PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Anti-cheat em Jogos Online

Matheus Basilio Gagliano

PROJETO FINAL DE GRADUAÇÃO

Orientação Prof. Bruno Feijó

CENTRO TÉCNICO CIENTÍFICO – CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Ciência da Computação

Rio de Janeiro, Novembro de 2021

Matheus Basilio Gagliano

Anti-cheat em Jogos Online

Relatório de Projeto Final, apresentado ao programa de Ciência da Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Ciência da Computação.

Orientador: Prof. Bruno Feijó

Rio de Janeiro
26 de Novembro de 2021.

“Perante um obstáculo, a linha mais curta entre dois pontos pode ser a curva.”

Bertolt Brecht

Resumo

Basilio, Matheus. Feijó, Bruno. Anti-cheat em Jogos Online. Rio de Janeiro, 2021. Projeto Final - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Este projeto apresenta, de forma abrangente, a situação atual de anti-cheats em jogos online, começando com a fundamentação de suas ameaças (trapaças) e culminando com o detalhamento de técnicas e métodos que podem ser empregados como contramedidas. Também há a discussão acerca de medidas genéricas e de alguns conselhos para proteger software.

Palavras-chave

Segurança, Segurança da Informação, Trapaça, Antitrapaça, Criptografia, Windows, Comunicação, Jogo, Jogo online, Proteção, Proteção de software, Computação distribuída, Sistema operacional, Sistemas operacionais

Abstract

Basilio, Matheus. Feijó, Bruno. Anti-cheat in Online Games. Rio de Janeiro, 2021. Final Project - Department of Informatics. Pontifical Catholic University of Rio de Janeiro.

This project presents, in a comprehensive way, the current situation of anti-cheats in online games, beginning with the fundamentals of their threats (cheats) and culminating with the detailing of techniques and methods that can be employed as countermeasures. There is also a discussion of generic measures and some advices to protect software.

Keywords

Security, Information Security, Cheat, Anti-cheat, Criptography, Windows, Communication, Game, Online game, Protection, Software protection, Distributed computing, Operating system, Operating systems

Sumário

1. Introdução

Jogos hoje em dia são multifacetados e envolvem áreas multidisciplinares para sua elaboração. Em jogos online/*multiplayer*, em que múltiplos jogadores interagem e/ou competem em tempo real, o desafio se torna ainda maior; agora não basta apenas construí-lo e publicá-lo, mas também necessita-se gerenciá-lo constantemente e arcar com custos de servidores a fim de hospedá-lo. Soma-se a isso a complexidade adicional de lidar com computação distribuída. Todos esses fatores devem ser pensados e considerados pelos desenvolvedores e pela empresa que decide iniciar o projeto.

Certamente, princípios e práticas de Segurança da Informação também devem ser apreciados quando o tema é desenvolvimento de jogos online. Confidencialidade, integridade, disponibilidade, autenticidade e controle de acesso — pilares de Segurança da Informação — influem diretamente sobre o modo pelo qual mecanismos eficazes de segurança são projetados. Sumarizando [1], confidencialidade ou sigilo garante que “informações privadas ou confidenciais não fiquem disponíveis nem sejam reveladas a indivíduos não autorizados” (confidencialidade de dados) e que “indivíduos controlem ou influenciem quais informações sobre eles podem ser coletadas e armazenadas, e por quem e para quem tais informações podem ser reveladas” (privacidade); integridade garante que “informações e programas sejam alterados somente de maneira especificada e autorizada” (integridade de dados) e que “um sistema desempenhe sua função livre de manipulações não autorizadas” (integridade de sistemas); disponibilidade garante que “sistemas funcionem prontamente e que não haja negação de serviço a usuários autorizados”; autenticidade é “a propriedade de ser genuína e poder ser verificada e confiável; confiança na validade de uma transmissão, de uma mensagem ou do originador de uma mensagem. Isso significa verificar que os usuários são quem dizem ser e que cada dado que chega ao sistema veio de uma fonte confiável”; e controle de acesso tem o objetivo de “limitar acesso a sistemas de informação a usuários autorizados, processos que agem em nome de usuários autorizados ou dispositivos (incluindo outros sistemas de informação) e aos tipos de transações e funções que usuários autorizados têm permissão de exercer”. Sem boas

medidas de segurança, o jogo (ou qualquer sistema) tende ao fracasso, já que variados tipos de ataque e trapaça impactarão a satisfação dos usuários.

Anti-cheat é um sistema que deve fornecer proteção preventiva e reativa a trapaças e, portanto, é um dos sistemas de segurança que qualquer jogo online precisa possuir. Para combater a crescente situação de trapaças em jogos online, vários anti-cheats foram desenvolvidos — tanto *in-house* quanto comerciais. O primeiro grande anti-cheat comercial, conforme um estudo [2], foi o *Punkbuster*, desenvolvido pela empresa *Even Balance*. Outros vieram após, a exemplo do *Valve Anti-Cheat* e do *Warden* [2]. Hoje em dia, há dois comerciais bastante conhecidos: o *BattlEye* e o *Easy Anti-Cheat*.

Existem outras táticas empregadas, como sistemas de voto que notificam a equipe do jogo sobre jogadores que podem estar utilizando trapaças, funcionando de modo que “um primeiro jogador em uma sessão de jogo online detecta um comportamento suspeito de trapaça por um segundo jogador numa sessão de jogo online, e o primeiro jogador comunica essa possível suspeita a uma entidade de monitoramento de trapaça do jogo” [3]. O jogo *Counter-Strike: Global Offensive* faz uso desse tipo de sistema, em que um jogador denunciado é votado por outros jogadores experientes, indicando à equipe se há forte suspeita de trapaça ou não.

Se um jogador for julgado como trapaceador, uma punição básica é ser banido, ficando sem acesso à conta do jogo durante certo intervalo de tempo ou até mesmo permanentemente. Outra estratégia para punir, utilizada em jogos que enfileiram jogadores para formar partidas, é colocar trapaceadores em uma mesma partida.

Atualmente um anti-cheat pode rodar na camada zero (kernel) ou na camada três (usuário), ou ter dois componentes: um executando na camada zero e o outro na camada três. Neste último caso, o componente envia comandos ao componente na camada zero, a fim de executar verificações que necessitam do modo kernel. Tanto o *BattlEye* quanto o *Easy Anti-Cheat* possuem componentes na camada kernel [4][5].

Sistemas executando em modo kernel têm mais privilégios no sistema operacional, ficando acima hierarquicamente da maioria dos programas presentes num computador comum (figura 1). Isso dá vantagens significativas ao anti-cheat, na medida em que ele poderá controlar o sistema operacional em sua totalidade, monitorando e controlando outros programas. Uma desvantagem, entretanto, é que componentes não

implementados corretamente que residem nesse modo podem abrir um buraco de vulnerabilidades passível de ser explorado por sistemas maliciosos (não apenas trapaças). Por exemplo, programas maliciosos podem utilizar esse componente, se houver vulnerabilidade, para injetar códigos maliciosos, que executarão em modo kernel. Outra desvantagem é desestabilizar completamente o sistema operacional — caso seja monolítico como o Windows — ao apresentar falhas, bem como diminuir o desempenho do sistema operacional, inteiramente, quando há módulos ineficientes ou que consomem muito poder de processamento.

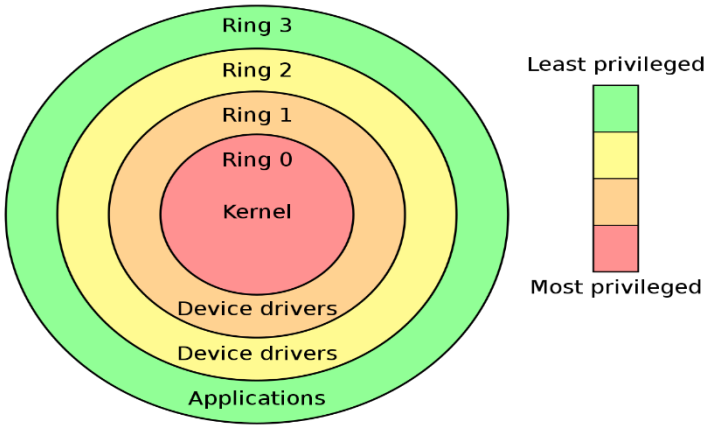

Figura 1 Anéis de privilégio comuns em sistemas operacionais.

Por outro lado, anti-cheats na camada três possuem menos privilégios, mas não têm as desvantagens descritas no parágrafo anterior. Apesar de fazerem menos do que anti-cheats em modo kernel, é possível utilizar várias técnicas, em modo usuário mesmo, para dificultar bastante a criação e o uso de trapaças.

Um dos objetivos deste trabalho é fazer um estudo aprofundado sobre tipos de trapaça, métodos e técnicas frequentes utilizados por trapaças e anti-cheats, e problemas e tendências de sistemas anti-cheat em jogos online. Outro objetivo é desenvolver um anti-cheat simples, enfatizando alguns dos problemas levantados e implementando alguns métodos e técnicas discutidos.

Há uma pesquisa abrangente em cima de métodos e técnicas usados por anti-cheats, com foco naqueles que residem na camada três.

Além disso, também há a pesquisa de como trapaças fazem o que fazem, buscando-se, assim, aprender a se proteger contra os mecanismos

empregados por elas. Como para anti-cheats, a direção aqui também será à camada três.

Métodos e técnicas genéricos de proteção de software que versem, por exemplo, sobre o uso de máquinas virtuais para ofuscar seções de código e sobre mecanismos para detectar a execução de software em ambientes virtualizados e para dificultar análises estáticas e dinâmicas também serão estudados. Esses métodos e técnicas que serão apresentados costumam ser bastante utilizados por programas maliciosos (*malwares*), logo o trabalho, indiretamente, estará fazendo vasta pesquisa em criação de *malwares* em geral — incluindo ferramentas usadas para dissecá-los.

Como mencionou-se anteriormente, não há a pretensão de estudar métodos e técnicas utilizados em modo kernel, porque esse estudo, por si só, é passível de ser feito por outro trabalho separado, dada sua complexidade. Este Projeto Final tem características fortes de pesquisa e, por causa disso, não está sendo apresentada uma modelagem detalhada e uma lista de requisitos (funcionais e não funcionais). A pesquisa tomou a maior parte do tempo do Projeto Final.

A implementação de um anti-cheat simples visa a demonstrar como toda a pesquisa feita funciona na prática. Um requisito não funcional desse anti-cheat é ser eficiente em termos computacionais, uma vez que, mesmo executando na camada três, é possível perder desempenho na jogabilidade caso as verificações sejam muito pesadas computacionalmente.

A motivação para o trabalho é entender melhor como trapaças e anti-cheats funcionam, estudando, paralelamente, mais a fundo a área de Segurança da Informação e compreendendo as idiossincrasias do sistema operacional Windows com mais clareza.

Este documento começa apresentando os objetivos do trabalho, resumindo o que foi escrito acima, e discorrendo sobre trapaças. Depois parte para desenvolvimento de trapaças e proteção de software. Em seguida, a partir de todo o conhecimento adquirido ao longo das seções, discute sobre proteção contra trapaças, apresenta a modelagem e uma implementação de um anti-cheat simples — cobrindo e ilustrando métodos e técnicas expostas —, e finaliza com conclusões.

1. Referências

[1] William Stallings e Lawrie Brown. 2014. Segurança de computadores: princípios e práticas (segunda edição). Elsevier, Rio de Janeiro, RJ.

[2] Samuli Lehtonen. 2020. *Comparative Study of Anti-cheat Methods in Video Games*. Tese de mestrado. Universidade de Helsínquia, Helsínquia, Finlândia.

[3] Gary Zalewski. 2013. *Moderation of cheating in on-line gaming sessions*. N.º Patente US8490199B2, Requisitada 29 de Outubro de 2007, Publicada 16 de Julho de 2013.

[4] *BattlEye — The Anti-Cheat Gold Standard*. 2021. Acessado em 24 de Junho, 2021 de https://www.battleye.com/about/.

[5] *Easy Anti-Cheat*. 2021. Acessado em 24 de Junho, 2021 de https://www.easy.ac/pt-br/#about.

2. Objetivos

O trabalho objetiva:

- Realizar estudos aprofundados em conhecimentos sistematizados pela Segurança da Informação;
- Realizar estudos aprofundados em trapaças, compreendendo:
    - Suas categorizações (tipos);
    - Métodos, técnicas e ferramentas frequentemente empregados para construí-las, restringindo o escopo àquelas que atuam na camada três (*user mode*) do sistema operacional Windows.
- Realizar estudos aprofundados em anti-cheats, compreendendo:
    - Mecanismos gerais de proteção de software;
    - Situação atual;
    - Métodos, técnicas e ferramentas comumente utilizados para construí-los, restringindo o foco àqueles que atuam na camada três (*user mode*) do sistema operacional Windows.
- Implementar um protótipo, que é um anti-cheat simples, na linguagem C++, contendo alguns métodos e técnicas expostos na parte teórica;
- Criar testes para o protótipo.

.

3. Trapaça

Uma trapaça (cheat) pode ser definida como qualquer vantagem injusta que um jogador faça uso para se beneficiar em um jogo, tornando sua jogabilidade mais fácil ou obtendo vantagens não acessíveis a outros jogadores. Ela não se limita apenas à utilização de sistemas paralelos que visem prover certa vantagem, mas também se mostra presente quando se tira proveito de falhas de jogo — que podem ser erros de programação (bugs) ou falhas de projeto de jogabilidade (falhas de design). Estas, por exemplo, acontecem quando um personagem sobe em cima de outro a fim de ter visão acima de determinado muro ou objeto que a obstrui (figura 1). Aqueles ocorrem quando se executa um procedimento para duplicar itens de inventário, por exemplo.

Figura 1 Jogadores apoiando-se uns nos outros para obter visão privilegiada, no jogo Counter-Strike.

É comum que fabricantes de jogos incluam códigos de jogo para facilitar o desenvolvimento e os testes de funcionalidades (como maneira de depuração). Esses códigos geralmente se encontram no formato de comandos em série, como uma sequência de botões de teclado e/ou de mouse a serem pressionados. Também podem vir como comandos de texto, em que a inserção ocorre em caixa de texto específica. No jogo *Sonic the Hedgehog* (1991), é possível habilitar o modo de depuração ao executar a sequência de comandos "*Up, C, Down, C, Left, C, Right, C, A + Start*"; em *The Sims* (2000), exibe-se uma caixa de texto ao pressionar "*Control + Shift + C*", na qual podem inserir-se comandos em texto, como "rosebud", para

ativar trapaças [1]. Em alguns casos, como em *Nier Automata* (2017), fãs descobrem códigos sigilosos fazendo engenharia reversa no binário do jogo [2].

Embora trapacear num jogo em que o jogador joga sozinho (jogos *single-player*) possa ser considerado uma postura válida, essa mesma conduta é estritamente proibida em jogos online.

1. Referências

[1] Barbara PM. 2016. *5 Famous Cheat Codes in Video Game History*. Acessado em 17 de Março, 2021 de https://blackshellmedia.com/2016/08/11/5-famous-cheat-codes-video-game-history/.

[2] Wesley Yin-Poole. 2021. *Modder reverse-engineers Nier Automata to discover its "final secret": a cheat code*. Acessado em 17 de Março, 2021 de https://www.eurogamer.net/articles/2021-01-04-modder-finds-nier-automatas-final-secret-a-cheat-code.

4. Tipos de Trapaça

Como visto, jogadores usam trapaças para obter benefícios em sua jogatina, visando derrotar oponentes mais facilmente ou, de forma geral, deixar a mecânica do jogo mais fácil ou facilitar progressões de jogo. Essa situação ocorre quando as mecânicas do jogo são muito repetitivas, deixando a jogabilidade cansativa, ou quando o jogador se sente em desvantagem em relação a outro jogador — seja devido a equipamentos que tornam o oponente mais forte, seja devido a habilidades inferiores em relação às do oponente. Hoje em dia, trapacear é um problema sério em jogos online, já que pode prejudicar a diversão de outros jogadores, tornar a competitividade desleal e trazer prejuízos às receitas da empresa proprietária do jogo (jogadores param de jogar quando percebem que trapaças são recorrentes).

Jianxin Jeff Yan e Hyun-Jin Choi [1] classificam tipos de trapaça como: trapaça por conluio; trapaça ao abusar de procedimentos ou políticas; trapaça relacionada a ativos virtuais; trapaça por comprometimento de senhas; trapaça ao negar serviço a outros jogadores; trapaça devido à falta de sigilo; trapaça devido à falta de autenticação; trapaça relacionada a usos internos indevidos; trapaça por engenharia social; trapaça ao modificar o binário do jogo ou os seus dados; e trapaça ao explorar bugs ou falhas de design. Todas essas categorias serão desdobradas adiante.

Trapaça por conluio acontece quando dois ou mais jogadores se unem para realizar alguma ação, sendo que essa união causa algum tipo de desbalanceamento. Não necessariamente precisam ser dois jogadores distintos; o mesmo jogador pode, por exemplo, abrir dois clientes do jogo e realizar ações com eles, agindo como dois jogadores. É o que pode acontecer num jogo de Poker, por exemplo. Quando há dois clientes participando da partida, a chance de se obter uma mão boa aumenta, gerando uma vantagem injusta.

Há também jogadores que abusam de procedimentos ou políticas do jogo para trapacear. É comum que muitos jogos de tiro online forneçam uma opção ao jogador para se retirar de certa partida. Caso essa opção não tenha sido projetada adequadamente pelos desenvolvedores, uma pessoa mal intencionada pode simplesmente sair do jogo na iminência de um abate por outro jogador; ou, numa partida ranqueada, sair dela porque seu time

está perdendo. Como esses abusos são bem conhecidos pelos desenvolvedores, é difícil encontrar um jogo que não tenha imposto restrições a essa funcionalidade. É possível alocar uma janela de tempo antes da saída: o jogador deverá esperar dez segundos antes de se retirar, por exemplo. Penalidades de perda de pontos, em partidas ranqueadas, também são bastante adotadas por jogos. Nesse esquema, o participante que sair de uma partida ranqueada perderá pontos (normalmente chamados pontos de liga, indicando que são pontos de ranqueadas); ainda, caso haja reincidência, poderá ficar impossibilitado de entrar em outras ranqueadas durante determinado período de tempo.

Em trapaças relacionadas a ativos virtuais — aqueles que estão no âmbito da economia do jogo, ou que proveem algum tipo de vantagem ao jogador —, há o que se chama de “comércio virtual”. Jogadores compram ativos de outros jogadores usando dinheiro real, e não do jogo. Isso porque alguns jogos, principalmente os de simulação (RPGs — *Role Playing Games*), dispõem de funcionalidade que permite a compra e a venda de ativos com a própria moeda do jogo. Essa funcionalidade, claro, não consta como trapaça. Este tipo de trapaça pode ser mais abrangente, quando um jogador compra ativos do jogo de uma empresa. Nesse tipo de relação, a empresa comumente é criada com o intuito de vender ativos do jogo, contratando funcionários que jogam exclusivamente para coletar ativos (ação chamada vulgarmente de *farming*) ou, ainda, utilizando outros tipos de trapaça, como *bots*, para coletar os ativos automaticamente.

Senhas roubadas, fracas e reutilizadas são a principal causa de violações de dados e uma maneira comprovada de obter acesso a recursos protegidos sob autenticação. Na categoria de trapaça por comprometimento de senha, atacantes utilizam métodos tradicionais para obter senhas de outros jogadores, como ataques de força bruta, *keylogger*, dicionário e *phishing*. A melhor forma de prevenir este tipo de trapaça é usar autenticação de dois fatores, quando possível, ou implementar mecanismos que forcem o uso de boas senhas pelos usuários.

Ataques de negação de serviço geram tráfego enorme de dados e podem facilmente exaurir os recursos de computação e comunicação de uma vítima em um curto período de tempo. Jogadores maliciosos que usam ferramentas para realizar esse tipo de ataque estão, também, trapaceando. Quando outro jogador, provavelmente o adversário do atacante, é desconectado de uma partida devido a ataques de negação de serviço, a partida fica mais

fácil de ser ganha pelo time do atacante. Jogos que utilizam a arquitetura de rede *peer-to-peer* sofrem dessa vulnerabilidade, embora existam soluções.

Em trapaças que são ocasionadas por falta de sigilo, pacotes trocados entre jogadores ou entre jogadores e servidor são monitorados, criados, alterados ou removidos intencionalmente por um jogador malicioso. Jogos vulneráveis a esse tipo de ataque não possuem comunicação segura de pacotes, o que pode ser facilmente resolvido com encriptação e mecanismos de controle de integridade e autenticidade.

Trapaças causadas por falta de autenticação são relacionadas às de comprometimento de senha. Novamente, o sistema não possui mecanismos adequados que garantem uma boa autenticação de jogadores. Particularmente, existem também os casos em que uma pessoa que está no mesmo ambiente que outra altera a senha desta sem ela saber. Isso é comum em *LAN houses*, onde jogadores estão próximos uns dos outros.

Muitos jogos online de grande porte possuem uma equipe para monitorar jogadores e prestar suporte a eles. Os membros dessa equipe normalmente são chamados de *gamemasters*. Quando o sistema não tem mecanismos de controle de acesso ou autorização para executar certas ações, esses membros podem utilizar seus privilégios indevidamente. Como mencionado, o jogo deve ter mecanismos de autorização — atributo fundamental de Segurança da Informação — para restringir atos passíveis de serem feitos por um *gamemaster*. Ou, ainda, ter algum mecanismo de consenso em ações críticas que possam ser feitas por ele. Quando há utilização indevida de privilégios a fim de beneficiar-se ou beneficiar outro, está-se diante de trapaça relacionada a usos internos indevidos.

Engenharia social, processo de enganar pessoas para que forneçam acesso ou informações confidenciais, é uma ameaça relevante em sistemas de informação que possuem sistemas de autenticação. Embora haja muitas informações sobre engenharia social e seus perigos, a ameaça é considerada muito real e não pode ser facilmente defendida. Em trapaça por engenharia social, jogadores maliciosos usam diversos métodos e canais para ludibriar outros jogadores, manipulando-os e influenciando-os. Medidas de proteção técnicas geralmente são ineficazes contra esse tipo de ataque e, além disso, as pessoas acham que são boas em detectá-lo [2]. É possível, por exemplo, enganar um jogador para obter sua senha usando métodos sociais, como os relacionados à persuasão, e canais digitais, como redes

sociais, sites falsos e e-mails. Outros tipos de canais também podem ser utilizados, como telefonemas e até mesmo contato físico com a vítima.

Em trapaças pela modificação do binário do jogo ou dos seus dados, programas e ferramentas modificam a memória ou os arquivos do jogo. É o tipo de trapaça mais comum quando se pensa em trapaças.

Uma ferramenta amplamente empregada para escanear e editar memória de outro programa é o Cheat Engine. Nela, é necessário, inicialmente, selecionar um programa alvo no qual se deseja trabalhar. Após selecionado, é possível escanear sua memória a partir de diversas técnicas providas pela ferramenta, ou editar um endereço diretamente, caso ele já seja conhecido. Além de escanear e editar memória, há outras funcionalidades fornecidas pela ferramenta; entre elas, existem, por exemplo, injeção de código, criação e execução de scripts — que utilizam APIs do Cheat Engine — com a linguagem LUA, visualizador de *assembly* (*disassembler*) e debugador [3].

Para editar arquivos, o processo geralmente é mais complexo, pois atacantes precisam conhecer como o sistema de arquivos do jogo funciona, principalmente como eles são lidos (seu formato). Caso um atacante consiga decifrar o formato, cria-se uma ferramenta para ler e escrever um determinado arquivo. Em jogos modernos, devido ao seu tamanho, o sistema de arquivos normalmente é composto por arquivos compactados e encriptados. O processo basicamente se torna, desse modo, descobrir qual é o algoritmo de compactação e encriptação aplicado. Exemplificando, se um jogo utilizar arquivos zipados sem encriptação e o atacante descobrir esse esquema, ele pode descompactar qualquer arquivo, editar o conteúdo que deseja e depois compactar novamente.

Como esse tipo de trapaça é bem conhecido pelos desenvolvedores, é difícil editar memória ou arquivos sem que ocorra detecção da respectiva violação. Checagens de integridade, por exemplo, são muito utilizadas em ambas as modalidades de modificação.

Trapaças ao explorar bugs ou falhas de design são autoexplicativas. A melhor solução para esta categoria é utilizar boas práticas de engenharia de software a fim de construir o sistema melhor, com o máximo de qualidades que ela preconiza. É claro, um processo adequado de desenvolvimento faz parte dessas práticas. Dadas as características que o jogo possui como sistema — como alto grau de incertezas e frequentes mudanças de requisitos —, o ideal é usar alguma metodologia ou *framework* ágil, como o Scrum.

1. Referências

[1] Jianxin Jeff Yan e Hyun-Jin Choi. 2002. *Security issues in online games*. *The Electronic Library*, 20 (Abr. 2002), 125-133. DOI: http://dx.doi.org/10.1108/02640470210424455.

[2] Katharina Krombholz, Heidelinde Hobel, Markus Huber e Edgar Weippl. 2015. *Advanced social engineering attacks*. *Journal of Information Security and Applications*, 22 (Jun. 2015), 113-122. DOI: https://doi.org/10.1016/j.jisa.2014.09.005.

[3] Cheat Engine Wiki. 2021. Acessado em 30 de Maio, 2021 de https://wiki.cheatengine.org/index.php?title=Main_Page.

5. Sistemas de Trapaça

Programas de trapaça são os que modificam dados (localizados na memória ou não) do jogo ou fornecem funcionalidades de auxílio que violam os termos de uso, a fim de prover alguma vantagem a quem utiliza. Eles podem ser classificados em duas categorias: internos e externos. Além disso, podem residir nas camadas três (*user mode*), zero (*kernel mode*), ou três e zero — simultaneamente — do sistema operacional.

Modificar o binário do jogo e os arquivos usados por ele é a forma mais tradicional que se pensa quando o termo trapaça vem à tona. Arquivos são alterados para, por exemplo, diminuir a opacidade de determinado material do jogo — possibilitando que inimigos sejam vistos através de objetos e paredes —, ou até mesmo aumentar o contraste do material, facilitando a identificação de objetos aos quais o material está vinculado. Adultera-se o binário do jogo principalmente para monitoramento e alteração de métodos e funções, bem como para edição de estruturas de dados utilizadas pelo jogo. A classe do personagem normalmente contém estruturas de dados e informações críticas (posição, direção, número de vidas restantes e quantidade de experiência, que determina o nível do personagem). Essa classe é muito visada por desenvolvedores de trapaça no início do desenvolvimento, pois ela geralmente é a base de informações de que a trapaça necessita e também a base a partir da qual se derivam outras informações.

Em trapaças de baixa complexidade, o mais comum é modificar valores usados pelo jogo (edição de memória), como a quantidade de munição que o personagem possui. Sistemas de apoio a desenvolvimento de trapaças que escaneiam memória, como o Cheat Engine, são empregados de modo a obter esses valores para posterior alteração. Outra ferramenta muito popular para escanear e editar memória é o ArtMoney. Uma forma, primitiva e antiga, de modificação de dados com hardware é utilizando dispositivos, a saber: Game Genie, Code Breaker e GameShark.

Por outro lado, naquelas que dotam de maior complexidade, várias técnicas podem ser utilizadas a fim de atingir variados tipos de modificação; entre esses tipos, exemplificando, tem-se a alteração de fluxo de execução — bastante aplicada em trapaças. Ao alterar o fluxo de execução ou apenas monitorá-lo, inserem-se *callbacks* em procedimentos invocados pelo jogo

para que a trapaça execute ações específicas assim que métodos e funções são chamados (técnica conhecida como *hooking*). Num procedimento que é executado quando o personagem recebe danos, um *callback* pode ser inserido para aplicar itens de cura automaticamente, por exemplo. Ainda, essa técnica também permite que argumentos passados a métodos e funções sejam modificados. Consequentemente, seria possível alterar o valor do dano recebido caso a função ou o método tenha como parâmetro a quantidade de dano a ser recebida.

Vale destacar que trapaças mais elaboradas, além dos traços apresentados no parágrafo anterior, fazem uso de métodos e técnicas similares aos que *malwares* utilizam, como: injeção de código com bibliotecas dinâmicas (DLLs no Windows); mecanismos que adulteram o próprio sistema operacional (*rootkits*), a fim de que elas se mantenham indetectáveis frente a anti-cheats mais sofisticados; manipulação remota de memória (em espaço de endereçamento de outro processo — o do jogo); e instalação de drivers na camada kernel que implementam, entre outras funcionalidades, operações presentes em *rootkits*. É claro, essa lista é meramente exemplificativa, já que existem muitos outros métodos e técnicas passíveis de serem empregados — inclusive aqueles que exploram falhas do sistema operacional que os possibilitam funcionar. Alguns desses métodos e técnicas serão discutidos em outro momento desta pesquisa.

Ferramentas de auxílio são aquelas que automatizam ações do jogador ou interferem de alguma forma na jogabilidade, sem necessariamente interagir diretamente com o jogo, e têm a característica de serem mais genéricas do que as trapaças descritas anteriormente — em relação a aplicabilidade e programabilidade. Elas podem ler endereços de memória do alvo ou não, para programar a lógica de suas funcionalidades. AutoIt, linguagem de programação interpretada para Windows, possibilita construções de scripts (macros) visando automatizar tarefas, inclusive as que se referem a um jogo qualquer. Com ela, é possível converter scripts em executáveis, facilitando distribuições de eventuais trapaças a computadores que não possuam um interpretador de AutoIt instalado. Funcionalidades como simulação de input de mouse e teclado são providas por essa linguagem e são muito utilizadas em trapaças. Ações podem ser automatizadas também externamente (em que a ferramenta não está dentro do ambiente do sistema operacional), utilizando dispositivos robóticos que monitoram e agem conforme aquilo que recebem desse monitoramento. Por exemplo, é possível que o dispositivo

escaneie pixels da tela e aja pressionando algum botão de teclado ou mouse ou, em aparelhos móveis com tela *touchscreen*, dispararando um mecanismo físico para simular inputs (figura 1). Em jogos de tiro, podem-se simular inputs ao detectar um pixel vermelho centralizado (indicando, em muitos casos, que a mira está perfeitamente apontada para um adversário). Um programa para hardware Arduino é uma maneira de construir esse tipo de ferramenta externa [1].

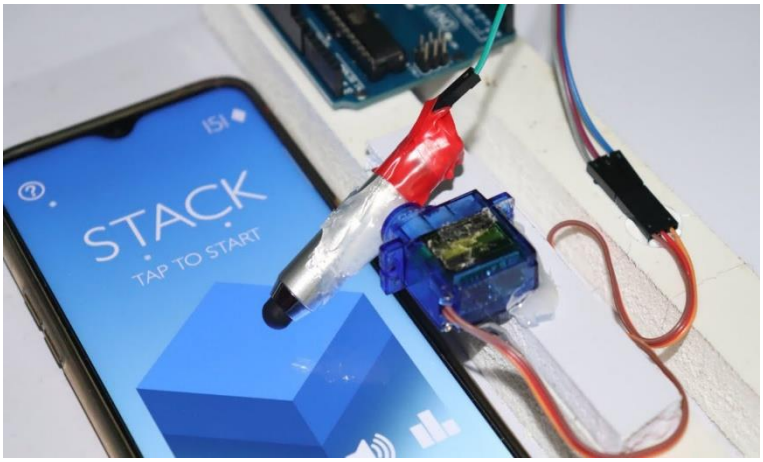

Figura 1 Dispositivo robótico com microcontrolador jogando sem intervenção humana.

1. Trapaças Internas e Externas

Trapaças internas são aquelas que habitam o espaço de endereçamento do jogo e comumente são compiladas em bibliotecas dinâmicas (DLLs), que geralmente são injetadas no processo do jogo por outro processo separado. Também é possível injetá-las sem usar esse outro processo. Como elas compartilham o espaço de endereçamento, é natural que sejam mais eficientes do que trapaças externas, que precisam passar por um processo mais burocrático a fim de interagir com o processo do jogo. Trapaças internas podem, pois, escrever e ler memória diretamente, bem como invocar métodos e funções do jogo.

Trapaças externas são processos separados e isolados, que possuem seu próprio espaço de endereçamento. Como descrito anteriormente, dependem de APIs providas pelo sistema operacional para efetuar ações em outro processo. Elas devem, primeiramente, abrir um canal de comunicação com

o processo alvo (*handles* no Windows) para, enfim, enviar suas ações — também por meio das APIs.

2. *User Mode* e *Kernel Mode*

No Windows, objeto de estudo desta pesquisa, a arquitetura do sistema operacional é constituída por camadas, em que as camadas três (*user mode*) e zero (*kernel mode*) são as mais importantes quando se deseja examinar trapaças.

De acordo com a documentação da Microsoft [2], o processador alterna entre esses modos dependendo do tipo de código sendo executado. Aplicações de usuário rodam na camada três, e componentes críticos do sistema operacional na zero. Compreende-se, portanto, que essa divisão foi pensada para que aplicações de usuário não acessem nem manipulem dados críticos do sistema operacional, visando a manter a estabilidade do sistema em face de maus comportamentos de determinada aplicação.

No modo kernel, o processador permite acesso a toda a memória do sistema e a todas as instruções da CPU. Ainda, drivers (que podem ser desenvolvidos por terceiros) residindo nesse modo têm, também, acesso total à memória do sistema — podendo, inclusive, burlar mecanismos de segurança do sistema operacional para acessar objetos internos. Desde o Windows 10, como medida de segurança, apenas drivers com certificação EV (*Extended Validation*) SHA-2 assinados por autoridades certificadoras específicas podem ser executados [3].

Outra peculiaridade notável é que aplicações de usuário alternam do modo usuário para o kernel quando fazem chamadas de serviços fornecidos pelo sistema operacional. Dessa forma, é normal que uma *thread* no modo de usuário tenha partes do seu tempo de execução em modos distintos (mas não em ambos simultaneamente) [3].

Visto isso, trapaças podem tirar vantagem do modo kernel para se blindarem de outros programas que tentem analisá-las, adulterar objetos internos do sistema operacional para dificultar sua identificação e, claro, usar todo o poder que o modo kernel oferta.

3. Referências

[1] Jason Poel Smith. *Hack a Video Game Controller With an Arduino for Greater Accessibility (or Cheating)*. Acessado em 10 de Junho, 2021 de https://www.instructables.com/Hack-a-Video-Game-Controller-with-an-Arduino-for-G/.

[2] Microsoft. 2017. *User mode and kernel*. Acessado em 11 de Junho, 2021 de https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode.

[3] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich e David A. Solomon. 2017. *Windows Internals Part 1: System architecture, processes, threads, memory management, and more* (sétima edição). *Microsoft Press*, Redmond, WA.

6. Desenvolvimento de Trapaças

Construir trapaças é um processo de variável dificuldade, dependendo de quais funcionalidades serão providas e também de haver algum anti-cheat de que o jogo se mune; neste caso, mecanismos que burlem esse anti-cheat poderão ser necessários para evitar detecções.

Esta pesquisa se propõe a apresentar os principais métodos e técnicas empregados por trapaças — bem como as ferramentas mais comuns usadas no desenvolvimento delas —, para, em momento ulterior, apresentar métodos e técnicas frequentemente utilizados por anti-cheats a fim de coibi-las.

Antes de começar, é importante destacar que há uma distinção entre trapaças e *exploits*. Estes são códigos que exploram vulnerabilidades ou bugs do jogo, como uma exploração de falha que permita duplicar itens. Greg Hoglund e Gary McGraw [1] discorrem sobre alguns *exploits* usados no jogo World of Warcraft; entre eles, estão bugs de trajeto de NPCs (personagens controlados pelo computador) que, ao serem explorados, permitiam que o jogador infligisse danos a esses NPCs sem receber dano algum deles. Trapaças não possuem essa intenção, mas sim utilizar funcionalidades e informações do jogo para construir suas próprias funcionalidades. Ou seja, a diferença está na intenção. Dessa forma, sublinha-se que esta pesquisa não abrange *exploits*.

1. Ferramentas

1.1. *Cheat Engine*

Cheat Engine é uma ferramenta de código aberto e, possivelmente, a ferramenta mais popular para criação de trapaças. Além de possibilitar escanear memória de qualquer processo, possui: depurador, *assembler, disassembler*, criador de scripts com a linguagem Lua e *speedhack*.

Quando se deseja encontrar o endereço que armazena a quantidade de munição de determinado personagem, por exemplo, é preciso escolher, primeiramente, qual tipo de dado escanear. Essa escolha pode ser trivial, mas às vezes o tipo correto só é achado por meio de tentativa e erro. Após selecionar o tipo de dado, basta inserir o valor a ser procurado. A melhor

estratégia a se adotar ao escanear memória é gerar o máximo de entropia possível no estado do jogo antes de fazer o próximo escaneamento [2]. À medida que escaneamentos consecutivos são feitos, a ferramenta restringe o espaço de endereços do valor pesquisado até chegar a uma quantidade de endereços bem pequena — momento que permitirá ao usuário decidir qual endereço realmente é o certo. Encontrado o endereço, altera-se seu valor para qualquer um desejado; também é possível “congelar” esse valor, o que, neste exemplo, não deixará a munição do personagem diminuir.

Caso não se saiba qual tipo de dado procurar, a ferramenta provê a opção de tipo “desconhecido”. Essa opção é poderosa, porém exige mais esforço do usuário para filtrar o endereço desejado.

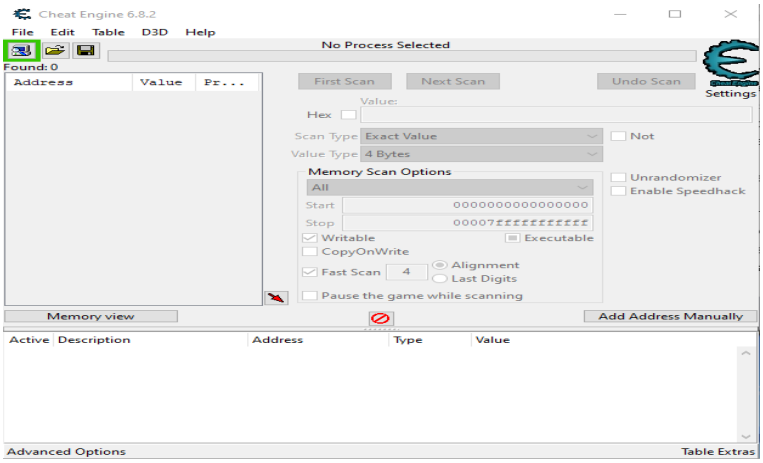

Figura 1 Interface do *Cheat Engine* 6.8.2.

1.2. OllyDbg e x64/x32 Dbg

OllyDbg e x64/x32 Dbg são depuradores de uso geral, embora possam ser utilizados para fazer engenharia reversa em jogos. Com engenharia reversa, descobrem-se estruturas de dados do programa e seu fluxo de execução. Quando se conhece como uma função funciona e como é invocada, é possível modificá-la a fim de inserir ou retirar comportamentos, bem como chamá-la artificialmente. Um método que tem a responsabilidade de infligir danos ao personagem pode ser alterado para retirar esse comportamento. Ainda, um método que faça o personagem pular pode ser chamado artificialmente para que execute essa ação em alguma situação de perigo, por exemplo. Esse segundo exemplo é particularmente explorado por

trapaças que possuem o objetivo de automatizar algum aspecto da jogabilidade do jogo.

O *disassembler* dessas ferramentas transforma código de máquina em *assembly*, que é uma linguagem mais fácil de ser compreendida por humanos.

O depurador permite, em meio a outras funcionalidades fornecidas, colocar *breakpoints* em partes do código. Esses *breakpoints* são poderosos porque é possível programá-los de tal maneira que apenas sejam ativados ao satisfazer determinada condição, ou quando algum endereço de memória for lido ou escrito.

Também há como visualizar a *stack* e outras informações do programa que está sendo analisado, a exemplo de suas *threads* ativas e suas seções de código (obtidas a partir do seu formato PE [3]).

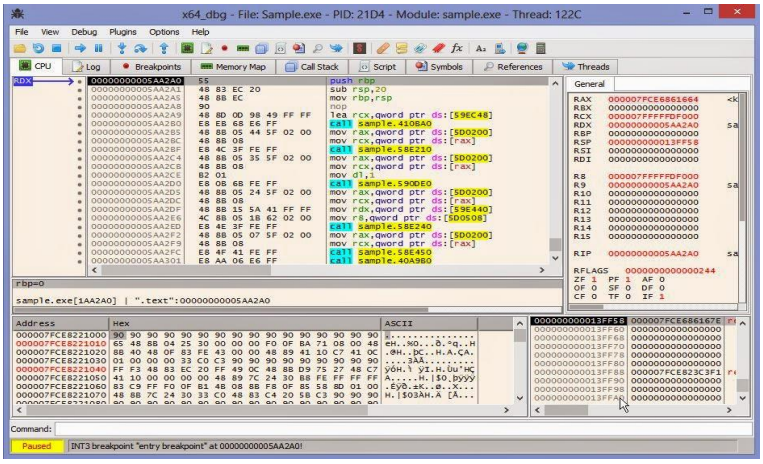

Figura 2 Interface do x64dbg.

1.3. *ReClass*

ReClass é uma ferramenta que analisa a estrutura da memória de um processo e aplica *offsets* automaticamente (identifica os deslocamentos dos campos de uma estrutura relativos ao endereço dessa estrutura) a uma região de memória especificada pelo usuário. É uma alternativa à função de dissecar dados e estruturas do *Cheat Engine*, que facilita encontrar informações necessárias para criar trapaças. Por exemplo, a tarefa típica de interpretar a estrutura de dados que armazena dados de personagens fica mais fácil.

Além da funcionalidade principal de interpretar regiões de memória, também é possível gerar código em C++ e C# que representa a região sob análise (figura 3). A região, de acordo com a figura, pode ser mapeada diretamente para classes (*class*) e estruturas (*struct*) da linguagem C++.

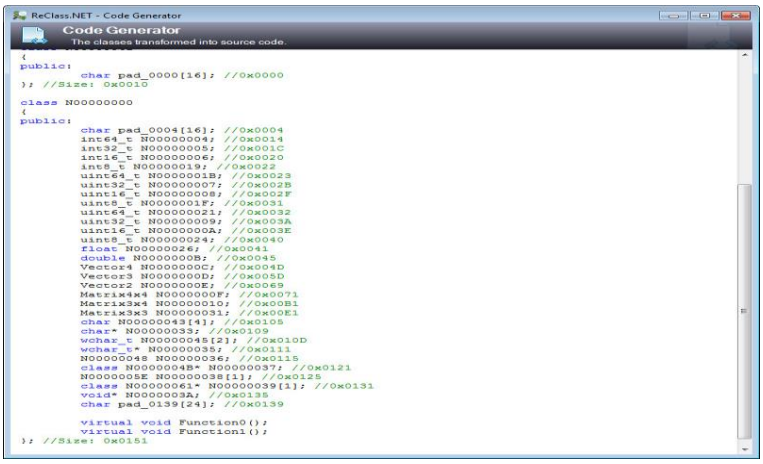

Figura 3 Geração de código com *ReClass* [4].

1.4. IDA Pro

IDA Pro é um *disassembler e debugger* muito utilizado em engenharia reversa de softwares. Ao importar um executável, a ferramenta executa uma análise automática para encontrar blocos de código que possivelmente representam funções. APIs do Windows são reconhecidos automaticamente e nomeados de acordo.

Uma funcionalidade (intitulada “Decompilador”) bem útil provida pelo IDA é a conversão de código de máquina em pseudocódigo na linguagem C (figura 4).

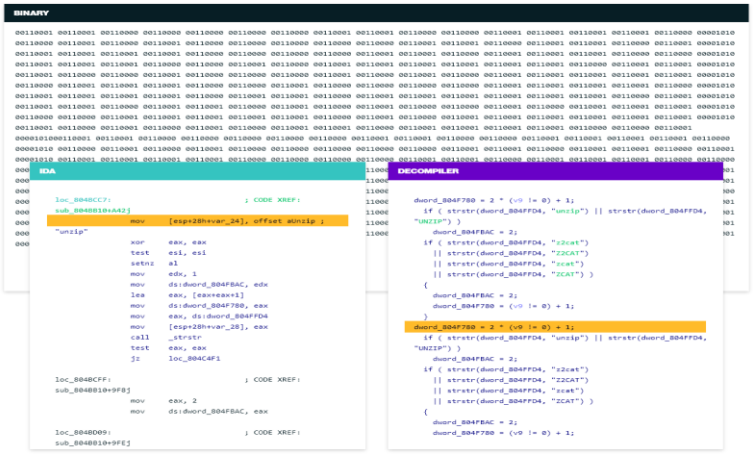

Figura 4 Código de máquina em pseudocódigo C, por meio da funcionalidade “Decompilador” do IDA Pro [5].

Outra função conveniente é poder visualizar gráficos de códigos de máquina. Com ela, entender o fluxo de execução do código fica mais fácil (figura 5). É possível identificar ramificações de código rapidamente, como mostra a figura.

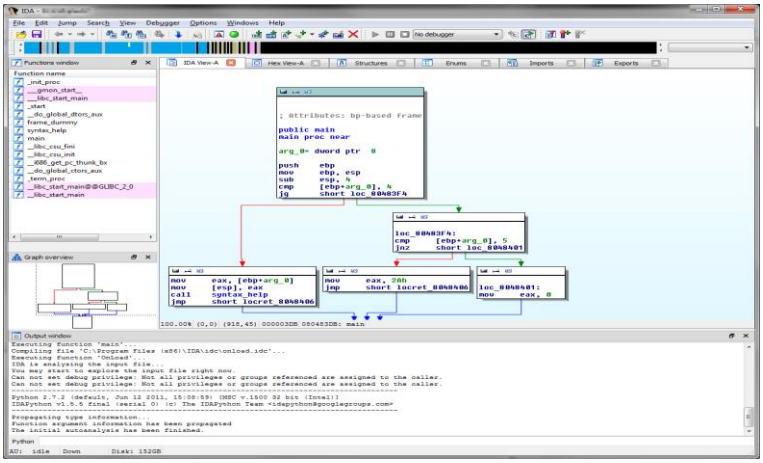

Figura 5 Gráfico de código de máquina, no IDA Pro.

1.5. *Process Monitor*

Process Monitor é uma ferramenta de monitoramento avançada para Windows que exibe atividades desempenhadas por um processo — como operações de rede e interações (criações, leituras e escritas) feitas com arquivos e registros do Windows —, em tempo real. Além disso, também mostra a pilha de cada *thread* do processo, o que torna possível entender, em alto nível, o fluxo de execução do programa (revelando o contexto de

operações que causou determinado evento) e ter uma ideia dos procedimentos mais chamados e onde residem (em qual biblioteca).

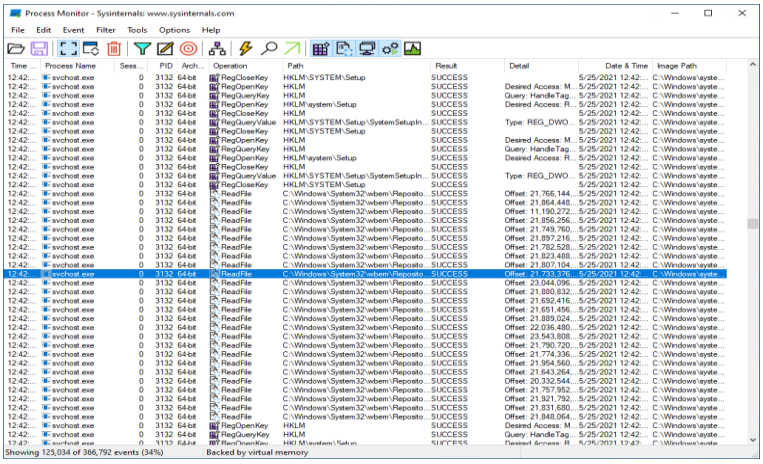

Figura 6 *Process Monitor*.

1.6. *Process Explorer*

Process Explorer é outra ferramenta de monitoramento que pode ser vista como um gerenciador de tarefas avançado. Ela provê diversas informações e estatísticas de determinado processo, como seus processos pais e filhos, sua lista de *handles* abertas, as bibliotecas carregadas e seu uso de CPU e memória.

Por meio da lista de *handles*, é possível identificar objetos importantes usados pelo programa, a exemplo de arquivos, *mutexes* e segmentos de memória compartilhada. Esses dois últimos, especialmente, são úteis quando se deseja burlar a restrição de haver apenas uma instância do sistema executando (geralmente implementada com *mutexes* [6]) e quando se deseja saber se há compartilhamento de dados com outros processos (comunicação entre processos), respectivamente. Por outro lado, a partir da exibição que mostra quais arquivos estão sendo usados pelo programa, fica fácil identificar a localização de arquivos críticos do jogo, como arquivos de *log* e *save*, para possivelmente manipulá-los posteriormente.

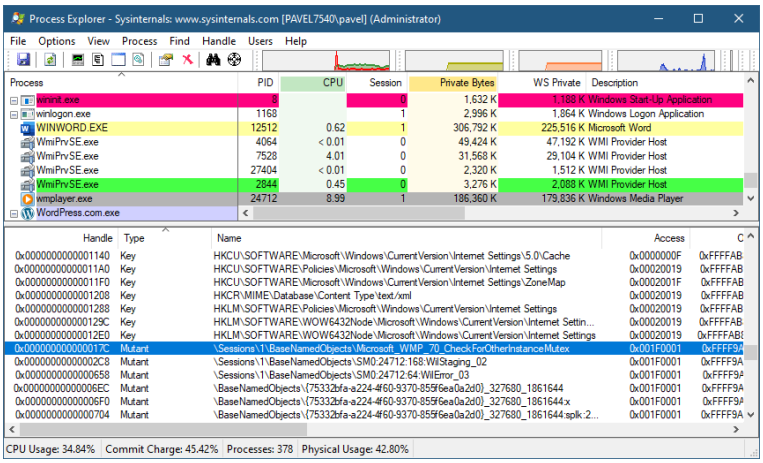

Figura 7 Visualização de *handles* com *Process Explorer*.

É possível aprender muito sobre o funcionamento do jogo (ou sistemas, de forma geral) ao fazer análises superficiais de suas interações e operações com o sistema operacional. Consequentemente, o *Process Monitor* e o *Process Explorer* são utilizados em etapas prévias a investigações mais detalhadas.

2. Técnicas

2.1. Lendo e alterando memória

Ler e escrever memória é fundamental para o desenvolvimento de qualquer trapaça.

Com leitura, trapaças checam determinados estados do jogo para tomar decisões e agir. Por exemplo, se a vida do personagem estiver abaixo de certa porcentagem, uma ação possível seria utilizar itens de cura automaticamente, ou fazer o personagem recuar ou se esconder de seus adversários.

Por outro lado, o propósito da escrita é alterar valores que armazenam dados e informações críticas de elementos que afetam a jogabilidade — a exemplo das propriedades do personagem e das armas equipadas por ele —, bem como modificar códigos de máquina; tudo a fim de obter algum tipo de benefício. Somente com a possibilidade de escrever memória, a jogabilidade pode ser impactada significativamente — tanto de quem usa a trapaça quanto de outros jogadores, caso o jogo seja online. Logo, *anti-*

cheats devem se esforçar ao máximo na construção de barreiras para edição de memória.

A saber, apenas com alteração de memória, é possível: modificar a vida do jogador para algum valor “infinito”, ou congelar esse valor, visando que o personagem não sofra danos; modificar a velocidade do personagem, implicando uma locomoção mais veloz, o que torna possível explorar mapas rapidamente e esquivar-se de inimigos com facilidade; modificar o dano da arma do personagem, fazendo com que, caso o valor seja alterado para algum muito grande, adversários sejam abatidos instantaneamente; modificar a quantidade de munição da arma, para que o personagem possa disparar continuamente, sem mais precisar recarregar, ou para que tenha munição ilimitada de certa arma que foi projetada para ser escassa em questão de munição, como granadas; modificar o recuo da arma para zero, para obter precisão perfeita em disparos; modificar o valor da quantidade de moedas que o personagem possui, a fim de comprar recursos de forma ilimitada; e modificar estados específicos do personagem para ficarem permanentes, como alterar sua visibilidade frente a inimigos (em alguns jogos, o jogador fica invisível quando se esconde em matas). Os exemplos acima são uma pequena parte do que é passível de ser feito com edição de memória: é muito importante existir uma proteção eficaz contra esse tipo de trapaça.

Alterar códigos de máquina é outra possibilidade quando o assunto é escrever memória. Trapaças podem modificar instruções de condicionais para pular certo comportamento, como uma checagem que verifica se determinada arma pode ser disparada novamente (verificação de cadência). Ainda, a instrução DEC, que reduz um valor em uma unidade, pode ser substituída por INC para incrementar, fazendo, por exemplo, em muitos jogos, munição ilimitada de armas. Outra maneira de descartar comportamentos indesejados é a partir da instrução NOP, substituindo todo o código de máquina alvo por uma sequência de NOPs (Anexo I, “Removendo código indesejado com NOPs”).

No Windows, processos externos ao jogo precisam adquirir uma *handle* do processo do jogo para efetuar operações de leitura e escrita de memória. Primeiramente, é preciso obter o ID do processo (PID), que pode ser coletado diretamente, por meio de ferramentas externas, ou por meios indiretos, a partir do nome da janela ou do nome do processo. Com o PID, basta fornecê-lo à função *OpenProcess* para adquirir a *handle*.

Em “Obtendo identificadores de um processo”, no Anexo I, há trechos de código que demonstram o que foi escrito acima.

No caso de trapaças internas, que residem no espaço de endereçamento do processo do jogo, é possível ler e escrever memória diretamente, sem precisar de uma *handle*. Nessa situação, é suficiente alterar as proteções da região de memória a ser lida ou escrita, usando a função *VirtualProtect*, e em seguida executar a operação desejada, com a função *ReadProcessMemory* ou *WriteProcessMemory*, ou diretamente, utilizando ponteiros.

Em “Lendo e escrevendo memória de um processo”, no Anexo I, alguns códigos exemplificam como ler e escrever memória.

É importante comentar que existe um mecanismo chamado *Address space layout randomization* (ASLR) desde o Windows Vista, cujo objetivo é prevenir vulnerabilidades de corrupção de memória (*overflow* e *underflow* em *heap* ou *stack*, formato inadequado de *strings*, *overflow* em índice de vetores e variáveis não inicializadas, por exemplo [14]). Em vez de carregar seções do executável (como *heap*, *stack* e código) em endereços fixos da memória, elas são carregadas em localizações aleatórias, diminuindo a previsibilidade do *layout* da memória.

No que tange a trapaças, o ASLR força que os endereços utilizados por elas sejam calculados relativos a uma localização (ou seja, deslocamentos a partir dela). Para endereços que estão na seção de código, o que costuma ser o caso em trapaças, a localização mencionada é o endereço base do processo, podendo ser obtido com o API *GetModuleHandle* (Anexo I, “Encontrando o endereço base de um processo”).

2.2. Encontrando endereços automaticamente

Trapaças precisam de endereços, que são obtidos durante a etapa de engenharia reversa e escaneamento de memória, para construir suas funcionalidades.

Um problema que surge ao utilizar endereços estáticos é que eles variam entre diferentes versões do jogo. Para que não seja necessário atualizar esses endereços após atualizações, trapaças sofisticadas derivam os endereços de que necessitam a partir de extratos de código de máquina do jogo, que sinalizam padrões para chegar a determinado endereço.

Dessa forma, em vez de capturar o endereço desejado diretamente e inseri-lo na trapaça, o código de máquina nas redondezas desse endereço — ou o código que o lê ou o modifica — é analisado e extraído de tal maneira que instruções com conteúdo variável (geralmente as que fazem referência à memória, ou a endereços dinâmicos) são substituídas por *wildcards*, indicando que essas instruções devem ser ignoradas quando a comparação no procedimento de busca por padrões estiver sendo feita.

Em “Comparando memória se baseando em padrões, para encontrar endereços de memória após atualizações”, no Anexo I, o código demonstra como achar endereços com essa técnica. Nele, os *wildcards* são representados com “*”.

2.3. *Hooking*

No contexto de trapaças, *hooking* é a técnica utilizada para alterar comportamentos de funções, inserindo código que executa processamentos imediatamente anteriores ou posteriores ao corpo da função que sofreu o *hook* (é possível colocar o *hook* no meio da função — técnica conhecida como *mid-function hooking* —, mas isso foge ao escopo desta seção). Os processamentos podem servir de gatilho para alguma funcionalidade de uma trapaça, ou alterar os argumentos ou o retorno da função, por exemplo.

Existem diversos métodos de *hook*, cada um com diferentes níveis de complexidade e para propósitos específicos. Vale ressaltar que, independentemente da técnica, o código que fará processamentos adicionais deve ser inserido de tal forma que seja capaz de voltar ao fluxo de execução original. Adiante, apenas os métodos *Byte Patching*, *IAT/EAT Hooking* e o método com *breakpoints* de hardware [7] serão expostos.

Byte patching é uma técnica simples e eficaz, que consiste em colocar instruções com comportamento de desvio.

Para ilustrar, pode-se usar a instrução JMP no intuito de pular para o endereço do código que fará processamentos adicionais e, finalizado o processamento, desviar novamente para restaurar o fluxo de execução original; a instrução CALL insere o endereço de retorno na pilha antes de pular, logo é outra forma de fazer o desvio; ou utilizar as instruções PUSH e RET, colocando, por exemplo, PUSH 0xDEADBEEF e em seguida RET (desviará para 0xDEADBEEF, pois esse endereço estará no topo da pilha).

Outra maneira mais rebuscada de desviar, com *byte patching*, é registrar um tratador de exceções e depois gerar uma exceção a partir de alguma instrução específica, como INT3. Assim, basta que o código do tratador seja programado para modificar o registro EIP para apontar para o endereço do código que fará processamentos adicionais, causando o redirecionamento de execução desejado.

IAT/EAT *hooking* é baseado na forma pela qual os executáveis do Windows são estruturados (formato PE [8]). IAT significa *Import Address Table* e EAT, *Export Address Table*. Essas tabelas contêm ponteiros para APIs, que são inicializados por um programa chamado carregador PE [9]. O carregador PE é responsável por carregar executáveis do disco, interpretando o formato PE.

A ideia desse método é substituir o ponteiro de algum API por um que aponta para uma função específica, redirecionando a execução para essa função, em vez de executar o API. É importante destacar que o método não funciona apenas para tabelas IAT ou EAT, mas para qualquer tabela de ponteiros.

Hooks com substituição de ponteiros são bastante utilizados por trapaças, já que bibliotecas gráficas, como DirectX, armazenam internamente ponteiros para os APIs que disponibilizam. Portanto, ao *hookar* APIs gráficos, fica possível manipular como o jogo é renderizado, abrindo espaço, por exemplo, para trapaças que exibem adversários ocultos por objetos ou paredes opacas (*wallhack* ou ESP).

Outra técnica é usar *breakpoints* de hardware para ativar um tratador de exceções, como foi discutido no método de *byte patching*. A diferença é que, com *breakpoints* de hardware, não é necessário modificar memória para pôr os *breakpoints*; em vez disso, *breakpoints* de hardware são inseridos diretamente no contexto de uma *thread* [10]. Logo, a ideia da técnica é suspender a *thread*, alterar seu contexto e depois resumi-la, gerando a interrupção.

2.4. Modificando arquivos

Nesta técnica, a trapaça altera arquivos no disco antes de serem carregados na memória, como o executável do jogo, os arquivos (de configuração, de conteúdo e outros) usados pelo jogo ou as bibliotecas de que o jogo depende.

Uma aplicação popular é modificar arquivos a fim de substituir texturas de modelos; no caso, torna-se uma textura opaca em transparente, para obter visão de adversários que estão atrás de paredes ou objetos opacos, ou editar uma textura de tal modo que ela fique com um contraste maior, facilitando a identificação de objetos importantes ou inimigos, por exemplo. Outra aplicação [11] é editar arquivos de *save* para melhorar atributos do personagem, avançar níveis ou adicionar itens ao inventário.

2.5. Injetando código

Há muitas maneiras de adicionar e executar códigos particulares em uma aplicação. Uma das formas é com *hooks*, que já foram descritos. Outra forma, mais útil para quantias maiores de código, é injetar bibliotecas dinâmicas.

Com injeção de bibliotecas, o código injetado, armazenado na biblioteca, é executado no espaço de endereçamento do processo que sofreu a injeção. São vários os métodos que forçam um processo a carregar código alheio, então apenas as técnicas Injeção Clássica, *Hook Injection* e *Thread Hijacking* serão descritas a seguir.

Um método direto e de fácil implementação e compreensão (apelidado de Injeção Clássica) é com o uso do API *LoadLibrary*, fornecido desde o Windows XP pela Microsoft. O ponto-chave é entender que uma biblioteca chamada kernel32.dll é mapeada no mesmo endereço em quase todos os processos. Assim, o *LoadLibrary*, que é exportado por essa biblioteca, também sempre fica no mesmo endereço, mesmo em processos distintos.

Para o método funcionar, primeiramente é preciso criar uma *string* no processo alvo para armazenar o caminho da biblioteca a ser carregada. Isso se faz com o API *VirtualAllocEx* — para alocar a região de memória —, e *WriteProcessMemory* — para colocar o valor (caminho) nessa região. Feita essa etapa, deve-se criar uma *thread* no processo alvo com o API *CreateRemoteThread*; ela iniciará sua execução no endereço do *LoadLibrary*, que receberá o endereço da *string* alocada anteriormente como argumento e carregará a biblioteca.

Outra forma de injetar é explorando o mecanismo de *hooks* do Windows (*Hook Injection*), que permite adicionar um pedaço de código a ser executado quando uma mensagem (*Window Messages*) é recebida pela aplicação [12]. O API *SetWindowsHookEx* faz o processo alvo carregar uma

DLL, especificada entre os argumentos do API, para posteriormente executar uma função exportada, também especificada nos argumentos, por essa DLL ao receber determinado evento.

O método *Thread Hijacking* [13] funciona suspendendo uma *thread* do processo com o API *SuspendThread*, e depois usando o API *SetThreadContext* para modificar o contexto da *thread*, fazendo seu registro EIP apontar para o endereço do *LoadLibrary*. O argumento desse API, que representa a localização da biblioteca, deve ser alocado e preenchido antecipadamente, podendo fazer isso de maneira análoga ao método de Injeção Clássica. Alterado o contexto, resume-se a execução da *thread* com o API *ResumeThread*.

2.6. Manipulando recursos

Manipulação de recursos se refere a tudo que uma trapaça pode manipular para atingir seus objetivos, como as *threads* do jogo, as *handles* abertas pelo jogo, as permissões do processo do jogo ou das suas *threads*, e outros objetos do sistema operacional, associados ao processo do jogo.

Por exemplo, se o desenvolvedor de uma trapaça chegar à conclusão de que uma *thread* específica tem a responsabilidade de realizar checagens a fim de detectar trapaças, ele simplesmente pode terminar a *thread* (com o API *TerminateThread*) ou suspendê-la.

3. Referências

[1] Greg Hoglund e Gary McDraw. 2007. *Exploiting Online Games: Cheating Massively Distributed Systems*. *Addison Wesley Professional*.

[2] Nick Cano. 2016. *Game Hacking: Developing Autonomous Bots for Online Games*. *No Starch Press*.

[3] Microsoft. *PE Format*. Acessado de https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.

[4] Github. *ReClassNET/ReClass.NET: More than a ReClass port to the .NET platform*. Acessado de https://github.com/ReClassNET/ReClass.NET.

[5] Hex-rays. IDA Pro. Acessado de https://hex-rays.com/ida-pro/.

[6] Stack Overflow. *What is the correct way to create a single-instance WPF application?*. Acessado de https://stackoverflow.com/questions/19147/what-is-the-correct-way-to-create-a-single-instance-wpf-application.

[7] Rohitab. *The different ways of hooking*. Acessado de http://www.rohitab.com/discuss/topic/41855-tutorial-the-different-ways-of-hooking/.

[8] Microsoft. Formato PE. Acessado de https://docs.microsoft.com/pt-br/windows/win32/debug/pe-format.

[9] Github. *A Windows PE format file loader*. Acessado de https://github.com/polycone/pe-loader.

[10] Ling. *Hardware breakpoints and exceptions on Windows*. Acessado de https://ling.re/hardware-breakpoints/.

[11] Fantasy Anime. *Save State Hacking*. Acessado de https://fantasyanime.com/save-state-hacking.

[12] Nick Harbour. *Stealth Secrets of the Malware Ninjas*. Conferência *Black Hat*, EUA, 2007.

[13] Amit Klein e Itzik Kotler. *Windows Process Injection in 2019*. Conferência *Black Hat*, EUA, 2019.

[14] Ollie Whitehouse. *An Analysis of Address Space Layout Randomization on Windows Vista*. Apresentado em *Black Hat DC*, Virgínia, EUA, 2007.

7. Proteção de Software

Esta seção apresentará, sem entrar em detalhes, algumas técnicas comumente usadas para proteger software. O foco será dado àquelas que estão no escopo de jogos.

Assumindo que o código do jogo rodará no computador do usuário (hoje em dia, com os avanços da computação em nuvem, jogos podem executar em máquinas remotas, apenas recebendo *inputs* e enviando a imagem do estado do jogo (*output*) ao computador do usuário — tudo pela rede [4]), é importante lembrar que ele possui controle total da máquina, então nenhuma técnica provê proteção absoluta. Entretanto, quanto mais técnicas forem empregadas corretamente, maior será a dificuldade de um atacante causar danos ou descobrir e utilizar informações para ganho próprio.

1. Ofuscação

Ofuscação refere-se ao conjunto de técnicas que aplicam transformações que fazem a execução do programa ser difícil de ser investigada. De acordo com Collberg e Thomborson [3], ofuscação transforma um programa em um equivalente, preservando suas funcionalidades e dificultando ações de engenharia reversa. Engenharia reversa visa analisar o sistema para descobrir seu funcionamento, seus mecanismos, seus algoritmos e informações ocultas ou sigilosas.

Para entender ofuscação, primeiramente é preciso conhecer o que é análise estática e análise dinâmica. Análise estática é o processo de analisar um programa sem precisar executá-lo [1]. É feita a fim de entender, em alto nível, a estrutura do código e da aplicação em si. Por outro lado, análise dinâmica observa o comportamento de um programa em execução [2], monitorando registros, valores na memória, *heap*, *stack* e instruções executadas. Ambas as análises fazem uso de *disassemblers*, para converter código de máquina em *assembly*, que é analisado.

Collberg e Thomborson [5] classificam transformações de acordo com o alvo que sofrerá a transformação. São quatro tipos de transformação: em estruturas léxicas (*layout* da aplicação), no fluxo de controle, em dados e preventivas.

Em estruturas léxicas, nomes de variáveis são trocados de modo a remover o valor semântico que o nome proporciona. Reordenar funções que estão próximas no código, remover comentários, remover símbolos de depuração e alterar a formatação do código também são outras técnicas.

Transformações no fluxo de controle modificam estruturas de controle (condicionais, repetições etc) sem alterar seu comportamento. Entre essas transformações, há as que tornam o grafo de fluxo de controle uniforme [6], as que substituem condicionais por sinais e posteriormente inserem instruções aleatórias (“lixo”, *junk code*) [7] e as que colocam predicados opacos, que não são executados em tempo de execução e são difíceis de entender quando se faz análise estática [8].

Ofuscação em dados abrange transformações que ocorrem em nível de dados, como codificação, ordenação, agregação e armazenamento. Obfuscar estruturas de dados, relações de herança, constantes e variáveis são exemplos dessas transformações.

Transformações preventivas ajudam outras transformações a ficarem mais resistentes a ataques. Técnicas de *anti-disassembly*, que fazem ferramentas de *disassembly* interpretarem incorretamente um código sob análise, e *anti-debug*, que detectam se um debugador está presente ou dificultam atividades de depuração, podem ser consideradas transformações preventivas.

Outra maneira de obfuscar é com encriptação de código. Durante a execução do programa, partes do código são descriptografadas antes de serem executadas. Nesse momento, o código estará em texto claro, podendo ser inspecionado e analisado. Essa é uma vulnerabilidade da técnica. Uma alternativa à encriptação é a compressão (*packing*), para obter código ilegível. A ferramenta UPX [9], um *packer*, pode ser utilizada para isso, reduzindo o tamanho do executável e servindo como uma forma indireta de ofuscação.

Outras técnicas bem interessantes, que são utilizadas em jogos [10], são: ofuscação de ponteiros, reordenação do *heap* e migração de variáveis da *stack* para o *heap*. Com essas técnicas, ferramentas que escaneiam memória se tornam ineficazes, pois elas funcionam justamente comparando estudos do *heap* para encontrar o valor desejado. Encriptação de ponteiros também é outra barreira para essas ferramentas, já que varreduras automáticas de ponteiros (*Pointer scan* no *Cheat Engine*, por exemplo) — que visam encontrar ponteiros estáticos entre execuções da aplicação —

ficam ineficazes. Finalmente, a migração de variáveis mencionada pode ser usada para colocar valores críticos ou constantes no *heap*, que estarão protegidos com os mecanismos citados anteriormente.

2. Controles de Integridade

Controles de Integridade fazem referência às técnicas que detectam quando dados ou alguma funcionalidade do sistema é alterada sem permissão.

Uma técnica simples para verificar a integridade do programa ao ser executado é com assinatura. O fabricante do software assina e o usuário, por meio do sistema operacional, verifica a assinatura, que agora é parte do software. A verificação depende de uma Infraestrutura de Chave Pública, a fim de checar a autenticidade da chave pública. Assinaturas são utilizadas pela Microsoft em seus *drivers*, e são obrigatórias em *drivers* desenvolvidos e distribuídos por terceiros [11].

Outra proteção trivial é com *checksum*, em que o *hash* do executável é calculado e comparado com o *hash* original desse executável. Se a comparação falhar, então se assume que o sistema foi corrompido. Apesar de ser fácil implementar esse controle, é muito fácil burlá-lo; basta que o atacante descubra a localização do *hash* original e altere para o *hash* desejado, ou modifique a função que calcula o *hash* no intuito de retornar o *hash* original sempre. Uma alternativa à comparação local de *hash* é fazê-la remotamente.

Chang and Atallah [13] apresentam um esquema em que pedaços de código calculam um *checksum* de outros pedaços de código. Dessa forma, pode-se imaginar uma corrente, em que os nós (pedaços de código) se verificam mutuamente. Nesse esquema, também é possível que um nó repare outro que esteja corrompido. Assim, como demonstrado no documento, é necessário atacar toda a rede de nós para modificar um programa com sucesso. Uma desvantagem desta técnica é que é difícil automatizar a construção dessa rede de nós — dificultando futuras manutenções —, visto que fica nas mãos do programador a missão de especificar em quais partes do código os nós de verificação (*guards*) devem residir.

Aucsmith [12] apresenta diversas técnicas de controle de integridade. Entre elas, há assinatura, encriptação e checagem mútua de segmentos de código.

Outra técnica para conseguir integridade é com virtualização de código. Nesse tipo de virtualização, trechos de código especificados são substituídos por instruções fictícias, que são decodificadas e executadas por uma máquina virtual interna. O software Themida, bem popular para proteger executáveis, possui essa função de virtualização. Embora virtualizar não seja diretamente relacionado à integridade, essa forma de ofuscação faz com que um atacante não consiga modificar a seção de código protegida diretamente. Ora, se houver uma alteração em instruções virtualizadas e a modificação (instruções substituídas) não for interpretável pela máquina virtual, é natural que ocorra o *crash* da aplicação. Assim, o atacante primeiramente precisa entender como a máquina virtual funciona antes de fazer as alterações pretendidas.

3. Análise Estática

Análise estática se refere a estudos que não exigem a execução real do código. Ela é utilizada por compiladores para otimizar o código [15].

O primeiro passo quando se faz engenharia reversa de um executável é torná-lo legível para o ser humano, desmontando-o. A desmontagem, ou *disassembly*, é o inverso da montagem, ou *assembly*, que um compilador realiza, em que o código fonte é convertido em instruções específicas da CPU. Ainda que seja uma análise estática, *disassemblers* dependem de suposições. Por exemplo, a menos que especificado de outra forma, um *byte* pode ser tanto código ou dado. Dessa forma, é possível enganar um *disassembler* ao adicionar *bytes* supérfluos (*junk code*) entre instruções [14].

Outro método para entender o código melhor é usando decompiladores. Em vez de apenas fazer engenharia reversa em cima do código de máquina, a decompilação facilita esse processo, procurando padrões que podem ser convertidos para pseudocódigo de alto nível. A ferramenta IDA Pro dispõe dessa funcionalidade, decompilando código de máquina em pseudocódigo na linguagem C.

Sem entrar a fundo, já que não é o objetivo principal deste trabalho, algumas técnicas para se proteger contra análise estática serão citadas a seguir.

Encriptação de código, como já mencionada, pode servir para dificultar análise estática. Como o código é tratado como uma sequência de *bytes* e cifrado para posteriormente ser decifrado em tempo de execução, isso impossibilita que um atacante veja o código original diretamente, apenas abrindo o programa e vendo o binário. Uma desvantagem desta técnica é que escaneadores de vírus podem detectar o sistema erroneamente como *malware* por meio de busca por padrões, pois bastantes *malwares* conhecem e empregam essa técnica. Ainda, é vulnerável a análises dinâmicas, bastando executar a aplicação, pausá-la (com *breakpoints*) na iminência do código cifrado ser executado (agora já decifrado) e, enfim, extrair o código original da memória (*memory dump*).

Ofuscação de código é outra maneira de impor barreiras à análise estática. Tanto alterar o fluxo de controle quanto as estruturas de dados, por exemplo, são meios de obstar a compreensão do código de máquina. É claro, como já descrito, inserir *junk code* também é outra possibilidade.

4. Análise Dinâmica

Em contraste à análise estática, a dinâmica se baseia na execução do sistema para analisar seu comportamento, examinando suas atividades (chamadas de sistema, de APIs etc) em tempo real.

Segundo Amir Afianian et al. [16], técnicas de evasão a análises dinâmicas se dividem em duas categorias: evasão manual (*anti-debugger*) e evasão automática de *sandboxes*. A primeira é quando um humano faz a análise, normalmente utilizando *debuggers*. A segunda, por outro lado, refere-se ao processo feito por uma máquina ou software.

Grande parte das técnicas de evasão manual têm os *debuggers* como alvo, que são os instrumentos principais em análises dinâmicas manuais. *Debuggers* são softwares que inspecionam a execução de outro programa, o que inclui percorrer o código instrução por instrução, pausar o sistema como um todo, examinar variáveis e registros, entre outras funções. Colocar *breakpoints*, por exemplo, permite que *debuggers* parem a execução da aplicação em qualquer endereço. Entre os mecanismos de evasão

apresentados para esta categoria, estão: leitura do PEB (*Process Environment Block*), varredura que procura *breakpoints* e procura por artefatos do sistema que indiquem a presença de um *debugger*.

O PEB é uma estrutura de dados que contém informações do processo. Assim, é possível inspecioná-lo para checar se um *debugger* está presente. APIs do Windows como *IsDebuggerPresent* e *CheckRemoteDebuggerPresent* leem o PEB para fazer essa verificação. Outros métodos baseados em PEB examinam os campos *NtGlobalFlags* e *ProcessHeap*.

Breakpoints podem ser tanto de software quanto de hardware. Nos de hardware, o endereço que sofrerá a pausa é armazenado nos registros da CPU. Nos de software, substitui-se o primeiro *byte* do endereço alvo para 0xCC (instrução INT3). Assim, para fazer a detecção, basta escanear a memória e checar se existe alguma instrução INT3, ou, no caso de hardware, utilizar o API *GetThreadContext* para verificar o estado dos registros da CPU.

Debuggers deixam traços no sistema de arquivos, nos registros e no nome do processo. Dessa maneira, pode-se descobrir sua presença procurando essas pistas. O API *FindWindow*, a saber, pode ser usado para checar se determinado processo está ativo no sistema.

Na categoria de evasões automáticas, existem os *sandboxes*, que são ambientes controlados e confinados que rodam programas. O sistema operacional, que executa o *sandbox*, fica, então, blindado de comportamentos maliciosos por parte do programa que está executando no *sandbox*. Entre as técnicas de evasão para esta categoria, está a técnica que recolhe impressões (*fingerprinting*), ou pistas, a fim de identificar *sandboxes*. A técnica procura sinais que indicam se a execução está sendo feita num ambiente simulado. Sinais incluem *drivers* (máquinas virtuais geralmente precisam de *drivers* específicos para que possam se comunicar com o sistema operacional hospedeiro), arquivos no disco, valores de registros, propriedades da CPU e outros processos que iniciam quando a máquina virtual é executada.

Além das técnicas apresentadas acima, outra opção é usar encriptação de código. A diferença em relação à análise estática é que a encriptação deve ser feita paulatinamente, descriptografando quando necessário e logo em seguida encriptando novamente, após executar toda a região protegida. Apesar de também ser suscetível a extrações em tempo de execução, pode

ser usada concomitantemente com as outras técnicas descritas nesta seção, principalmente as que estão relacionadas a inibir a utilização de *debuggers*.

5. Referências

[1] Deepsource. *What is Static Analysis*? Acessado de https://deepsource.io/glossary/static-analysis.

[2] NTT Application Security. *Dynamic Analysis*. Acessado de https://www.whitehatsec.com/glossary/content/dynamic-analysis.

[3] Christian S. Collberg e Clark Thomborson. *Watermarking, tamper-proofing, and obfuscation-tools for software protection*. *IEEE Transactions on software engineering*, 28.8, págs. 735-746, 2002.

[4] The New York Times. *'Crucial Time' for Cloud Gaming, Which Wants to Change How You Play*. Acessado de https://www.nytimes.com/2021/07/01/technology/cloud-gaming-latest-wave.html.

[5] Christian Collberg, Clark Thomborson e Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Universidade de Auclanda, Auclanda, Nova Zelândia.

[6] Tımea László e Ákos Kiss. *Obfuscating C++ programs via control flow flattening*. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae*, Seção de Informática, 30.1, págs. 3-19, 2009.

[7] Zhi Wang et al. *Branch obfuscation using code mobility and signal*. *IEEE 36th Annual Computer Software and Applications Conference Workshops*, 2012.

[8] Anirban Majumdar e Clark Thomborson. *Manufacturing opaque predicates in distributed systems for code obfuscation*. *Proceedings of the 29th Australasian Computer Science Conference*, 48, 2006.

[9] UPX. *UPX: the Ultimate Packer for eXecutables*. Acessado de https://upx.github.io/.

[10] Riot Games. *Riot's Approach to Anti-Cheat*. Acessado de https://technology.riotgames.com/news/riots-approach-anti-cheat.

[11] Microsoft. *Driver Signing*. Acessado de https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing.

[12] David Aucsmith. *Tamper resistant software: An implementation*. *International Workshop on Information Hiding*, págs. 317-333, 1996.

[13] Hoi Chang e Mikhail J. Atallah. *Protecting software code by guards*. *ACM Workshop on Digital Rights Management*, págs. 160-175, 2001.

[14] Christopher Kruegel et al. *Static disassembly of obfuscated binaries*. *USENIX security Symposium*, pág. 18, 2004.

[15] Anjana Gosain e Ganga Sharma. *Static analysis: A survey of techniques and tools*. *Intelligent Computing and Applications*, págs. 581-591, 2015.

[16] Amir Afianian et al. *Malware dynamic analysis evasion techniques: A survey*. *ACM Computing Surveys (CSUR)*, 52.6, págs. 1-28, 2019.

8. Proteção contra Trapaças

Existem duas maneiras de se proteger contra trapaças: a partir de proteções implementadas na parte do servidor e de proteções na parte do cliente, que fica no mesmo sistema operacional que o executável do jogo.

Verificações no servidor são sempre as mais eficazes, pois um possível adversário não possui acesso ao servidor, logo não há a preocupação de colocar controles para modificações não autorizadas. Embora nem todo tipo de jogo permita que toda a sua lógica seja checada no servidor — antes de repassar as ações para outros usuários —, é uma boa prática que os desenvolvedores estudem a fundo o *design* do jogo para projetarem a jogabilidade de uma forma que a maior quantidade de lógica possível passe pelo servidor previamente.

Como descrito, muitos jogos, devido a sua dinamicidade e requisitos de latência, requerem que as checagens sejam feitas no próprio cliente. Entre esses jogos, a categoria que mais se sobressai é a de jogos de Tiro em Primeira Pessoa (FPS), como o popular jogo *Counter-Strike*, por exemplo. Por outro lado, há os jogos que não possuem essa exigência, como MOBAs (*Multiplayer online battle arena*). O exemplo mais popular dessa categoria é o jogo *League of Legends*, considerado um dos jogos mais populares atualmente.

Apesar dessa vantagem de muitas verificações poderem residir no servidor, ainda há a necessidade de implementar soluções no cliente, pois trapaças que visam automatizar a jogabilidade ainda podem ser feitas com relativa facilidade.

Este trabalho não discute acerca de proteções de servidor, porque elas dependem diretamente do *design* e da lógica do jogo. Dessa forma, torna-se relevante discorrer apenas a respeito de contramedidas no cliente.

Por restrições de tempo até a entrega deste documento, essas proteções são apresentadas e detalhadas na seção seguinte, cujo conteúdo descreve a modelagem e a implementação de um anti-cheat simples, que é o protótipo deste Projeto Final.

9. Modelagem e Implementação de um Anti-cheat simples

Sucintamente, a arquitetura do protótipo consiste na divisão em duas grandes partes: componentes que residem no servidor e outros no cliente. Para facilitar o desenvolvimento e os testes, simula-se a comunicação entre as duas partes, com serialização, sem utilizar pacotes de rede.

A parte do servidor é necessária neste trabalho porque se deseja exemplificar como realmente um anti-cheat funciona em jogos online. À parte do cliente incumbe, é claro, notificar o servidor a respeito de violações de medidas de segurança pelo jogador. Dessa forma, o componente de anti-cheat presente no servidor poderá repassar quaisquer notificações ao servidor do jogo, a fim de que tome providências, como suspensões de conta ou alertas.

Os módulos do cliente executam no mesmo sistema operacional que o jogo, prevenindo uso de trapaças e reagindo quando detecta alguma. Boa parte do esforço da implementação é dedicada a essa parte, visto sua importância.

Não há interface no protótipo. Por causa de restrições de tempo até a entrega deste Projeto Final, não foi possível implementar testes automatizados nem implementar interações via linha de comando, para que pudesse ser demonstrado na apresentação. Dessa maneira, o protótipo pode ser encarado como uma Prova de Conceito. Apenas alguns métodos e técnicas estão no código, que serão explicados nesta seção.

1. Estrutura e mecanismos basilares

O protótipo divide-se em duas partes: cliente e servidor. Ambos iniciam a comunicação com uma troca de chaves de Diffie–Hellman. Concluída a troca, pacotes subsequentes são cifrados com RC4, com a chave trocada (figura 1).

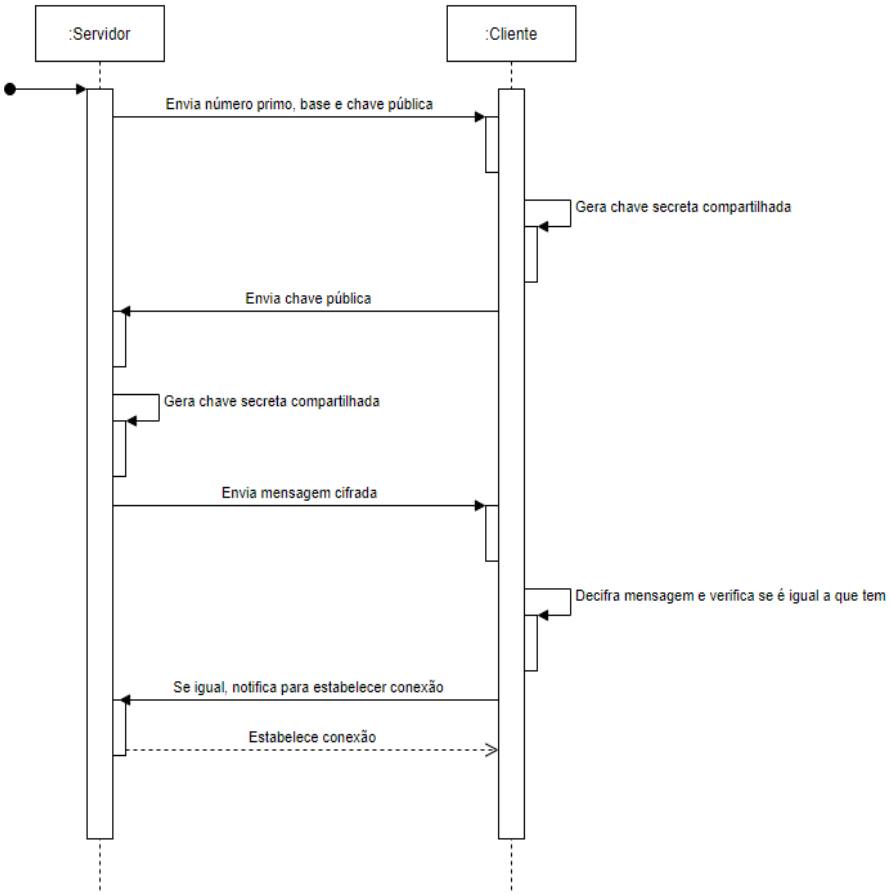

Figura 1 Iniciando comunicação, com troca de chaves de Diffie-Hellman.

Os pacotes entre o cliente e o servidor possuem confidencialidade, por meio de encriptação com RC4 se estabelecida a conexão, e verificação de integridade. Sublinha-se que a troca de chaves implementada é susceptível a ataques *man-in-the-middle*, violando o pilar da autenticidade. Dessa maneira, é necessário obter a chave pública a partir de certificados digitais, nas duas pontas, para não tornar possível esses ataques. Outra implementação pendente no protocolo é a proteção contra ataques de *replay*.

Há três partes, em formato JSON, que compõem um pacote: o cabeçalho (*header*), o conteúdo (*payload*) e a cauda (*tail*). O cabeçalho armazena um número (ID) que identifica o tipo de pacote e uma descrição do pacote, que é opcional. O conteúdo armazena dados acerca do pacote, que podem variar entre pacotes do mesmo tipo. A cauda armazena o *digest* (em MD5) do pacote inteiro, o que inclui o *header* e o *payload*. O MD5 é muito susceptível a colisões e a ataques de tabela *rainbow*, portanto uma opção

melhor é substitui-lo pelo algoritmo SHA, de *digest* com tamanho de 256 *bits* ou 512 *bits*.

Assim, feitas as inserções do *header* e do *payload*, coloca-se o *hash* no final, que leva em consideração todos os *bytes* desses dois elementos precedentes. Se a conexão está estabelecida, o pacote é codificado em Base64, visto que os três componentes agora estão cifrados com RC4.

Todo esse esquema está representado na figura 2.

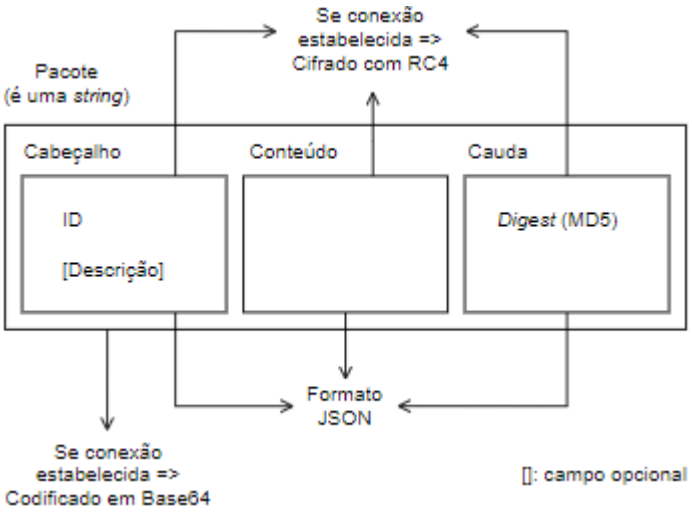

Figura 2 Estrutura de um pacote.

No protótipo, há um mecanismo de *heartbeat*, que é um pacote enviado pelo servidor para o cliente com o intuito de checar se a conexão ainda está ativa e, de certa forma, também verificar se o cliente está executando, protegendo contra suspensões ou desativações intencionais feitas por um adversário no cliente.

O pacote de *heartbeat* contém, em sua área de conteúdo, um campo que armazena uma *string* aleatória gerada pelo servidor. O cliente, ao receber o pacote, deve enviá-lo de volta ao servidor com essa mesma *string*. Se as *strings* forem diferentes, o servidor rejeita o pacote. O servidor envia o *heartbeat* periodicamente e o cliente tem um número limitado de chances (tentativas) para enviar o pacote de volta. Se o pacote não for devolvido dentro do número máximo de tentativas, incluindo rejeições, o servidor desfaz sua conexão com o cliente.

Um panorama desse funcionamento está representado na figura 3.

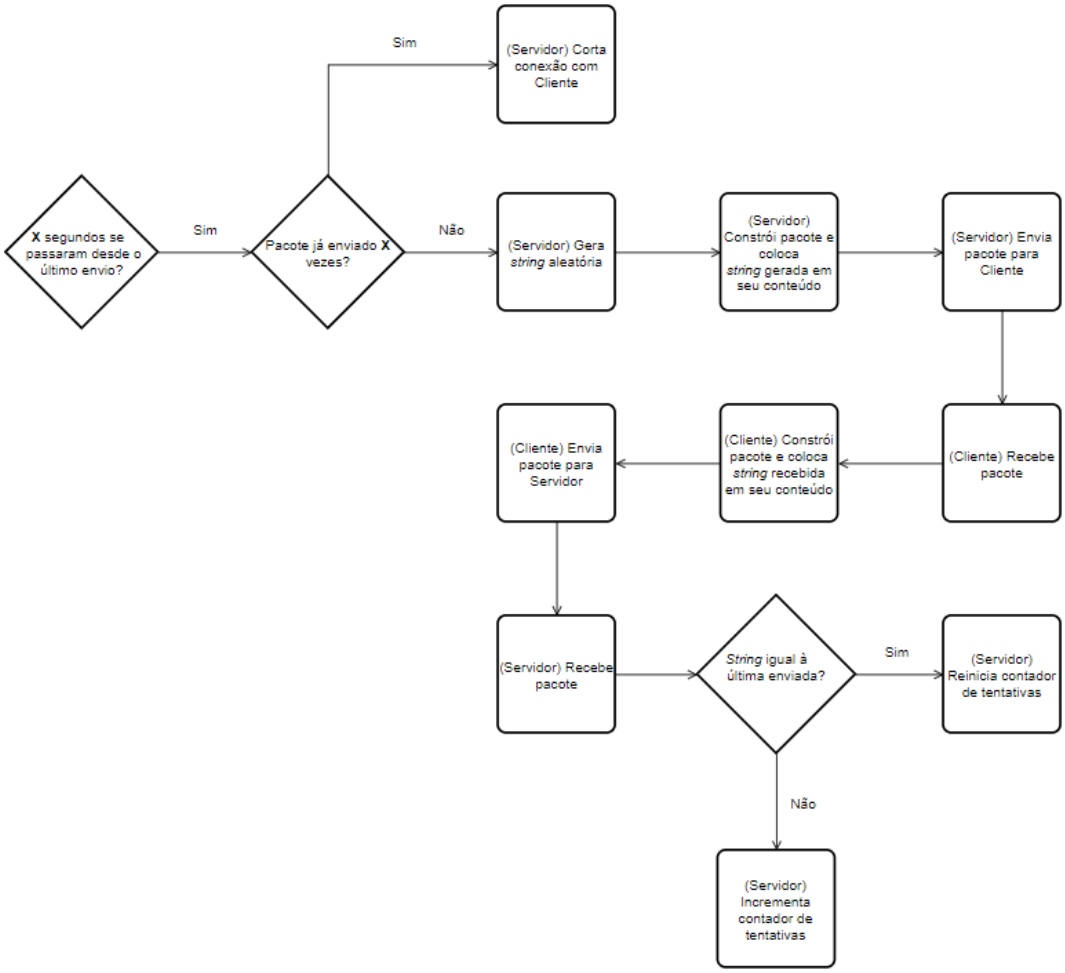

Figura 3 Mecanismo de *heartbeat*.

2. Contramedidas

Devido a restrições de tempo até a entrega deste trabalho, apenas foi possível implementar três proteções de integridade.

A primeira proteção, chamada Verificador de Memória, checa se a imagem do executável carregada na memória é igual à do disco.

A segunda proteção, Protetor de Memória, empacota valores e variáveis para blindá-los contra escaneamentos de memória e alterações estranhas à lógica do jogo.

A terceira proteção, Verificador de Arquivos, compara *hashes* de arquivos selecionados com seus *hashes* originais, para verificar se há modificação não autorizada.

O funcionamento de cada uma dessas proteções será detalhado adiante.

2.1. Verificador de Memória

Esta medida de segurança, quando o sistema de anti-cheat é iniciado, salva o *digest* a seção mapeada na memória relativa à área do executável que armazena o código da aplicação — em código de máquina —, determinando sua execução, a exemplo da lógica do jogo e das suas funções.

No Windows, o formato do executável é dado pelo *Portable Executable* (PE) *format* (figura 4), que também é o formato de DLLs. Esse formato nada mais é que uma estrutura de dados que armazena todos os dados necessários para que o carregador de executáveis do Windows carregue o executável na memória e o execute. Esse formato abrange referências a DLLs, carregadas concomitantemente à aplicação, de modo que funções externas utilizadas pelo programa, que se localizam na tabela de importação (*import table*) dessa estrutura, sejam *linkadas* em tempo de execução (as entradas da tabela são preenchidas com o endereço correspondente à função da entrada, obtido após o carregamento da respectiva DLL). Ainda, essa estrutura também armazena outros recursos, como ícones do programa, e a área de dados locais de *threads* (*thread-local storage* — TLS) [1].

De acordo com o *layout* da estrutura, o formato inclui diversos *headers* e seções que informam ao carregador do Windows como fazer o mapeamento para a memória. Um executável possui várias regiões, para diferentes finalidades, que exigem que o início de cada seção esteja alinhado com a próxima página de memória virtual. Parte do trabalho do carregador é fazer esse alinhamento, já que, para poupar espaço, as seções não estão alinhadas no disco. Cada seção tem atributos diferentes no que tange à proteção de memória [1].

Para o Verificador de Memória, a seção mais importante a ser avaliada é apelidada *.text*, cuja proteção de memória é de apenas execução e leitura. Dessa forma, trapaças precisam, primeiramente, alterar essa proteção para suportar escrita previamente às modificações, conforme já discutido em outra seção deste documento. Outra seção notável do formato é a *.rdata*, que armazena valores de variáveis globais utilizados no código do sistema [1].

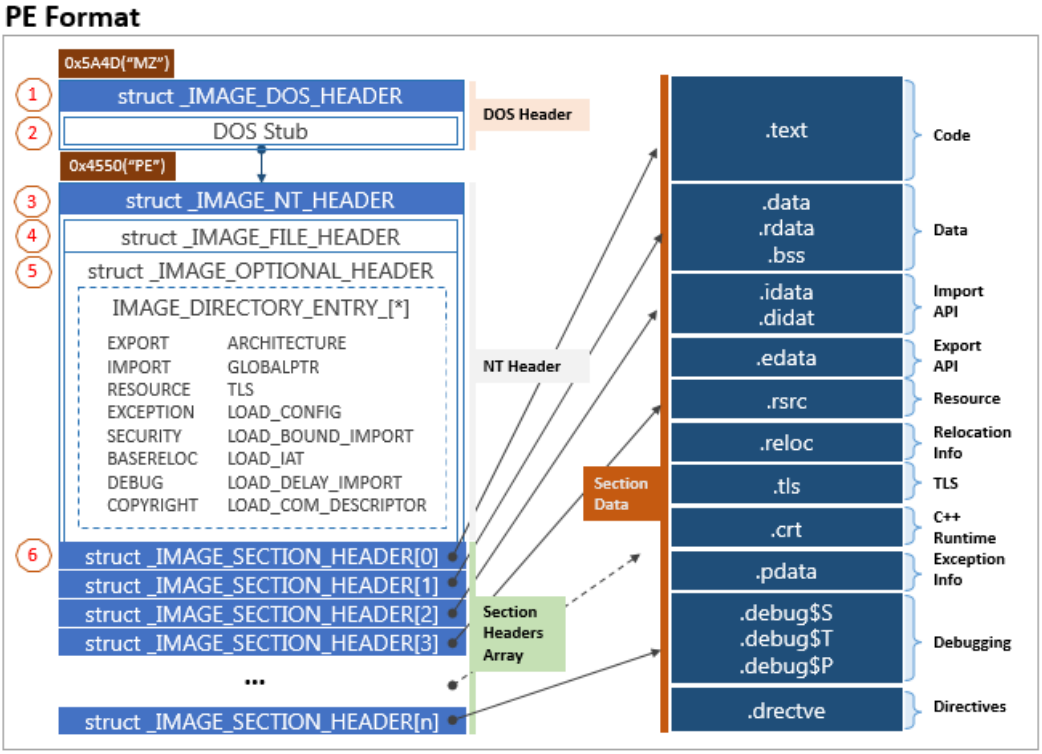

Figura 4 Visão simplificada do formato PE [2].

Visto isso, o Verificador de Memória, no momento em que o processo que está sendo protegido é iniciado, faz uso de APIs do Windows para obter informações da seção *.text*, como seu endereço inicial (endereço base ou *base address*) e seu tamanho. Adquiridas essas informações, calcula-se e armazena-se o *digest* com MD5.

Dessa forma, a verificação deste módulo se resume em comparar o *digest* da seção no momento da checagem com o *digest* armazenado, que é o original. Se os *digests* não forem iguais, há quebra de integridade.

2.2. Protetor de Memória

Como já escrito no trabalho, escaneadores de memória funcionam restringindo o espaço de endereçamento do programa à procura do endereço que contém o valor desejado.

Esta medida de segurança, baseada no artigo da empresa *Riot Games* [3], tem dois usos principais: proteger variáveis dinâmicas, que mudam ao longo da execução do sistema, e variáveis estáticas, ou constantes, que são imutáveis. Outra maneira de proteger variáveis estáticas é com o Verificador

de Memória, incidindo sobre a seção *.rdata*. Essa outra forma é mais complexa de fazer funcionar porque a seção *.rdata* pode conter dados mutáveis, principalmente em seu início; assim sendo, haveria a necessidade de configurar o Verificador para ignorar os *bytes* mutáveis, levando em consideração, para fins do *digest*, apenas a parte imutável.

A ideia por trás do Protetor é embalar variáveis. No protótipo, codificado na linguagem C++, o Protetor utiliza a funcionalidade de *template* da linguagem para ser uma classe genérica, que pode ser instanciada a partir de qualquer tipo de variável.

Quando se instancia a classe para proteger uma variável, o módulo aloca uma variável no *heap* e esconde o endereço retornado com manipulação de ponteiros (trecho de código 1).

```
MemoryProtector() {
        m_pData = GetEncodedPtr(new T);
}
~MemoryProtector() {
        delete GetDecodedPtr(m_pData);
}

...

PBYTE GetEncodedPtr(T* pDecodedPtr) {
        return PBYTE(pDecodedPtr) + PtrToUlong(this);
}
T* GetDecodedPtr(PBYTE pEncodedPtr) {
        return (T*)(pEncodedPtr - PtrToUlong(this));
}
```

Trecho de código 1 Mascarando ponteiros.

Feita essa inicialização, o chamador pode invocar o método *VerifyCRCAndSetData*, que atua tanto para inicializar quanto para modificar futuramente a variável. Esse método, além de alterar o conteúdo da variável de maneira segura e reconhecida, realiza, se existente um CRC, a comparação do CRC da variável nesse momento com o CRC armazenado internamente, detectando alterações indevidas. Em relação ao CRC, o método, após a modificação, calcula e armazena o novo CRC da variável, levando em conta todos os *bytes* na memória que a representam. Esse método e seus auxiliares estão ilustrados no trecho de código 2.

```
void BuildCRC() {
        using namespace CryptoPP;
```

```
                auto pData = (BYTE*)GetDecodedPtr(m_pData);
                g_CRC32.Update((const byte*)pData, sizeof(T));
                m_strGoodCRC.resize(g_CRC32.DigestSize());
                g_CRC32.Final((byte*)&m_strGoodCRC[0]);
        }


        void SetData(const T &data) {
                GetData() = data;
        }

        void VerifyCRC() {
                using namespace CryptoPP;

                auto pData = (BYTE*)GetDecodedPtr(m_pData);
                g_CRC32.Update((const byte*)pData, sizeof(T));

        if (!g_CRC32.Verify((const byte*)m_strGoodCRC.data())) {
                        std::cout << "VARIABLE
CORRUPTED!" << std::endl;
                }
        }

        void VerifyCRCAndSetData(const T &data) {
                if (!m_strGoodCRC.empty()) VerifyCRC();
                SetData(data);
                BuildCRC();
        }
```

Trecho de código 2 Inicializando e alterando uma variável protegida.

Os métodos expostos previamente garantem a verificação de integridade mencionada no início da discussão. Agora, para que não seja possível localizar o valor em escaneamentos de memória, o método *MoveHeapLoc* (trecho de código 3) tem a tarefa de mover o valor protegido para uma nova região de memória.

Dessa forma, quando um escaneador estiver restringindo seu escopo de busca, o endereço antes presente em sua lista de possíveis endereços agora não será mais válido; ainda, o escaneador também não terá nessa lista o valor na nova região, já que ele não estava previamente. Para que aparecesse, seria necessário reconstruir a lista de possíveis endereços do valor desejado, o que não é possível fazer, porque vai contra à redução de escopo descrita.

```
        void MoveHeapLoc() {
                T* p = GetDecodedPtr(m_pData);
                m_pData = GetEncodedPtr(new T);
                GetData() = *p;
                delete p;
        }
```

```
        void VerifyCRCAndMoveHeapLoc() {
                if (!m_strGoodCRC.empty()) VerifyCRC();
                MoveHeapLoc();
        }
```

Trecho de código 3 Movendo valor para nova região de memória.

2.3. Verificador de Arquivos

O Verificador de Arquivos é um componente cujo objetivo é a checagem da integridade de arquivos predefinidos.

Primeiramente, a fim de indicar ao Verificador o que recai em sua verificação, faz-se necessário construir um arquivo que contém a lista de arquivos que devem sofrer a verificação, bem como seus *digests* originais. No protótipo, para facilitar o desenvolvimento, esse módulo de construção pode ser ativado com o argumento de linha de comando *--buildcrc*.

O módulo é configurável — opcionalmente —, a partir de um arquivo nomeado *opt_crcbuilder.json*, para ignorar diretórios e certas extensões de arquivos. Um exemplo de configuração está no trecho de código 4.

```
{
  "ignore": {
    "ext": [
      "json",
      "bat",
    ],
    "dir": [
      "some_directory"
    ]
  }
}
```

Trecho de código 4 Configurando o módulo de construção do Verificador.

Executado o módulo de construção, é gerado o arquivo *server_files_hash.json*, que armazena o caminho relativo e o *digest* de cada arquivo sob proteção. Para gerar, o sistema percorre recursivamente a pasta raiz, onde se encontra o executável do protótipo, visando incluir todos os

arquivos, inclusive de quaisquer subdiretórios existentes, ignorando aquilo que foi definido no arquivo de configuração. Esse arquivo criado pertence somente ao servidor, não cabendo, portanto, colocá-lo junto a outros arquivos do cliente.

Com o arquivo gerado, agora é possível fazer as comparações de integridade. Assim, quando a requisição de integridade de arquivos é feita pelo servidor, o cliente coleta os *digest*s dos arquivos definidos e os envia ao servidor. Caso algum *digest* não seja igual, a verificação falha e o servidor alerta sobre qual arquivo está diferente.

3. Referências

[1] Microsoft. *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. Acessado de https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN.

[2] *Kevin's Attic for Security Research*. *PE File Format*. Acessado de https://dandylife.net/blog/archives/388.

[3] *Riot Games*. *Riot's Approach to Anti-cheat*. Acessado de https://technology.riotgames.com/news/riots-approach-anti-cheat.

10. Conclusões

Neste projeto, houve a discussão acerca de trapaças e seus impactos em jogos, bem como a apresentação de alguns métodos e técnicas para tornar um software ou um jogo mais resiliente a ataques.

A maior contribuição do trabalho foi sistematizar alguns conhecimentos de proteção de software e métodos e técnicas que trapaças utilizam para atingir seus objetivos. A literatura sobre anti-cheats e trapaças é bem escassa, o que fez com que a pesquisa entrasse em campos relacionados para obter a fundamentação necessária, como Segurança da Informação, de forma ampla, e, em especial, sua subárea de programas maliciosos.

Uma trapaça, conforme vista ao longo do trabalho, é qualquer vantagem não acessível a outros jogadores, podendo assumir diferentes facetas. Atenção foi dada à sua faceta de manipulação de executável, já que é a mais recorrente e mais perceptível pelos jogadores em jogos. Essa forma requer proteções em diversos níveis do executável, necessitando do desenvolvedor competências múltiplas no que tange a sistemas operacionais e código de baixo nível, para compreender aquilo que deve ser feito para tornar a vida de um adversário mais difícil.

De fato, proteção de software nunca será completamente eficaz, já que o código do software, suas dependências e sua execução estão na máquina do jogador. Assim sendo, o dono da máquina tem a vantagem e possui uma ampla gama de ferramentas para auxiliá-lo na jornada de desmistificar o funcionamento do software e, assim, manipulá-lo a seu bel-prazer. Anti-cheats e proteção de software, então, apenas sobem a barra frente a um atacante, dificultando sua atuação, mas nunca protegem inteiramente ao ponto de bloquear todos os possível ataques.

A proteção mais eficaz, como descrito na pesquisa, é colocar o máximo de lógica do jogo no servidor, para que as verificações necessárias sejam feitas ali — sem que seja possível a interferência de um atacante e antes de as ações que sofrem as verificações refletirem em outros jogadores. Jogos que não possuem o requisito de alta responsividade e latência podem ser facilmente projetados e implementados pelos desenvolvedores para que a lógica de ações críticas passem pelo servidor previamente. Por outro lado, jogos de alta responsividade são mais díficeis de serem implementados dessa maneira, o que exige dos desenvolvedores outra abordagem para

dificultar ataques. Em ambos os casos, contudo, um anti-cheat no cliente é necessário, porque adversários ainda conseguem, com facilidade, construir trapaças que automatizam a jogabilidade, sem nem necessariamente interagir com o processo do jogo, apenas obtendo as informações testando a cor de pixels da tela.

Entre as proteções genéricas mais importantes no cliente, destacam-se ofuscação e técnicas de *anti-debug*. São relevantes porque dificultam o uso de depuradores, ferramentas essenciais para analisar o software, e também, no caso da ofuscação, tornam a análise do código bem mais difícil, o que cria uma barreira para adversários menos experientes.

Existem várias outras proteções importantes quando o tema é anti-cheat, como verificação e prevenção de inputs simulados e verificações de integridade em dependências do jogo, como bibliotecas gráficas. Por restrições de tempo, essas contramedidas não foram implementadas no protótipo, embora as que foram implementadas são tão importantes quanto e estão presentes em todos os anti-cheats atuais de grande relevância, especialmente a técnica de proteção de memória, que protege constantes e variáveis e faz a moção de seus endereços no *heap*.

Em Projeto Final I, tive a oportunidade de ampliar muito meus conhecimentos a respeito de Segurança da Informação, perpassando áreas essenciais, como controle de acesso, criptografia, comunicação segura e ataques comuns.

Além disso, estudei vários artigos acadêmicos e livros relacionados a trapaças e anti-cheats, obtendo, dessa forma, ampla visão sobre dificuldades enfrentadas por anti-cheats e principais métodos, técnicas e ferramentas empregados por trapaças e anti-cheats.

Indiretamente, também aprendi conceitos e mecanismos específicos do sistema operacional Windows, visto que os métodos e as técnicas se fundamentam neles.

O protótipo implementado deve ser visto como uma Prova de Conceito e foi feito para executar no Windows, outra razão que justifica um estudo mais abrangente desse sistema operacional.

Este Projeto Final tem características fortes de pesquisa e, por causa disso, não estão sendo apresentadas uma modelagem e uma lista de requisitos (funcionais e não funcionais) detalhadas. A pesquisa tomou a maior parte do tempo do Projeto Final.

Anexo I — Códigos de Métodos e Técnicas de Trapaças

1. Comparando memória se baseando em padrões, para encontrar endereços de memória após atualizações [1]

```
bool _f_memcmp(const char *in, const char *pat, int len) {
        for (int i = 0; i < len; i++) {
                if (*pat == '*') {
                }
                else if (*pat != *in) {
                        return false;
                }
                pat++;
                in++;
        }
        return true;
}

DWORD ScanForBytes(const char *haystack,
        DWORD haystack_size, const char *needle,
        DWORD needle_size) {
        const char *curr = haystack;
        assert(haystack_size >= needle_size);
        while (curr <= (haystack + haystack_size)) {
                if (*curr == *needle) {

        if (true == _f_memcmp(curr, needle, needle_size))
{

        DWORD offset = curr - haystack;
                                return(offset);
                        }
                }
                curr++;
        }
        return -1;
}

DWORD FindOffset(char *theName) {
        if (!strcmp(theName, "NetClient::ProcessMessage"))
 {
                /*
        .text:00514630 arg_0 = dword ptr 8.text:00514630 a
rg_4 = dword ptr 0Ch
                .text:00514630
                .text:00514630 push ebp
                .text:00514631 mov ebp, esp
                .text:00514633 mov edx, _aCounter ****
                .text:00514639 push ebx
                .text:0051463A mov ebx, [ebp+arg_4]
                .text:0051463D push esi
                .text:0051463E push edi
                .text:0051463F lea eax, [ebp+arg_4+2]
                .text:00514642 mov esi, ecx
```

```
                .text:00514644 inc edx
                .text:00514645 push eax

        55 8B EC 8B 15 * * * * 53 8B 5D 0C 56 57 8D 45 0E
8B F1
                */

        char s[] = { 0x55, 0x8B, 0xEC, 0x8B, 0x15, '*', '*
',

        '*', '*', 0x53, 0x8B, 0x5D, 0x0C, 0x56, 0x57, 0x8D
, 0x45, 0x0E,
                        0x8B, 0xF1 };

        int offset = ScanForBytes(g_binBuf, g_binBufSize,
                        s, sizeof(s));
                if (offset != -1) return offset;
        }
}
```

2. Obtendo identificadores de um processo

2.1. Obtendo o PID do processo a partir do nome da sua janela [2]

```
HWND myWindow =
FindWindow(NULL, "Title of the game window here");

DWORD PID;
GetWindowThreadProcessId(myWindow, &PID);
```

2.2. Obtendo o PID do processo sem o nome da sua janela [2]

```
PROCESSENTRY32 entry;
entry.dwSize = sizeof(PROCESSENTRY32);

HANDLE snapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);
if (Process32First(snapshot, &entry) == TRUE) {
        while (Process32Next(snapshot, &entry) == TRUE) {
                wstring binPath = entry.szExeFile;

        if (binPath.find(L"game.exe") != wstring::npos) {

        printf("game pid is %d\n", entry.th32ProcessID);
                        break;
                }
        }
}
CloseHandle(snapshot);
```

2.3. Obtendo uma *handle* do processo [2]

```
DWORD PID = getGamePID();
HANDLE process = OpenProcess(
        PROCESS_VM_OPERATION |
        PROCESS_VM_READ |
        PROCESS_VM_WRITE,
        FALSE,
        PID
);
if (process == INVALID_HANDLE_VALUE) {
        printf("Failed to open PID %d, error code %d",
                PID, GetLastError());
}
```

3. Encontrando o endereço base de um processo [2]

```
DWORD newBase;

HMODULE k32 = GetModuleHandle("kernel32.dll");

LPVOID funcAdr = GetProcAddress(k32, "GetModuleHandleA");
if (!funcAdr)
funcAdr = GetProcAddress(k32, "GetModuleHandleW");

HANDLE thread =
CreateRemoteThread(process, NULL, NULL,
(LPTHREAD_START_ROUTINE)funcAdr,
NULL, NULL, NULL);

WaitForSingleObject(thread, INFINITE);

GetExitCodeThread(thread, &newBase);

CloseHandle(thread);
```

4. Lendo e escrevendo memória de um processo

4.1. Alterando a proteção de determinada região de memória do processo [2]

```
template<typename T>
DWORD protectMemory(HANDLE proc, LPVOID adr, DWORD prot) {
        DWORD oldProt;
        VirtualProtectEx(proc, adr, sizeof(T), prot, &oldP
rot);
        return oldProt;
}
```

4.2. Acessando memória do processo [2]

```
template<typename T>
T readMemory(HANDLE proc, LPVOID adr) {
        T val;
        ReadProcessMemory(proc, adr, &val, sizeof(T), NULL
);
        return val;
}

DWORD value = readMemory<DWORD>(proc, adr);
```

4.3. Modificando memória do processo [2]

```
template<typename T>
void writeMemory(HANDLE proc, LPVOID adr, T val) {
        WriteProcessMemory(proc, adr, &val, sizeof(T), NUL
L);
}

DWORD oldProt =
protectMemory<DWORD>(process, address, PAGE_READWRITE);
writeMemory<DWORD>(process, address, newValue);
protectMemory<DWORD>(process, address, oldProt);
```

5. Injetando bibliotecas dinâmicas (DLLs) em um processo [2]

```
wchar_t* dllName = "c:\\something.dll";
int namelen = wcslen(dllName) + 1;
LPVOID remoteString =
VirtualAllocEx(process, NULL, namelen * 2, MEM_COMMIT, PAG
E_EXECUTE);
WriteProcessMemory(process, remoteString, dllName, namelen
 * 2, NULL);

HMODULE k32 = GetModuleHandleA("kernel32.dll");
LPVOID funcAdr = GetProcAddress(k32, "LoadLibraryW");
HANDLE thread =
CreateRemoteThread(process, NULL, NULL,
(LPTHREAD_START_ROUTINE)funcAdr,
remoteString, NULL, NULL);

WaitForSingleObject(thread, INFINITE);
CloseHandle(thread);
```

6. Controlando o fluxo de execução de um processo

6.1. Removendo código indesejado com NOPs [2]

```
template<int SIZE>
void writeNop(DWORD address) {
        auto oldProtection =
        protectMemory<BYTE[SIZE]>(address, PAGE_EXECUTE_RE
ADWRITE);

        for (int i = 0; i < SIZE; i++)
                writeMemory<BYTE>(address + i, 0x90);

        protectMemory<BYTE[SIZE]>(address, oldProtection);
}

writeNop<2>(0xDEADBEEF);
```

7. Referências

[1] Greg Hoglund e Gary McDraw. 2007. *Exploiting Online Games: Cheating Massively Distributed Systems*. *Addison Wesley Professional*.

[2] Nick Cano. 2016. *Game Hacking: Developing Autonomous Bots for Online Games*. *No Starch Press*.

Anexo II — Códigos de Métodos e Técnicas de Anti-cheats

1. Verificando se uma *thread* está suspensa

```
bool IsThreadSuspended(DWORD dwTID) {
        bool bIsSuspended = false;

        HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FAL
SE, dwTID);
        if (hThread == NULL) {
                bIsSuspended = true;
                return bIsSuspended;
        }

        if (SuspendThread(hThread) > 0)
                bIsSuspended = true;

        ResumeThread(hThread);
        CloseHandle(hThread);

        return bIsSuspended;
}
```

2. *Hooks*

2.1. Escaneando *hooks* no IAT

```
typedef struct ModAPIsInfo {
        LPCSTR lpModName;
        char* pszModAPIs;
};

bool AreAPIsInsideModule(HMODULE hMod, char* pszAPIs) {
        HANDLE hProcess = GetCurrentProcess();
        MODULEINFO hModInfo;
        if (GetModuleInformation(hProcess, hMod, &hModInfo
, sizeof(MODULEINFO)) != TRUE)
                return false;

        LPVOID pModLow = hModInfo.lpBaseOfDll;
        auto pModHigh = (LPVOID*)((unsigned char*)pModLow
+ hModInfo.SizeOfImage);

        char* pszCurrAPI = NULL;
        char* pszNextAPI = NULL;
        pszCurrAPI = strtok_s(pszAPIs, " ", &pszNextAPI);

        bool bInsideModule = false;
```

```
        while (pszCurrAPI != NULL) {

        LPVOID pAPIAddr = GetProcAddress(hMod, pszCurrAPI)
;
                if (pAPIAddr != NULL &&

        (pAPIAddr < pModLow || pAPIAddr > pModHigh)) {
                        bInsideModule = true;
                        break;
                }


        pszCurrAPI = strtok_s(NULL, " ", &pszNextAPI);
        }

        return bInsideModule;
}

HMODULE GetModHandle(LPCSTR lpModName) {
        HMODULE hMod = GetModuleHandle(lpModName);
        if (hMod == NULL)
                hMod = LoadLibrary(lpModName);

        return hMod;
}

bool IsIATHookPresent(const std::vector<ModAPIsInfo> &vMod
APIsInfos) {
        bool bIsIATHookPresent = false;

        for (auto modAPIsInfo : vModAPIsInfos) {

        HMODULE hMod = GetModHandle(modAPIsInfo.lpModName)
;

        bIsIATHookPresent = AreAPIsInsideModule(hMod, modA
PIsInfo.pszModAPIs);
                if (bIsIATHookPresent) break;
        }

        return bIsIATHookPresent;
}

std::vector<ModAPIsInfo> vModAPIsInfos = {
        { "kernel32.dll", pszKernel32APIs },
        { "ntdll.dll", pszNTDLLAPIs },
        { "user32.dll", pszUser32APIs }
};
IsIATHookPresent(vModAPIsInfos);
```

3. Enumerações

3.1. Procurando janelas julgadas como maliciosas

```
bool IsWindowPresent(const std::vector<std::string> &vMali
ciousWindowNames) {
        int iNumWindows = 0;
```

```
        for (HWND hWnd = GetTopWindow(NULL);
                hWnd != NULL;
                hWnd = GetWindow(hWnd, GW_HWNDNEXT)) {

                DWORD dwWindowPID = 0L;

        GetWindowThreadProcessId(hWnd, &dwWindowPID);


        DWORD dwCurrProcessID = GetCurrentProcessId();
                if (dwCurrProcessID == dwWindowPID)
        continue;

                const int iMaxWindowNameLen = 128;
                char szWindowName[iMaxWindowNameLen];

        GetWindowText(hWnd, szWindowName, iMaxWindowNameLe
n);

                std::string strWindowName = szWindowName;
                for (const auto
&strMaliciousWindowName : vMaliciousWindowNames) {

        if (strWindowName.find(strMaliciousWindowName) !=
std::string::npos) {
                                return true;
                        }
                }

                ++iNumWindows;

        }

        if (iNumWindows <= 1) {
                // Something is hooked. Take action.
                return true;
        }

        return false;
}
```

4. Referências

[1] Microsoft. *Programming reference for the Win32 API*. Acessado de https://docs.microsoft.com/en-us/windows/win32/api/.