coordenadas no plano de visão para coordenadas de tela (Fonte: [51]).

Ao multiplicar o vetor posição do objeto no mundo pela matriz *ViewProjection*, realizar a **divisão de perspectiva** e transformar as coordenadas normalizadas para coordenadas em tela, a função obtém a posição do objeto na tela, sendo esta expressa em píxels.

4.9 ESP

Extra**S**ensory **P**erception é uma funcionalidade que confere ao usuário habilidades de ver seus inimigos e amigos através da parede, o que configura uma vantagem expressiva em jogos de tiro competitivos como o Assault Cube. A Figura 4.19 ilustra a tela do jogo com a funcionalidade ativada numa sessão multijogador.

Figura 4.19: Tela do jogo com a funcionalidade de ESP ativada.

A primeira etapa da implementação é filtrar o vetor de jogadores alocado pelo jogo, cujo o endereço foi encontrado via de engenharia reversa. Este vetor armazena uma série de ponteiros para objetos de jogadores contendo informações sobre os mesmos, como vida, posição da cabeça, posição dos pés, arma utilizada e outros. Nesta filtragem inicial, a DLL seleciona somente jogadores que estão vivos, ou seja, cuja a vida seja maior do que 100, para a renderização da caixa.

O programa então itera o vetor de jogadores renderizáveis criado na etapa anterior e inicializa o processo renderização da caixa por meio da obtenção das coordenadas em tela da cabeça e dos pés do jogador sendo iterado. Para isso, a funcionalidade acessa

a posição no mundo destes membros contidos na instância do jogador e utiliza a função **WorldToScreen()** para obter as suas respectivas coordenadas em tela. Tais coordenadas são utilizadas para calcular as dimensões da caixa conforme as Equações 4.3 e 4.4, onde h é a altura da caixa, w é a largura da caixa, $\overrightarrow{H_S}$ é a posição na tela da cabeça e $\overrightarrow{F_S}$ a posição na tela dos pés.

$$h = F_{Sy} - H_{Sy} \tag{4.3}$$

$$w = \frac{h}{2} \tag{4.4}$$

Repare que no cálculo da altura a posição no eixo Y, dada em pixels, dos pés é maior do que a da cabeça. Isso acontece pois a origem do sistema de coordenadas na projeção criada pelo hack para renderizar as caixas de ESP é no campo superior esquerdo da tela, ou seja, quanto mais embaixo da tela um pixel está, maior o seu valor no eixo Y. A motivação da escolha de tal sistema foi simplesmente por questões de afinidade do desenvolvedor.

Com as dimensões da caixa obtida, o programa calcula as coordenadas do canto superior esquerdo $\overrightarrow{B_{tl}}$ e do canto inferior direito $\overrightarrow{B_{br}}$ da caixa, conforme as Equações 4.5 e 4.6, para posteriormente renderizá-la.

$$\overrightarrow{B_{tl}} = (H_{Sx} - \frac{w}{2})\hat{i} + H_{Sy}\hat{j} \tag{4.5}$$

$$\overrightarrow{B_{br}} = (H_{Sx} + \frac{w}{2})\hat{i} + (H_{Sy} + h)\hat{j} \tag{4.6}$$

Uma vez definidos os cantos $\overrightarrow{B_{tl}}$ e $\overrightarrow{B_{br}}$ da caixa, o programa também define a cor na qual a caixa será renderizada, escolhendo a cor azul para jogadores aliados e vermelho para jogadores inimigos. Os parâmetros de posição, cor e largura de linha são então passados para uma função auxiliar implementada pelo hack que realiza a renderização da caixa por meio de APIs de geometria simples da biblioteca de renderização OpenGL, utilizada pelo jogo.

4.10 Aimbot

O Aimbot é uma funcionalidade que permite o usuário mirar automaticamente na cabeça de inimigos tanto em sessões locais quanto em sessões multijogador.

4.10.1 Política de seleção de alvo

Para a implementação da política de seleção, foi necessário encontrar o endereço base do vetor de jogadores e iterá-lo com o intuito de selecionar o inimigo vivo mais próximo da mira do jogador. Por meio da transformada **WorldToScreen()** na posição da cabeça do alvo foram obtidas as coordenadas equivalentes em tela, que foram comparadas com as coordenadas da mira (que está ao centro da tela) para selecionar o inimigo cuja a distância entre os pontos fosse a menor.

4.10.2 Lógica de rastreamento

Para forçar a mira do jogador na cabeça do alvo, o hack calcula o ângulo ideal para o alvo e o utiliza para sobrescrever o valor em memória do ângulo apontado pelo jogador.

A primeira etapa no cálculo do ângulo ideal é a obtenção da posição relativa do alvo em função do olho do jogador ($\overrightarrow{T}$), que pode ser obtida a partir da subtração da posição no mundo da cabeça do alvo ($\overrightarrow{H}$) pela posição no mundo do olho do jogador ($\overrightarrow{P}$), conforme explicitado pela Equação 4.7

$$\overrightarrow{T} = \overrightarrow{H} - \overrightarrow{P} \tag{4.7}$$

Uma vez obtido o vetor $\overrightarrow{T}$ que aponta para a cabeça do alvo a partir do olho do jogador, é preciso extrair a sua angulação para finalmente forçar os ângulos da mira para este valor. Para isso, o vetor $\overrightarrow{T}$, dado em coordenadas cartesianas, foi transformado para o seu equivalente em coordenadas esféricas, conforme as Equações 4.8, 4.9 e 4.10

$$r = \sqrt{{T_x}^2 + {T_y}^2 + {T_z}^2} \tag{4.8}$$

$$\varphi = \arctan \frac{T_y}{T_x} \tag{4.9}$$

$$\theta = \arctan \frac{T_z}{\sqrt{{T_x}^2 + {T_y}^2 + {T_z}^2}} = \arctan \frac{T_z}{r} \tag{4.10}$$

onde r é o raio, φ é o ângulo azimute e θ é o angulo polar. Para melhor compreensão do que cada ângulo representa no sistema de coordenadas esférico referencie a Figura 4.20, que demonstra a convenção da física, usada pelo hack, para este sistema de coordenadas.

Após a conversão do vetor para coordenadas esféricas, as componentes são convertidas de radianos para graus e ajustadas para o sistema de coordenadas do jogo, que possui pontos de referência diferentes do padrão ISO. Os ajustes de referência realizados podem ser observados abaixo:

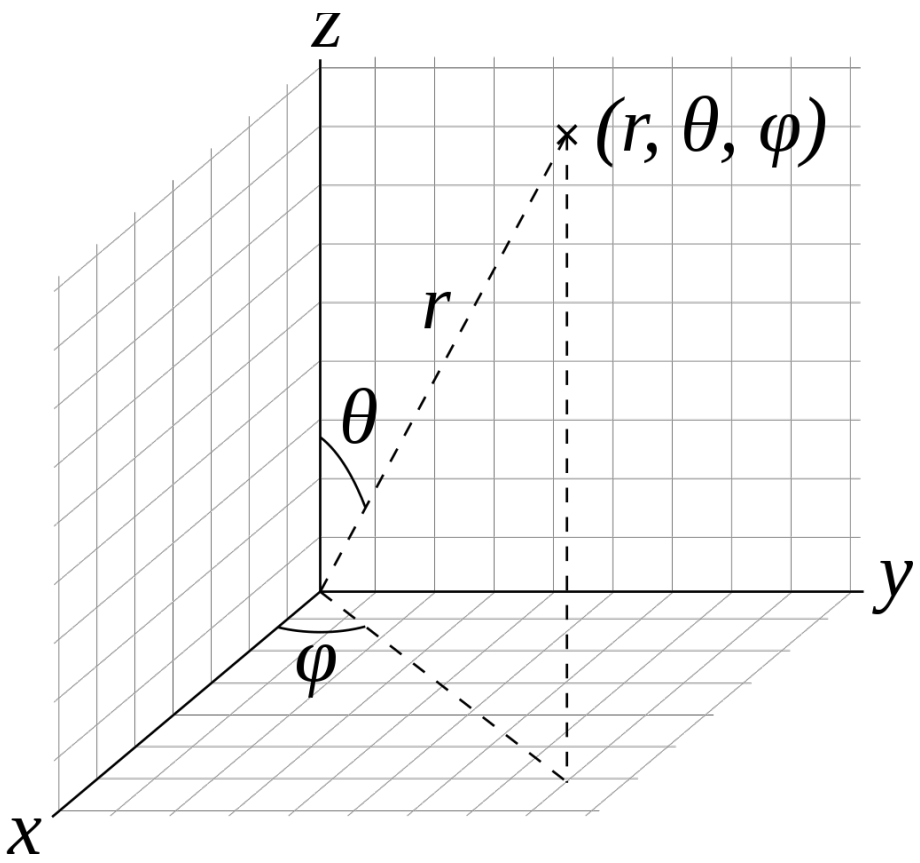

Figura 4.20: Sistema de coordenadas esféricas (Fonte: [53]).

```
resultingAngle->Z = angle.Z + 90;
resultingAngle->Y = angle.Y - 90;

if (target->HeadPosition.Z > LocalPlayer->HeadPosition.Z)
        resultingAngle->Y = fabs(resultingAngle->Y);
```

```
uintptr_t FindDynamicMemoryAddress
(uintptr_t ptr, std::vector<unsigned> offsets)
{
        uintptr_t address = ptr;
        for (unsigned int offset : offsets)
        {
                address = *(uintptr_t*)address;
                address += offset;
        }
        return address;
}
```

Listing 4.1: Função que encontra endereços alocados dinâmicamente a partir de uma cadeia de ponteiros

Capítulo 5

Conclusão e trabalhos futuros

5.1 Conclusão

Esta monografia demonstrou na prática como uma injeção de DLL pode ser conduzida em um processo evitando vetores de detecção comumente utilizado por soluções especializadas de detecção e, para demonstrar o poder da técnica, a DLL implementada modificou de forma significativa o funcionamento do jogo Assault Cube, utilizado como processo alvo neste trabalho.

Primeiro foi definida a motivação deste trabalho e o seu objetivo, estabelecendo o que seria abordado em termos conceituais e o que seria implementado em termos práticos.

Em seguida foram apresentados os principais conceitos necessários para que o conceito da injeção fosse compreendido, sendo este apresentado no final do capítulo, após a definição de **Shared Library** e **Handles**, a descrição de **Dynamic Loading** e **Inline Hooking** e a apresentação do **Formato PE**.

Após uma revisão conceitual, o injetor implementado foi exposto. Com ele, é possível configurar cada etapa da injeção conforme o usuário desejar, possibilitando o seu uso tanto para injeções legítimas, que ocorrem por intermédio de sistema operacional deixando uma trilha de rastros, quanto para injeções furtivas, que objetivam evadir mecanismos especializados de detecção.

Com a ferramenta desenvolvida foram conduzidas 2 injeções no jogo, um utilizando uma configuração completamente gerenciada pelo sistema operacional e o outro utilizando os mecanismos de evasão do injetor. No primeiro teste ficou evidente os diversos rastros deixados pelas APIs do sistema operacional encarregadas de realizar a injeção. Tais rastros facilitam a detecção e a determinação do caminho em disco da DLL injetada, que pode ser usado por uma ferramenta de segurança para enviar o arquivo a um servidor de análise.

No segundo teste foram utilizadas as técnicas de evasão discutidas nesta monografia, como o **Pyjack Injection**, **Manual Mapping** e **Thread Hijacking**, resultando, con-

forme exposto pelos dados coletados, numa injeção não suscetível aos vetores de detecção abordados. Vale lembrar que, como na configuração furtiva a DLL não precisa estar salva em disco, são abertas diversas possibilidades para o seu provisionamento em tempo de execução, o que dificulta significativamente o envio da DLL a um centro de análise para a construção de uma assinatura de detecção.

No capítulo seguinte foi abordado o desenvolvimento da DLL injetada e as suas funcionalidades, demonstrando os alto grau de conhecimento sobre o alvo requerido para que a técnica possa ser utilizada com eficácia. Caso um atacante seja determinado suficiente para compreender o funcionamento interno de seu alvo, é possível modificá-lo como quiser, conforme demonstrado neste trabalho.

5.2 Trabalhos futuros

Os seguintes trabalhos futuros são propostos:

- **Pyjack Injection em arquiteturas diferentes:** a ferramenta implementada consegue explorar somente handles de processos da mesma arquitetura que o alvo, havendo espaço para melhorias neste aspecto. Com adaptação dos elementos de baixo nível para o emulador WOW64, utilizado pelo Windows para executar aplicações 32 bits em sistemas 64 bits, é possível que tal funcionalidade seja desenvolvida;
- **Implementação de formas de provisionamento alternativas:** o injetor implementado neste trabalho necessita que a DLL utilizada esteja salva em disco, o que não precisa ser um requisito obrigatório. É possível suportar múltiplas formas de provisionamento utilizando a técnica de **Mapeamento Manual** e adaptando o injetor para permitir que bytes de qualquer fonte (sockets, arquivos, vetores de bytes *hardcoded*) possam ser interpretados como uma DLL e injetados no alvo;
- **Implementação do injetor proposto a nível de Kernel:** o injetor implementado executa à nível de usuário e é possível criar uma implementação a nível de Kernel, que disponibiliza ao desenvolvedor um leque maior de funcionalidades;
- **Implementação de novas contramedidas para vetores de detecção:** o combate às ameaças digitais é uma constante corrida armada entre softwares de segurança e ferramentas para quebrá-los, em que aqueles evoluem conforme o surgimento de novas ameaças e técnicas de exploração. O mesmo é valido para o injetor desenvolvido neste trabalho, que pode estar obsoleto após alguns anos de melhorias nas técnicas de detecção atuais. Visando a constante melhoria da segurança no ambiente virtual, novos vetores de ataque precisam ser estudados para que novas contramedidas sejam desenvolvidas.

Referências

[1] Moore, Gordon E.: *Cramming more components onto integrated circuits*. Electronics, 38(8):0–4, 1965. 1

[2] Kushner, David: *The real story of stuxnet*, Fevereiro 2013. https://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet/. 1

[3] Thabet, Amr: *Stuxnet malware analysis paper*, September 2011. https://www.codeproject.com/Articles/246545/Stuxnet-Malware-Analysis-Paper#ch4.3.1. 1

[4] Albright, David, Paul Brannan e Christina Walrond: *Did Stuxnet Take Out 1,000 Centrifuges at the Natanz Enrichment Plant?* Institute for Science and International Security, 2010. 2

[5] OppenheimerFunds: *Investing in the soaring popularity of gaming*, Junho 2018. https://www.reuters.com/sponsored/article/popularity-of-gaming. 2, 3

[6] *Program library howto - introduction.* http://tldp.org/HOWTO/Program-Library-HOWTO/introduction.html, acesso em 2019-10-30. 6

[7] Microsoft: *Creating a resource-only dll*, Março 2016. https://docs.microsoft.com/en-us/cpp/build/creating-a-resource-only-dll?view=vs-2019, acesso em 2019-10-30. 6

[8] Zoltán, Kővágó: *Playing an ogg vorbis file using the libvorbisfile library*, Junho 2014. https://commons.wikimedia.org/wiki/File:Ogg_vorbis_libs_and_application_dia.svg, acesso em 2019-06-05. 7

[9] Grover, Sandeep: *Linkers and loaders*, Novembro 2002. https://www.linuxjournal.com/article/6463, acesso em 2019-06-05. 7

[10] *Linker input-output diagram*, Junho 2009. https://commons.wikimedia.org/wiki/File:Linker.svg, acesso em 2019-06-05. 8

[11] Jones, M.: *Static vs. dynamic linking*, Agosto 2008. https://developer.ibm.com/tutorials/l-dynamic-libraries/, acesso em 2019-07-05. 9

[12] *Dll hell - a article about dll hell*, 2007. https://web.archive.org/web/20080703211550/http://dllcity.com/dll-hell.php, acesso em 2019-1-06. 9

[13] *Side-by-side assemblies*, Junho 2010. https://docs.microsoft.com/en-us/previous-versions//aa376307(v=vs.85), acesso em 2019-06-1. 9

[14] Dassen, J.H.M.: *Dynamic loading*, Agosto 1995. http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/node7.html, acesso em 2019-28-05. 10

[15] Michael Satran, Mark LeBlanc, Colin Robertson Drew Batchelor Cristopher Warrington: *Pe format*. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format, acesso em 2019-08-24. 11

[16] Benhut1: *Portable executable 32 bit structure in svg fixed*, September 2016. https://commons.wikimedia.org/wiki/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg, acesso em 2019-08-29. 12, 13

[17] Gordon Hogenson, Mike Jones, Colin Robertson Saisang Cay Mike B: */dynamicbase (use address space layout randomization)*, Novembro 2018. https://docs.microsoft.com/en-us/cpp/build/reference/dynamicbase-use-address-space-layout-randomization?view=vs-2019, acesso em 2019-08-29. 14

[18] Plosceanu, Dennis: *Function hooking and windows dll injection*, March 2016. https://ocw.cs.pub.ro/courses/so/laboratoare/resurse/injections, acesso em 2019-06-02. 14, 42

[19] Hoffman, Chris: *What is code injection on windows?*, Agosto 2018. https://www.howtogeek.com/363845/what-is-code-injection-on-windows/, acesso em 2019-06-02. 14, 42

[20] Antoniewicz, Brad: *Windows dll injection basics*, January 2013. http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html, acesso em 2019-06-01. 14, 15, 16, 17

[21] John Kennedy, Michael Satran: *Traversing the module list*, May 2018. https://docs.microsoft.com/en-us/windows/win32/toolhelp/traversing-the-module-list, acesso em 2019-08-23. 16, 26

[22] Hosseini, Ashkan: *Ten process injection techniques: A technical survey of common and trending process injection techniques*, Julho 2017. https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process, acesso em 2019-06-07. 16, 18

[23] *Worm:win32/rebhip.a*, September 2017. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Worm:Win32/Rebhip.A, acesso em 2019-06-07. 16

[24] *Remote thread execution in system process using ntcreatethreadex for vista windows7*. https://securityxploded.com/ntcreatethreadex.php, acesso em 2019-06-10. 17, 18

[25] 3gstudent: *Inject-dll-by-apc*, Junho 2018. https://github.com/3gstudent/Inject-dll-by-APC/blob/master/NtCreateThreadEx.cpp, acesso em 2019-10-30. 18

[26] ntquery: *Anti-debug ntcreatethreadex*, Março 2014. https://ntquery.wordpress.com/2014/03/29/anti-debug-ntcreatethreadex/, acesso em 2019-10-30. 18

[27] K-Max, Vinicius: *Burlando antivirus com dll side-loading por diversão e vantagem*, April 2018. https://medium.com/@viniciuskmax/burlando-antivirus-com-dll-side-loading-por-divers%C3%A3o-e-vantagem-parte-1-ffcd26058239, acesso em 2019-06-10. 18

[28] Martin, Robert C.: *The principles of ood*. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, acesso em 2019-07-31. 23

[29] MacCabe, T.J., McCabe, Associates e IEEE Computer Society: *Structured testing*. Tutorial Texts Series. IEEE Computer Society Press, 1983. https://books.google.com.br/books?id=vtNWAAAAMAAJ. 24

[30] Microsoft: *Virtualallocex function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex, acesso em 2019-09-15. 30, 40

[31] Kowalczyk, Krzysztof: *Portable executable file format*, Julho 2018. https://blog.kowalczyk.info/articles/pefileformat.html, acesso em 2019-08-30. 30

[32] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3*. Intel Corporation, 2019. https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf. 30

[33] *Visual Studio, Microsoft Portable Executable and Common Object File Format Specification*. Microsoft Corporation, Dezembro 2015. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v83.docx. 32

[34] Zeltser, Lenny: *How malware defends itself using tls callback functions*, Julho 2009. https://isc.sans.edu/diary/How+Malware+Defends+Itself+Using+TLS+Callback+Functions/6655, acesso em 2019-09-03. 34

[35] Team, OneCode: *File handle operations demo*, Fevereiro 2012. https://code.msdn.microsoft.com/windowsapps/CppFileHandle-03c8ea0b/sourcecode?fileId=52762&pathId=1613038731, acesso em 2019-09-10. 35

[36] adamkramer: *Handle monitor*, Março 2015. https://github.com/adamkramer/handle_monitor/blob/master/handle_monitor.cpp, acesso em 2019-09-10. 35

[37] Microsoft: *Ntquerysysteminformation function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation, acesso em 2019-09-10. 35, 39

[38] Chappel, Geoff: *System information structures, system_process_information*, Agosto 2019. https://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/ex/sysinfo/process.htm, acesso em 2019-09-14. 39

[39] Nowak, Tomasz: *Undocumented functions of ntdll, system_process_information*, Abril 2002. http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FSystem%20Information%2FStructures%2FSYSTEM_PROCESS_INFORMATION.html, acesso em 2019-09-14. 39

[40] Microsoft: *Openthread function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openthread, acesso em 2019-09-15. 40

[41] Microsoft: *Suspendthread function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-suspendthread, acesso em 2019-09-15. 40

[42] Microsoft: *Getthreadcontext function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getthreadcontext, acesso em 2019-09-15. 40

[43] Microsoft: *Writeprocessmemory function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory, acesso em 2019-09-15. 41

[44] Microsoft: *Resumethread function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread, acesso em 2019-09-15. 41

[45] Microsoft: *Readprocessmemory function*, Abril 2018. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-readprocessmemory, acesso em 2019-09-15. 41

[46] Microsoft: *Process monitor*, Março 2019. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon, acesso em 2019-09-18. 42

[47] *Process hacker 2*. https://processhacker.sourceforge.io/, acesso em 2019-09-18. 42

[48] *Ollydbg*, August 2019. http://www.ollydbg.de, acesso em 2019-09-22. 53, 61

[49] *Cheat engine*, August 2019. https://www.cheatengine.org, acesso em 2019-09-22. 53

[50] Stewart, Samuel: *What is the best fps for gaming?*, July 2019. https://www.gamingscan.com/best-fps-gaming/, acesso em 2019-09-21. 59

[51] *Open gl coordinate systems*. https://learnopengl.com/Getting-started/Coordinate-Systems, acesso em 2019-10-06. 64, 65, 66, 67, 68, 69

[52] *Tutorial 3 : Matrices*. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/, acesso em 2019-10-06. 64

[53] Geek3: *3d spherical.svg*, August 2019. https://commons.wikimedia.org/wiki/File:3D_Spherical.svg, acesso em 2019-10-08. 73