Engenharia Reversa de **Código Malicioso**

Guilherme Venere

Engenharia Reversa de **Código Malicioso**

Guilherme Venere

Engenharia Reversa de **Código Malicioso**

Guilherme Venere

Rio de Janeiro
Escola Superior de Redes
2013

Copyright © 2013 – Rede Nacional de Ensino e Pesquisa – RNP
Rua Lauro Müller, 116 sala 1103
22290-906 Rio de Janeiro, RJ

Diretor Geral
Nelson Simões

Diretor de Serviços e Soluções
José Luiz Ribeiro Filho

Escola Superior de Redes

Coordenação
Luiz Coelho

Edição
Pedro Sangirardi

Revisão
Lincoln da Mata

Revisão Técnica
Thiago Oliveira Marques

Coordenação Acadêmica de Segurança e Governança de TI
Edson Kowask Bezerra

Equipe ESR (em ordem alfabética)
Celia Maciel, Cristiane Oliveira, Derlinéa Miranda, Elimária Barbosa, Evellyn Feitosa, Felipe Nascimento, Lourdes Soncin, Luciana Batista, Luiz Carlos Lobato, Renato Duarte e Yve Abel Marcial.

Capa, projeto visual e diagramação
Tecnodesign

Versão
1.0.1

Este material didático foi elaborado com fins educacionais. Solicitamos que qualquer erro encontrado ou dúvida com relação ao material ou seu uso seja enviado para a equipe de elaboração de conteúdo da Escola Superior de Redes, no e-mail info@esr.rnp.br. A Rede Nacional de Ensino e Pesquisa e os autores não assumem qualquer responsabilidade por eventuais danos ou perdas, a pessoas ou bens, originados do uso deste material.
As marcas registradas mencionadas neste material pertencem aos respectivos titulares.

Distribuição
Escola Superior de Redes
Rua Lauro Müller, 116 – sala 1103
22290-906 Rio de Janeiro, RJ
http://esr.rnp.br
info@esr.rnp.br

Dados Internacionais de Catalogação na Publicação (CIP)

V454e Venere, Guilherme
Engenharia reversa de código malicioso / Guilherme Venere. – 1. ed. aum. – Rio de Janeiro: RNP/ESR, 2013.
128 p. : il. ; 28 cm.

Bibliografia: p. 127.
ISBN 978-85-63630-26-1

1. Engenharia de software. 2. Assembly (Linguagem de programação). 3. Segurança de software. 4. IDA Pro (software). 5. OllyDBG (software). I. Título.

CDD 005.12

Sumário

Escola Superior de Redes

A Escola Superior de Redes (ESR) é a unidade da Rede Nacional de Ensino e Pesquisa (RNP) responsável pela disseminação do conhecimento em Tecnologias da Informação e Comunicação (TIC). A ESR nasce com a proposta de ser a formadora e disseminadora de competências em TIC para o corpo técnico-administrativo das universidades federais, escolas técnicas e unidades federais de pesquisa. Sua missão fundamental é realizar a capacitação técnica do corpo funcional das organizações usuárias da RNP, para o exercício de competências aplicáveis ao uso eficaz e eficiente das TIC.

A ESR oferece dezenas de cursos distribuídos nas áreas temáticas: Administração e Projeto de Redes, Administração de Sistemas, Segurança, Mídias de Suporte à Colaboração Digital e Governança de TI.

A ESR também participa de diversos projetos de interesse público, como a elaboração e execução de planos de capacitação para formação de multiplicadores para projetos educacionais como: formação no uso da conferência web para a Universidade Aberta do Brasil (UAB), formação do suporte técnico de laboratórios do Proinfo e criação de um conjunto de cartilhas sobre redes sem fio para o programa Um Computador por Aluno (UCA).

A metodologia da ESR

A filosofia pedagógica e a metodologia que orientam os cursos da ESR são baseadas na aprendizagem como construção do conhecimento por meio da resolução de problemas típicos da realidade do profissional em formação. Os resultados obtidos nos cursos de natureza teórico-prática são otimizados, pois o instrutor, auxiliado pelo material didático, atua não apenas como expositor de conceitos e informações, mas principalmente como orientador do aluno na execução de atividades contextualizadas nas situações do cotidiano profissional.

A aprendizagem é entendida como a resposta do aluno ao desafio de situações-problema semelhantes às encontradas na prática profissional, que são superadas por meio de análise, síntese, julgamento, pensamento crítico e construção de hipóteses para a resolução do problema, em abordagem orientada ao desenvolvimento de competências.

Dessa forma, o instrutor tem participação ativa e dialógica como orientador do aluno para as atividades em laboratório. Até mesmo a apresentação da teoria no início da sessão de aprendizagem não é considerada uma simples exposição de conceitos e informações. O instrutor busca incentivar a participação dos alunos continuamente.

As sessões de aprendizagem onde se dão a apresentação dos conteúdos e a realização das atividades práticas têm formato presencial e essencialmente prático, utilizando técnicas de estudo dirigido individual, trabalho em equipe e práticas orientadas para o contexto de atuação do futuro especialista que se pretende formar.

As sessões de aprendizagem desenvolvem-se em três etapas, com predominância de tempo para as atividades práticas, conforme descrição a seguir:

Primeira etapa: apresentação da teoria e esclarecimento de dúvidas (de 60 a 90 minutos).
O instrutor apresenta, de maneira sintética, os conceitos teóricos correspondentes ao tema da sessão de aprendizagem, com auxílio de slides em formato PowerPoint. O instrutor levanta questões sobre o conteúdo dos slides em vez de apenas apresentá-los, convidando a turma à reflexão e participação. Isso evita que as apresentações sejam monótonas e que o aluno se coloque em posição de passividade, o que reduziria a aprendizagem.

Segunda etapa: atividades práticas de aprendizagem (de 120 a 150 minutos).
Esta etapa é a essência dos cursos da ESR. A maioria das atividades dos cursos é assíncrona e realizada em duplas de alunos, que acompanham o ritmo do roteiro de atividades proposto no livro de apoio. Instrutor e monitor circulam entre as duplas para solucionar dúvidas e oferecer explicações complementares.

Terceira etapa: discussão das atividades realizadas (30 minutos).
O instrutor comenta cada atividade, apresentando uma das soluções possíveis para resolvê-la, devendo ater-se àquelas que geram maior dificuldade e polêmica. Os alunos são convidados a comentar as soluções encontradas e o instrutor retoma tópicos que tenham gerado dúvidas, estimulando a participação dos alunos. O instrutor sempre estimula os alunos a encontrarem soluções alternativas às sugeridas por ele e pelos colegas e, caso existam, a comentá-las.

Sobre o curso

O curso apresenta técnicas de análise de malware para apoiar a investigação forense digital e a resposta a incidentes envolvendo programas mal-intencionados. O objetivo é fornecer aos administradores de TI habilidades práticas para a análise destes programas. São abordados os conceitos, procedimentos, práticas e ferramentas para a análise de um código malicioso, com o uso de uma ferramenta para a realização das atividades práticas, que consolidam o conhecimento teórico. São apresentados os comandos básicos de Assembly para que o aluno faça a engenharia reversa de worms que afetaram milhares de computadores. O aluno aprenderá as melhores práticas antiengenharia reversa, desenvolvendo competências para a criação de defesas mais eficazes contra códigos maliciosos.

A quem se destina

Curso destinado a profissionais com sólido conhecimento em segurança de redes e sistemas e análise forense, interessados em desenvolver competências na área de análise de artefatos maliciosos e engenharia reversa. Profissionais de outras áreas que possuam conhecimento compatível com os requisitos básicos também poderão participar.

Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico
Indica nomes de arquivos e referências bibliográficas relacionadas ao longo do texto.

```
Largura constante
```

Indica comandos e suas opções, variáveis e atributos, conteúdo de arquivos e resultado da saída de comandos. Comandos que serão digitados pelo usuário são grifados em negrito e possuem o prefixo do ambiente em uso (no Linux é normalmente # ou $, enquanto no Windows é C:\).

Conteúdo de slide 
Indica o conteúdo dos slides referentes ao curso apresentados em sala de aula.

Símbolo 
Indica referência complementar disponível em site ou página na internet.

Símbolo 
Indica um documento como referência complementar.

Símbolo 
Indica um vídeo como referência complementar.

Símbolo 
Indica um arquivo de aúdio como referência complementar.

Símbolo 
Indica um aviso ou precaução a ser considerada.

Símbolo 
Indica questionamentos que estimulam a reflexão ou apresenta conteúdo de apoio ao entendimento do tema em questão.

Símbolo 
Indica notas e informações complementares como dicas, sugestões de leitura adicional ou mesmo uma observação.

Permissões de uso

Todos os direitos reservados à RNP.
Agradecemos sempre citar esta fonte quando incluir parte deste livro em outra obra.
Exemplo de citação: VENERE, Guilherme; MARQUES, Thiago Oliveira. *Engenharia reversa de código malicioso*. Rio de Janeiro: Escola Superior de Redes, RNP, 2013.

Comentários e perguntas

Para enviar comentários e perguntas sobre esta publicação:
Escola Superior de Redes RNP
Endereço: Av. Lauro Müller 116 sala 1103 – Botafogo
Rio de Janeiro – RJ – 22290-906
E-mail: info@esr.rnp.br

Sobre os autores

Guilherme Venere é Team Leader da equipe de Análise de Malware do McAfee Lab e atua como analista de malware pelos últimos 4 anos, trabalhando com análise avançada de malware a partir do escritório de pesquisa do McAfee Labs no Chile, criando detecções para novas ameaças a clientes McAfee. Tem 7 anos de experiência em segurança de redes e tratamento de incidentes, trabalhando para o backbone acadêmico Brasileiro. É bacharel em Ciência da Computação pela Universidade Federal de São Carlos e certificado em análise forense pelo Instituto SANS.

Thiago Oliveira Marques é formado em Ciências da Computação pelo Centro Universitário de Barra Mansa. Trabalha com engenharia reversa desde 2007, desenvolvendo proteções e ministrando treinamentos e palestras relacionadas à análise de malware, no Brasil e no exterior.

Edson Kowask Bezerra é profissional da área de segurança da informação e governança há mais de quinze anos, atuando como auditor líder, pesquisador, gerente de projeto e gerente técnico, em inúmeros projetos de gestão de riscos, gestão de segurança da informação, continuidade de negócios, PCI, auditoria e recuperação de desastres em empresas de grande porte do setor de telecomunicações, financeiro, energia, indústria e governo. Com vasta experiência nos temas de segurança e governança, tem atuado também como palestrante nos principais eventos do Brasil e ainda como instrutor de treinamentos focados em segurança e governança. É professor e coordenador de cursos de pós-graduação na área de segurança da informação, gestão integrada, de inovação e tecnologias web. Hoje atua como Coordenador Acadêmico de Segurança e Governança de TI da Escola Superior de Redes.

1

Introdução à engenharia reversa

objetivos

Explicar o que é a engenharia reversa e como ela pode ser utilizada na análise de malwares; entender as diferenças entre os tipos de códigos maliciosos e conhecer os tipos de ambientes de análise.

conceitos

Engenharia reversa, programas maliciosos, formas de propagação, classificação do código malicioso, ambiente de análise.

Introdução

Engenharia reversa:

- Processo de descoberta do funcionamento interno de um programa de computador sem ter acesso ao código-fonte.
- Este processo pode ser tão simples como executar o programa e monitorar suas ações e resultados, ou tão complexo quanto analisar as instruções de código de máquina, uma a uma.

Um especialista em engenharia reversa precisa responder perguntas como:

- O que exatamente este software faz?
- Qual parte do código é responsável pelo acesso à rede?
- Quais parâmetros ele aceita?
- Quais ações ativam seu funcionamento?

De acordo com a literatura existente, a engenharia reversa pode ser descrita como o processo de analisar, compreender e identificar as funções de um equipamento, software ou dispositivo, de forma a ser capaz de fornecer manutenção ou desenvolver outro produto compatível com o analisado, ou então descobrir o modo de funcionamento de algum programa desconhecido.

Este processo pode envolver diversas técnicas, incluindo as mais simples, como a pura execução do software em um ambiente controlado, para identificar possíveis ações e modificações causadas por ele, até a análise do código de máquina, possibilitando o entendimento das funções do programa no nível mais baixo de código.

Alguns exemplos de situações onde a engenharia reversa pode auxiliar, além da análise de programas maliciosos:

- Desenvolvimento de driver para um dispositivo;
- Entendimento do funcionamento de um programa proprietário, para desenvolvimento de versão livre que não use o mesmo código do programa proprietário;
- Entendimento de protocolos para permitir compatibilidade entre produtos existentes e futuros produtos desenvolvidos;
- Documentação de código legado, para o qual não exista mais o código-fonte.

Exercício de nivelamento 1
Arquivos maliciosos

Todos os arquivos maliciosos podem ser classificados como vírus? Qual a principal característica de um vírus?

Qual a diferença entre virtualização e emulação?

Exercício de fixação 1
Objetivo da engenharia reversa

Qual a finalidade da Engenharia Reversa?

Definição

Engenharia reversa de arquivos maliciosos é um estudo de código suspeito ou malicioso com o intuito de descobrir:

- Suas funcionalidades e características.
- Seu modo de ação.
- Possíveis ameaças.

A engenharia reversa de arquivos maliciosos envolve não somente a análise de código executável, mas em muitos casos, de protocolos de comunicação, criptografia de dados, entre outros. Este estudo permite identificar as funções executadas por determinado código malicioso, permitindo assim sua identificação em dispositivos infectados, e em muitos casos a remoção dos mesmos do sistema.

Outro objetivo na análise de programas maliciosos é permitir aos times de tratamento de incidentes a identificação e o combate à ação destes programas em redes e sistemas on-line.

Exercício de fixação 2
Diferença entre vírus e trojan

Qual a diferença entre um vírus e um trojan?

Tipos de programas maliciosos

- Vírus.
- Trojan.
- Worm.
- Spyware.
- Rootkit.

É muito difícil catalogar todos os tipos de programas maliciosos existentes atualmente. Apresentamos abaixo uma lista com exemplos de possíveis pragas que podem ser encontradas ao analisar um programa desconhecido. Muitas vezes, o programa vai se encaixar em mais de uma classificação:

- **Vírus**: software que infecta outras aplicações e as usa como vetor de infecção de outros arquivos.
- **Trojan**: aplicação que tem um funcionamento malicioso baseado na diferença daquilo que se espera que ela faça.
- **Worm**: código malicioso com capacidade de se espalhar de um computador para outro através de diversos protocolos de rede.
- **Spyware**: aplicação maliciosa usada para coletar informações pessoais sobre o usuário de um computador.
- **Rootkit**: ferramenta capaz de esconder a presença de programas, usuários ou serviços maliciosos.

Hoje em dia os programas maliciosos utilizam técnicas diversas para descobrir e infectar sistemas vulneráveis, e para se esconder de possíveis mecanismos de detecção. Ao infectar um sistema, eles costumam instalar diversos outros programas maliciosos, comprometendo ainda mais o computador.

Ações de um programa malicioso

- Infecção de arquivos.
- Exploração de vulnerabilidades.
- Engenharia social.
- Instalação de outros programas maliciosos.
- Destruição de dados.
- Roubo de informações.
- Ocultação de atividade maliciosa.
- Agente de ações coordenadas.
- Propagação.

Um ponto a se destacar é que nos últimos anos os programas maliciosos têm cada vez mais sido utilizados para suporte a atividades criminosas, através de ataques coordenados a sites em troca de pagamento, distribuição de spywares, fraudes em sistemas de pay-per-click ou de ranking de sites, roubo de informações pessoais e de dados sigilosos e bancários.

Exercício de fixação 3
Engenharia social

O que é engenharia social?

Formas de ataque

Vetores de ataques de baixo nível:

- Estouros de buffers (Stack e Heap).
- Vulnerabilidades de formato de strings.

Vetores de ataque de alto nível:

- Exploração de vulnerabilidades em aplicativos.
- Exploração de falhas existentes de configuração do sistema ou de proteções.

Ataques de mais alto nível:

- Engenharia social.
- E-mails em massa.

Os vetores de ataque usados por programas maliciosos apresentam variações, e muitas vezes um único código malicioso pode explorar diversas vulnerabilidades, tanto de baixo como de alto nível.

Os ataques de estouro de buffer (buffer overflow) exploram vulnerabilidades na forma como os programas tratam dados gravados em buffers (espaços de memória reservados para variáveis). Muitas vezes, o programador não verifica o conteúdo de um parâmetro gravado em um buffer, o que permite que o parâmetro seja gravado nesta área de memória com um tamanho maior do que é suportado. Como a área de memória seguinte à área de armazenamento de variáveis é reservada para a pilha de execução do programa, estes dados serão sobrescritos, permitindo ao atacante controlar o fluxo de execução do programa, e consequentemente permitindo a execução de código malicioso.

Os ataques contra formatos de string exploram uma vulnerabilidade em códigos escritos em C/C++, onde um parâmetro para uma função de string (*printf* por exemplo) não é verificado, permitindo a manipulação da memória escrita no Stack e consequentemente a execução do código malicioso.

Finalmente, ataques de engenharia social são difíceis de combater, pois dependem exclusivamente da ingenuidade do usuário do computador. Estes ataques induzem o usuário a executar o programa malicioso ou divulgar inconscientemente informações pessoais ou sigilosas. É hoje em dia um dos principais vetores de ataque utilizados por programas maliciosos.

Exemplo disso são os milhares de spams enviados diariamente, com “fotos” ou “relatórios” falsos anexados, ou links para páginas e arquivos maliciosos para serem executados.

Como é a propagação?

- Se é por rede, qual a velocidade de propagação?
- Existe algum padrão na forma de se propagar (é randômica ou viciada)?
- O programa contém um módulo para reenvio de e-mails em massa?
- Ele se replica por compartilhamentos?
- Ele se replica por P2P ou outros protocolos de troca de arquivos?
- Quais vulnerabilidades ele explora?

A seguir, conheceremos algumas perguntas mais específicas que devem ser respondidas pelo analista fazendo a engenharia reversa. O método de propagação de um programa malicioso pode envolver diversas técnicas e etapas, e descobrir essas informações pode ajudar não somente a encontrar outras possíveis máquinas vulneráveis, mas também descobrir novas vulnerabilidades que possam estar sendo exploradas pelo programa.

Quais funcionalidades o programa tem?

- Backdoor?
- Servidor IRC?
- Abre alguma porta de serviço?
- Busca dados em alguma página web?
- Recebe algum tipo de comando remoto?
- Tem algum temporizador para executar funções programadas?

A lista de funcionalidades de um programa malicioso atual, como os bots, pode ser gigantesca. Um programa desse tipo pode dar controle total ao invasor sobre a máquina invadida, e permitir que ele execute praticamente qualquer função remotamente.

Descobrir estas funções pode ajudar na identificação de comportamentos estranhos em um sistema, ou a detecção de informações que possam estar sendo coletadas pelo invasor.

Quais modificações o programa causa?

- Ele cria/modifica/remove alguma chave de registro?
- Quais arquivos ele cria/modifica/remove?
- Ele modifica algum processo em execução?
- Ele se auto modifica?
- Ele modifica algum dado ou função do sistema?

Um programa malicioso, ao invadir o computador, pode modificar o sistema a ponto de somente uma restauração completa do mesmo ser capaz de corrigir os problemas.

Descobrir as modificações causadas pelo programa pode ajudar a recuperar o estado sadio do sistema, ou identificar possíveis maneiras de combater novas infecções.

Qual o objetivo do código?

Para quê o programa foi escrito?

- Ataque direcionado?
- Roubo de informações?
- Rede de computadores infectados?

Quem escreveu o código?

- Qual linguagem foi usada?
- Em que data foi escrito?
- Existe alguma característica na forma de escrever o código, que permita a associação a alguém ou a algum grupo?

A motivação por trás de quem escreve um programa malicioso pode dar pistas sobre seu autor, ou de possíveis esquemas criminosos. Quase sempre, existe uma motivação financeira dos autores desse tipo de programa.

Outra informação importante a ser pesquisada é a característica do código escrito. Muitas vezes é possível identificar um padrão no código que possa ser associado a uma pessoa ou grupo.

Classificação de código malicioso

O código pode ser classificado em três categorias:

- **Código constante**: executado constantemente durante a execução do programa.
  - Ex: loop de coleta de endereços de e-mails, ou de scan por máquinas vulneráveis
- **Código reativo**: executado em resposta a um evento específico
  - Ex: quando um usuário acessa determinada página de um banco
- **Código dormente**: executado em data determinada
  - Ex: ataque DDoS coordenado

As categorias descritas abrangem quase a totalidade de códigos que podem ser encontrados em um programa malicioso.

Muitas vezes, durante a análise de um binário, o analista vai encontrar porções do código que são executadas em um loop constante e em alguns casos podem causar a degradação do sistema.

Códigos reativos são muito utilizados em programas que coletam informações bancárias, e que agem no momento em que o usuário acessa a página do seu banco. Eles podem então sobrepor a tela de digitação dos dados, ou então monitorar o teclado e cliques do mouse para capturar as informações digitadas.

O código dormente costuma ser usado em situações que exijam uma ação coordenada das máquinas infectadas. Exemplos disso são diversos programas maliciosos que agem somente em determinadas datas, ou em determinado período do dia.

Exercício de fixação 4
Ambiente virtualizado

Qual a finalidade de um ambiente virtualizado?

Ambiente de análise

Como analisar estes programas?

- Ambiente instalado para finalidade de análise:
  - Forma mais segura.
  - Gasta muito tempo e recurso, pois a cada execução do programa malicioso, precisamos recuperar o estado inicial do sistema.
  - Podemos utilizar hardware especial: CoreRestore, Norton Ghost, e outros hardwares/softwares para recuperação de disco.
- Ambiente virtual:
  - VMWare, Xen, Virtual PC, entre outros.
  - Mais barato para implantar uma estrutura de análise.
  - Pode simular um ambiente de rede completo.
  - Facilita a recuperação do estado inicial, agilizando a análise, mas pode prejudicar a análise.

Para analisar um programa malicioso em tempo de execução, pode-se utilizar basicamente três formas:

1. Um ambiente físico preparado para isso, com computadores e sistemas operacionais reais, que permitam simular com fidelidade o ambiente comum de um usuário. É a forma mais garantida de conseguir um comportamento normal do programa analisado, pois este não saberá jamais que o computador infectado está sendo examinado. Suas desvantagens são o alto custo de implementação e manutenção, já que a cada execução do programa malicioso, todo o ambiente precisará ser restaurado. Existem diversos equipamentos que podem ser usados para realizar essa operação. Basicamente, eles mantêm o estado do conteúdo em disco em seu formato original, direcionando todas as operações de gravação para um disco "spare". Para restaurar o ambiente, eles simplesmente descartam esse conteúdo gravado, e reiniciam o sistema a partir do disco original.
2. Outra técnica é a utilização de imagens binárias dos discos utilizados, que são gravados novamente no disco, sobrescrevendo quaisquer modificações causadas pelo programa malicioso. O ambiente virtual tem a vantagem de facilitar muito a restauração do sistema, agilizando assim a análise. Além disso, é muito mais barato de implantar, permitindo assim a simulação de um ambiente de rede completo, com diversas máquinas virtuais realizando funções específicas para enganar o programa malicioso e analisar interações remotas do mesmo. A desvantagem desses sistemas é que muitos programas maliciosos estão cientes desse tipo de ambiente, e modificam seu comportamento, parando a execução, executando funções diferentes ou até mesmo tentando explorar vulnerabilidades no gerenciador do sistema virtual. Por isso o analista deve tomar cuidado ao usar esse tipo de técnica.
3. Uma terceira técnica envolve a utilização de um simulador de sistema operacional, que não deixa de ser um ambiente virtual. Mas diferente dos ambientes virtuais simulados, estes programas simulam apenas as chamadas específicas do sistema operacional, não executando as funções propriamente ditas. Exemplos destes softwares são o Norman Sandbox e o CWSandbox.

Virtualização x Emulação

- Tecnologias de virtualização tais como VMWare capturam todo o acesso a hardware, e simulam toda a placa-mãe, menos o processador.
- Tecnologias de emulação, tais como BOCHS ou QEMU, emulam completamente o processador, dispositivos de hardware e memória.
- Emulação é mais lenta, mas permite o controle total sobre o ambiente, e é muito mais segura para impedir que o programa malicioso fuja do ambiente virtual.
- Emulação parcial pode ser muito útil na análise estática.

A emulação tem algumas vantagens sobre a virtualização simples, pois permite controlar a forma como o código binário vai ser executado pelo processador.

Devido ao fato de sistemas de virtualização simularem todo o hardware com exceção do processador, o código é executado diretamente pelo processador do sistema. Em um ambiente emulado, até o processador é simulado, permitindo assim que uma instrução seja interpretada da forma como o analista quiser.

Um exemplo disso são instruções de TRAP de processamento, que normalmente param a execução de um programa executado dentro de um debugger. Um programa que use essa técnica para impedir a engenharia reversa é facilmente "enganado" em um sistema emulado, que pode simplesmente ignorar esse tipo de instrução.

Análise "ao vivo" x Engenharia reversa

Análise ao vivo:

- Significa executar o programa malicioso em um ambiente e monitorar sua atividade.
- Permite identificar rapidamente algumas funções do programa.
- Complicado de identificar a forma como um programa executa essas funções, mas é um bom ponto de início para a engenharia reversa.

Engenharia reversa:

- Análise do código do programa.
- Permite entender seu funcionamento, e descobrir funções escondidas, ou código dormente ou reativo.

A análise ao vivo de um programa pode ser um bom início na análise de um programa malicioso, já que pode dar pistas importantes sobre as funcionalidades de um programa.

Mas ela não permite descobrir facilmente as funcionalidades que sejam ativadas por uma ação específica do usuário ou uma função dormente ativada em determinada data.

Roteiro de Atividades 1

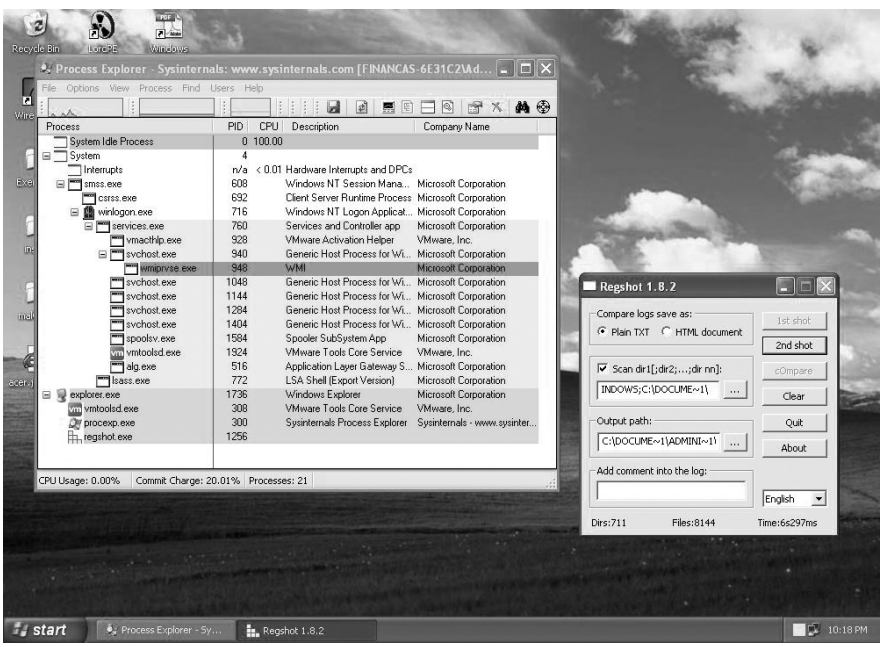
Atividade 1.1 – Análise dinâmica de binário desconhecido

Esta atividade demonstra uma possível análise dinâmica de um binário desconhecido, em um ambiente virtual. O objetivo é identificar mudanças no sistema causadas pela execução de um programa desconhecido. Para isso, utilizamos a ferramenta RegShot, um utilitário livre que permite monitorar mudanças em arquivos e no registro, e comparar dois estados do sistema, para identificar mudanças.

O aluno deve abrir a máquina virtual disponibilizada para o curso, que contém um sistema operacional Windows XP Pro com Service Pack 2.

Para garantir que todos os alunos tenham um ambiente igual, foi gerado um snapshot chamado “Início das aulas”.

1. Inicie o VMWare e abra a imagem “D:\Windows_XPSP2_Malware\WinXPPro.vmx”.
2. Restaure o snapshot “Início das aulas”.
3. Execute o Process Explorer, o atalho encontra-se no desktop, e mantenha-o aberto durante o exercício, para examinar os processos em execução.
4. Execute a ferramenta RegShot no desktop;
5. Marque a opção “HTML Document” e “Scan Dir”, acrescentando a linha abaixo:
6. C:\Windows;C:\Docume~1\
7. Clique em “1st Shot” e escolha “Shot”;
8. Execute o ícone “acer.jpg.exe” no desktop;
9. Clique em “2nd Shot” e depois em “Shot”, e quando terminar, clique em “Compare” para gerar a diferença entre as execuções.



Ao final do exercício, o RegShot abrirá uma janela do Internet Explorer com o conteúdo das modificações identificadas por ele.

Perguntas:

1. Que modificações o programa causou no sistema?

2. Você percebeu algum outro comportamento estranho no sistema?

3. Faça um breve relatório com suas suspeitas, e descreva dicas para auxiliar uma possível engenharia reversa desse programa.

4. Após terminar, restaure novamente o snapshot “Início das aulas”, pois o sistema está comprometido.

2

Ferramentas

objetivos

Criar um ambiente de análise e utilizar as ferramentas para auxiliar a análise do código e entender as diferenças entre elas, identificando a melhor utilização de cada uma.

conceitos

Ambiente de análise, virtualização, debuggers, decompiladores e disassemblers.

Montagem do ambiente de análise

Antes de iniciar a análise, deve-se montar um ambiente completo, composto de:

- Máquinas virtuais.
- Ferramentas de análise dinâmica.
- Debuggers.
- Decompiladores.
- Disassemblers.
- Ferramentas de manipulação de arquivos executáveis.

A análise de binários suspeitos pode envolver diversas técnicas, e por isso recomenda-se que o ambiente usado pelo analista atenda a todas essas necessidades. O uso de máquinas virtuais facilita muito esse tipo de análise, pois permite ao analista não somente ter as ferramentas disponíveis, como também diversas configurações de ambientes, como sistemas operacionais, patches, configurações de firewall, rede etc. A partir daí, o analista deve compor um ambiente com as ferramentas que atendam a suas necessidades e com as quais tenha familiaridade.

Neste capítulo veremos algumas ferramentas de apoio à análise de binários suspeitos, tanto para análise dinâmica quanto para análise estática.

Exercício de nivelamento 1 — Ferramentas

O que é um debugger?

Qual a diferença entre decompilador e disassembler?

VMWare

- Salvamento de estado em background.
- Diversos níveis de snapshot.
- Clones.
- Simulação de infraestrutura de redes.
- Gravação de vídeos.

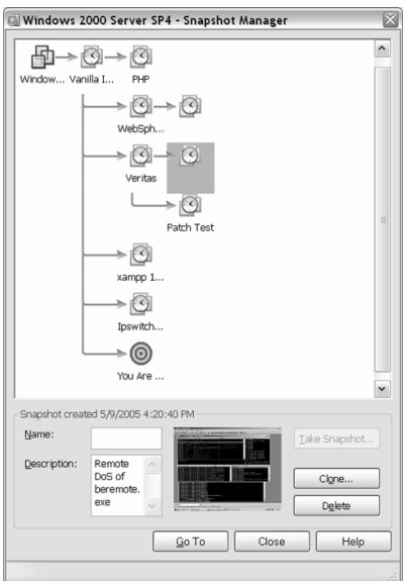


Figura 2.1
Snapshot Manager.

Marcando a opção "disable acceleration" nas configurações da máquina virtual, alguns programas maliciosos deixam de detectar o VMWare.

O ambiente virtual que usaremos durante o curso é baseado no VMWare, um ambiente de virtualização que simula todo o hardware de um computador, mas passa diretamente à execução das instruções ao processador físico da máquina. Por isso ele não é considerado um emulador.

Dentre as características que podemos destacar no VMWare, podemos citar o gerenciamento de snapshot, que permite a criação de uma hierarquia de estados salvos do sistema. Isto permite ao analista testar diversas configurações ou situações diferentes usando apenas uma máquina virtual.

Outra característica importante é a possibilidade de simular uma infraestrutura de rede básica, através do uso das interfaces virtuais do VMWare (vmnet1, vmnet2 etc). Cada máquina virtual pode estar conectada a uma ou mais redes virtuais, e podem se comunicar entre si, permitindo a criação, por exemplo, de um roteador/firewall entre duas redes.

Finalmente, o VMWare tem a capacidade de tirar um snapshot da tela da máquina virtual em execução, ou de gravar um vídeo da atividade na tela, uma ótima ferramenta para estudar o comportamento de determinado programa, ou para utilização em treinamentos.

O VMWare implementa um hook virtual na CPU do sistema virtual, para que haja comunicação entre a máquina virtual e o processador físico. É dessa forma que ele implementa a execução de instruções diretamente no processador. Mas esse hook é passível de detecção

no ambiente virtual. Alguns programas maliciosos têm a capacidade de identificar esse hook e mudam sua execução por estarem dentro do ambiente virtual. Muitas vezes, esses programas não vão executar dentro de uma máquina virtual. Ao configurar a máquina virtual, o analista pode marcar a opção "Disable Acceleration" no menu "VM→Settings→Hardware →Processors→Disable acceleration for binary translation", fazendo com que o hook não seja criado e estes programas passem a funcionar.

Exercício de fixação 1
VMWare

O VMWare é um software de emulação ou virtualização?

Debugger

Debugger é uma ferramenta de análise em tempo de execução, que permite a manipulação do programa no nível de código de máquina (assembly).

Funcionalidades:

- Informações sobre estado da CPU.
- Execução passo-a-passo.
- Pontos de parada (breakpoints).
- Visualização e manipulação de memória e registros.
- Visualização de threads.

A partir de agora conheceremos algumas ferramentas que podem auxiliar na análise de um binário desconhecido. Elas são utilizadas para realizar tarefas específicas, e isoladas não seriam de grande serventia.

O primeiro tipo de ferramenta que veremos são os debuggers, programas utilizados para analisar o código de máquina (assembly) de um programa em execução. Com essa ferramenta é possível:

- Examinar informações como estado de registradores e posições de memória;
- Executar o programa instrução por instrução;
- Visualizar threads;
- Criar pontos de parada em determinadas posições do código ou então em caso de acesso a determinada posição de memória.

Os debuggers permitem ainda modificar o conteúdo de registradores e memória, alterando assim a execução do programa. Esta funcionalidade é importante para permitir ao analista desviar a linha de execução do programa, evitando códigos de proteção ou direcionando o programa para uma função específica que se queira analisar.

Debuggers para Windows

- Microsoft WinDBG.
- OllyDBG.
- IDA Pro.
- PyDBG.

Entre os exemplos de debugger para Windows, o SoftIce foi durante muito tempo considerado o melhor debbuger existente, pois era extremamente poderoso. Ele foi descontinuado em 2006, mas continua sendo uma referência.

Muitos compiladores vêm com suas próprias versões de debugger. Em http://www.thefreecountry.com/compilers/cpp.shtml podem ser encontrados diversos compiladores gratuitos, muitos dos quais possuem debugger interno e outras ferramentas de debugging.

Exercício de fixação 1
Decompilador e disassembler

Qual a diferença entre um decompilador e um disassembler?

Decompilador

Ferramenta que tenta traduzir dados binários em uma linguagem de alto nível, geralmente apresentando falhas (decompilação de verdade é impossível). Realiza otimizações do compilador, remoção dos símbolos, nomes de variáveis e funções perdidas.

Existem ferramentas que basicamente deixam o disassembly mais legível.

Decompiladores são ferramentas que auxiliam o analista na tentativa de recuperar um código de alto nível, a partir de um código em linguagem de máquina. Eles normalmente tentam atingir esse objetivo com a identificação de estruturas dentro do código binário, e posterior transformação dessas estruturas em uma representação de alto nível.

Por exemplo, ao analisar um código binário, a ferramenta pode identificar uma estrutura representando um switch/case, e então substituí-la por uma representação em alto nível (geralmente C ou C++) dessa estrutura.

Mas esse tipo de abordagem falha na maioria dos casos. O problema é que durante o processo de compilação, muitas informações são perdidas, como nomes de variáveis e de funções, e a otimização feita pelo compilador modifica a forma como a estrutura em linguagem de alto nível é mapeada para a linguagem de máquina.

Ferramentas de decompilação:

- REC e REC Studio.
- Desquirr.
- Boomerang.
- Hex-Rays Decompiler.

Entre as ferramentas que tentam decompilar código, Desquirr e Hex-Rays Decompiler são plugins para o IDA Pro. As demais são ferramentas stand-alone, sendo que o REC Studio é a mais estável.

Exercício de fixação 2
Arquivos binários

O que são arquivos binários?

Manipuladores de binários

Ferramentas de manipulação de binários:

- Permitem examinar e modificar a estrutura dos arquivos executáveis.

Arquivos executáveis:

- Arquivos estruturados, geralmente baseados em um padrão comum dentro de uma arquitetura de processador.
- Informações que podem ser encontradas nessas estruturas:
  - *Entry point* do programa.
  - Tamanho das seções.
  - Lista de funções importadas.

Exemplos de padrões de executáveis:

- PE, ELF, COM, MACH-O, COFF, entre outros.

Uma parte importante da análise de arquivos suspeitos inclui a identificação de informações básicas sobre os arquivos analisados. O primeiro passo é identificar o tipo de arquivo analisado: se é arquivo de dados, executável, se está compactado ou não.

Ao identificar um arquivo executável e examinando a estrutura desse binário, o analista pode conseguir informações importantes. O formato de executável que estudaremos durante esse curso é o formato Portable Executable (PE), o padrão de executável usado no Microsoft Windows.

No formato PE, a estrutura do binário contém informações importantes sobre o executável.

Ferramentas de manipulação de binários

- LordPE
  - http://www.woodmann.net/collaborative/tools/index.php/LordPE
- PEiD
  - http://www.peid.info/
- ImportREC
  - http://vault.reversers.org/ImpRECDef
- PE Explorer (comercial)
  - http://www.heaventools.com/overview.htm

Durante o curso veremos exemplos de ferramentas para manipulação de binários. O último link fornecido contém uma lista de ferramentas de manipulação de binários e de engenharia reversa.

Disassembler

- O que é um disassembler?
  - É uma ferramenta de análise estática que transforma *bytes* em linguagem assembly.
- Praticamente todos os debuggers podem realizar o disassembly de um executável.
- O aspecto mais difícil em relação ao disassembly é diferenciar o que é código executável do que é apenas dado.

O disassembly é usado para realizar a análise estática de um binário. A principal vantagem de realizar uma análise estática é ter acesso ao código do programa. Em uma análise dinâmica, ficamos restritos apenas ao que o programa decide executar baseado nos seus parâmetros. Já em uma análise estática, temos acesso a toda a estrutura do código do programa, podendo seguir o caminho que melhor nos convier.

Apesar disso, a dificuldade encontrada na análise estática é diferenciar o que é código executável do que é apenas dado. Em um arquivo binário, tudo é apresentado por uma sequência de bits. Cabe ao sistema operacional, na hora da execução, identificar o ponto inicial do programa (através dos cabeçalhos PE) e transferir o controle para este ponto. A partir daí, o programa decide o fluxo de execução, e com isso faz a diferenciação entre código e dado.

Uma técnica comum anti-engenharia reversa é armazenar partes do código na área do executável reservada aos dados do programa, e vice-versa.

Os disassemblers, por não executarem o programa, não têm a informação do que é código e do que é dado, e precisam descobrir isso seguindo o fluxo de instruções, mas sem executá-las.

Um exemplo seriam os programas executáveis compactados. Normalmente eles contêm uma rotina de descompactação, e ao final dela, o fluxo de execução é transferido para uma área de memória que ainda não contém código executável. Este tipo de situação muitas vezes confunde o disassembler, que pode marcar aquela posição como código, quando na verdade ela ainda não é.

Ferramentas de Disassembly:

- IDA Pro
  - http://www.hex-rays.com/idapro/
- OllyDBG
  - http://www.ollydbg.de/
- Fenris
  - http://lcamtuf.coredump.cx/fenris
- PE Browser
  - http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html

Entre as ferramentas de disassembly, a mais conhecida e considerada a melhor é o IDA Pro. Apesar disso, outra boa ferramenta que pode ser usada em conjunto com o IDA Pro é o OllyDBG. Com essas duas ferramentas, é possível analisar praticamente qualquer tipo de arquivo suspeito, mesmo aqueles que tentam se proteger de análises.

No link abaixo há uma lista com diversas ferramentas de disassembly para download:

- http://www.woodmann.net/collaborative/tools/index.php/Category:Disassemblers

Roteiro de Atividades 2

Atividade 2.1 – Configuração do VMWare

Esta atividade instrui o aluno a iniciar o sistema e prepará-lo para as próximas sessões. Por isso, este roteiro descreve todas as ações que o aluno deverá executar à risca, para ter um sistema pronto para as próximas sessões.

Será utilizado o sistema operacional Windows XP Pro com Service Pack 2, com uma licença temporária de 60 dias.

Para modificar as configurações do sistema operacional, ele deve estar parado, isto é, o aluno deve fazer o shutdown do sistema operacional se ele ainda estiver ativo desde a última sessão, ou se estiver em estado de pausa.

- Execute o VMWare;
- Abra a máquina virtual fornecida:
  - "D:\Windows_XPSP2_Malware\winXPPro.vmx"
- Desabilite a aceleração nas configurações da VM;
- Compartilhe um diretório para ser acessado pela máquina virtual;
- Inicie o sistema, faça um snapshot e salve como "Início das atividades".

Na aba "Hardware", o aluno deve confirmar que a configuração da interface de rede ethernet está como "Host-only", pois assim a máquina virtual não terá acesso à internet.

Na aba "Options", opção "Shared Folders", o aluno deverá verificar se existe um compartilhamento ativo, apontando para o diretório "malware" presente no CD do aluno ou no disco da máquina "host".

Apenas como exemplo, está configurado na máquina virtual disponível no CD do aluno um diretório compartilhado apontando para um diretório em meu sistema. Este compartilhamento deverá ser removido ou modificado para apontar para um diretório existente.

Repare que o aluno poderá acessar esse compartilhamento depois através do Explorer, indicando no lugar do IP da máquina host, o endereço "\\vmware-host\" e o nome do compartilhamento, no formato "\\vmware-host\Shared Folders\nome_compartilhamento". Após encontrar a máquina, o aluno poderá mapear o compartilhamento em um drive local para facilitar o acesso.

Finalmente, o aluno deverá ligar o sistema operacional e fazer um snapshot, a partir do qual serão executadas todas as atividades daqui para a frente. Ao executar uma atividade que exija a execução de código malicioso e o consequente comprometimento da máquina, o aluno poderá retornar a este snapshot para ter novamente um sistema limpo.

Atividade 2.2 – Ferramentas de decompilação

1. Execute “Rec/RecStudio”;
2. Abra “Exemplos\Sessão 2\mal.upx.6dddd7e8e5ff88a15b7884a833ff893b.dat” no desktop;
3. Qual a sua opinião sobre o resultado obtido? Discuta com seus colegas situações onde este tipo de ferramenta pode ser útil.

Para ilustrar a dificuldade de se trabalhar com ferramentas de decompilação, utilizaremos uma destas ferramentas para analisar um binário suspeito, e visualizar o resultado da decompilação.

Ao abrir o executável, o programa analisará a estrutura do binário e mostrará uma estrutura em árvore à esquerda, e o código “assembly” à direita. Clicando em algum dos procedimentos identificados pelo programa, é possível visualizar o código decompilado daquela função.

Atividade 2.3 – Ferramentas de manipulação de binários

Os alunos devem abrir, com cada ferramenta, dois ou mais arquivos suspeitos presentes no diretório “Malware” no desktop da máquina virtual, para que possam comparar diferentes informações sobre cada um.

1. Utilize as ferramentas PEiD, Lord PE e ImportREC, disponíveis no desktop da máquina virtual para examinar os arquivos maliciosos presentes no diretório “Malware”, também no desktop.
2. Utilize todas as funcionalidades de cada um dos programas. Quais funcionalidades você considera que seriam mais úteis em uma análise de um programa malicioso? Discuta com seus colegas as informações encontradas com cada ferramenta, e como essas informações podem ser úteis para identificar um programa malicioso. Faça um relatório descrevendo as funcionalidades que você considera mais úteis e as situações em que elas podem ajudar.

3

Introdução ao IDA Pro e OllyDBG

objetivos

Apresentar as ferramentas de disassembly mais utilizadas para engenharia reversa, mostrando suas diferenças e principais funcionalidades.

conceitos

IDA Pro, OllyDbg.

Introdução ao IDA Pro

DataRescue IDA Pro:

- Ferramenta padrão para análise estática.
- Suporta múltiplas arquiteturas de processadores.
- Cross-plataforma, interface de console e gráfica.
- Expansível através de plugins em diversas linguagens.
- Manipulável através de scripts.
- Possui dezenas de plugins e ferramentas escritas para ele.
- Versão comercial custa pouco mais de US$ 500.

Neste capítulo, vamos conhecer as ferramentas que usaremos ao longo do curso, como o Interactive Disassembler Pro, ou IDA Pro, reconhecido como um dos melhores programas do mercado para engenharia reversa. Custando pouco mais de 500 dólares, a quantidade de funcionalidades disponíveis compensa o valor pago.

Trata-se de um programa de disassembly capaz de manipular executáveis de praticamente qualquer tipo de processador, por causa da sua arquitetura modular. Basta o analista ter ou desenvolver os módulos de acesso a um processador, que o IDA identificará binários daquele processador.

Além de suportar processadores diferentes, o IDA suporta também diversos tipos de binários, através de módulos de reconhecimento de executáveis. Esta propriedade torna possível, por exemplo, analisar programas escritos para microprocessadores PIC, programas escritos em linguagens variadas como Java, .Net, Linux ELF ou executáveis de outras arquiteturas, como Mac, Sun etc.

A licença do IDA dá direito ao uso da versão Windows e também da versão Linux ou Mac do programa. O IDA também suporta um sistema de plugins e scripts externos, que faz com que suas possibilidades de análise sejam praticamente ilimitadas. Os scripts podem

ser desenvolvidos tanto em sua própria linguagem, o IDC, como em Python, C e Perl, entre outras. Justamente por isso, existem na internet centenas de ferramentas escritas para integração com o IDA Pro.

Interactive Disassembler Pro:

- IDA é uma das melhores ferramentas para engenharia reversa.
- Controlável através de scripts IDC IDA Perl ou IDA Python.
- Interativo, pois é possível interagir com seu banco de dados, corrigindo o que ele não conseguir identificar automaticamente.

FLIRT:

- Fast Library Identification and Recognition Technology.
- Bibliotecas de padrões, permitem que o IDA reconheça as chamadas padrão de bibliotecas.

Como já foi comentado, o IDA Pro pode ser controlado e expandido através de scripts IDC, ou através de outras linguagens como Python e Perl.

O termo interativo advém do fato de que o IDA funciona na verdade como um gerenciador de banco de dados. Ao abrir um executável, as informações sobre ele são inseridas em um banco de dados, e a partir daí todas as operações são realizadas nele.

Por isso, o executável não é mais necessário após a criação da base de dados, facilitando a distribuição da análise entre um grupo de pessoas, sem a necessidade de enviar o binário malicioso propriamente dito, que só se faz necessário no caso de se realizar a execução do binário.

As operações realizadas na base permitem ao analista modificar informações no disassembly, adicionar comentários, consertar códigos que tenham sido identificados erroneamente pelo IDA, criar ou remover funções, estruturas e tipos, renomear objetos, entre outras operações.

Uma das características mais vantajosas do IDA é a utilização de bibliotecas de reconhecimento de padrões. O FLIRT é um sistema de biblioteca de padrões aplicados ao banco de dados analisado, que permite identificar automaticamente funções pertencentes a bibliotecas padrão do Windows. Por exemplo, o sistema pode identificar automaticamente funções padrão do Delphi ou do Visual C, o que facilita o trabalho do analista, já que ele não vai precisar analisar novamente essas funções.

Exercício de nivelamento 1 
IDA Pro e OllyDGB

Utilizando o IDA podemos analisar o código do binário sem que o processo esteja em execução. Como isso é possível?

Qual o plugin do OllyDBG que nos permite gerar um dump do executável?

Durante o curso:

- Será utilizado IDA Pro 5.0 Free.
- Os arquivos de exemplo estão nas pastas “Exemplos” e “Malware” no desktop.
- Sempre que houver comprometimento da máquina, restaurar o último snapshot “sadio”.

- Por isso, evite executar os arquivos maliciosos durante o curso, a não ser quando indicado
- Para começar, vamos configurar o IDA Pro Free para facilitar o uso.

Neste curso vamos utilizar o IDA Pro Free. Esta versão é distribuída livremente através do site do IDA Pro, mas não conta com uma série de funcionalidades da versão atual (6.x).

A principal característica faltante é a visualização da listagem assembly em forma de gráfico, que facilita a identificação de estruturas e fluxo. Na versão Free não é possível usar os plugins compilados para as versões comerciais, o que é mais um incentivo para a compra da versão comercial.

Os arquivos de exemplo vão estar disponíveis dentro da máquina virtual, nos diretórios "Exemplos" e "Malware" no desktop. Estes arquivos são realmente maliciosos, coletados pelo CAIS através do projeto de banco de dados de malwares. Estes arquivos devem ser manipulados em observância aos preceitos de segurança e ética.

Durante o curso será necessário, em alguns casos, executar o arquivo malicioso de forma a auxiliar a análise. Nestes casos, onde ocorre o comprometimento da máquina virtual, deve-se recuperar o último estado salvo do VMWare, para retornar a máquina virtual a um estado sadio.

Por isso é importante relembrar a criação de um diretório compartilhado entre a máquina virtual e a máquina host, para que ele sempre grave qualquer arquivo que deseja manter neste diretório, pois ao recuperar o estado salvo, os arquivos armazenados dentro da máquina virtual serão perdidos.

Salve o estado da máquina logo no início desse capítulo, para ter um ponto de retorno mais atualizado para recuperar a máquina após uma infecção.

O primeiro passo antes de iniciar este capítulo é configurar o IDA Pro.

Exercício de fixação 1
Conhecendo o IDA Pro

O IDA Pro é um decompilador ou um disassembler?

IDA Pro

Ao abrir o IDA Pro e um executável para análise, esta é a tela que você irá encontrar (Figura 3.1).

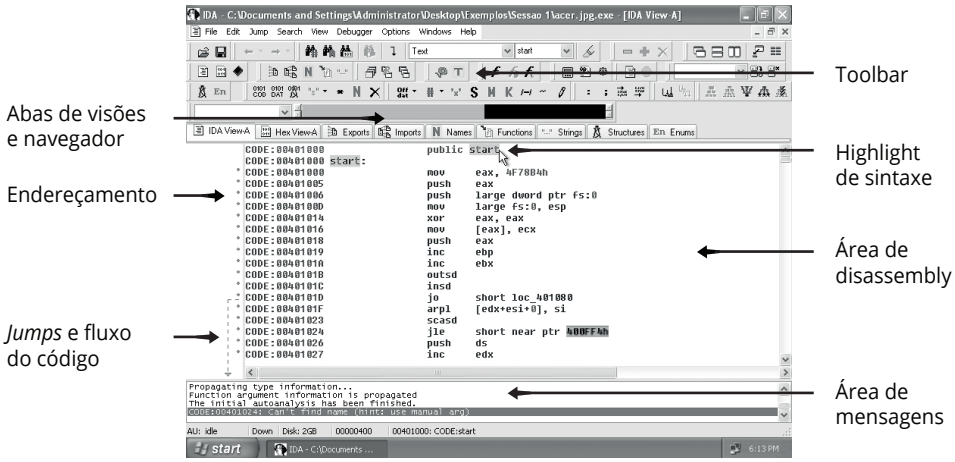


Figura 3.1 IDA Pro.

Na parte superior estão as barras de ferramentas, com acesso a muitas funções do IDA, e no menu, acesso a outras ferramentas.

Abaixo da área de toolbar, temos a barra de navegação. Ao abrir um executável, essa área representa o arquivo completo. Ela é colorizada, e cada cor indica um tipo de informação. Ao remover a barra de navegação da toolbar (puxando-a para o centro da tela por exemplo, destacando da área de toolbar) ela mostrará uma legenda com o significado de cada cor. A caixa *dropdown* à esquerda permite ao aluno encontrar rapidamente alguns tipos de informação, como o *entry point* do programa ou algum padrão de texto ou hexadecimal.

Abaixo da barra de navegação temos as abas com as janelas abertas. É possível abrir outras janelas através do menu "Views->Open Subviews". Estas janelas mostram informações diferentes presentes no banco de dados, como todas as funções identificadas ou todas as strings encontradas.

Na aba IDA View A temos a área de disassembly. Cada aba IDA View pode representar uma parte diferente do código, e você poderá abrir quantas abas IDA View precisar (View → Open Subview → Disassembly).

Nesta área, temos o código assembly do programa que está sendo analisado. À esquerda, temos uma área reservada para mostrar o fluxo do programa. As setas indicam JUMPs dentro do código. Ao lado, temos o endereço de cada instrução, precedido do nome da seção onde o código está. Logo após, qualquer identificador, como o nome de função, labels etc.

Em seguida, vem o código propriamente dito. Veja que o IDA faz um highlight de objetos selecionados, marcando outras ocorrências para facilitar a visualização. Uma colorização é aplicada para identificar diferentes tipos de informações no código, como por exemplo azul para instruções e parâmetros nomeados, verde para endereços ou dados brutos, e vermelho para áreas com problemas na identificação. O IDA permite navegar no código dando um duplo clique em algum objeto nomeado.

Por último, na parte mais à direita, o IDA mostra as referências e comentários.

Na parte inferior da tela são mostradas as mensagens do programa e de saída dos plugins e scripts. Essa área deve ser examinada para descobrir se a função executada teve sucesso ou não.

Funcionalidades

- Visão hexadecimal.
- Lista de funções.
- Lista de nomes e símbolos.
- Lista de funções importadas.
- Lista de funções exportadas.
- Lista de strings.
- Estruturas e enumerações.
- Debugger.
- Referências cruzadas.
- Scripts IDA.

- **Visão hexadecimal**: mostra o conteúdo dos bytes do programa. Pode ser sincronizada com a listagem disassembly, de forma a sempre exibir o mesmo local que está sendo mostrado na listagem. Isto pode ser muito útil para identificar partes do código que não foram identificadas automaticamente pelo IDA, ou que foram identificadas erroneamente. Por exemplo, o analista pode com o IDA identificar uma string de texto como código, e observando a visão hexadecimal fazer a correção no banco de dados.
- **Lista de funções**: mostra todas as funções identificadas pelo IDA, tanto as pertencentes a bibliotecas, quanto aquelas nomeadas automaticamente. Um duplo clique em qualquer uma delas leva à função na listagem disassembly; clicando com o botão direito é possível visualizar e modificar informações sobre as flags da função.
- **Lista de nomes**: mostra todos os nomes identificados e criados automaticamente pelo IDA ou criados pelo analista. Através dessa lista você pode chegar rapidamente a qualquer dado identificado dentro do programa.
- **Lista de funções importadas**: mostra a lista de funções que o IDA identifica como sendo importadas de DLLs externas, que são representadas no código apenas por suas declarações.
- **Lista de funções exportadas**: as funções que o programa exporta. Para uma DLL, conterá todas as funções exportadas, mas para um binário, ela normalmente contém todos os *entry points* do programa.
- **Lista de strings**: mostra todos os textos reconhecidos pelo IDA. Essa lista é gerada dinamicamente, e normalmente o IDA reconhece somente um tipo específico e básico de string (estilo C, 7 bit ASCII null terminated). Para configurar outros tipos de string, deve-se clicar com o botão direito na janela de string, e escolher *Setup*. Nesta tela também é possível escolher a opção "Ignore instruction/data definition", que fará com que o IDA procure por strings até dentro de funções. Isto pode aumentar os falsos positivos, mas também pode resolver o problema do IDA não encontrar certas strings misturadas com código.
- **Estruturas e enumerações**: nas janelas *Structures e Enums* são encontradas as estruturas e enumerações identificadas automaticamente pelo IDA. Estas estruturas são identificadas através da análise do código, biblioteca de assinaturas, ou pelo uso de determinadas variáveis em chamadas a funções de bibliotecas. É possível também criar novas estruturas ou enumerações conforme a necessidade.
- **Debugger**: permite executar de forma controlada o programa analisado, mas não esqueça de que ao executar um programa malicioso, a máquina poderá ser comprometida.

- **Referências cruzadas**: janela que pode ser acessada clicando em um objeto, nome ou função dentro da listagem disassembly, e pressionando Control+X.
- **Scripts**: podem ser acessados através do menu "File->IDC File" ou do toolbar correspondente. Ao executar um script, o output dele será enviado para a área de mensagens. Algumas vezes, o script pode pedir a intervenção do usuário, outras vezes, ele adicionará uma função ao IDA Pro.

Como exemplo, a execução do script "C:\Program Files\Ida Free\IDC\call_count_prefixer.idc" o fará escanear todo o banco de dados do programa analisado, e renomear as funções colocando como prefixo o número de vezes que a função é chamada; desta forma se pode saber as funções mais importantes e que devem ser analisadas primeiro.

Atalhos

Atalhos e navegação:

- '**R'**: transforma valor hexa em caractere.
- **'H'**: transforma valor hexa em decimal.
- **'N'**: renomeia objeto atual.
- **'X'**: mostra referências de código.
- **F12**: mostra gráfico de controle de fluxo.
- **Ctrl+F12**: mostra gráfico de chamada de função.
- **Enter**: segue uma referência (duplo clique).
- **ESC**: retorna para a referência anterior.
- **: (dois pontos)**: adiciona comentário.
- **; (ponto-vírgula):** adiciona comentário repetitivo.

O mais importante a lembrar aqui são os atalhos para navegação no código. ENTER em cima de um XREF faz com que a listagem disassembly pule para aquela referência, e ao pressionar ESC, o IDA retorna para o ponto anterior. Com isso, você pode navegar rapidamente por uma sequência de chamadas a funções, e retornar ao ponto inicial com facilidade.

Ao analisar uma função e entender seu funcionamento, você poderá renomeá-la, e assim identificá-la rapidamente depois, procedimento que deve sempre ser usado.

- **CTRL+X:** mostra referências à função corrente.
- **CTRL+UP/DOWN:** para navegar sem perder a marcação de highlight de texto.
- **CTRL+LEFT/RIGHT:** parar pular entre itens.
- **SHIFT+ENTER:** para marcar o item atual.
- **ALT+UP/DOWN:** para encontrar a próxima ocorrência do item atual.
- **ALT+M:** marca uma posição.
- **CTRL+M:** pula para a posição marcada.
- Para apagar marcas:
  - No menu "Jump -> Clear Mark", escolha a marca a apagar.

Algoritmo de autoanálise

Algoritmo de autoanálise do IDA:

1. Carregue o arquivo e crie os segmentos.
2. Adicione o ponto de entrada e as DLLs exportadas à fila de análise.
3. Encontre todas as sequências de código e marque-as.
4. Pegue um endereço da fila para "desassemblar", adicionando na fila as referências de código.
5. Enquanto a fila não estiver vazia, repita o procedimento anterior.
6. Analise uma última vez todo o código, convertendo para dado o que não foi analisado no segmento de texto.

Agora que você já conhece um pouco sobre a interface do IDA, vamos entender o que é feito no momento em que um programa é carregado pelo IDA para ser analisado. Deve ser escolhido um programa para ser carregado, como o arquivo "\Desktop\Exemplos\Sessao 3\mal.Down.nopack.worm.exe". Ao escolher um programa para ser carregado, poderá ser visualizada uma tela de configuração do arquivo. Ao pressionar OK, o IDA irá proceder com a autoanálise do arquivo.

No primeiro passo, o arquivo é carregado na base de dados e os segmentos são criados, de acordo com o descrito no cabeçalho do arquivo.

A seguir, o *entry point* do programa e todas as DLLs requeridas são adicionados à fila de análise. Em seguida, o IDA encontra todas as sequências de código dentro do arquivo, e seleciona-as para análise.

O próximo passo inicia um loop. Uma instrução é retirada da fila, e o fluxo de instruções a partir dela é analisado, até que não seja mais possível encontrar uma instrução executável. Por exemplo, a partir do *entry point*, o IDA examinará a sequência de instruções seguintes, até uma instrução de saída ou de retorno. Após isso, se não houver mais código, ele pega outra instrução da fila, e começa novamente. Se for feita uma referência de fluxo a uma posição de memória que não tenha sido marcada como código, o IDA adiciona essa posição à lista de instruções a analisar. Ao final, o código inteiro é analisado mais uma vez, e aquilo que não for identificado é marcado como dado.

Ao carregar o arquivo, você poderá acompanhar o estado da autoanálise através da área de mensagens do IDA, e da modificação na barra de navegação, que passará a mostrar novas cores dependendo do que for identificado.

Gráficos

Gráficos de chamadas:

- Os programas analisados podem ser visualizados como gráficos.
- A função são os nós, e as chamadas da função são as arestas.
- Útil para visualizar relações entre funções.

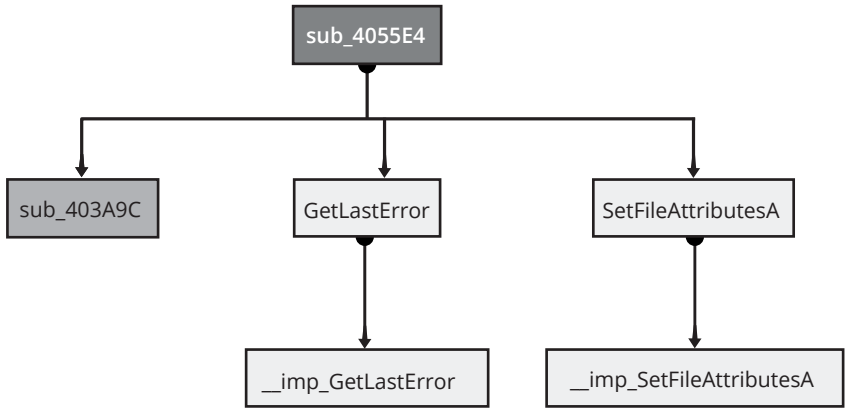


Figura 3.2
Gráficos de chamadas

Os gráficos de chamada são muito úteis para visualizar informações de fluxo dentro do programa. As informações de fluxo servem para encontrar relações entre funções diferentes, e descobrir pontos importantes no programa analisado.

No IDA Pro comercial, a partir da versão 5, existe um modo de visão disassembly que já incorpora uma visão gráfica, sem a necessidade de gerar o gráfico toda vez.

Nestes gráficos, os nós são as funções, e as arestas são as chamadas entre as funções.

- Abra o arquivo:
  \Desktop\Exemplos\Sessao3\mal.Down.nopack.worm.exe
- Dê duplo clique em uma função na lista de funções.
- Pressione Control-Shift-T para gráficos até a função atual.
- Control-Shift-F para gráficos a partir da função atual.

Para visualizar um gráfico de chamada de funções, abra o arquivo indicado e escolha uma das funções identificadas pelo IDA, na janela *Funções*, de preferência alguma identificada pelo nome como "sub_xxxxxx", sendo "xxxxxx" qualquer número. Clique uma vez sobre o nome da função na listagem disassembly.

Após isso, basta pressionar Control+Shift+T para gerar um gráfico de chamadas a partir do *entry point* até a função atual. Se o atalho não estiver configurado, basta clicar no ícone identificado ao lado do texto acima.

Para gerar um gráfico de chamada a partir da função atual, basta pressionar Control+Shift+F ou pressionar o ícone correspondente.

- Gráficos de fluxo de controle:
  - Funções podem ser visualizadas como gráficos.
  - Blocos básicos são representados como nós.
  - Os caminhos lógicos são representados pelas arestas.
- Procure a função sub_403D70:
  - CTRL+P e digite o nome da função.
- F12 para ver gráfico de fluxo.

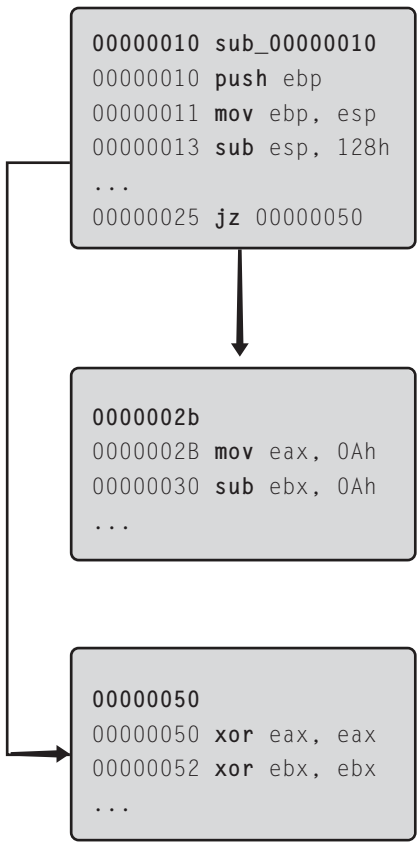


Figura 3.3
Gráficos de fluxo de controle.

Outra funcionalidade importante para o analista é poder visualizar, dentro de uma função, como se comporta o fluxo de controle do programa.

Saltos condicionais, como blocos *if/then/else* ou *switch/case* podem ser difíceis de identificar dentro da listagem assembly. Por isso, o IDA Pro permite visualizar uma função como um gráfico de fluxo de controle.

Neste gráfico, blocos básicos são representados pelos nós, e as arestas indicam os caminhos lógicos possíveis. Linhas verdes indicam o caminho seguido quando a condição for verdadeira, e linhas vermelhas quando a condição for falsa.

Um bloco básico é a maior sequência de instruções executadas antes de um salto condicional.

Como um pequeno exercício, você pode usar um atalho para procurar na lista de funções pela função *sub_403D70*. Pressionando Control+P, uma janela é aberta com as funções reconhecidas. Nesta tela, bem como em outras parecidas, basta digitar o nome da função, e a lista será percorrida até o cursor parar na função desejada.

Ao encontrar a função, basta clicar duas vezes sobre ela, e uma vez sobre o nome da função na listagem disassembly. Após isso, basta pressionar F12 para visualizar o gráfico.

Signatures e Type Library

- Muitas vezes, funções dentro do programa podem pertencer a bibliotecas padrão de compiladores ou do sistema.
- O IDA Pro usa bibliotecas de assinaturas para identificar essas funções.
- Algumas vezes, o IDA reconhece erroneamente o compilador usado, e acaba não identificando corretamente essas funções.

Muitas vezes, ao compilar um programa, o autor pode optar por embutir nele as DLLs necessárias para seu funcionamento. Ou seja, ele compila estaticamente as bibliotecas necessárias.

Além disso, dependendo do compilador usado, diversos módulos característicos desse compilador também podem ser embutidos no programa. Por exemplo, em linguagens visuais como Visual C++ ou Delphi, as funções para tratamento dos objetos visuais, como formulários e botões, são padronizadas.

Ao analisar um programa suspeito, identificar corretamente o compilador usado pode oferecer um grande auxílio ao analista, que pode então procurar por funções padronizadas e eliminá-las da lista a ser analisada.

O IDA Pro tem duas funcionalidades muito importantes para auxiliar o analista nesta tarefa. A primeira delas é a biblioteca de tipos. Uma biblioteca de tipos é um conjunto de assinaturas que auxiliam o IDA a identificar o compilador usado no programa, e na identificação de variáveis padrão, métodos de chamada, funções padrão etc.

Signatures (ou assinaturas de função) são bibliotecas de assinaturas que auxiliam o IDA a reconhecer funções padrão de bibliotecas. Ele realiza essa tarefa através da pesquisa por hash de funções existentes. Se uma função desconhecida tem o mesmo hash de uma função de uma biblioteca do sistema, então elas possivelmente são a mesma função.

Dessa forma, o IDA consegue reconhecer automaticamente partes do código que de outro modo permaneceriam desconhecidas, e caberia então ao analista identificar a funcionalidade daquela função. Algumas vezes, porém, o IDA reconhece erroneamente o compilador, e não aplica corretamente as assinaturas. Veremos a seguir um exemplo disso.

Veja como a barra de navegação muda, quando as assinaturas são aplicadas:

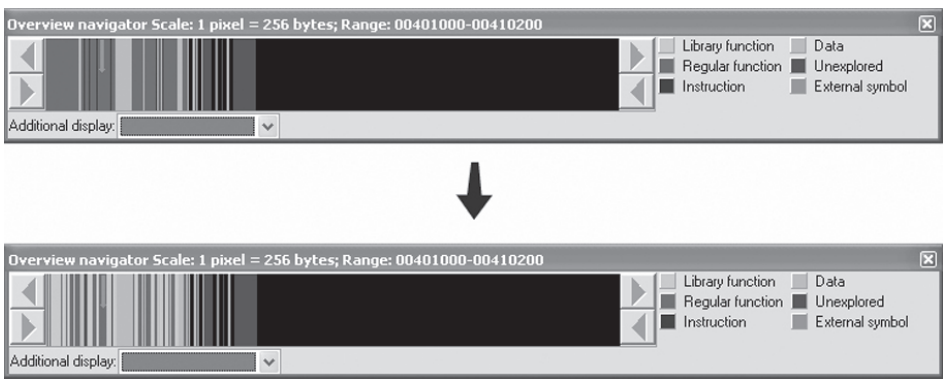


Figura 3.4
Mudanças na barra de navegação.

Finalmente, após todas as assinaturas aplicadas, você vai perceber que muitas funções que antes estavam marcadas como desconhecidas, agora estão marcadas como pertencendo às bibliotecas do Borland. Isto facilita a análise, pois agora não é mais preciso analisar essas funções.

Demonstraremos a seguir a utilidade das funções de reconhecimento automático de padrões no IDA. As assinaturas podem ser encontradas em diversas fontes pela internet ou desenvolvidas pelo próprio analista, permitindo assim expandir a funcionalidade do IDA.

Introdução ao OllyDBG

- Freeware.
- Contém diversas funcionalidades *cracker friendly.*
- Expansível através de diversas linguagens.

- Controlável através de scripts OllyScript.
- Contém um disassembler poderoso.
- Há centenas de funções escondidas.
- A documentação existente é excelente.

Cuidado: o programa carregado é colocado em estado de execução pausada, e qualquer erro pode comprometer seu sistema!

Veremos a partir de agora outra ferramenta muito útil na análise de programas maliciosos. O OllyDBG é um debugger e disassembler, distribuído livremente, que contém uma grande quantidade de funcionalidades úteis para a análise.

O Olly é uma ferramenta *cracker friendly*, pois suas funcionalidades são especialmente desenvolvidas para facilitar a engenharia reversa de programas protegidos. Com a possibilidade de expandir seu funcionamento através de scripts e plugins, assim como o IDA, o Olly complementa as características do IDA. O conjunto das duas ferramentas atende a praticamente qualquer necessidade que o analista possa ter durante uma análise.

Uma característica do Olly é que a maioria das funções não são acessíveis pelo menu do programa. Menus contextuais no botão direto contêm muitas outras funcionalidades, assim como atalhos de teclado acessam outras tantas funcionalidades inacessíveis de outro modo. Felizmente, a documentação existente é excelente e o analista poderá aprender muito com ela.

Um ponto a ser destacado é que o Olly, ao abrir um executável para análise, diferentemente do IDA, coloca o programa em execução e pausa a execução na primeira instrução a ser executada. Isto significa que o programa vai estar carregado em memória, pronto para ser executado. Qualquer ação errada por parte do analista pode colocar o programa em execução, e comprometer a máquina, por isso tome cuidado.

Conhecendo o OllyDBG

Ao abrir um programa para análise, esta é a janela principal do Olly (Figura 3.5). Basicamente nessa tela estão as principais informações que o analista precisa saber durante a análise. Outras janelas podem ser abertas através do menu *View.*

Janela principal da CPU:

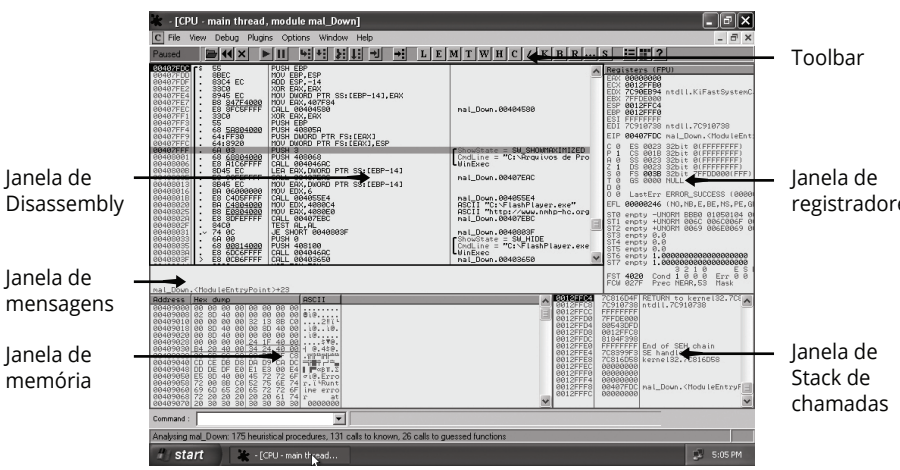


Figura 3.5 OllyDBG.

Na parte superior, há uma pequena toolbar, com botões para abrir novas janelas e para a execução controlada do programa.

Abaixo, à esquerda, temos a janela de disassembly, dividida em colunas. Da esquerda para a direita:

- Endereço da instrução;
- Bytes relativos à instrução;
- Instruções assembly e parâmetros;
- Comentários, referências, tipos de parâmetros etc.

À direita da janela de disassembly, temos a primeira mudança em relação ao IDA. Nesta tela, pelo fato do programa ser colocado em modo de execução pelo Olly, ele pode mostrar o estado atual dos registradores e flags da CPU. Isto só é possível durante a execução. É recomendada a pesquisa na seção de ajuda do Olly para saber o significado de cada um desses campos. É importante conhecer bem pelo menos os registradores (nove primeiros valores no topo dessa janela) e as flags da CPU (CPAZSTDO, a seguir na janela), pois são importantes durante a análise.

Abaixo da janela de disassembly, há uma pequena janela de mensagens, que serve também para mostrar o estado de variáveis monitoradas (watches). Abaixo, a representação da área de memória do programa. Através dessa janela, é possível navegar na memória do programa em tempo real. Isto é importante para monitorar áreas de memória que estejam sendo modificadas pelo programa durante a execução.

À direita, uma área representa o Stack de chamadas do programa. Outra informação importante é a estrutura de Structured Exception Handling (SEH), que será vista adiante.

Exercício de fixação 2
OllyDGB

Como o OllyDBG armazena as informações obtidas durante a análise?

_____

_____

Funcionalidades do OllyDBG

Módulos principais:

- Disassembly comentável.
- Enumeração de argumentos.
- Registradores de CPU.
- Pilha da CPU.
- Dump da memória.
- Janela de log.
- Pilha de chamada.
- Módulos executáveis.
- Mapa de memória.
- Threads.
- Breakpoints.
- Bookmarks.

- Cadeia de SEH.
- Expressões monitoradas.
- Handles.
- Plugins.
- Call Stack Graph.

Abra o arquivo "\Desktop\Exemplos\Sessao 3\mal.Down.nopack.worm.exe" novamente, para usar como exemplo.

Os sete primeiros itens da lista acima são as janelas padrão mostradas na tela principal. Aqui, como no IDA, o disassembly pode ser comentado, tanto automaticamente pelo Olly, como manualmente pelo usuário.

Sempre que possível, o Olly tenta também identificar parâmetros e anotá-los no disassembly, como pode ser visto nas chamadas ao WinExec no início do programa.

O log pode ser acessado clicando-se no botão 'L' na toolbar.

Os módulos executáveis são todas as DLLs e demais executáveis usados pelo programa. Podem ser acessados clicando no botão 'E' na toolbar.

O mapa de memória, que mostra a estrutura das seções do programa na memória, pode ser acessado com o botão 'M'.

Os threads criados pelo programa podem ser vistos com o botão 'T'.

Os breakpoints marcados no programa, para controlar a execução do mesmo, podem ser vistos clicando-se no botão 'B'.

Os bookmarks são marcações feitas pelo analista para relembrar de posições importantes no código. Eles podem ser acessados através do plugin *Bookmarks* (Plugins → Bookmarks → Bookmarks).

A cadeia de Exception Handling pode ser vista clicando-se em "View->SEH chain".

Expressões monitoradas podem ser visualizadas na janela de log da tela principal, ou na tela de watches acessível pelo menu "View → Watches".

Os handles mostram os arquivos mantidos abertos pelo programa, incluindo arquivos normais, diretórios, pipes, sockets, conexões de rede etc. Eles podem ser acessados pelo botão 'H' na toolbar.

Os plugins instalados nesta versão do Olly incluem não somente os padrões, mas também um grande conjunto de plugins encontrados na internet em sites especializados no Olly.

Um plugin interessante é o plugin OllyFlow, capaz de gerar diversos gráficos, como no IDA. Mas além dos tipos de gráfico do IDA, um tipo especial que o Olly disponibiliza é o *Call Stack Graph*. Este gráfico mostra as modificações feitas no Stack de uma função para enviar parâmetros a ela, isto é, com ele é possível descobrir rapidamente quais instruções no código foram usadas para passar parâmetros à uma função.

Plugins

O OllyDBG tem um suporte excelente a plugins:

- OllyDump

- OllyScript
- OllyFlow
- Hide Debugger
- Heap Vis
- Breakpoint Manager

O suporte a plugins do Olly é muito bom, sendo que diversos estão instalados na versão distribuída na máquina virtual:

- **OllyDump**: exporta o processo atual para disco, reconstruindo a IAT;
- **OllyScript**: permite a execução de scripts externos;
- **OllyFlow**: gera gráficos de funções, como no IDA Pro;
- **Hide Debugger**: implementa diversas funções para esconder o OllyDBG do programa analisado;
- **Heap Vis**: enumera e procura por heaps do processo atual;
- **Breakpoint Manager**: importa e exporta marcações de breakpoint.

A lista acima mostra apenas alguns dos plugins disponíveis. O OllyDump é um plugin importante, pois permite copiar para disco a imagem de um programa em memória. Muitas vezes, os programas maliciosos são compactados ou criptografados, e fica impossível analisá-los diretamente. Neste caso, é necessário executar o programa dentro de um ambiente controlado, e após a rotina de descompactação/decriptografia, usar o plugin OllyDump para copiar o novo executável em memória para o disco. Adiante veremos como usar essa funcionalidade para descompactar um programa desconhecido.

O OllyScript é o plugin que permite a execução de scripts externos para executar diversas funções. Juntamente com a distribuição do Olly, encontraremos centenas de scripts para descompactar ou desproteger alguns tipos de sistema de proteção comumente usados por programas maliciosos, em "\Desktop\Install\OllyDbg\OllyScripts\".

Atalhos

- **F9:** executa o programa atual.
- **CTRL+F9:** executa até retornar.
- **ALT+F9:** executa até código de usuário.
- **F12:** pausa.
- **F7:** step into.
- **F8:** step over.
- **F2**: marca/apaga breakpoint.
- **CTRL+G**: vai para endereço.
- **Qualquer tecla**: modifica assembly.
- **: (dois pontos):** adiciona label.
- **; (ponto e vírgula):** adiciona comentário.
- **CTRL+F7:** animate.
- **ENTER:** segue um *jump.*

Aqui estão alguns atalhos úteis durante o uso do OllyDBG. O Help do programa disponibiliza uma lista completa de atalhos, além de algumas funções interessantes.

Roteiro de Atividades 3

Atividade 3.1 – Signatures e Type Library

O objetivo desta atividade é verificar como o reconhecimento correto do compilador usado pode auxiliar na análise. Como exemplo, utilizaremos o mesmo programa malicioso dos exemplos anteriores: "\Desktop\Exemplos\Sessao 3\mal.Down.nopack.worm.exe".

1. Com o arquivo "\Desktop\Exemplos\Sessao 3\mal.Down.nopack.worm.exe" aberto, examine a barra de navegação para verificar o que foi reconhecido. Examinando as strings, podemos ver algumas referências ao Borland Delphi no código do programa.
2. Abra a janela "Type Libraries" (View → Open Subviews → Type Libraries), observando que o compilador identificado é "vc6win".
3. Abra a janela "Signatures" (View → Open Subviews → Signatures) e adicione as assinaturas que julgar pertinentes, observando aquelas que identificam funções de bibliotecas.

Observe que as partes azuis representam quase a totalidade do código, e devem ser analisadas, pois representam funções desconhecidas.

Ao adicionar as assinaturas, perceba que enquanto elas forem aplicadas ao código, a janela vai indicar quantas funções são reconhecidas pelas assinaturas. Ao mesmo tempo, a barra de navegação vai mostrar as funções identificadas como pertencendo a uma biblioteca, mudando as cores de azul escuro para azul claro.

Pesquise na internet o modo como essas assinaturas são criadas, e outras vantagens existentes no uso de assinaturas. Faça um relatório descrevendo as páginas que encontrar sobre o assunto, explicando a razão de terem sido consideradas interessantes.

Atividade 3.2 – IDA Pro

Abra o arquivo "\Desktop\malware\mal.Down.nopack.winlogin.exe" no IDA Pro, navegue pelas funcionalidades que acabamos de estudar no módulo teórico e responda as seguintes questões:

1. Na janela de funções exportadas existe alguma função? Caso positivo, qual o endereço desta função?

2. Navegue pela janela de strings e encontre a string "task32.exe". Qual o endereço da função que faz referência a esta string?

Atividade 3.3 – OllyDBG

1. Abra "\Desktop\Exemplos\Sessão 3 \mal.upx.6dddd7e8e5ff88a15b7884a833ff893b.dat";
2. Vá para 0x50EDB6h (Control+g) e marque um breakpoint (F2);
3. Encontre o endereço de memória onde o binário vai ser descompactado, e vá para este endereço no dump de memória (Control+g);
4. Execute com F9 até breakpoint;
5. CTRL+F7 para executar Animate;
6. Clique no botão de Pause (**||**) para parar ou no botão **<<** para restaurar o programa para o estado inicial.

Nesta atividade, conheceremos uma funcionalidade interessante do Olly, e começaremos a entender o processo de análise de um binário suspeito.

O arquivo que usaremos como exemplo é "\Desktop\Exemplos\Sessão 3\mal.upx.6dddd7e8e5ff88a15b7884a833ff893b.dat". Este arquivo está compactado com UPX, um compactador muito popular em arquivos maliciosos. Este compactador instala uma rotina de descompactação em uma área não usada pelo programa, e modifica o *entry point* do programa para realizar um desvio para essa função assim que o programa é iniciado. Esta rotina de descompactação grava o programa descompactado em outra área de memória, já reservada no cabeçalho do executável para esta finalidade.

Veja o arquivo "\Desktop\Install\Unpacking\OpenRCE UPX Notes.txt" para consultar um pequeno roteiro de como descompactar arquivos UPX.

De acordo com o arquivo TXT indicado acima, um arquivo compactado com UPX tem no seu *entry point* o seguinte conteúdo:

```
60                          PUSHAD
BE   [4 Bytes]          MOV ESI, [ADDRESS]
8DBE [4 bytes]          LEA EDI, DWORD PTR DS:[ESI+Value]
57                          PUSH EDI
83CD FF                 OR EBP, FFFFFFFF
EB 10                   JMP SHORT [Relative Jump]
90                          NOP
```

Leia o documento descrevendo o funcionamento do compactador UPX e tente descobrir o endereço de memória que deve monitorar para ver o código descompactado.

Agora veremos o OllyDBG executando cada linha do código (uma a uma), de forma animada (uma instrução a cada 0,25 segundos, aproximadamente), e ao mesmo tempo, as modificações sendo gravadas na memória na posição indicada.

Pressione F9 para executar o programa até o breakpoint marcado. Após isso, basta pressionar Control+F7 para iniciar a animação. O aluno poderá acompanhar a descompactação dos bytes do programa pela área de memória descoberta. Esta funcionalidade é muito útil durante a análise de um binário desconhecido, para a identificação de partes do código executadas em um loop.

4

Formato de arquivos executáveis

objetivos

Apresentar o formato PE e mostrar como extrair informações relevantes das seções, tabela de importação e cabeçalhos para a análise estática do arquivo.

conceitos

Formato e cabeçalho PE, tabela de importação, cabeçalho de seção.

Exercício de fixação 1
Arquivos executáveis

Todo arquivo binário é um arquivo executável? E todo arquivo executável é um arquivo binário?

Formato de arquivos executáveis

- Para um programa ser executado em um sistema operacional, ele deve obedecer a certos formatos padronizados.
- Dessa forma, o sistema sabe onde encontrar o código, recursos, bibliotecas, e como mapear tudo isso no espaço de memória reservado ao programa.
- Alguns formatos de executáveis:
  - Portable Executable PE (Windows, DOS).
  - ELF (Linux, Unix).
  - ABI Mach-O (MacOS X).

Neste capítulo, conheceremos um pouco sobre o formato usado por arquivos executáveis. Estudaremos rapidamente a estrutura do formato ELF do Linux, mas vamos nos concentrar no formato PE, usado pelos executáveis do Windows.

Um formato de arquivo executável descreve a estrutura dos dados e códigos de um programa, de forma a permitir ao sistema operacional identificar cada parte do programa, alocando o espaço necessário em memória e mapeando a estrutura física do arquivo na memória virtual, iniciando a execução do programa a partir do ponto de entrada do código.

Os formatos executáveis descrevem então como organizar a informação dentro de um arquivo, descrevendo onde ficam as áreas de código, de recursos, as bibliotecas que devem

ser carregadas junto com o programa, o endereço inicial de execução (entry point) e a quantidade de memória que deve ser reservada ao programa, entre outras informações.

Os formatos PE, ELF e ABI Mach-O descrevem como deve ser um executável em ambientes Windows, Linux/Unix e Mac OS X, respectivamente.

Exercício de nivelamento 1
Execução de código

Em que campo está localizado o endereço do início da execução do código?

Qual o cabeçalho que contém informações relacionadas às seções do executável?

Formato PE

- Formato de arquivos Portable Executable (PE):
  - A Microsoft baseou o formato PE no antigo formato COFF do Unix.
- Por que "portável"?
  - Suporta 32 bits e 64 bits.
  - Suporta arquiteturas MIPS, PowerPC, DEC Alpha e ARM.
- Arquivos PE:
  - Extensão *.exe* ou *.dll*

O formato PE é o principal formato de executáveis usado em sistemas Windows. Ele foi baseado no antigo formato COFF usado em versões mais antigas do Unix. Outros formatos suportados no Windows são os formatos DOS (formato de executável de 16 bits do antigo DOS, reconhecido pela string "MZ" no começo do arquivo), COM (executável de 16 bits do DOS com extensão COM) e NE (New Executable, formato usado pelo DOS Multitasking, que usualmente pode ser executado como 32 bits).

O formato PE é considerado portável, pois pode ser utilizado tanto por arquivos de 32 bits como de 64 bits, e pode ser executado sem modificações em outras arquiteturas de processador, como MIPS, PowerPC, DEC Alpha e ARM.

Uma característica muito conhecida desse formato, e que causa muitos problemas para os usuários mais leigos, é que ele é reconhecido pelo sistema operacional pela sua extensão, que pode ser .EXE ou .DLL. Muitas vezes, um usuário malicioso disfarça o fato de um programa ser executável, mudando sua extensão para outra que não seja EXE, ou adicionando uma extensão extra, levando a vítima a acreditar que o programa não passa de um arquivo comum, como por exemplo nomeando um arquivo como *fotos.jpg.exe*.

Estrutura do cabeçalho PE

- Inicia com a assinatura PE:
  - E_magic=4D5Ah 'MZ'
- Cabeçalho NT compreende o cabeçalho FILE e o OPTIONAL
  - Assinatura = 5045h 'PE'

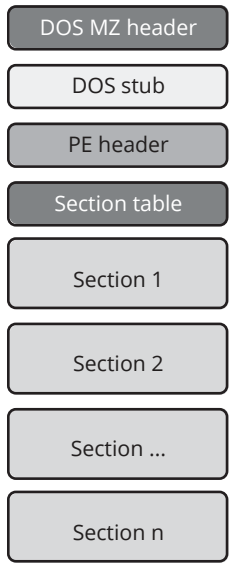


Figura 4.1
Formato de arquivos Portable Executable (PE).

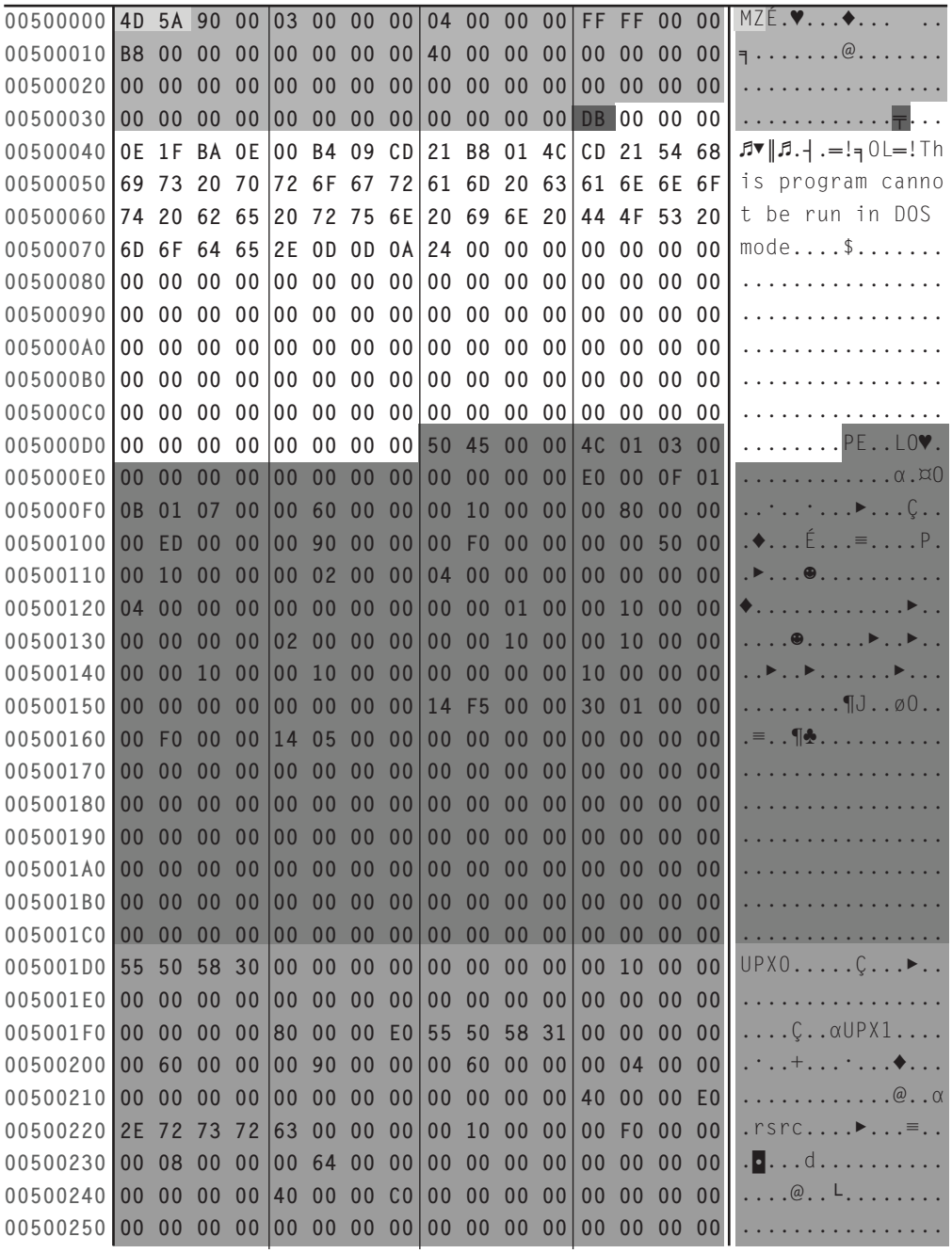
A Figura 4.1 representa a estrutura do cabeçalho de um arquivo PE. A primeira coisa a se notar é a assinatura de um arquivo executável. Em sistemas Windows, arquivos executáveis sempre começam com a string "MZ". Este fato permite identificar um arquivo executável mesmo que ele não esteja com a extensão certa, ou também identificar executáveis embutidos dentro de outros arquivos.

A seguir, temos o *stub* DOS, que é o código responsável por imprimir a mensagem "This program cannot be run in DOS mode".

O campo a seguir é o cabeçalho NT ou cabeçalho PE propriamente. Ele é sempre iniciado pela string "PE". É nele que ficam contidas as informações sobre o executável, incluindo endereço do *entry point*, endereço das tabelas de importação e exportação, número de seções, entre outras informações.

A última parte do cabeçalho PE descreve as seções existentes no binário. Cada descrição contém o nome da seção, tamanho real (em disco) e virtual (em memória), realocações (se forem necessárias), atributos da seção (leitura/escrita, executável, contém código etc).

Estas características permitem ao analista identificar alguns casos onde o binário foi modificado, como por exemplo, se foi usado um compactador. Estes programas costumam compactar somente as seções de código, e criam seções extras para armazenar o código descompactado. Estas seções extras geralmente têm tamanho 0 em disco, e um tamanho virtual grande o suficiente para armazenar o código descompactado. Outras características dessas seções vazias é que elas geralmente têm o atributo de execução setado, pois após a descompactação, o controle do programa passará ao código armazenado ali.



00500000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
00500010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
00500020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00500030	00	00	00	00	00	00	00	00	00	00	00	00	DB	00	00	00
00500040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
00500050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
00500060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
00500070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
00500080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00500090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005000D0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	03	00
005000E0	00	00	00	00	00	00	00	00	00	00	00	00	E0	00	0F	01
005000F0	0B	01	07	00	00	60	00	00	00	10	00	00	00	80	00	00
00500100	00	ED	00	00	00	90	00	00	00	F0	00	00	00	00	50	00
00500110	00	10	00	00	00	02	00	00	04	00	00	00	00	00	00	00
00500120	04	00	00	00	00	00	00	00	00	00	01	00	00	10	00	00
00500130	00	00	00	00	02	00	00	00	00	00	10	00	00	10	00	00
00500140	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00	00
00500150	00	00	00	00	00	00	00	00	14	F5	00	00	30	01	00	00
00500160	00	F0	00	00	14	05	00	00	00	00	00	00	00	00	00	00
00500170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00500180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00500190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005001D0	55	50	58	30	00	00	00	00	00	00	00	00	00	10	00	00
005001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
005001F0	00	00	00	00	80	00	00	E0	55	50	58	31	00	00	00	00
00500200	00	60	00	00	00	90	00	00	00	60	00	00	00	04	00	00
00500210	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	E0
00500220	2E	72	73	72	63	00	00	00	00	10	00	00	00	F0	00	00
00500230	00	08	00	00	00	64	00	00	00	00	00	00	00	00	00	00
00500240	00	00	00	00	40	00	00	C0	00	00	00	00	00	00	00	00
00500250	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Na Figura 4.2 podemos ver como se parece o início de um arquivo PE quando visto em um editor hexadecimal. Podemos ver a assinatura de arquivo executável "MZ" logo no começo do dump, seguida pelo cabeçalho DOS. O campo em vermelho indica o Offset, a partir do começo do arquivo, do início do cabeçalho PE. Logo após está o *stub* DOS com a mensagem impressa quando o programa é executado em DOS.

Figura 4.2
Arquivo PE quando visto em um editor hexadecimal.

O cabeçalho PE inicia com a string "PE" e no fim vemos a região que descreve as seções do programa. Neste caso podemos notar pelas seções que se trata de um arquivo compactado com o compactador UPX, um programa muito utilizado para compactação de executáveis de programas maliciosos.

Cabeçalhos do formato PE

- Cabeçalhos DOS e NT:
  - O arquivo PE começa com o *stub* DOS, usualmente responsável pela mensagem "*This program cannot be run in DOS mode*", e por apontar para o cabeçalho NT (Offset to PE signature).
- São necessárias no mínimo duas seções em um arquivo PE: uma para dados e uma para código.
  - Podem existir mais seções, dependendo do tipo de arquivo, sendo as mais comuns:
    - .text: código executável.
    - .data, .rdata ou .bss: dados.
    - .rsrc: seção de recursos.
    - .idata, .edata: importação e exportação de funções.

Como vimos, o cabeçalho de um arquivo PE está dividido em três partes:

- Cabeçalho DOS, responsável por imprimir a mensagem quando o arquivo é executado em DOS, e por indicar o offset do cabeçalho PE;
- O próprio cabeçalho PE;
- Os cabeçalhos de seção, que descrevem as características das sessões existentes.

A seguir são mostrados alguns exemplos de seções padrão que podem existir em um executável. Muitas vezes, quando um binário é compactado ou criptografado com um compactador de executáveis, ele pode modificar o número ou alguma característica das sessões, e por isso você deve ficar atento a essas informações quando for analisar um arquivo.

Seções mais comuns:

- .text – código executável;
- .data, .rdata ou .bss – dados; normalmente BSS é usado para armazenar dados dinâmicos durante a execução, por exemplo na descompactação do binário.
- .rsrc – seção de recursos; programas maliciosos costumam guardar parte do código ou tabelas de importação aqui, para dificultar a análise dos binários.
- . idata, .edata – importação e exportação de funções.

- Cabeçalhos NT: cabeçalho de arquivo.
  - Primeiro cabeçalho NT, logo após a assinatura PE.
- Contém alguns campos interessantes:
  - **Machine**: indica o tipo de arquitetura.
  - **NumberOfSections**: o número de seções no executável.
  - **TimeDateStamp**: não é crítico, mas pode indicar a data em que o arquivo foi criado.
  - **SizeOfOptionalHeader**: indica o tamanho exato do cabeçalho opcional.

A primeira parte do cabeçalho NT descreve características gerais do binário, como tipo de arquitetura para o qual ele foi compilado, número de seções existentes, timestamp da criação do arquivo e tamanho do cabeçalho PE.

O timestamp existente aqui pode ser uma indicação de quando o binário foi compilado, e por isso os autores dos programas maliciosos costumam zerar esse valor, para apagar seus rastros.

O tamanho do cabeçalho opcional indica o tamanho em bytes do restante do cabeçalho, a partir da assinatura PE, até o início dos cabeçalhos de seções.

- **Cabeçalho NT:** cabeçalho Optional.
- Assinatura = 10 Bh (*magic number* PE32)
- **AddressOfEntryPoint:** endereço de início da execução.
  - É possível que outros códigos sejam executados antes do ponto de entrada.
- **ImageBase:** endereço usado como base para todos os endereços relativos no código.
  - É possível algumas vezes encontrar o cabeçalho PE neste endereço na memória.
- **SizeOfHeaders**: indica o tamanho do cabeçalho completo, a partir do início do arquivo, inclusive dos cabeçalhos de seções.
- **SectionAlignment:** alinhamento das seções na memória.
- **FileAlignment:** alinhamento das seções no disco.

No cabeçalho opcional estão localizadas a maioria das informações sobre o executável. Um dos principais campos desse cabeçalho é o endereço do *entry point* do programa. Este é o endereço da primeira instrução que será executada após o programa ser carregado na memória. Ou pelo menos, esta é a descrição "oficial" do formato PE.

Durante a análise de um programa malicioso o analista deve tomar muito cuidado, pois existem casos onde um código pode ser executado ANTES do sistema transferir o controle de execução para a instrução no *entry* point, ou antes do debugger paralisar a execução do programa. Em capítulo posterior, analisaremos exemplos destes casos, mas por ora devemos apenas ficar conscientes de que mesmo executando o programa em um ambiente controlado, não se deve jamais confiar que o sistema está seguro e que o programa malicioso não foi executado.

Seguindo com a descrição dos campos mais importantes do cabeçalho opcional, a seguir temos o campo que indica o endereço base a partir do qual os offsets fazem referência na imagem. É neste endereço que a imagem do programa será carregada na memória virtual, e onde podemos encontrar o início do cabeçalho PE do programa carregado. É possível conferir isso, examinando no Olly o valor indicado em *ImageBase* e depois procurando pelo endereço do início do cabeçalho do executável.

O campo *SizeOfHeaders* indica o tamanho completo dos cabeçalhos incluindo os cabeçalhos de seções.

Os campos que indicam o alinhamento das seções permitem descobrir quanto de espaço deve realmente existir entre as seções. Em casos raros, um programa malicioso pode utilizar esse espaço entre seções para armazenar dados ou código. Por exemplo, se uma seção tem 500 bytes e o alinhamento de seções é 1000, a próxima seção só irá começar a partir do endereço 1000, e não do endereço 500.

- Cabeçalho NT: Cabeçalho Optional.
  - Export Table Address/Size.
    - Indica o endereço da tabela de exportação; normalmente usada em DLLs.
  - Import Table Address/Size.
    - Indica o endereço da tabela de importação.
  - Resource Table Address/Size.
    - Endereço da área de recursos, onde são armazenados ícones, imagens, strings etc.
  - TLS Table Address/Size.
    - Endereço da tabela de chamadas da Thread Local Storage.

Abaixo estão descritas algumas das principais tabelas existentes no cabeçalho PE. Essas tabelas são as mais importantes durante a análise de um binário malicioso:

- **Export Table Address/Size**: indica o endereço da tabela de exportação; normalmente usada em DLLs.
- **Import Table Address/Size**: indica o endereço da tabela de importação. Se o offset indicar um endereço fora da área de código, pode significar que o binário foi modificado para dificultar a análise. A tabela de importação indica as DLLs e funções delas que o programa utilizará, e por isso muitas vezes o autor do programa modifica a localização da tabela, para dificultar a análise.
- **Resource Table Address/Size**: endereço da área de recursos, onde são armazenados ícones, imagens, strings etc. Pode também conter código ou a tabela de importação em binários modificados. A tabela de recursos indica como estão organizados os recursos dentro da seção destinada a eles. A seção de recursos normalmente é usada por autores de programas maliciosos para armazenar código ou dados do programa.
- **TLS Table Address/Size**: endereço da tabela de chamadas da Thread Local Storage. Esta tabela pode indicar funções que serão executadas antes do *entry point* do programa. A tabela de chamadas do TLS (Thread Local Storage) serve normalmente para guardar informações relativas aos threads criados pelo programa, mas devido a sua característica de ser possível criar funções de chamada a TLSs executadas antes do *entry point* do programa, elas são usadas para execução de código malicioso, como por exemplo uma função para descompactar ou decriptografar código, ou mesmo infectar o sistema se o programa for carregado em um debugger.

Tabela de importação

- A função primária da Tabela de Endereços de Importação (IAT) é prover informações para o loader do sistema localizar as funções de API e outros símbolos utilizados por um executável.
- Pode também prover informações sobre as ações que um executável pode tomar.
- Ocultar ou "obfuscar" a IAT é uma técnica comumente utilizada por arquivos maliciosos.
- A IAT pode ser reconstruída por diversas ferramentas.

A Tabela de Endereços de Importação (Instruction Address Table, ou IAT), é usada pelo sistema para saber as DLLs que o programa precisa para funcionar, e as funções dessas DLLs que serão utilizadas.

Estas informações podem ser úteis para a identificação do tipo de comportamento que o programa pode ter. Por isso, normalmente a IAT é modificada ou "obfuscada" pelos autores de programas maliciosos, dificultando o trabalho de análise do programa, mas não o impede de funcionar corretamente. Em alguns casos, o próprio código do programa carrega dinamicamente as DLLs e funções que vai utilizar. Isso pode ser notado em um programa malicioso, pois as únicas funções que ele vai importar serão "KERNEL32.LoadLibraryA" e "KERNEL32.GetProcAddress".

Essas duas funções são utilizadas para carregar uma DLL em memória, e descobrir o endereço base da mesma, a partir do qual o programa poderá acessar as funções que precisar.

Mesmo assim, existem diversas ferramentas que podem ser usadas para recriar a tabela de importação e descobrir as funções que o programa utiliza. Veremos um exemplo disso em capítulo posterior.

Cabeçalho de seção

- **VirtualSize:** tamanho da seção quando carregada na memória.
  - Pode ser maior que *SizeOfRawData*, e neste caso será preenchida com zeros.
- **VirtualAddress:** endereço da seção na memória, relativo a *ImageBase.*
- **SizeOfRawData:** tamanho da seção no disco.
- **PointerToRawData:** offset no arquivo do conteúdo da seção a ser carregado na memória.
- **Characteristics:** contém flags como "execução", "leitura" e "escrita", entre outras.

A última parte do cabeçalho de um executável descreve as seções existentes, onde ficam armazenados os códigos e dados do programa. Existem dois estados para uma seção:

- Estado físico: modo da seção no arquivo físico gravado no disco.
- Estado virtual: modo como a seção vai ser mapeada em memória.

Existem casos em que o tamanho da seção em disco é nulo, e quando a seção é carregada em memória, fica com um tamanho maior. Isto pode ser usado, por exemplo, para indicar uma seção onde serão armazenados dados dinâmicos do programa.

- **VirtualSize** e **VirtualAddress** – campos que indicam o tamanho e o offset em que será carregada a seção a partir do *ImageBase*.
- **SizeOfRawData** – indica o tamanho real da seção em disco.
- **PointerToRawData** – define o offset onde a seção está armazenada a partir do início do arquivo.
- **Characteristics** – campo que indica os atributos da seção, como por exemplo UNINITIALIZED_DATA, EXECUTE, READ, WRITE, entre outros.

Roteiro de Atividades 4

Atividade 4.1 – Navegando pela estrutura de um arquivo PE

1. Execute o OllyDBG;
2. Abra o arquivo:
   \Desktop\Exemplos\Sessao 4\mal.upx.6dddd7e8e5ff88a15b7884a833ff893b.dat
3. Abra a Janela de Memória (Alt-M ou botão M na barra de ferramentas);
4. Identifique a área de memória do programa, e clique duas vezes sobre o item “PE Header”;
5. Examine as estruturas dos cabeçalhos DOS e NT. Você pode alternar entre visualização no formato de cabeçalho e no formato hexadecimal, clicando com o botão direito e escolhendo “Special->PE Header” ou “Hex->Hex/ASCII (16 bytes)”;
6. Faça uma lista dos campos que considera mais importantes durante a análise;
7. Mantenha o Olly DBG aberto para acompanhar o módulo teórico, acompanhando no arquivo os campos explicados pelo instrutor.

Atividade 4.2 – Examinando arquivos PE

1. Examine os arquivos em “\Desktop\malware\”, utilizando o LordPE e o OllyDBG;
2. Navegue pelas informações de cabeçalho dos executáveis;
3. Procure por inconsistências nos campos do cabeçalho que possam dificultar a análise ou indiquem comportamento malicioso ou suspeito;
4. Discuta alguns casos encontrados pela turma que tenham dificultado a análise do programa;
5. Pesquise na internet problemas e soluções relacionadas a mudanças na estrutura de arquivos executáveis;
6. Faça um relatório descrevendo os arquivos analisados e as inconsistências encontradas, indicando se elas dificultaram a análise do arquivo e possíveis soluções para contornar os problemas. Complemente o relatório com as informações encontradas na internet.

Para mais informações sobre o formato PE, consulte os arquivos:

- \Desktop\PE File Format - A Reverse Engineer's View.pdf
- \Desktop\PE Format Poster.pdf

5

Assembly básico – parte 1

objetivos

Apresentar a linguagem assembly e seus pontos principais, como análise de stack e análise de memória, para extrair o máximo de informações em uma análise de código malicioso.

conceitos

Assembly, registradores, stack, mapeamento de memória, structured exception handling.

Exercício de fixação 1
Assembly

Por que devemos entender o assembly para trabalhar com engenharia reversa?

Exercício de nivelamento 1
Identificação de parâmetros

Qual o registrador responsável por armazenar o endereço do topo da pilha?

A partir de um endereçamento baseado em EBP, como podemos identificar os parâmetros passados e as variáveis locais definidas em uma função?

Assembly básico

Para fazer a engenharia reversa de um programa, é necessário saber um pouco de assembly. Não tem como escapar. Vamos começar pelo básico:

- Os componentes principais da arquitetura x86 são CPU, memória, registradores e disco.
- A CPU recupera uma instrução, decodifica e executa.
- Uma aplicação é implementada como uma sequência de instruções assembly.

A partir desse capítulo começaremos a estudar o assembly em um nível básico. Para esta parte do curso, é importante ter um conhecimento mínimo de assembly e do funcionamento do sistema operacional no nível de código de máquina.

Não está no escopo deste curso ensinar tudo sobre a linguagem; o conhecimento mínimo dela é pré-requisito para o curso, mas somente relembraremos alguns conceitos úteis durante a análise de programas maliciosos.

Primeiro, devemos relembrar os componentes macro de um processador: CPU, memória, registradores e disco.

A CPU processa as instruções de um programa, lendo cada instrução a partir da memória principal para uma memória interna à CPU (cache), decodificando-a e executando-a. Para isso, utiliza registradores, que são áreas de memória internas à CPU, para armazenar dados necessários à execução da instrução. Os programas são armazenados em disco, e dele são lidos através de instruções de acesso a dispositivos físicos do sistema operacional.

Um programa é implementado como uma sequência de instruções assembly. Mas como veremos daqui para a frente, essa sequência nem sempre é muito clara.

- As instruções podem:
  - Referenciar ou manipular posições de memória.
  - Fazer cálculos.
  - Controlar o fluxo do programa.
  - Controlar a entrada e saída de dados.
- Uma única linha de código de alto nível é convertida em diversas instruções assembly.

É importante lembrar que essas instruções são de baixo nível, isto é, perfazem as ações mais simples. Por isso, uma única linha de código em linguagem de alto nível, como C por exemplo, pode se transformar em diversas linhas em assembly.

```
int main () {
      printf("Hello World");
}
```

```
HelloWorld proc near
var_8= dword ptr -8
var_4= dword ptr -4
push    ebp
mov     ebp, esp
sub     esp, 8
and     esp, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    sub_401088
call    __main
mov     [esp+8+var_8], offset aHelloWorld ; "Hello World"
call    printf
leave
retn
HelloWorld endp
```

Figura 5.1
Comparação de linha de código em linguagem de alto nível com assembly.

Este código foi complicado com o Cygwin GCC, com diversas funções extras de inicialização inseridas pelo compilador. Outros compiladores poderiam adicionar outros tipos de código. Esta é uma característica importante durante a análise, pois é possível reconhecer o tipo de compilador usado apenas estudando as funções de inicialização adicionadas ao código do usuário.

No exemplo, vemos que um simples código "Hello World" pode se transformar em funções e instruções diversas.

O arquivo no IDA Pro está disponível no diretório "\Desktop\Sessao 5\HelloWorld.exe".

Registradores

- Registradores são áreas de memória reservadas para uso pela CPU.
- Existem oito registradores de uso geral em plataformas x86 de 32 bits, cada um também com 32 bits.
- O acesso aos registradores é muito rápido.
  - Como o acesso é rápido e eles são limitados, os compiladores tentam utilizá-los de forma eficiente.
  - Isso se torna um fator importante durante a análise.

Durante a execução de uma instrução, a CPU precisa armazenar dados que possam ser necessários para executá-la. A CPU não tem acesso direto à memória da máquina, e portanto precisa utilizar uma memória interna para isso. Por exemplo, para copiar dados de uma posição de memória para outra, a CPU precisa copiar cada byte da posição de memória de origem, para dentro da memória da CPU, e depois para a posição de memória de destino.

Estas áreas de memória interna são os registradores. Existem oito registradores de uso geral em processadores de 32 bits, e cada registrador tem 32 bits de espaço para armazenar informações.

Em plataformas x64 temos os registradores de 64 bits que além dos registradores de uso geral existentes na plataforma 32 bits existem mais 8 registradores de R8 até R15. Os registradores já existentes na plataforma x86 são acessados como: RAX, RBX, RCX, RDX, RSI, RDI, RBP e RSP.

Os registradores especiais possuem uma forma especial para acesso aos conteúdos de 32 bits, 16 bits e 8 bits. Por exemplo, o registrador R8 acessa o seu conteúdo 64 bits, para acessar o seu respectivo DWORD (32 bits) utilizamos R8D, para acessar o WORD (16 bits) utilizamos R8W e para acessar os 8 bits utilizamos R8L.

O ponteiro de instrução (EIP na plataforma 32 bits) também recebeu seu equivalente 64 bits que é acessado como RIP.

Como esses registradores estão localizados dentro da CPU, o acesso a eles é muito rápido. Devido ao fato de estarem em número restrito, o uso dos compiladores precisa ser otimizado. Dessa forma, ao analisar um programa, poderemos entender o que se passa quando um registrador é usado para realizar operações distintas, e como a CPU faz para não perder as informações armazenadas em um registrador quando precisar usá-lo para outro fim.

Registradores de uso geral na arquitetura x86:

- **EAX**: acumulador, volátil, utilizado para retorno de valores de funções.
- **EBX**: não volátil, utilizado para endereçamento indireto.
- **ECX**: utilizado em contadores e funções de loop, volátil.
- **EDX**: volátil, de uso geral.

8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits
RAX							
				EAX			
						AX	
						AH	AL

Tabela 5.2
Organização de um registrador.

A Tabela 5.2 é de um gráfico que mostra como está organizado um registrador. É importante notar que um mesmo registrador pode ser acessado por diversos nomes em um programa. Cada nomenclatura indica o tamanho do dado que queremos acessar, mas todos eles se referem ao mesmo registrador. Olhando a figura, vemos que o mesmo registrador RAX (64 bits) pode ser acessado como EAX (32 bits), AX (16 bits) e AH e AL se referem aos 8 bits mais significativos e 8 bits menos significativos, respectivamente, do registrador AX.

É importante ter esse conceito bem entendido, pois isso se torna importante durante a análise e quase sempre é motivo de confusão. Se temos um registrador EAX, por exemplo, com valor 0x09ABCD01, e executamos uma operação que zere o conteúdo de AL, o registrador EAX terá seu valor modificado para 0x09ABCD00, pois AL se refere a uma posição do registrador EAX.

Tendo isso em mente, veremos que normalmente os compiladores usam determinados registradores para algumas funções comuns. Por exemplo, EAX geralmente é usado como acumulador para instruções, ou para retorno de funções.

EBX costuma ser usado para endereçamento indireto de dados, ECX em contadores e funções de loop, e EDX é um registrador de uso geral.

Mesmo assim, em alguns casos o compilador pode precisar de mais de um registrador para executar uma instrução, e pode usá-los de forma diferente da descrita acima.

Registradores de uso geral:

- ESI
  - Armazena dados de origem em operações com strings (não volátil).
- EDI
  - Armazena dados de destino em operações com strings (não volátil).
- ESP/EBP
  - Ponteiro do Stack (volátil) / ponteiro de frame (não volátil).
- EIP
  - Ponteiro de instrução, indica a próxima instrução a ser executada.

Mais alguns registradores e seus usos mais comuns:

- ESI e EDI: registradores normalmente usados em operações que envolvem strings, como origem e destino das mesmas, respectivamente.Normalmente, estes registradores contêm o endereço de memória onde as strings estão armazenadas, e as instruções vão ler cada byte do endereço de origem e escrever no endereço de destino.
- ESP e EBP: normalmente usados como apontadores do Stack e de frame.
- EIP: usado para controlar o fluxo de execução, apontando sempre para a instrução seguinte que deve ser executada.

Você pode examinar o funcionamento dos registradores abrindo o arquivo "\Desktop\Sessao 5\HelloWorld.exe", e executando passo a passo o programa com F7, examinando a janela de registradores (alto à direita) e do Stack (embaixo à direita). Algumas instruções modificam os valores dos registradores (MOV, ADD), e outras modificam o Stack (PUSH xxx; SUB ESP,xxx). O Olly mostra em vermelho os registradores e flags que foram modificados pela última instrução.

Stack

- Estrutura de dados abstrata baseada em uma combinação de hardware e software.
- Utilizado para armazenar dados, em uma estrutura LIFO (Last In First Out).
  - Passagem de parâmetros para funções.
  - Armazena endereços de retorno de chamadas.
  - No Windows, serve também para armazenar a Structured Exception Handling (SEH).
- A instrução PUSH é utilizada para colocar um valor de 32 bits no Stack.
- A instrução POP remove um valor de 32 bits do Stack.
- Dados podem ser acessados no Stack através de referência indireta usando o registrador EBP.

Entender o funcionamento do Stack é fundamental para realizar uma análise de qualquer tipo de programa em nível de código de máquina. O Stack é usado pela CPU para passar parâmetros a funções, armazenar endereços de retorno de chamadas, e outras estruturas mais complexas como a Structured Exception Handling (SEH).

Ele funciona como uma estrutura LIFO, isto é, o último dado a ser colocado no Stack vai ser o primeiro a sair. O Stack é comparado a uma "pilha de pratos", que cresce de baixo para cima. O registrador ESP é utilizado para apontar para o topo da pilha, e normalmente, quando um programa precisa reservar espaço de memória para um parâmetro de função, ele subtrai o tamanho necessário de ESP. Essa subtração adiciona espaço ao Stack. Para limpar o Stack, deve-se somar o tamanho desejado a ESP, o que removerá este tamanho do Stack.

Para gravar um dado de 32 bits na pilha, usamos a função PUSH, e para remover esse dado, usamos a função POP.

Os dados também podem ser acessados através de referência indireta usando o registrador EBP, como mostrado no exemplo abaixo:

```
MOV EAX,DWORD PTR SS:[EBP-14]
```

Move para EAX o valor armazenado na posição 14 do Stack.

Layout de memória

Tabela 5.3
Layout de memória no Microsoft Windows.

Espaço na memória
232 = 4.294.967.296 bytes = 4 gigabytes disponíveis

Separação de memória:

- Sistemas baseados em Windows dividem estes 4 GB em duas partes: memória do kernel e do usuário.
- Cada processo vê apenas seu próprio bloco de 2 GB.
- Processos não podem sair de seu espaço de memória.

- Os endereços virtuais são divididos em páginas de 4096 bytes.
- O acesso à memória é feito no nível de páginas.

O sistema operacional Windows, em sua versão de 32 bits, disponibiliza um total de 4 GB de memória virtual para os aplicativos de kernel e de usuário. Cada processo vê somente 2 GB de memória virtual, que é compartilhada entre todos os processos.

O compartilhamento ocorre através da paginação de memória, isto é, cada processo vê sua memória como sendo de 2 GB contínuos em memória, mas o sistema organiza esses 2 GB em páginas de 4096 bytes, e somente as páginas em uso de um processo ficam em memória. Duas páginas não ocupam nunca o mesmo espaço físico na memória real, e assim o sistema consegue fazer com que os processos compartilhem a memória física, sem que um tenha acesso à memória de outro. Este esquema permite que um processo tenha disponíveis 2 GB sempre, independente da memória física do computador. As páginas que não couberem na memória física ficam armazenadas na partição de *swap* até que sejam necessárias.

- No início da execução, a imagem do programa é carregada na memória.
- As DLLs necessárias são carregadas.
- As variáveis de ambiente são mapeadas.
- As pilhas do processo são iniciadas.

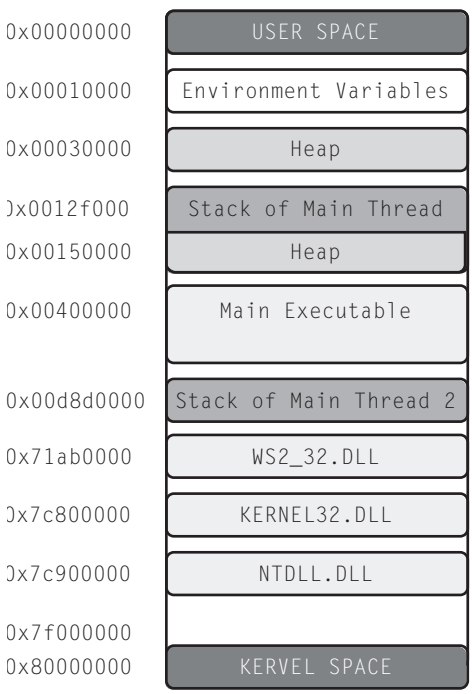


Figura 5.4
Mapeamento de um processo na memória.

Na Figura 5.4 podemos observar como é o mapeamento de um processo nos 2 GB reservados para ele. Podemos notar que o código executável do programa não é carregado necessariamente no início da memória. O endereço onde ele é carregado dentro da memória virtual é definido pelo seu atributo *ImageBase* de acordo com o que está determinado no cabeçalho do executável. Normalmente (mas nem sempre) esse endereço é 0x00400000.

Podemos notar ainda que as variáveis de ambiente são as primeiras a serem carregadas, seguido do Heap e do Stack (normalmente disponível a partir de 0x0012f000) do thread principal do programa. Após o executável, existe um espaço reservado aos Stacks dos threads adicionais, e finalmente o espaço reservado pelo programa às DLLs necessárias.

Durante o carregamento, essas estruturas são mapeadas em memória, as DLLs são carregadas, as pilhas iniciadas e o programa começa sua execução, com o controle passado ao *entry point*.

Conhecer essa estrutura é importante, principalmente se o analista se deparar com alguma vulnerabilidade em um programa. Um tipo de vulnerabilidade muito explorada é o estouro de pilha, onde o atacante cria uma variável com valor muito grande, que estoura o espaço reservado para o Stack, e acaba sobrescrevendo áreas importantes de memória. Isto pode causar o término inesperado do programa, ou a execução de código malicioso.

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000				Priv 00021004	RW	RW	
00020000	00001000				Priv 00021004	RW	RW	
0022D000	00001000				Priv 00021104	RW Guarded	RW	
0022E000	00002000			stack of main thread	Priv 00021104	RW Guarded	RW	
00230000	00003000				Map 00041002	R	R	
00240000	00006000				Priv 00021004	RW	RW	
00340000	00006000				Priv 00021004	RW	RW	
00350000	00003000				Map 00041004	RW	RW	
00360000	00016000				Map 00041002	R	R	\Device\Harddi
00380000	0003D000				Map 00041002	R	R	\Device\Harddi
003C0000	00006000				Map 00041002	R	R	\Device\Harddi
003D0000	00001000				Priv 00021040	RWE	RWE	
003E0000	00001000				Priv 00021040	RWE	RWE	
00400000	00001000	HelloWo:		PE header	Imag 01001002	R	RWE	
00401000	00001000	HelloWo:	.text	SFX	Imag 01001002	R	RWE	
00402000	00001000	HelloWo:	.rdata		Imag 01001002	R	RWE	
00403000	00001000	HelloWo:	.bss	data	Imag 01001002	R	RWE	
00404000	00001000	HelloWo:	.idata	imports	Imag 01001002	R	RWE	
00410000	00041000				Map 00041002	R	R	\Device\Harddi
0065B000	00001000				Priv 00021004	RW Guarded	RW	
0065C000	00004000			stack of main thread	Priv 00021004	RW Guarded	RW	
60000000	00005000				Map 00041004	RW	RW	
60010000	0000B000				Map 00041004	RW	RW	
60040000	00003000				Map 00041004	RW	RW	
61000000	00001000	cygwin1		PE header	Imag 01001002	R	RWE	
61001000	000FE000	cygwin1	.text	code	Imag 01001002	R	RWE	
610FF000	00003000	cygwin1	/4		Imag 01001002	R	RWE	
61102000	0000B000	cygwin1	.data	data	Imag 01001002	R	RWE	
6110D000	00034000	cygwin1	.rdata		Imag 01001002	R	RWE	
61141000	0000A000	cygwin1	.bss		Imag 01001002	R	RWE	
61141B00	00008000	cygwin1	.edata	exports	Imag 01001002	R	RWE	
61153000	00001000	cygwin1	.rsrc	resources	Imag 01001002	R	RWE	
61154000	00011000	cygwin1	.reloc	relocations	Imag 01001002	R	RWE	
61165000	00001000	cygwin1	/19		Imag 01001002	R	RWE	
61166000	00001000	cygwin1	/38		Imag 01001002	R	RWE	
61167000	00009000	cygwin1	.idata	imports	Imag 01001002	R	RWE	
61170000	00090000	cygwin1	.cygheap		Imag 01001002	R	RWE	
77DD0000	00001000	ADVAPI3;		PE header	Imag 01001002	R	RWE	
77DD1000	00075000	ADVAPI3;	.text	code, imports, exports	Imag 01001002	R	RWE	

Figura 5.5
Layout da memória de processo.

Podemos ver no Olly como fica esse mapeamento de memória de um programa real. Basta abrir o arquivo "\Desktop\Sessao 5\HelloWorld.exe", e examinar o mapa de memória com o comando Alt-M ou clicando no botão M da barra de ferramentas.

Podemos ver acima os endereços da memória do processo, a região reservada ao Stack, a região reservada ao executável (cada seção carregada em uma região), seguida do Stack do thread adicional, e finalmente a área das DLLs.

Se quiser examinar uma região específica, basta clicar duas vezes sobre ela. Por exemplo, para ver quais variáveis de ambiente estão sendo utilizadas pelo programa, basta clicar duas vezes sobre a primeira região de memória, e o *dump* dessa região irá abrir.

É importante se familiarizar com essa tela, pois ela é muito importante para encontrar informações na hora da análise. Por exemplo, se o programa carrega dinamicamente DLLs durante a sua execução, usando as funções *LoadLibraryA* e *GetProcAddress*, o aluno poderá ver onde essas DLLs foram mapeadas em memória e acessá-las através dessa tela. Isto pode ser interessante, por exemplo, quando o programa carrega uma DLL que não é padrão no sistema. Uma DLL embutida, por exemplo. É muito comum um programa malicioso descarregar em um sistema uma DLL com funções maliciosas e carregá-la dinamicamente, como se fosse um plugin.

Exercício de fixação 2
Tratamento de exceção

É possível utilizar o tratamento de exceção de forma maliciosa?

Structured Exception Handling

- SEH:
  - Manipuladores de exceção são funções registradas para tratar problemas.
  - Podemos registrar uma função para tratar erros de E/S ou divisões por zero, por exemplo.
- Os endereços destas funções são armazenados no Stack.
- Quando ocorre uma exceção, a lista de manipuladores é percorrida para encontrar uma apropriada.
- Existe um manipulador que trata todas as exceções, responsável pela mensagem:
  - "Windows has detected a general protection fault".

Sistemas Windows utilizam um sistema de tratamento de exceções chamado Structured Exception Handling (SEH). Essa estrutura é formada por ponteiros para funções, que são chamadas toda vez que uma exceção ocorre dentro do programa.

Um programa pode registrar funções para tratar algumas exceções, como por exemplo uma divisão por zero. Quando ocorre a exceção, o Windows percorre essa lista, passando o controle para cada função registrada, e aquela que conseguir tratar o erro será executada.

O endereço dessas funções fica armazenado no Stack, em uma lista encadeada. Existe sempre uma função de tratamento de exceções registradas pelo Windows, e geralmente ela é responsável por mostrar a mensagem "Windows has detected a general protection fault".

Alguns programas utilizam essa estrutura para controlar seu fluxo de execução. Veremos nas atividades práticas um exemplo de um compactador de executáveis chamado PECompact, que utiliza essa funcionalidade para executar sua rotina de descompressão.

Instruções básicas assembly

- Assembly básico:
  - O padrão que adotaremos é o assembly Intel (outro padrão seria AT&T).
  - Ex: MOV destino, origem.

- Ponteiros são indicados por []
  - Ex: [EAX] = *EAX = Conteúdo do endereço apontado por EAX.
- Load Effective Address (LEA)
  - Transfere o endereço de offset da origem para o registrador de destino.
  - É mais rápido que usar MOV.
    - Ex: LEA EAX, [EBX] vs. MOV EAX, EBX.
  - LEA é geralmente utilizado em operações matemáticas.
    - Ex: LEA EAX, [EAX*EAX] é equivalente a EAX^2.

A seguir faremos uma rápida revisão de alguns comandos mais usados em assembly.

Primeiramente vemos dois exemplos de uso de ponteiros em assembly. O primeiro caso é usado para indicar o conteúdo do endereço armazenado em um registrador.

Assim, se EAX=0x00401000, e a partir do endereço 0x00401000 temos os valores 0xDE 0x41, 0xBE e 0x00, o comando MOV EAX, [EAX] vai fazer EAX=0x00BE41DE. Isto porque EAX é um registrador de 32 bits (4 bytes são copiados) e em sistemas *little endian* os valores são armazenados de trás para a frente. Nesse caso, o comando MOV não copiou o valor de EAX, mas sim o conteúdo apontado pelo endereço em EAX.

No caso do LEA, o indicador de ponteiro funciona de forma diferente. Load Effective Address (LEA) indica que o que vamos copiar para o registrador de destino é o endereço armazenado no registrador de origem. Assim, se EBX=0x00401000, o comando LEA EAX,[EBX] vai copiar o valor 0x00401000 em EAX. Isto é o equivalente a fazer MOV EAX, EBX.

LEA geralmente é usado em operações matemáticas envolvendo os valores armazenados nos registradores. Um exemplo disso é o início do código em um arquivo compactado com UPX:

```
0050ED00 PUSHAD                 <- Salva flags

0050ED01 MOV ESI,509000         <- Move para ESI endereço 0x00509000

0050ED06 LEA EDI,DWORD PTR DS:[ESI+FFFF8000]  <- Salva em EDI o
endereço onde será descompactado o programa (ESI+0xFFFF8000 => ESI-
0x8000 => 0x00501000)

0050ED0C PUSH EDI               <- Salva EDI
```

- XOR executa a operação OU-EXCLUSIVO.
  - Passar o mesmo registrador nos dois parâmetros zera o conteúdo do registrador.
    - Ex: XOR EAX, EAX é igual a EAX = 0.
- PUSH/POP salva ou recupera o conteúdo de um registrador no Stack.
  - Ex: PUSH EAX/POP ECX.
- PUSHF(D)/POPF(D) salva ou recupera os flags de CPU.
  - Ex: PUSHF.
- As versões terminadas em D trabalham em 32 bits; o comando normal em 16 bits.

O comando XOR é usado basicamente para zerar registradores, mas pode indicar o uso de criptografia. Qualquer operação XOR onde o operando da direita é diferente do da esquerda pode ser considerada uma criptografia simples. Os valores da esquerda são mascarados com o valor da direita, que pode ser chamado de chave da criptografia. Nas duas últimas sessões, veremos um exemplo de criptografia usando essa técnica.

Os comandos PUSHAD e POPAD são usados para salvar todas as flags e registradores no Stack. Isso geralmente é feito antes de uma operação que vá modificar muitos registradores, e o programa precisa salvar essa informação antes de executar as operações. O compactador UPX usa essas funções para salvar o estado inicial do programa antes de descompactá-lo, e recuperar esse estado antes de passar o controle para o programa descompactado.

- Instruções de matemática básica:
  - INC, DEC, ADD, SUB, MUL, DIV.
- Manipulação de registradores:
  - MOV, LEA.
- Instruções de chamadas de função:
  - CALL / RET, ENTER / LEAVE.
- Comparação de valores:
  - CMP, TEST.
- Controle de fluxo:
  - JMP, JZ, JNZ, JG, JL, JGE etc.
- Manipulação de Stack:
  - PUSH, POP.
- Operação binária:
  - AND, OR, XOR, SHL, SHR.

Diversas instruções realizam operações básicas em assembly, como operações matemáticas, manipulação de registradores, chamada de funções, comparações e controle de fluxo, manipulação de Stack e operações binárias.

Um manual mais completo sobre essas operações pode ser obtido através do IDA Pro, se tiver configurado corretamente o Help adicional conforme foi sugerido (opcodes.hlp), bastando posicionar o cursor em um comando e pressionar Control-F1.

Instruções assembly: Números

- Números podem ser representados com sinal ou sem.
- Números positivos sem sinal variam de 0 a 0xFFFFFFFF.
- Números com sinal ocupam metade desse valor:
  - Positivos vão de 0x00000000 a 0x7FFFFFFF.
  - Negativos vão de 0x80000000 a 0xFFFFFFFF (-1).

É importante guardar bem esses conceitos sobre formato dos números em assembly. Como assembly não mantém os tipos de variáveis, muitas vezes o analista se confunde ao considerar um valor como número quando na verdade ele não é mais, ou considerar um número positivo quando na verdade ele é negativo. Um exemplo disso foi dado quando mostramos que 0xFFFF8000 é na verdade -0x8000.

Instruções assembly: Endereçamento

Modos de endereçamento de memória:

- Endereçamento direto.
- Endereçamento indireto.

- Endereçamento indexado.
- Endereçamento indexado escalado.

Formas de endereçamento usadas em assembly:

- **Endereçamento direto**
  MOV EAX, 0xDACAFE
  int big num = 0xDACAFE;
- **Endereçamento indireto**
  MOV EAX, [EBX]
  char inicial = *nome;
- **Endereçamento indexado**
  MOV EAX, [EBX+0xB4D]
  int opcao = char array [0xB4D];
- **Endereçamento indexado escalado**
  MOV EAX, [EBX+ECX*20]
  struct *mystruct = struct array[20];

O endereçamento direto é aquele onde o valor do operando da direita é copiado diretamente para o operando da esquerda.

No endereçamento indireto, o valor do operando da direita é o endereço copiado para o operando da esquerda.

Os dois últimos exemplos mostram como usar endereçamento indireto para criar estruturas como *arrays* e estruturas mais complexas.

No último caso, temos que EBX é o endereço base da estrutura, ECX é o tamanho de cada registro, e 20 é o número do registro que queremos acessar.

Instruções assembly: Chamada de função

Existem diversos padrões para a chamada de função:

- cdecl
  - Quem chama limpa o Stack.
- stdcall
  - Função chamada limpa o Stack.
- fastcall
  - Argumentos podem ser passados em registradores diferentes.
- thiscall
  - Código gerado por C++, ponteiro para objetos é passado em registrador.
    - Microsoft: ECX.
    - Borland / Watcom: EAX.
- naked

As chamadas a funções podem indicar o tipo de compilador usado no programa, pois cada compilador tem uma preferência por um tipo diferente de chamada.

As chamadas mais comuns são *cdecl* e *stdcall*. Normalmente os parâmetros para essas funções são passados através do Stack, por meio de instruções PUSH que precedem

a chamada, na ordem inversa dos parâmetros esperados. A diferença entre as duas é que nas chamadas *cdecl* a função que chamou limpa o Stack após a finalização da função, e no padrão *stdcall* é a função chamada que deve limpar o Stack.

Os outros tipos de chamada são usados em casos especiais, como por exemplo as chamadas *thiscall* que são usadas em código orientado a objetos para passagem nos registradores de ponteiros para objetos.

Tanto o IDA Pro quanto o OllyDBG tentam reconhecer o padrão de chamada utilizado em um programa e nomeiam automaticamente as variáveis passadas como parâmetro quando a função for conhecida (por exemplo, chamadas a funções de bibliotecas do sistema).

Argumentos de função são empilhados na ordem inversa:

```
stdcall                                    cdecl
funcao(arg1, arg2);                        funcao(arg1, arg2);
push arg2                                  push arg2
push arg1                                  push arg1
call funcao                                call funcao
mov [ebp-0xc], eax                         add esp, 8
                                           mov [ebp-0xc], eax
```

Figura 5.6
Argumentos de função são empilhados na ordem inversa.

Abaixo estão dois exemplos de chamada de funções com passagem de parâmetros e os respectivos modos de tratamento do Stack:

```
stdcall

funcao(arg1, arg2);

push arg2

push arg1

call funcao

mov [ebp-0xc], eax

cdecl

funcao(arg1, arg2);

push arg2

push arg1

call funcao

add esp, 8

mov [ebp-0xc], eax
```

Podemos observar que no caso da função *cdecl* o Stack é limpo após a chamada à função com o comando ADD ESP, 8. Como vimos anteriormente, esse comando remove 8 bytes do Stack, removendo efetivamente os dois parâmetros passados anteriormente com o comando PUSH. No caso da chamada *stdcall,* isso é feito dentro da própria função.

No final dos códigos, o valor de retorno da função, em EAX, é salvo na posição de memória indicada por [EBP-0xc] (posição 0xC do Stack). Vamos ver a seguir como funciona esse tipo de endereçamento.

Instruções assembly: Enquadramento

- Enquadramento baseado em EBP:
  - Registrador utilizado para indicar a localização de argumentos e variáveis em uma função.
- Modo tradicional:

```
push ebp
mov ebp, esp
sub esp, 0x100
```

- DLLs mais recentes:

```
mov edi, edi
push ebp
mov ebp, esp
```

- Compiladores otimizados podem omitir o apontador de frame.
  - As variáveis são referenciadas por ESP.

O endereçamento baseado em EBP permite ao programa acessar o Stack independentemente dos valores que possam ter sido adicionados ao mesmo. Por exemplo, quando uma função é chamada, seus parâmetros ficam armazenados no Stack. Durante a execução da função, outros valores podem ser gravados ou removidos do Stack, modificando o ponteiro de topo da pilha. Para acessar os parâmetros, a função normalmente salva o valor de ESP em EBP no começo da execução, e passa a acessar os parâmetros usando EBP como valor relativo.

Vimos um exemplo disso quando o programa salva o retorno da chamada de função EAX em sua variável [EBP-0xC], ou seja, na posição 0xC relativa a EBP.

É importante notar que apesar dessa ser a maneira mais comum de acesso às variáveis, nem sempre esse formato é usado. Compiladores mais recentes ou otimizações durante a compilação podem omitir o uso de EBP.

Instruções assembly: Parâmetros de função

- Parâmetros são positivos.
  - Parâmetros passados para funções são referenciados positivamente a partir de EBP.
- [EBP + valor]
  - Normalmente indica argumentos no Stack.
- [EBP - valor]
  - Indica variáveis locais.
- O retorno de funções normalmente é feito em EAX quando o valor é 32 bits.

Finalmente, para diferenciar o uso de variáveis locais e variáveis passadas como parâmetros, os compiladores geralmente usam valores positivos ou negativos referentes a EBP.

Vimos o uso de [EBP-0xC], que indica uma variável local. Lembrando dos exemplos de chamada *cdecl* e *stdcall*, poderíamos escrever aquele código em assembly da seguinte forma:

```
localvar=funcao(arg1,arg2);
```

Ou seja, o valor de retorno de "funcao(arg1,arg2)" seria armazenado em "localvar".

Roteiro de Atividades 5

Atividade 5.1 – Structured Exception Handling

```
00401000 MOV EAX,4BA070
00401005 PUSH EAX
00401006 PUSH DWORD PTR FS:[0]
0040100D MOV DWORD PTR FS:[0],ESP
00401014  XOR EAX,EAX
00401016  MOV DWORD PTR DS:[EAX],ECX
00401018  PUSH EAX
```

1. Com OllyDBG, abra o arquivo:

```
\Desktop\Malware\mal.Banker-Down.PECompact.worm.scr
```

2. Examine o Stack quando executar 0x0040100D;
3. Execute com F7 cada instrução acima, até chegar em 0x00401016;
4. Ao executar 0x00401016, será gerada uma exceção, que deve ser passada ao programa com Shift-F7;
5. Descreva o que aconteceu com o fluxo do programa e os possíveis efeitos na análise do programa.

Atividade 5.2 – Estrutura de memória dos executáveis

1. Examine os arquivos no diretório "\Desktop\Malware" com o OllyDBG;
2. Verifique o mapeamento das seções na memória e o conteúdo da memória nos endereços das seções;
3. Examine as DLLs carregadas. Os processos que usam "LoadLibraryA" e "GetProcAddress" têm alguma diferença em relação aos demais?

4. Compare os executáveis que usam o mesmo compactador, e veja se consegue descrever o padrão de código e de mapeamento de memória usado por pelo menos três deles.

6

Assembly básico – parte 2

objetivos

Identificar blocos de código que são normalmente encontrados em códigos analisados, visando facilitar a análise do código e a identificação de funcionalidades.

conceitos

Blocos condicionais, estruturas de repetição, manipulação de memória e strings, operações matemáticas, recuperação de tipos, lógica branchless.

Exercício de fixação 1
Código assembly

É possível identificar estruturas de programação como *if*, *for* e *switch* em linhas de código assembly?

Exercício de nivelamento 1
Código assembly

Sabendo que os compiladores evitam utilizar a instrução DIV por ela ser muito lenta, qual a instrução utilizada para efetuar divisões em casos em que o resto da divisão não é necessário?

Ao abrir um executável em um disassembler, é possível recuperar informações referentes aos tipos de variáveis?

Ramificações

- Instruções utilizadas no controle de decisões do programa.
- Em uma instrução CMP, o operando da esquerda é o parâmetro de comparação:
  - CMP EBX, [EAX+20h]
  - EBX:
    - > [EAX+20h]? JA *endereço*
    - < [EAX+20h]? JB *endereço*
    - = [EAX+20h]? JE *endereço*

Ramificações podem ser entendidas como código de controle de fluxo dependente do valor de uma variável. No exemplo do slide, temos o uso de uma instrução de comparação CMP EBX, [EAX+20h]. Essa instrução compara EBX com o valor armazenado no endereço apontado por EAX+20h. A seguir, existem três pulos condicionais, que vão depender do valor dos flags configurados por CMP.

Existem outras maneiras de realizar tais comparações, como por exemplo o código abaixo, encontrado no endereço 0x00407354 do programa malicioso em análise:

```
CODE:00407354                call    sub_40729C
CODE:00407359                sub     al, 1           ; Se
(sub_40729C()-1)
CODE:0040735B                jb      short loc_407372 ; < 0? va
para location1
CODE:0040735D                jz      short loc_407361 ; = 0? Va
para location2
CODE:0040735F                jmp     short loc_407362 ; qualquer
outro valor? Va para location3
```

Neste código, o valor de retorno da função “sub_40729C” é diminuído em 1, e o resultado é testado. Se for menor que 0, o controle passa para “loc_407372”, se for 0, passa para “loc_407361”, e qualquer outro valor faz o programa pular para “loc_40736”.

Blocos condicionais

```
if (var > 128)
   var = 128;
mov eax, [ebp+8]
cmp eax, 0x80
jbe skip
mov eax, 0x80
skip:
```

- Jump Below or Equal (JBE) é uma comparação sem sinal.
- Jump if Less than or Equal (JLE) é uma comparação com sinal.

Blocos condicionais são parecidos com as ramificações, mas normalmente contêm apenas uma comparação e um único desvio de fluxo. No código existem diversos exemplos desse tipo de estrutura.

```
if (var > 128)
   var = 128;
mov eax, [ebp+8]
cmp eax, 0x80
jbe skip
```

```
mov eax, 0x80
skip:
```

Loops

```
for (i = 0; i < 100; i++)
  do_something();
xor eax, eax
start:
cmp eax, 0x64
jge exit
call do_something
inc eax
jmp start
exit:
```

- Loops são fáceis de visualizar em gráficos.
- No IDA Pro 5, o modo normal de visualização já é em forma gráfica.
- Pode-se usar F12 para visualizar o fluxo de uma função.

Loops são estruturas de código executadas um certo número de vezes, normalmente com uma condição de saída. Por exemplo, a função FOR executa a sequência de código *do_something()* cem vezes.

```
for (i = 0; i < 100; i++)
  do_something();
xor eax, eax
start:
cmp eax, 0x64
jge exit
call do_something
inc eax
jmp start
exit:
```

Um loop é facilmente identificado em um gráfico de controle de fluxo. No IDA, basta posicionar o cursor dentro da função desejada e pressionar F12 para gerar um gráfico como mostrado na Figura 6.1, onde é possível facilmente identificar o loop e as suas condições de saída.

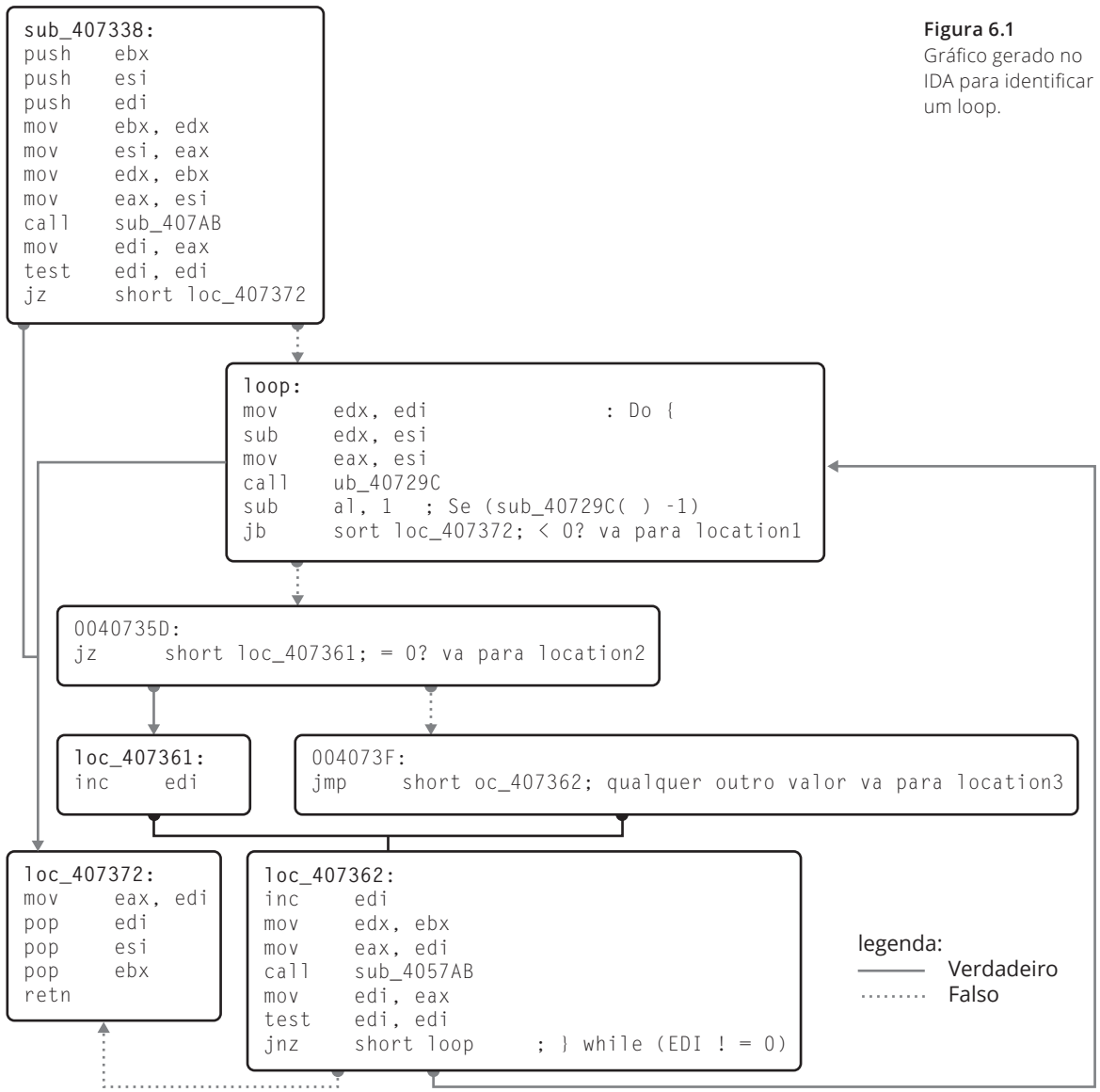


Figura 6.1
Gráfico gerado no IDA para identificar um loop.

Switch

Switch Statements:

```
switch (var) {

  case 0: var = 100; break;

  case 1: var = 200; break;

jmp switch_table[eax*4]

  case_0:

  mov [ebp-4], 100

jmp end

  case_1:

  mov [ebp-4], 200
```

```
jmp end
end:
```

IDA Pro e OllyDBG têm excelente identificação de *switch/case.*

Estruturas *Switch* são muito comuns para controlar o fluxo de execução de um programa quando é necessário testar uma variável para comparar com diversos valores. Existem basicamente duas maneiras de se fazer isso em assembly, e o formato que o analista encontrará depende muito do compilador usado.

```
switch (var) {
   case 0: var = 100; break;
   case 1: var = 200; break;
jmp switch_table[eax*4]
   case_0:
   mov [ebp-4], 100
jmp end
   case_1:
   mov [ebp-4], 200
jmp end
end:
```

No exemplo acima, a estrutura switch é criada através de uma tabela de JMP. Neste caso, o JMP é executado de acordo com o valor de EAX*4 (4 é o tamanho do endereço armazenado na tabela *switch_table*, e EAX é o índice da localização que será chamada). Por exemplo, se quisermos passar o controle para o sexto valor da tabela de switch, faremos o seguinte:

```
MOV EAX,6h
JMP switch_table[EAX*4]
No IDA, uma tabela de switch pode ser reconhecida pela sequência de
offsets, seguindo a definição de uma variável:
MOV EAX, 3h
JMP off_407F8C[EAX*4]
…
off_407F8C              dd offset loc_4045C4 ; DATA XREF:
CODE:00407F88
          dd offset loc_404394
          dd offset loc_4045FC
          dd offset loc_4046E4
```

No caso acima, o controle do fluxo é direcionado para a terceira localização a partir de “off_407F8C”, ou seja, “loc_4045FC”.

Memcpy()/Strcpy()

memcpy() / strcpy() inline

```
mov esi, origem
mov edi, [ebp-64]
mov ebx, ecx
shr ecx, 2
rep movsd
mov ecx, ebx
and ecx, 3
rep movsb
```

- REP MOVSD copia ECX DWORDS de ESI para EDI.
- REP MOVSB copia o restante dos bytes.

Este tipo de estrutura pode ser encontrada em qualquer programa que trabalhe com strings.

```
mov esi, origem
mov edi, [ebp-64]
mov ebx, ecx
shr ecx, 2
rep movsd
mov ecx, ebx
and ecx, 3
rep movsb
```

Os comandos REP são usados para ler/copiar strings em geral. Mais informações sobre os comandos REP SCA* podem ser encontradas no *Help* do IDA Pro (opcodes.hlp).

O importante a notar aqui é que o registrador ESI é usado para indicar o endereço da string de origem; EDI é usada para indicar o endereço da string de destino, e ECX é usado para indicar quantos caracteres serão lidos. Se a string não for um múltiplo de DWORD, os bytes restantes são copiados por REP MOVSB.

Strlen()/Strstr()

strlen()/strstr() inline

```
mov edi, string
or ecx, 0xFFFFFFFF
xor eax, eax
repne scasb
not ecx
dec ecx
```

- REPNE SCASB percorre a string em EDI procurando pelo caractere em AL.
- Para cada caractere percorrido, ECX é decrementado e EDI é incrementado.
- No fim da sequência REPNE:
  - EDI = posição do caractere encontrado + 1 (mais um).
  - ECX = tamanho negativo da string - 2 (menos dois).
- Tamanho da string:
  - not ECX - 1

As funções *strlen()* e *strstr()* são implementadas através do comando REPNE SCASB, que percorre a string apontada por EDI, procurando pelo caractere armazenado em AL (byte menos significativo de EAX).

```
mov edi, string

or ecx, 0xFFFFFFFF

xor eax, eax

repne scasb

not ecx

dec ecx
```

Para *strstr()*, cada caractere percorrido é comparado com AL (ECX é decrementado e EDI é incrementado). Ao final, ECX vai conter a posição do caractere em AL na string. Se AL=0x00, o comando procura pelo caractere de fim de string, transformando o comando em strlen().

Como ECX vai ser um número negativo, normalmente o código transforma o mesmo em número positivo fazendo NOT ECX; DEC ECX.

Estruturas

Acesso a estruturas:

```
push ebp
amov ebp, esp
mov eax, off_struct
mov ebx, [ebp+arg_0]           ; variável arg_0 local
push esi
cmp ebx, [eax+14h]             ; byte 0x14 a partir de EAX
mov eax, off_object
ja short loc_12345678
cmp [eax+8*var_4], ebx  ; if (struct[var_4]==EBX), cada struct[]
                        ; tem 8 bytes
sbb esi, esi
```

- EAX é carregado de uma variável global.
- EAX é acessado utilizando [], o que indica que é um ponteiro para a estrutura.

O acesso a estruturas geralmente é feito através de endereçamento indireto usando o registrador EAX ou EBP. No código abaixo podemos ver alguns exemplos de uso de estruturas:

```
push ebp
mov ebp, esp
mov eax, off_struct
mov ebx, [ebp+arg_0]            ; variável arg_0 local
push esi
cmp ebx, [eax+14h]              ; byte 0x14 a partir de EAX
mov eax, off_object
ja short loc_12345678
cmp [eax+8*var_4], ebx   ; if (struct[var_4]==EBX), cada struct[]
                         ; tem 8 bytes
sbb esi, esi
```

O primeiro caso é o salvamento do endereço da estrutura "off_struct" em EAX. Este valor é usado posteriormente para acessar o décimo quarto byte da variável, com a chamada CMP EBX, [EAX+14h]. Isso significa que deve-se comparar o valor de EBX com o décimo quarto byte a partir de EAX.

Outro uso de estruturas mostrado acima é o acesso a variáveis locais da função, através de endereçamento indireto de EBP. O comando MOV EBX, [EBP+arg_0] acessa o argumento "arg_0" passado à função.

Finalmente, temos um exemplo de acesso à uma estrutura mais complexa. Neste caso, podemos considerar "off_object" como um array de estruturas. No caso, cada estrutura deve ter 0x8 bytes de tamanho, e o acesso a cada membro da estrutura é feito como mostrado: CMP [EAX+8*var_4], EBX, ou seja, EBX vai ser comparado à estrutura com índice "var_4" em nosso array.

Operações matemáticas

- A divisão usando a instrução DIV é extremamente lenta.
  - Compiladores só a utilizam quando necessitam do resto.
- Divisões são feitas utilizando uma combinação de instruções de *Shift*.
  - Ex: divisão por 4.
    - SHR EAX, 2.
- SHL faz a operação oposta (multiplicação).

As operações de multiplicação e divisão em assembly são geralmente realizadas através das funções de Shift binário. O comando DIV é extremamente lento, e portanto o compilador só o utiliza quando precisar de resto.

Para casos onde o resto não é necessário, ou em que realizamos uma multiplicação, o uso das funções SHR e SHL é escolhido pelo compilador. Tomando como exemplo o número 8 representado em binário por 00001000:

```
SHR EAX, 2 => 00000010 => 2

SHL EAX, 3 => 00010000 => 16
```

No caso de algum bit significativo ser movido para fora do tamanho do byte, ele será descartado, e as flags de CARRY, ZERO e OVERFLOW podem ser setadas. Exemplo:

```
EAX=10001001

SHL EAX, 2 => 00100100

SHR EAX=3 => 00000100
```

Números randômicos

Pseudo geradores de números randômicos.

```
; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||

SaveCurrentTime proc near
                call    GetTickCount
                mov     ds:CurrentSavedTime, eax
                retn
SaveCurrentTime endp

; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||


RandomNumberGenerator proc near
                mov     eax, ds:CurrentSavedTime
                imul    eax, 343FDh
                add     eax, 269EC3h
                mov     ds:CurrentSavedTime, eax
                mov     ax, word ptr ds:CurrentSavedTime+2
                retn
RandomNumberGenerator  endp
```

Figura 6.2 Pseudo geradores de números randômicos.

Muitas vezes, um programa pode precisar gerar números randômicos para usar em suas funções. Como o Windows não provê um gerador de números randômicos real, os programas costumam implementar essa função através do uso do relógio do sistema. A função *GetTickCount* devolve a hora atual em número de segundos desde uma data específica (no Linux, UnixTime, contado a partir de 1 de janeiro de 1970).

Como esse número está sempre mudando, e geralmente é um número grande, o programa realiza algumas operações matemáticas com ele, para causar o estouro do espaço reservado para o número, e pega algumas partes do número final, para contar como número randômico. Este tipo de operação é chamada de geração de números pseudo randômicos.

O exemplo foi retirado do programa malicioso que analisaremos nas duas últimas sessões; ele retorna o número randômico de 32 bits em EAX.

Exercício de fixação 2
Variáveis em códigos assembly

Existe representação de tipos de variáveis em códigos assembly?

Variáveis

- Não existe representação de tipos de variáveis em assembly.
- Então, como diferenciar:
  - Strings de dados brutos?
  - Strings UNICODE de strings ASCII?
  - Inteiros de ponteiros?
  - Inteiros de valores booleanos?
    - Precisamos analisar a variável através das funções que a utilizam.

A identificação de tipos em assembly é uma tarefa difícil, porque o assembly não representa tipos de variáveis, dificultando essa tarefa do analista. Para identificar e diferenciar tipos de variáveis, é necessário estudar como esses valores são usados durante a execução do programa.

A forma indicada de fazer isso é estudando os parâmetros passados para funções ou operações realizadas com valores armazenados em registradores.

Variáveis de API

O IDA Pro tem uma excelente identificação de tipos usados em chamadas de funções de API:

```
CODE:004056A4 loc_4056A4:                ; CODE XREF: sub_405694+C
CODE:004056A4    lea     edx, [ebp+TotalNumberOfClusters]
CODE:004056A7    push    edx             ; lpTotalNumberOfClusters
CODE:004056A8    lea     edx, [ebp+NumberOfFreeClusters]
CODE:004056AB    push    edx             ; lpNumberOfFreeClusters
CODE:004056AC    lea     edx, [ebp+BytesPerSector]
CODE:004056AF    push    edx             ; lpBytesPerSector
CODE:004056B0    lea     edx, [ebp+SectorsPerCluster]
CODE:004056B3    push    edx             ; lpSectorsPerCluster
CODE:004056B4    push    eax             ; lpRootPathName
CODE:004056B5    call    GetDiskFreeSpaceA
```

Se compararmos com a dificuldade em encontrar os tipos das variáveis de funções não integrantes de uma API, para APIs documentadas o procedimento é mais fácil.

Para identificar o tipo de uma variável, é preciso descobrir um ponto no código onde essa variável seja usada, e ver como ela é tratada, isto é, se é usada em uma soma ou multiplicação, se pode ser um inteiro, se é usada para indicar uma área de memória com strings, se é uma variável de texto, e assim por diante.

Para funções presentes em APIs e DLLs do sistema, descobrir essas informações é fácil, bastando consultar uma referência sobre essa função, para descobrir as variáveis que ela recebe como parâmetro. Em seguida, basta olhar no código e ver o momento em que essas variáveis são armazenadas no Stack antes de chamar a função, o que equivale a passar esses parâmetros para ela. Sabendo a ordem em que as variáveis estão no Stack, é possível associá-las aos tipos de variáveis que a função espera como parâmetro de entrada, e com isso descobrir o tipo das variáveis no código.

No exemplo, uma chamada à função *GetDiskFreeSpaceA()* do sistema. De acordo com o IDA, essa função está definida na API como:

```
BOOL __stdcall GetDiskFreeSpaceA(LPCSTR lpRootPathName,LPDWORD
lpSectorsPerCluster,LPDWORD lpBytesPerSector,LPDWORD
lpNumberOfFreeClusters,LPDWORD lpTotalNumberOfClusters)
```

Examinando o código que antecede a chamada à função, observamos diversos comandos PUSH colocando esses parâmetros no Stack. Lembre-se sempre de que a ordem de passagem dos parâmetros é inversa à ordem descrita na chamada da função na API.

Recuperação de tipos

Inteiros x Ponteiros:

- Inteiros nunca são "desreferenciados".
- Inteiros são frequentemente comparados, ponteiros não.
- Ponteiros são comparados com zero ou com outros ponteiros.
- Aritmética com ponteiros é geralmente simples, mas com inteiros costuma ser complexa.

No caso específico de diferenciação de números e ponteiros, é importante notar a forma como as variáveis são usadas. Ponteiros costumam ser "desreferenciados" (dereferenced), e geralmente são comparados com zero ou com outros ponteiros. A aritmética realizada com ponteiros também costuma ser simples.

No caso de números, eles costumam ser comparados com outros números e nunca são "desreferenciados". A aritmética envolvendo números é mais complexa. O uso dos comandos INC e DEC pode indicar a presença de um número, já que não teria muito sentido incrementar ou decrementar o endereço de um ponteiro.

- Recuperação de tipos: Strings x Dados brutos.
- Cópia de strings é geralmente precedida por um *strlen()* ou o equivalente inline, cópia de dados brutos não.
- Strings são normalmente comparadas a caracteres legíveis ou a outras strings, mas dados brutos normalmente não.
- Confirme parâmetros strings até para as chamadas a APIs.

```
loc_504546:                             ;CODE XREF: sub_50450E+16↑j
                                        ; sub_50450E+1A↑j
                cmp     byte ptr [edi], 40h ; 40h == '@'
                mov     [ebp+esi+String1], bl
                jnz     short loc_5045C3

                cmo     ds:off_5090D0, ebx
                jz      short loc_50457D

                mov     eax, offset off_5090D0
                mov     esi, eax
```

Figura 6.3
Strings x Dados brutos.

No caso de strings, a identificação se faz através de chamadas a funções *strlen()* e *strcpy()*. Dados brutos também podem ser copiados com *strcpy() inline*, mas normalmente somente cópias de strings são precedidas por *strlen()*.

Caso a variável analisada seja usada como parâmetro em uma chamada de função de API, confirme na documentação da API se o parâmetro equivalente aceita mesmo uma string.

O exemplo compara o valor apontado por EDI com o caracter "@", e move o valor em BL para a posição ESI da variável String1, passada como parâmetro à função. ESI possivelmente foi carregado com o tamanho de String1, e o código faz com que o caractere em BL seja adicionado à string.

Código orientado a objetos

- Código orientado a objeto C++
  - O ponteiro *this* é normalmente passado através de ECX.

```
lea ecx, [esp+16]
call rotina_membro
```

- Tabelas virtuais, ou *VTables*, são comuns em C++

```
mov eax, esi
push 0x100
call dword ptr [eax+50]
```

- Procure identificar os construtores/destrutores e rotinas de inicialização ao analisar programas em C++, pois eles podem ser úteis.

A análise de código orientado a objeto costuma confundir o analista, principalmente ao tentar identificar as funções a variáveis membro de um objeto, e quando o mesmo é instanciado.

Como já vimos, o ponteiro *this* de um objeto é geralmente passado à uma função membro através de ECX.

Outra característica, neste caso usada em código C++, é o uso de tabelas virtuais, onde o endereço da função chamada é armazenado em um *array*, e a chamada é feita ao índice desejado dentro dessa tabela.

Lógica branchless

- Fluxo lógico do programa não conta com saltos ou desvios condicionais.
- Compiladores evitam o uso de desvios condicionais quando possível.
- A maioria dos pares CMP/JMP podem ser convertidos em uma sequência de operações aritméticas.
  - O uso da instrução SBB é um indicador típico.
- O código resultante roda muito mais rápido, mas não é tão legível.

O processo de decisão de fluxo é muito custoso em assembly. Por isso, em alguns casos os compiladores escolhem utilizar uma lógica sem desvios. Neste caso, os flags da CPU são usados para controlar o fluxo do programa.

O uso da instrução SBB é um indicador típico de que o compilador escolheu usar lógica *branchless* no código. Geralmente ela substitui casos simples de estruturas IF/THEN.

```
cmp eax, 1
sbb eax, eax
inc eax
pop esi
pop ebx
retn
```

- CMP EAX, 1 vai setar o flag CF se EAX for 0.
- SBB EAX, EAX faz EAX= EAX - (EAX+CF).
- Portanto, se EAX=0, logo EAX = 0 - (0+1) = -1.
- Se EAX é maior que zero, então EAX = EAX - EAX+0 = 0.
- INC EAX torna os valores possíveis de EAX iguais a 0 ou 1.

Neste exemplo podemos ver o uso da instrução SBB para executar um código que transforma o valor de retorno de uma função em 0 ou 1.

```
cmp eax, 1
sbb eax, eax
inc eax
pop esi
pop ebx
retn
Inicialmente, EAX é comparado com 1.
Se EAX=0, o retorno da função será 0.
Se EAX for diferente de 0, o retorno da função será sempre 1.
Este código pode ser representado pelo seguinte código de alto nível:
Function x()
```

```
If (EAX>0)
   return 1
Else
   return 0
end
```

Assembly básico

Engenharia reversa pode parecer complicada, mas é uma questão de treino e familiaridade

- Treine os conceitos.
- Mantenha um guia de referência à mão (opcodes.hlp).
- Examine alguns programas com o debugger, executando passo a passo com F7.
- Use um emulador ou máquina virtual.
- Compile um código simples e tente analisá-lo.
- Sabendo o que o código faz, é muito mais fácil entender o assembly.
- Vale a pena lembrar que somente a experiência vai ajudar o analista na hora de estudar um código malicioso desconhecido. Portanto, treine sempre que puder e com a maior variedade possível de códigos.
- Procure manter sempre à disposição guias de referência de linguagem assembly e linguagens de alto nível, para entender exatamente o que os comandos significam e como as estruturas funcionam.
- Executando os programas em um ambiente virtual, passo a passo, você poderá estudar o comportamento de cada parte do código, sem se preocupar em comprometer o sistema.

Finalmente, para entender estruturas mais simples sem ter que ficar procurando por elas no código malicioso, crie alguns programas em linguagem de alto nível, com a estrutura que estiver analisando, e compile o código com diferentes compiladores, para ver como o código assembly é gerado e conhecer as diferenças entre os compiladores.

Roteiro de Atividades 6

Atividade 6.1 – Assembly básico

Examine com o IDA Pro o arquivo "\Desktop\malware\mal.Down.nopack.worm.exe".

Identifique as estruturas vistas na sessão teórica marcando a posição (Alt-M) e inserindo um comentário (:) ao lado das que conseguir encontrar. Ao final, grave o arquivo IDB do IDA Pro (mal.Down.nopack.worm.idb) com as marcações e comentários.

Atividade 6.2 – Reconhecimento de códigos

Examine os códigos de exemplo disponíveis no CD do aluno, no diretório "Sessão 6\Exemplos". Abra os arquivos C em um editor de textos e os executáveis correspondentes no IDA Pro.

Para cada arquivo, encontre o código assembly correspondente ao código C apresentado, e insira os comentários (:) e marcações (Alt-M) conforme a atividade anterior.

7

Import Address Table

objetivos

Apresentar a tabela de importação e como podemos reconstruí-la em caso de análise de um arquivo compactado.

conceitos

Tabela de importação, reconstrução da tabela de importação.

Exercício de fixação 1
Tabela de importação

Como as informações da tabela de importação podem ser úteis durante a análise de um código malicioso?

Exercício de nivelamento 1
Tabela de importação

Por qual motivo os autores de códigos maliciosos tentam obfuscar a tabela de importação?

Import Address Table

- A tabela de endereços de importação (IAT) é uma tabela com endereços dos símbolos importados.
- Normalmente executáveis escondem estes símbolos utilizando diversos truques para dificultar a análise.
- Geralmente eles resolvem por conta própria estes símbolos, ou movem a tabela de símbolos para um local não padrão.
- Depois de mapeadas dinamicamente, estas funções são integradas aos binários através de tabelas de *jumps*.
- Isto dificulta a identificação das DLLs envolvidas, por não passarem mais pelo *entry point* delas.

Neste capítulo, conheceremos uma característica dos programas maliciosos que costuma dificultar a análise. Como já foi comentado antes, os autores de programas maliciosos tentam de toda forma dificultar o trabalho do analista, usando técnicas de obfuscação de código, compactação e criptografia, entre outras.

Uma das formas mais comuns de obfuscar o código é remover qualquer referência a funções e DLLs do sistema. Dessa forma, ao examinar rapidamente o binário, o analista não sabe quais funções são usadas por ele, e terá que descobrir isso de outra forma. Isso é conseguido removendo a tabela de endereços de importação (IAT).

Mesmo assim, o binário ainda precisa acessar essas funções de alguma forma. A solução é montar essa tabela dinamicamente, durante a execução do programa, ou então movendo essa tabela para locais não padrão, como a seção de recursos do binário, e depois acessando esse dado dinamicamente.

O carregamento das funções é feito pelas chamadas à *LoadLibraryA* e *GetProcAddress*. Isso evita que as DLLs sejam inicializadas corretamente através da chamada a *DLLMain()*, e dificulta também a ação do analista.

Mas como em algum momento essas funções estarão mapeadas para uso do programa, é possível recuperar essa informação. Veremos a seguir exemplos desse tipo de técnica, e formas de recuperar a informação de importação de funções.

Para recriar essa tabela, podemos utilizar algumas técnicas:

- Procurar manualmente pela sequência de comandos *LoadLibrary/GetProcAddress*
- Procurar por hashes dos nomes e funções mais comuns
- Procurar por assinaturas dos códigos das funções
- Ou então podemos utilizar o próprio executável para isso!

Para recriar essa tabela, podemos usar as técnicas descritas acima, que normalmente exigem muito trabalho. Mas conforme comentado, o programa realiza as chamadas a *LoadLibrary/GetProcAddress* em algum momento, então basta esperar que esse momento ocorra, e recuperar o binário da memória já com as funções mapeadas.

Reconstrução da IAT

- Por que reconstruir a IAT?
- Analisar um arquivo malicioso sem estes símbolos pode ser muito trabalhoso.
- Existe um número de formas diferentes de se fazer isso.
- Normalmente, é necessário executar o programa, para que o mapeamento de funções seja feito por ele.
- Por isso, é importante salvar o estado da sua máquina virtual antes, para não comprometê-la.
- Existem diversas ferramentas para isso. Veremos duas a seguir:
  - OllyDBG/OllyDump.
  - ImportRec.

Para realizar a primeira atividade, vamos usar o OllyDBG, especificamente o plugin de dump de memória, para recuperar uma cópia do programa mapeado na memória.

A seguir, usaremos a ferramenta ImportRec para percorrer o programa e recuperar as informações de funções importadas presentes nele.

É importante lembrar que a execução de qualquer tipo de programa malicioso compromete a máquina. Por isso, neste momento crie um snapshot da sua máquina virtual, para recuperar o estado sadio após a execução do programa.

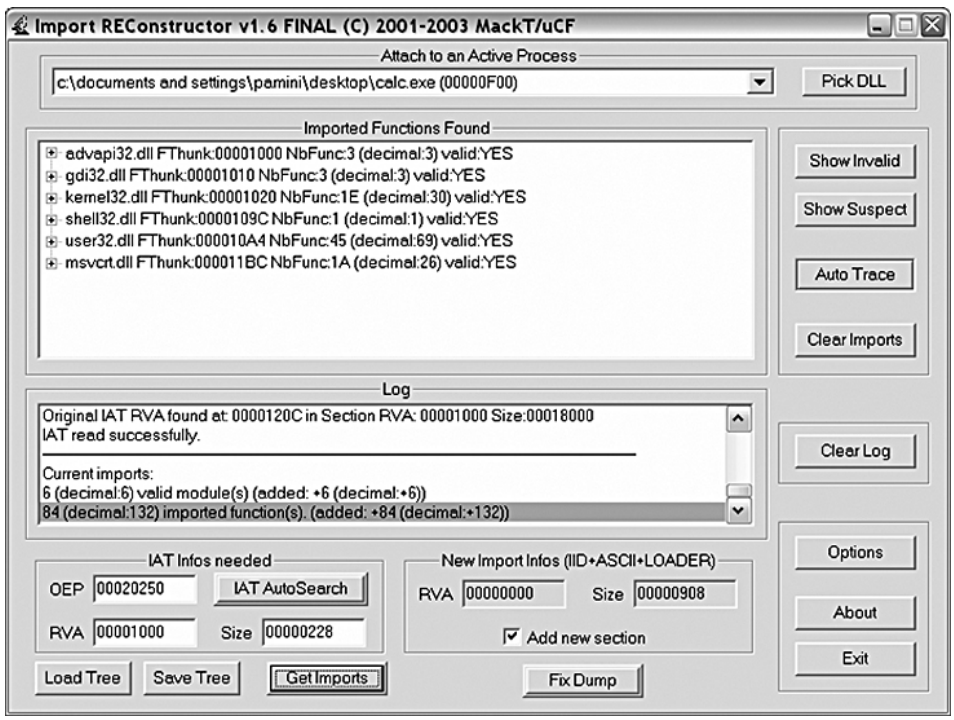


Figura 7.1
Feramenta ImportRec.

Neste capítulo aprendemos a recuperar informações ofuscadas em um programa malicioso. Vimos também algumas rotinas de descompressão de executáveis, e aprendemos a encontrar o OEP dos programas. Estas técnicas servem para recuperar outros tipos de informação, como arquivos criptografados.

Estude outros arquivos compactados, e os scripts do OllyDBG disponíveis em:

\Desktop\install\OllyDbg\OLLYSCRIPTS\

E treine sempre e o máximo que puder!

Roteiro de Atividades 7

Atividade 7.1 – Recuperando a IAT com ferramentas

Primeira parte

1. Faça snapshot da máquina virtual;
2. Abra com o OllyDBG o arquivo:
   \Desktop\malware\mal.upx.6dddd7e8e5ff88a15b7884a833ff893b.dat
3. Encontre o OEP do código descompactado, e pause a execução do programa nesta instrução;
4. Faça um dump do processo deixando marcada a opção "Rebuild Import". Para cada método de reconstrução da IAT disponível, faça um dump diferente;
5. Faça um relatório descrevendo o que fez até agora, incluindo o processo para encontrar o OEP e para fazer o dump do programa.

Segunda parte

1. Execute o ImportRec, e depois execute o programa malicioso original (renomeie de *.dat* para *.exe*);
2. Escolha o programa malicioso em execução no ImportRec, clique em "IAT Autosearch", depois em "Get Imports", e finalmente em "Fix Dump". Escolha um dos arquivos dump feito na primeira parte;
3. Abra o arquivo gerado acima e os dois arquivos de dump da primeira parte no IDA Pro e examine as funções importadas e o código analisado pelo IDA;
4. Verifique os arquivos com o LordPE para comparar as informações sobre os arquivos gerados;
5. Faça um relatório destacando as principais diferenças entre os métodos utilizados e as funções detectadas:

6. Recupere o snapshot inicial, pois a máquina virtual foi comprometida.

Atividade 7.2 – Recuperando a IAT manualmente

Agora que já vimos como fazer a recuperação automaticamente, vamos entender o procedimento de recuperação manual.

Primeira parte

1. No OllyDBG, abra o arquivo "\Desktop\Malware\mal.Banker.Upack.abrir-fotos.scr";
2. Descubra a área de memória na qual o programa será descompactado;

3. Execute o programa com F7 por um tempo, e monitore a área de memória em que serão gravados os dados descompactados na janela de dump, para descobrir a rotina de descompressão;
4. Encontre o endereço do último byte gravado para descobrir a instrução de saída da rotina de descompactação;
5. Descubra onde começa e onde termina a rotina de descompressão do Upack;
6. Faça um relatório descrevendo os passos executados e os métodos utilizados para a resolução das questões acima.
7. Após gravar o último byte, o programa realiza uma operação modificando alguns bytes no código descompactado. Possivelmente trata-se de um último passo de desobfuscação do código:

```
0093789E    MOV AL,BYTE PTR DS:[EDI]            ; Loop fixa código
009378A0    INC EDI
009378A1    ADD AL,18
009378A3    CMP AL,2
009378A5  ^JNB SHORT 0093789E                   ; mal_Bank.0093789E
009378A7    MOV EAX,DWORD PTR DS:[EDI]
009378A9    CMP AL,16
009378AB  ^ JNZ SHORT 0093789E                  ; mal_Bank.0093789E
009378AD    MOV AL,0
009378AF    BSWAP EAX
009378B1    ADD EAX,DWORD PTR DS:[ESI+14]
009378B4    SUB EAX,EDI
009378B6    STOS DWORD PTR ES:[EDI]
009378B7  ^ LOOPD SHORT 0093789E                ; fim do loop
```

Analise este código com seus colegas e faça um relatório com suas conclusões.

Segunda parte

1. Tente encontrar a posição de memória em que ficam armazenadas as DLLs e as funções que serão carregadas para recriar a tabela de importação.
2. Qual código é utilizado para executar essa operação?
3. O que acontece com o mapa de memória do processo após carregar todas as DLLs?

4. Faça um dump do processo após finalizar essa análise.
5. Faça um relatório descrevendo todos os passos executados nesta parte e os métodos usados para encontrar as informações. Entregue o dump acima junto com o relatório.

Terceira parte

1. Você consegue encontrar a chamada ao OEP do programa? Se encontrar, faça um dump do processo após atingir o OEP.
2. Restaure o estado da máquina virtual para o snapshot inicial.

Atividade 7.3 – Análise estática

Abra o mesmo arquivo da atividade anterior no IDA Pro. Encontre as partes de código responsáveis pela descompressão, reconstrução da IAT e OEP, comente (:) e marque (Alt+M).

8

Truques anti-engenharia reversa

objetivos

Apresentar técnicas anti-debugging normalmente utilizadas e demonstrar técnicas para detecção e bypass destas proteções.

conceitos

Detecção de debugger, thread local storage, detecção de máquinas virtuais, structured exception handling.

Exercício de fixação 1
Protetores de código

Qual o objetivo dos protetores de código utilizados em artefatos maliciosos?

Exercício de nivelamento 1
Debugger

Por que alguns códigos maliciosos utilizam técnicas de detecção de debugger?

É possível a execução de um código antes do entry point?

Truques anti-engenharia reversa

- Os autores de programas maliciosos utilizam compactadores e encriptadores para reduzir o tamanho dos arquivos, embaralhar o código e dificultar a análise.
  - Código compactado passa a aparecer como dado, confundindo os disassemblers.
- Algumas ferramentas usam técnicas anti-debugger.
- Existem centenas de compactadores.
- OllyDBG tem dezenas de scripts para descompactar estes arquivos.

Neste capítulo, estudaremos algumas técnicas usadas por autores de programas maliciosos para tentar impedir a análise ou dificultar a identificação do programa. Veremos como identificar essas técnicas e algumas ferramentas e medidas que podem ser tomadas para contra atacar.

Os autores de programas maliciosos utilizam compactadores e encriptadores para reduzir o tamanho dos arquivos, embaralhar o código e dificultar a análise.

Algumas ferramentas usam técnicas anti debugger, como detecção de execução passo a passo, modificações das funções de tratamento de exceções ou inserção de funções de callback na tabela TLS.

Existem centenas de compactadores, como UPX, PECompact, Yoda, tElock, ASPack, BurnEye, entre tantos outros.

Este tipo de técnica é usada justamente porque os autores de programas maliciosos sabem que empresas de antivírus e de combate ao crime digital utilizam técnicas de engenharia reversa para analisar o código e descobrir as características que identificam o programa, e as ações tomadas quando um usuário é infectado.

Entre as técnicas mais simples, podemos citar o uso de compactadores de executáveis, ou então técnicas mais avançadas como detecção de debugger, modificação de exceções, detecção de ambiente virtual, entre outras técnicas. Veremos como usar as ferramentas estudadas até agora para reverter essa situação e analisar qualquer tipo de programa malicioso.

Detecção de debugger

- Kernel32.IsDebuggerPresent()
- INT 3 (0xCC)
- Timers
- Exemplos:
  - start = GetTickCount();
  - alguma_funcao();
  - if (GetTickCount()-start > valorX)
  - Debugger_detectado();

A primeira técnica que veremos é também uma das mais perigosas no uso de máquinas virtuais para a análise de programas maliciosos. Alguns programas maliciosos tentam identificar se estão sendo executados dentro de um debugger, e podem modificar seu comportamento ou mesmo interromper seu funcionamento se conseguem detectar o debugger. Eles podem realizar essa operação de diversas formas, e implementar verificações periódicas para dificultar ainda mais o trabalho do analista.

A biblioteca do sistema Kernel32.dll possui uma função chamada *IsDebuggerPresent()* que retorna "verdadeiro" caso o programa esteja sendo executado ao mesmo tempo que um debugger.

Este tipo de proteção pode ser facilmente identificada e evitada. Como é uma função de uma biblioteca padrão que deverá ser importada pelo programa, uma simples análise das funções importadas permite identificar seu uso. Mesmo que a tabela de importação esteja modificada ou escondida, é possível recuperá-la, conforme foi visto anteriormente.

A segunda forma de detectar um debugger é através da execução da instrução assembly INT 3. Essa instrução é usada por debuggers para implementar breakpoints no código analisado. Ao ser executada na presença de um debugger, esta instrução gera uma exceção no sistema. Um programa malicioso pode executar essa instrução para gerar a exceção e parar o funcionamento do programa. Quando o programa for executado sem um debugger, ele funcionará normalmente.

Podemos descrever ainda a técnica de utilização de timers para identificar pausas na execução de um programa. Quando o programa é executado em modo passo a passo ou é pausado por um debugger, o relógio do sistema continua contando os segundos da mesma forma. Um programa pode usar essa característica para identificar se determinada parte do código é executada em um tempo razoável. Assim, se o código deveria ser executado em 1 segundo, mas demorou 5 segundos para rodar, o programa pode achar que foi executado em modo passo a passo, e parar seu funcionamento. Vimos um exemplo de como isso poderia ser feito.

Exercício de fixação 2
Entry point

É possível executar um código antes do entry point do arquivo?

Execução de código antes do Entry Point

- Existem formas de executar código antes que um debugger assuma a execução do programa.
  - Nunca assuma que um programa malicioso não pode executar código somente por carregar em um debugger.
- *DLLMain()* é executado antes do debugger assumir.
  - http://www.insomniasec.com/publications/PreDebug.pdf
- O código de inicialização da Thread Local Storage (TLS) em arquivos PE é executado antes do debugger.

Quando um programa é carregado, o sistema operacional mapeia os dados e o código na memória, inicializa variáveis e passa o controle ao *entry point* do programa para iniciar a execução.

Ao executar este programa dentro de um debugger, a execução do programa é pausada quando este atinge o *entry point*. Mas ao atingir esse ponto, uma brecha muito perigosa aparece. No momento em que o sistema operacional inicializa as variáveis do programa, algumas funções do mesmo podem ser chamadas para realizar essa tarefa. Isto pode permitir que o código do programa seja executado antes que o debugger assuma a execução do programa no *entry point*.

Um exemplo disso é o caso de DLLs. Como elas são arquivos PE, ao carregá-las no debugger, sua execução será paralisada no *entry point*. Mas em DLLs, a função *DLLMain()* é chamada antes disso, permitindo ao programa malicioso executar código antes da parada pelo debugger.

O TLS, como vimos anteriormente, é a estrutura responsável por manter a separação de dados entre diferentes threads de um mesmo processo. Rotinas de inicialização do TLS são chamadas antes do *entry point*, no momento da criação dos threads.

Thread Local Storage

Funcionamento da técnica do Thread Local Storage (TLS):

- Utilizada para alocar informações sobre threads.
- A estrutura importante no TLS é a IMAGE_TLS_DIRECTORY.

- Dentro dessa estrutura, existe um campo importante: *AddressOfCallBacks.*
- Ele aponta para uma lista de funções de callback.
- Estas funções podem ser executadas no momento da inicialização do arquivo PE na memória, antes de executar o *entry point.*

Inserir código como uma função de callback no TLS permite executar código antes do *entry point* do programa. Isto permite descompactar, decriptografar, ou mesmo modificar o código de forma a confundir quem tentar analisá-lo.

O IDA Pro detecta estas funções de callback e as oferece como *entry points* alternativos. Para visualizá-los, pressione Control+E.

Códigos não alinhados

- JUMP ou CALL no meio de uma função.
- Criptografia ou compactação de executável.
- Otimizações do compilador.
- Virtualização de código.
- Vulnerabilidades.

Outra técnica usada pelos autores de malware inclui truques de controle de fluxo do programa, com JUMPs e CALLs para endereços intermediários de código. Estas técnicas não modificam realmente o fluxo do programa, mas são truques de instruções assembly, que apenas dificultam a visualização do código, mas que durante a execução, permitem ao programa executar normalmente seu fluxo. No exemplo, vemos uma chamada a um endereço intermediário (CALL near ptr loc_AE0007+3), que na verdade equivale a um CALL a um endereço não identificado pelo IDA. Examinando o código da Figura 8.1 percebe-se que o CALL cairia no meio do comando “JMP NEAR PTR 460b04F7h” em 0x00AE0007.

Além desta técnica, o uso de compressores e protetores de executáveis, otimizações do compilador, virtualização de código (como as usadas pelo compactador Themida e pelo protetor ExeCrypter) e *shellcodes* que exploram vulnerabilidades, entre outras, dificultam muito ou tornam impossível a leitura do código assembly. Nestes casos, o analista vai ter um trabalho muito grande para identificar os truques usados, e quase sempre o trabalho não poderá ser replicado para outros arquivos.

```
.aspack:00AE0000 _aspack         segment para public 'DATA' use32
.aspack:00AE0000                 assume cs:_aspack
.aspack:00AE0000                 ;org 0AE0000h
.aspack:00AE0000                 db  90h ; É
.aspack:00AE0001
.aspack:00AE0001 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||
.aspack:00AE0001
.aspack:00AE0001
.aspack:00AE0001                 public start
.aspack:00AE0001 start           proc near
.aspack:00AE0001                 pusha
.aspack:00AE0002                 call    near ptr loc_AE0007+3
.aspack:00AE0007
.aspack:00AE0007 loc_AE0007:                             ; CODE XREF: start+1↑p
.aspack:00AE0007                 jmp     near ptr 460B04F7h
.aspack:00AE0007 start           endp
.aspack:00AE0007
.aspack:00AE0007 ; ---------------------------------------------------------------
.aspack:00AE000C                 db  55h ; U
.aspack:00AE000D                 db 0C3h ; +
```

Figura 8.1 Truques de controle de fluxo do programa.

Modificações no cabeçalho PE

- Valores inválidos.
- Valores de *ImageBase* inválidos.
- Dados inválidos em *LoaderFlags* e *NumberOfRvaAndSizes.*
- Tamanho excessivo em *SizeOfRawDataValues.*
- Técnicas de realocação:
  - Realocação serve para modificar ponteiros no código, caso a base da imagem seja realocada na memória.
  - Pode-se usar essa característica para modificar praticamente qualquer dado.
  - Abusando desta técnica, um autor de programa malicioso pode modificar todo seu código antes do programa ser carregado.

Outra técnica de autores de malware utiliza características dos arquivos PE para dificultar o carregamento do arquivo em programas de debugger e obfuscar o código. Algumas entradas no cabeçalho PE aceitam valores não padronizados, e algumas ferramentas de análise simplesmente não conseguem abrir tais arquivos, apesar de o sistema operacional abri-los normalmente e ignorar os valores inválidos.

Outra técnica usada é a realocação de código. Esta tabela, presente no cabeçalho PE, permite ao programa trocar alguns bytes do seu código, antes do início da execução, para possibilitar a realocação do endereço base do programa. Dessa forma, ele poderá trocar os offsets de algumas chamadas, para refletir o novo endereço base.

Examine o arquivo "\Desktop\malware\mal.Down.crypt.winss.jpg" no LordPE, e veja a tabela de realocação nos diretórios do programa. Veja que as realocações indicam o endereço original do dado (RVA e Offset), e para onde ele vai ser realocado (Far Address).

Usando essa técnica, o autor do programa malicioso consegue modificar código (veja as realocações de funções, que começam com os bytes 55 8B EC A1 0A 5A), dificultando a ação de análise automática do binário.

Structure Exception Handling

Structured Exception Handling (SEH):

- Esta tabela mantém um registro de funções executadas quando "coisas ruins acontecem".
- Por exemplo, podemos registrar uma função para tratar divisões por 0.
- Quando ocorre uma exceção, a cadeia SEH é percorrida para encontrar uma função apropriada.
- Um autor de programa malicioso pode instalar uma função de tratamento de exceção, que pode ser usada para modificar o código.
- Exemplo: PECompact2.

A estrutura SEH permite ao programa executar funções específicas quando ocorre alguma exceção durante a execução normal do código. Estas funções podem ser registradas pelo programa, e sempre que ocorrer uma exceção, a lista de funções registradas será chamada, e aquela que puder tratar a exceção em questão, será executada.

Compiladores visuais como Visual C e Delphi usam essa estrutura para instalar funções de controle dos formulários, botões e outros tipos de objetos visuais usados no programa.

Além disso, ferramentas como o compressor PECompact2 usam esse tipo de estrutura para obfuscar o código gerado. No exemplo, vemos como ocorre a instalação de uma função para tratar exceções, que na verdade é usada para desviar o controle do fluxo do programa, pois logo após instalar a função, o programa gera uma exceção (XOR EAX, EAX; MOV [EAX], ECX), e a função instalada é chamada. Esta função é na verdade a rotina de descompactação do programa.

Figura 8.2
Uso da ferramenta PECompact2.

```
CODE:00401000 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
CODE:00401000
CODE:00401000
CODE:00401000                public start
CODE:00401000 start          proc near
CODE:00401000                mov   eax, offset sub_469830 ; endereco da funcao de excecao
CODE:00401005                push  eax
CODE:00401006                push  large dword ptr fs:0
CODE:0040100D                mov   large fs:0, esp ; instala a excecao no SEH
CODE:00401014                xor   eax, eax        ; EAX == 0
CODE:00401016                mov   [eax], ecx      ; Segmentation fault! Dispara excecao
CODE:00401016 ; ---------------------------------------------------------------------------
CODE:00401018 aPecompact2    db 'PECompact2',0
CODE:00401023                db 0AFh ; >>
CODE:00401024                db  7Eh ; ~
CODE:00401025                db 0CEh ; +
```

Exercício de fixação 3
Ambiente virtualizado

Um código malicioso pode ter um comportamento diferente dentro de um ambiente virtualizado?

Detecção de máquinas virtuais

Detecção de VMWare:

- Alguns programas maliciosos detectam o VMWare e podem modificar seu comportamento.
- Instruções privilegiadas.
- Instrução SIDT e Interrupt Descriptor Table Register (IDTR).
  - http://invisiblethings.org/papers/redpill.html
- Diretórios ou chaves de registro.
- Algumas técnicas podem ajudar, como desabilitar a aceleração da VM ou não instalar o VMWare Tools.
- Algumas pessoas já chegaram a editar o binário do VMWare para modificar nomes de dispositivos típicos.

Durante este curso, vimos que analisar um binário dentro de um ambiente virtual pode facilitar o trabalho, além de ser uma maneira mais segura do que fazer a análise em uma máquina real. Mesmo assim, como já foi comentado antes, o analista precisa estar ciente de que um ambiente virtual, por mais que pareça seguro, pode ser detectado e explorado, prejudicando a análise ou até mesmo comprometendo o sistema hospedeiro.

Existem diversos programas maliciosos e ferramentas que exploram vulnerabilidades na forma como as máquinas virtuais são implementadas. Alguns programas conseguem detectar o VMWare, por exemplo, e modificar seu funcionamento ou simplesmente terminar a execução.

A detecção pode ser feita por diversos mecanismos, como os exemplos listados. O problema com máquinas virtuais criadas pelo VMWare é que em algum momento, elas precisam ter uma interação com o sistema hospedeiro, e essa interação pode ser detectada de dentro do sistema virtual.

Por isso, existem algumas técnicas que podem ser usadas para tentar diminuir a ocorrência deste tipo de detecção. A instalação do VMWare Tools, por exemplo, instala uma série de ferramentas e chaves de registro que são facilmente detectadas por um programa malicioso.

Outra forma de evitar a detecção é desabilitar a aceleração do processador da máquina virtual. Essa aceleração faz com que certas instruções não sejam emuladas pelo VMWare, e sim passadas diretamente à CPU. Para isso, o VMWare instala no sistema virtual uma interrupção que é chamada a cada instrução. Essa interrupção pode ser detectada por um programa que esteja rodando no sistema virtual. Ao desabilitar a aceleração, todas as instruções passam a ser emuladas, tornando o sistema mais lento, mas dificultando a detecção.

Em casos mais extremos, onde o programa malicioso detecta o VMWare através da enumeração de dispositivos, algumas pessoas chegaram a editar o binário do VMWare para modificar o nome destes dispositivos, e impedir o uso desse método de detecção.

Técnicas anti-engenharia reversa

- Apesar de todas as dificuldades apresentadas, as ferramentas apresentadas têm capacidade de combater técnicas anti-engenharia reversa.
- O IDA Pro, por carregar o binário para um banco de dados, e não executá-lo, permite o carregamento de quase todos os binários que for analisar.
- Lembre-se: para cada técnica usada pelos autores de programas maliciosos, sempre será possível desenvolver uma técnica contrária!

Lembramos que mesmo com todas as dificuldades, a escolha correta das ferramentas permite a análise de praticamente qualquer binário.

Os plugins do OllyDBG são de grande ajuda para analisar alguns arquivos protegidos por técnicas de obfuscação, valores inválidos em estruturas, técnicas de identificação de debugger etc. Habilite todas as opções do plugin Olly Advanced, Hide Debugger e do IsDebuggerPresent e Hide Caption, para que você possa analisar até os programas mais problemáticos.

Um plugin alternativo do Olly é o Phant0m. Com algumas proteções a mais, esse plugin permite analisar binários compactados com Themida ou ExeCryptor.

Para utilizar o plugin Phant0m (com um zero), é necessário marcar a opção “Expand plugin limit to 127 plugins” no Olly Advanced. Depois disso, basta renomear o arquivo “Phant0m.dl_” no diretório do Olly e reiniciá-lo.

Finalmente, é importante lembrar que toda técnica de obfuscação de código pode ser quebrada e revertida de algum modo, dependendo da capacidade do analista e das ferramentas que utiliza, e do tempo que se quer investir nessa tarefa.

Roteiro de Atividades 8

Atividade 8.1 – Detecção de debugger

1. Execute o arquivo "\Desktop\malware\IsDebuggerPresent.exe". O que acontece?

2. Abra o arquivo no IDA Pro, e execute novamente. Alguma coisa mudou?

3. Procure pela função que detecta o debugger;
4. Remova esta proteção usando o OllyDBG;
5. Faça um relatório explicando o que aconteceu nas duas execuções, e como você fez para remover a proteção, com a parte do código relevante comentada;
6. Faça um dump do programa corrigido.

No OllyDBG, basta digitar qualquer caractere na linha que deseja mudar, e ele insere estes caracteres nessa linha.

Atividade 8.2 – Thread Local Storage

1. Abra o arquivo "\Desktop\malware\tls.exe" no IDA Pro.
2. Marque um breakpoint na função "Start" e execute. O que aconteceu?

3. Encontre a função executada antes do *entry point*;
4. Marque um breakpoint e execute novamente. E agora?
5. Comente e marque o código relevante no IDA, incluindo o local onde essa função é chamada.
6. Discuta com seus colegas como combater este tipo de técnica. Escreva um relatório com o resultado das execuções acima e suas conclusões sobre a discussão.

Atividade 8.3 – Códigos não alinhados

1. Abra o arquivo "\Desktop\malware\mal.Banker.aspack.ActiveX.exe" no IDA Pro;
2. Identifique as partes do código que usam técnicas anti-debugger (comentários no IDA);
3. Modifique o banco de dados do IDA para mostrar o fluxo correto do programa;
4. Procure na internet por uma análise sobre o compressor usado, e comente no IDA os pontos do código que fazem a descompactação do arquivo;

5. Com o OllyDBG, encontre o *entry point* original e descompacte o programa (como vimos no Capítulo 7). Faça um dump do binário.
6. Faça um relatório descrevendo os passos executados neste exercício, o arquivo *.IDB* gerado pelo IDA e o binário extraído com o Olly, com uma descrição das informações importantes encontradas no binário.

Atividade 8.4 – Structured Exception Handling

1. Abra o arquivo "\Desktop\malware\mal.Down.PECompact.WindowsUpdate.exe" no OllyDBG;
2. Identifique as partes do código que usam a técnica de SEH (comentários no Olly);
3. Procure na internet por uma análise sobre o compressor usado. Comente no Olly os pontos do código que fazem a descompactação do arquivo;
4. Encontre o *entry point* original e descompacte o programa (como visto no Capítulo 7). Faça um dump do binário;
5. Faça um relatório com os passos executados nesta atividade, o binário extraído com o Olly, e com uma descrição das informações importantes encontradas no binário.

9

Análise de um worm

objetivos

Apresentar um roteiro de análise e pontos importantes durante a análise de um código malicioso, assim como técnicas de detecção das principais funcionalidades.

conceitos

Emulação, análise de strings, análise da tabela de importação, visualização gráfica, análise de backtrace.

Exercício de nivelamento 1
Análise de strings

Por que a análise das strings encontradas no executável pode facilitar o processo de identificação de funções de um código malicioso?

A aplicação de assinaturas de código ao IDB pode facilitar a análise? Por quê?

Descobrindo o básico

Roteiro de engenharia reversa:

- Comece sempre pelo mais simples:
  - Strings.
  - Funções importadas.
  - Aplicação de assinaturas.
- Examine o gráfico de chamadas e nomes de estruturas identificadas automaticamente.
- Passe rapidamente pelo binário nomeando as funções mais óbvias.
- Se necessário, identifique strings obfuscadas.
- Examine o backtrace de funções importadas.
  - Quem as utiliza?
  - Quais parâmetros são passados?

Nos Capítulos 9 e 10, veremos como realizar uma análise detalhada de um programa malicioso real, identificando suas funcionalidades e capacidades. Neste ponto, o aluno já tem a base teórica necessária para começar a se aventurar, porque as atividades anteriores deram um bom exemplo de como usar as ferramentas para analisar programas.

A partir de agora, podemos então começar a usar um método durante nossa análise, para facilitar o trabalho e permitir a reprodução dos resultados, caso necessário.

Como a engenharia reversa pode ser uma tarefa demorada e complexa, a melhor estratégia é começar a análise do programa pelas ferramentas mais fáceis, procurando as informações mais prontamente disponíveis. Por isso, comece pelo mais simples:

- Extraia as strings do arquivo, para ver se elas dão alguma dica do que o programa faz, e para usá-las como ponto de partida em sua análise. É muito mais fácil identificar uma função que usa a string "HKLM\Software\Microsoft\Windows\CurrentVersion\Run", do que uma que não utilize nenhuma string visível.
- Examine as funções importadas. Uma chamada à *CreateFileA()* ou *LoadLibraryA()* deve conter alguma informação importante sobre o funcionamento do programa.
- Aplique as assinaturas apropriadas. Use a funcionalidade do IDA de detectar código padrão, e evite o trabalho de analisar código de bibliotecas do sistema.
- Examine os nomes de estruturas detectadas automaticamente e o gráfico de chamadas de funções. Elas podem dar indicações das operações que aquela função executa.
- Examine as funções mais óbvias e dê nomes claros. Ao analisar os gráficos de funções que chamam essas funções mais simples, o funcionamento delas pode se tornar mais claro.
- Se o código contiver strings "obfuscadas", identifique-as e tente desobfuscá-las usando os plugins do IDA.
- Examine chamadas a funções importadas. Quais funções as utilizam? Quais parâmetros são passados? Isso pode ajudá-lo a identificar funcionalidades e descobrir os tipos de variáveis.

Veremos neste capítulo um pouco mais sobre essas técnicas mais simples.

Responda às perguntas específicas:

- Como ele se espalha?
  - Por e-mail? Como são enviados?
  - Por rede? Compartilhamentos ou redes P2P?
  - Ele explora alguma vulnerabilidade?
- Ele contém um backdoor?
  - Qual porta é aberta?
  - Ele conecta em algum lugar?
- Que modificações são feitas no sistema?
  - Quais arquivos são criados?
  - Quais chaves de registro são criadas?
- Existe alguma data especial?
  - Existe alguma data para ativar uma atividade específica?

Analisar o binário simplesmente por analisar não é eficiente. Tenha em mente perguntas específicas que queira responder. Existem tópicos mais genéricos com os quais devemos nos preocupar ao analisar um programa malicioso. Pense nessas perguntas, escolha a melhor sequência de ações para respondê-las, e fique focado em seu objetivo.

Analisar um binário desconhecido pode ser tão complexo que o analista poderia levar meses realizando essa tarefa. Se seu objetivo é somente identificar formas de se proteger contra tal praga, não é necessário analisar todo o código, função por função.

Exercício de fixação 1
Identificação do código

Quando não for possível identificar o código, como podemos proceder?

Emulação

- Plugin X86Emu para IDA Pro Free disponível em:
  - http://www.idabook.com/ida-x86emu/
- Para usá-lo, copie para o diretório Plugins do IDA e:
  - Carregue o malware no IDA.
  - Execute o plugin IDA-x86-meu.
  - Clique em "push data".
  - Entre com "0 0 0 0 0 0 0 ...." para criar um espaço no Stack.
  - Dê duplo clique em ESP e copie o valor para EBP.
  - Pronto, basta executar o programa com Run ou Step.
- Com ele, é possível emular apenas partes do código.
  - Coloque os valores apropriados no Stack.
  - Modifique os registradores, se necessário.
  - Posicione o cursor na primeira instrução a emular.
  - Execute com Run ou Step.

Em último caso, se não for possível identificar uma parte do código, seja por ele estar obfuscado ou simplesmente por ser muito complexo, tente executá-lo em um ambiente emulado.

O IDA Pro tem um plugin para emular uma CPU X86. Com ele, é possível executar o programa malicioso sem correr o risco de comprometer a máquina, pois as instruções serão executadas por um processador virtual.

Os passos descritos acima ensinam a usar o emulador. Uma das vantagens de executar o código emulado é que não é necessário executar todo o código para chegar a um ponto específico.

Por exemplo, se o analista descobrir uma função que faz a descompactação de um código ou criptografia de um dado, pode executar somente aquela parte do código. Basta identificar corretamente os registradores utilizados pelo código, os parâmetros que a função precisa ter no Stack, e criar esses dados de acordo. Então, ao executar aquelas linhas de código, o programa vai pensar que está sendo executado normalmente.

Strings

- Quais funções usam essas strings?
- Procure e tente identificar suas funcionalidades.
- Lembre-se de usar os comentários e de renomear as funções e variáveis que identificar.

Durante estes dois capítulos, usaremos um arquivo criado no Roteiro de Atividades 7, em que vimos como descompactar um arquivo UPX e realizar o dump do processo em memória para um arquivo. É este arquivo descompactado que iremos usar em nossos exemplos. Se você não guardou o resultado daquela sessão, deve fazê-lo agora, e guardar o arquivo em um diretório fora da máquina virtual, para poder restaurá-la a um estado sadio.

Ao abrir o arquivo no IDA Pro, perceba que o programa realiza uma análise prévia do arquivo e identifica uma parte de código (em azul) e muitas áreas de dados (em cinza). Utilizando o navegador para examinar essas regiões em cinza, você poderá ver que estas áreas quase sempre correspondem a strings usadas pelo programa. Ao examinar a aba de strings do IDA, é possível identificar diversos textos característicos de mensagens de e-mail, nomes de usuários, comandos de comunicação com o servidor SMTP, entre outras mensagens.

Address	Length	Type	String
"...."UPX0:0...	00000007	C	%s.zip
"...."UPX0:0...	00000005	C	html
"...."UPX0:0...	0000000C	C	%d%d%d%d
"...."UPX0:0...	00000041	C	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345...
"...."UPX0:0...	0000009B	C	X-Pririty: Normal\r\nX-Mailed: Microsoft Outlook Expr...
"...."UPX0:0...	00000031	C	Content-Type: multipart/mixed;\r\n\tboundary=\"%s\"\r\n
"...."UPX0:0...	00000016	C	\r\nMINE-Version: 1.0\r\n
"...."UPX0:0...	0000000	C	Date:
"...."UPX0:0...	0000000E	C	Subject:%s\r\n
"...."UPX0:0...	00000009	C	To:%s\r\n
"...."UPX0:0...	0000000B	C	From:%s\r\n
"...."UPX0:0...	0000001D	C	----=_%s_%.3u_%.4u_%8.X.%8.X
"...."UPX0:0...	00000009	C	NextPart
"...."UPX0:0...	0000000F	C	\r\n\r\n--%s--\r\n\r\n
"...."UPX0:0...	0000008B	C	--%s\r\nContent-Type: application/octet-stream;\r\n\tname=\"%s\"\"\r\nCont...
"...."UPX0:0...	00000007	C	inline
"...."UPX0:0...	00000007	C	\r\n\r\n\r\n
"...."UPX0:0...	00000058	C	--%s\r\nContent-Type: text/plain;\r\n\tcharset=us-ascii\r\nContent-Transfer-...
"...."UPX0:0...	00000031	C	This is a multi-part message in MINE format \r\n\r\n
"...."UPX0:0...	00000007	C	QUIT\r\n
"...."UPX0:0...	00000006	C	\r\n.\r\n
"...."UPX0:0...	00000007	C	DATA\r\n
"...."UPX0:0...	0000000D	C	RC%s0:<%s>\r\n
"...."UPX0:0...	00000005	C	PT T

Funções importadas

Figura 9.1
Aba de strings do IDA.

- Quais funções usam essas chamadas?
- Tente identificar a funcionalidade delas.
- Você consegue encontrar os parâmetros passados?

Outra forma de descobrir rapidamente os objetivos de uma função é examinando as chamadas a bibliotecas importadas. Estas chamadas geralmente permitem identificar os parâmetros passados e a operação que as funções tentam executar.

Uma maneira simples de realizar essa operação no IDA é examinar a aba de funções importadas e acessar a área do disassembly onde elas estão localizadas. A partir de cada função, basta usar os comandos de referências cruzadas e identificar as funções que fazem chamadas a essas funções importadas.

Procure identificar todas as funções chamadas, pois elas fornecem uma ótima indicação do funcionamento do programa.

00501004	RegOpenKeyExA	ADVAPI32
00501008	RegSetValueExA	ADVAPI32
0050100C	RegQueryValueExA	ADVAPI32
00501010	RegEnumKeyA	ADVAPI32
00501014	RegCreateKeyExA	ADVAPI32
0050101C	FindClose	kernel32
00501020	GetFileSize	kernel32
00501024	FindNextFileA	kernel32
00501028	MapViewOfFile	kernel32
0050102C	UnmapViewOfFile	kernel32
00501030	FileFirstFileA	kernel32
00501034	GetEnvironmentVariableA	kernel32
00501038	GetDriveTypeA	kernel32
0050103C	GetSystemTime	kernel32
00501040	WriteFile	kernel32
00501044	CreateFileMappingA	kernel32
00501048	LoadLibraryA	kernel32
0050104C	CreateProcessA	kernel32
00501050	GlobalAlloc	kernel32
00501054	GetLastError	kernel32
00501058	CreateMutexA	kernel32
0050105C	IstrcatA	kernel32
00501060	GetFileAttributesA	kernel32
00501064	CopyFileA	kernel32
00501068	DeleteFileA	kernel32

Figura 9.2
Aba funções importadas do IDA.

Assinaturas de código

Aplicação de assinaturas:

- Use assinaturas para identificar automaticamente funções de bibliotecas padrão.
- Compiladores visuais costumam inserir muitas funções de controle de objetos que não precisam ser analisadas. Infelizmente, nem sempre elas são úteis.

Além de funções importadas de DLLs do sistema, um programa pode conter código padrão inserido pelos compiladores, ou pode ter sido compilado estaticamente com suas bibliotecas. Neste caso, essas funções estarão misturadas ao código inserido pelo autor do programa. Exemplos desse tipo de código são inseridos por compiladores visuais (Visual C, C++ Builder, Delphi). Eles inserem funções de manutenção dos controles usados no formulário.

Para distingui-las do código inserido pelo autor, e evitar o trabalho de analisá-las, usaremos as assinaturas do IDA. Normalmente essas assinaturas estão associadas a um compilador específico.

Descobrir no código qual foi o compilador usado pode ser muito útil. Por exemplo, programas compilados com compiladores visuais da Borland tendem a apresentar diversas strings referenciando chaves de registro com o nome Borland. Isto pode ser uma indicação de que este compilador foi usado. Mesmo assim, às vezes é difícil identificar o compilador, como é o caso do programa analisado. Neste caso, o analista deverá inserir todas as assinaturas que achar prováveis, e ver quais delas identificam código.

```
d3vcl       Applied   7   Delphi 3 Visual Component Library
b32vcl      Applied   3   Borland Visual Component Library & Packages
bcb5rt      Applied   3   CBuilder 5 runtime
bds8vcl     Applied   3   BDS2008 Component Library & Packages
bcb5rt      Applied   3   CBuilder 5 runtime
c4vcl       Applied   2   CBuilder 4 and Delphi 4 VCL
d4vcl       Applied   2   Delphi 4 Visual Component Library
d5vcl       Applied   2   Delphi 5 Visual Component Library
bds2006     Applied   2   Delphi2006/BDS2006 Visual Component Library
bds2007     Applied   2   Codegear Rad Studio 2007 Visual Component Library
bds40       Applied   2   BDS 4.0 RTL and VCL
msmfc2      Applied   1   MFC32 WinMain detector
bds         Applied   1   BDS 2005-2007 and Delphi6-7 Visual Component Library
bds         Applied   1   BDS 2005-2007 and Delphi6-7 Visual Component Library
exe         Applied   1   Startups of EXE/COM files
bh32ocf     Applied   0   Borland OCF 32 bit
bh32dbe     Applied   0   Borland DBE 32 bit
bh32rw32    Applied   0   BCC v4.x/5.x & BCB v1.0/v7.0 BDS2006 win32 runtime
bp32_2      Applied   0   Borland Delphi/C++Builder VCL
msddk32     Applied   0   DDK Windows 32bit
pe          Applied   0   Startups of PE files
```

Visualização gráfica

Figura 9.3
Uso das assinaturas do IDA.

Verificar estrutura do *entry point*:

- Use a opção "View → Graphs → User xref chart", mudando os valores de "Recursion Depth".
- Tente identificar funções que distribuem o fluxo do programa, ou aquelas que concentram as chamadas, nas pontas do gráfico.

Após identificar algumas funções usando os métodos descritos até agora, o analista começa a ter uma ideia das funcionalidades do programa. Ele já pode tentar visualizar as referências cruzadas e descobrir quais funções chamam ou são chamadas pelas que foram identificadas até o momento.

Para isso, os gráficos de chamada podem ser úteis. Por exemplo, se o analista optar por usar uma metodologia de pesquisa *top-down,* poderá usar os gráficos de referências cruzadas editáveis para criar gráficos mais específicos.

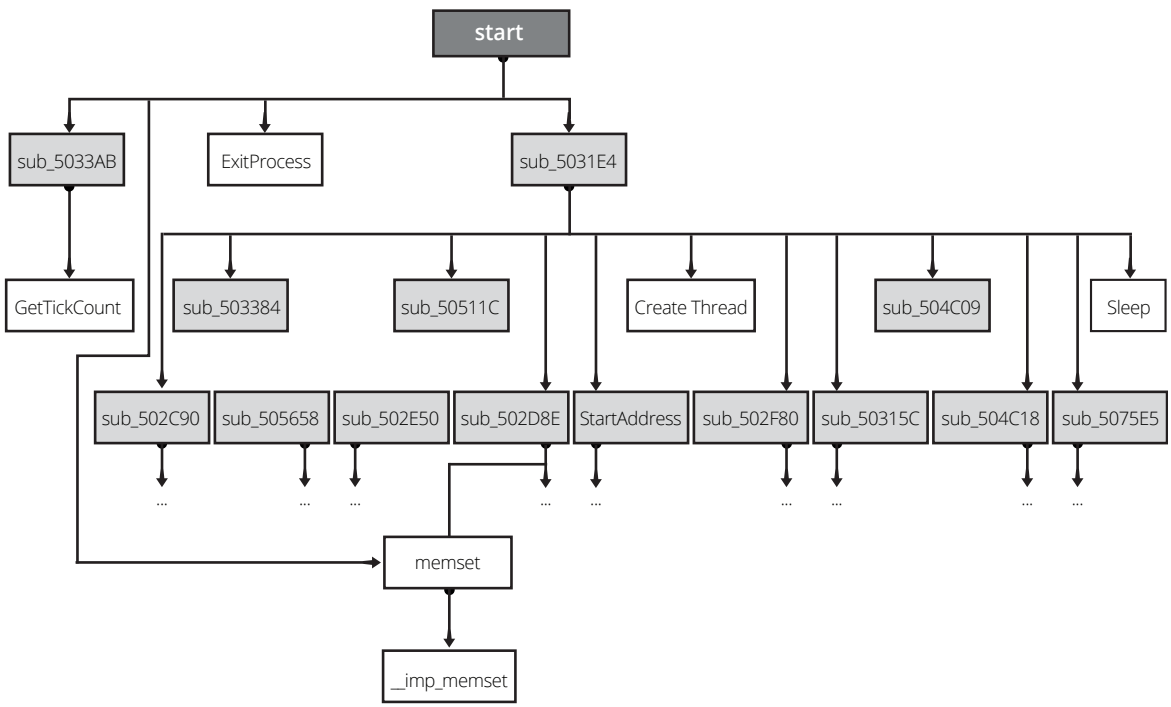
Por exemplo, para gerar um gráfico com as funções chamadas a partir da função "Start", com dois níveis de recursão, o analista pode mudar a opção "Recursion Depth" para 2, desmarcar a opção "Cross Reference To" e, se quiser, desmarcar a opção "Ignore Data" e marcar a opção "Print Comments".

No gráfico, duas funções de bibliotecas e duas funções desconhecidas são chamadas a partir do *entry point*, sendo que uma dessas últimas chama diversas outras funções desconhecidas. Você pode escolher analisar a mais simples primeiro, ou partir para a identificação da função que chama as demais funções do programa.

```
                public start
start           proc near               ; CODE XREF:

var_298         = dword ptr -298h
ThreadId        = dword ptr -108h

                push    ebp
                mov     ebp, esp
                sub     esp, 298h
                lea     eax, [ebp+var_298]
                push    eax
                push    101h
                call    dword_50116C
                call    sub_5033A8
                push    108h
                lea     eax, [ebp+ThreadId]
                push    0               ; int
                push    eax             ; void *
                call    memset
                push    eax             ; ThreadId
                call    sub_5031E4
                add     esp, 10h
                push    0               ; uExitCode
                call    ExitProcess
```



Exercício de fixação 2
Identificação do código II

Muitas vezes partir do *entry point* do arquivo para descobrir um funcionamento é muito trabalhoso. Conhecendo algum comportamento do artefato em questão, o que podemos utilizar para facilitar este processo de identificação do código?

Figura 9.4
Chamada de funções de biblioteca e funções desconhecidas.

Análise de backtrace

Verificar backtrace de funções importadas:

- Navegue até a área de funções importadas.
- CTRL-X abre janela de referências para uma função.
- Identifique como o código faz a chamada à função importada.

A análise do código através de backtrace (análise reversa) das funções importadas permite ao analista descobrir rapidamente funções que fazem uso das funções importadas de DLLs.

Para encontrar o código que faz referência às funções importadas, você deve procurar pela região onde estas funções estão armazenadas no código, e usando o atalho Control+X, descobrir os códigos que fazem referência àquela função. Esta técnica vale também para tentar descobrir as partes do código que usam a função analisada.

No exemplo, observamos que uma referência à função *RegOpenKeyExA()* é usada para criar uma chave no registro. O texto passado como parâmetro à função não aparece na lista de strings, pois cada caractere foi passado individualmente para o Stack, e dessa forma a string nunca aparece inteira no código. Esta é uma técnica que os autores de programas maliciosos usam para dificultar a identificação de características do código.

No exemplo, os caracteres referentes à chave de registro criada são normalmente apresentados em bytes hexadecimais. Para transformar bytes em caracteres, selecione o byte e pressione 'R'.

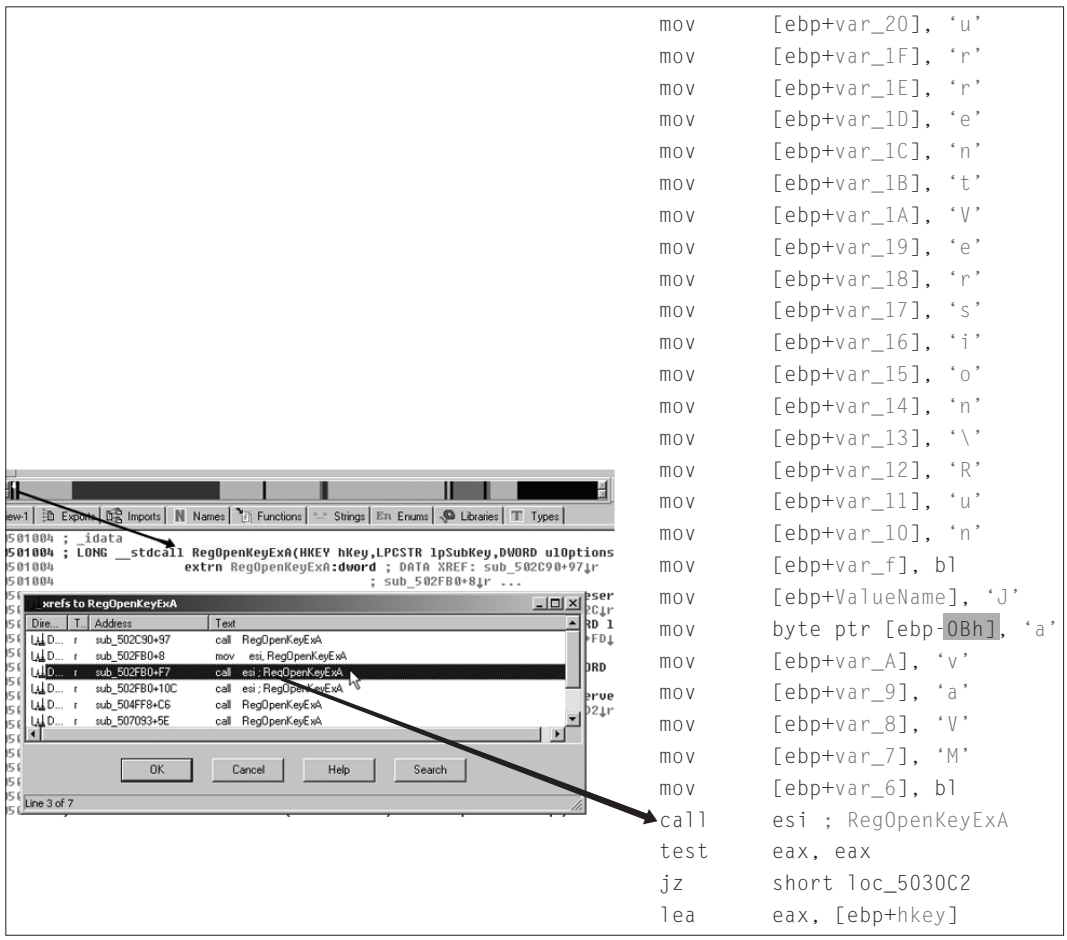


Figura 9.5
Referência à função RegOpenKeyExA.

Funções básicas

Para identificar funções básicas:

- Comece pelas funções do *entry point* e por aquelas que utilizam funções importadas.
- Lembre-se de comentar o código e renomear funções e variáveis (selecione o nome e use 'N').
- Use F12, CTRL-SHIFT-F e CTRL-SHIFT-T para examinar o gráfico de fluxo e o gráfico de chamadas das funções que analisar.
- Algumas vezes este procedimento ajuda a entender o funcionamento do código.

```
SaveCurrentTime proc near               ; CODE XREF: start+1B↑p
                call   GetTickCount
                mov    ds:nSavedCurrentTime, eax ; EAX contem o valor de retorno de GetTickCount()
                retn
SaveCurrentTime endp

; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||


RandomNumberGenerator proc near         ; CODE XREF: sub_5031E4+57↑p
                mov    eax, ds:nSavedCurrentTime ; Usa SavedCurrentTime
                imul   eax, 343FDh     ; Multiplica por um numero grande
                add    eax, 269EC3h    ; adiciona a outro numero grande
                mov    ds:CurrentSavedTime, eax ; Salva em uma variavel
                mov    ax, word ptr ds:nSavedCurrentTime+2 ; Copia 2 bytes do SavedCurrentTime
                                       ;para AX (parte de EAX)
                retn
RandomNumberGenerator endp
```

Figura 9.6
Identificação de funções básicas.

Análise de código

Agora é a sua vez! Consegue chegar até aqui?

Function	Segment	Length	R	F	L	S	B	T	=
CheckfPunctuation	UPX0	00000022	R	.	.	.	.	.	.
CompareStringsWithTable	UPX0	00000036	R	.	.	.	.	.	.
CompareTwoStrings	UPX0	00000030	R	.	.	.	B	.	.
CreateRegKeyMutex	UPX0	000000FE	R	.	.	.	B	.	.
CreateRegKeyRun	UPX0	00000140	R	.	.	.	B	.	.
CreateSystemMutex	UPX0	000000C2	R	.	.	.	B	.	.
CreateTempFile	UPX0	00000160	R	.	.	.	B	.	.
FindOutlookIEWindow	UPX0	0000003F	.	.	.	.	.	T	.
FindOutlookAddressBook	UPX0	00000124	R	.	.	.	B	.	.
GenerateRandomEmailID	UPX0	00000016	R	.	.	.	.	.	.
GenerateTimestamp	UPX0	000000DD	R	.	.	.	B	T	.
GetInternetConnectionState	UPX0	00000077	R	.	.	.	B	.	.
MainThread	UPX0	0000009C	R	.	.	.	B	T	.
MaybeConvertStringToLowercase	UPX0	0000004A	R	.	.	.	B	.	.
RandomNumberGenerator	UPX0	0000001C	R	.	.	.	.	.	.
RegisterProcessAsService	UPX0	00000088	R	.	.	.	B	.	.
SaveCurrentTime	UPX0	0000000C	R	.	.	.	.	.	.
SendMsgToOutLookAndIE	UPX0	0000002C	R	.	.	.	.	T	.
memcpy	UPX0	00000006	R	.	.	.	.	T	.
memset	UPX0	00000006	R	.	.	.	.	T	.
start	UPX0	0000004A	.	.	.	.	B	.	.
sub_5034BB	UPX0	00000168	R	.	.	.	B	T	.
sub_503697	UPX0	000000E6	R	.	.	.	B	T	.
sub_50377D	UPX0	0000009C	R	.	.	.	.	.	.

Figura 9.7
Análise de código.

No próximo capítulo, começaremos a descrever as características gerais do programa, seus métodos de ação, as informações que podem ser usadas para identificar pragas na rede e as características do programa que permitem a criação de métodos de proteção contra novas infecções.

Roteiro de Atividades 9

Atividade 9.1 – Strings

1. Abra o arquivo “mal.upx.6dddd7e8e5ff88a15b7884a833ff893b.dumped.dat” criado no Roteiro de Atividades do Capítulo 7;
2. Examine as strings contidas no executável;
3. Encontre e analise pelo menos três funções que usem estas strings;
4. Examine as funções que utilizam essas strings, e tente identificar como elas funcionam;
5. Use os comentários no código e renomeie as variáveis e nomes de função de acordo com o que encontrar.

Atividade 9.2 – Funções importadas

1. Examine as funções importadas pelo programa;
2. Encontre e analise pelo menos três funções que utilizem estas chamadas a bibliotecas.

Atividade 9.3 – Assinatura de código

1. Aplique as assinaturas que achar relevantes ao código;
2. Identifique aquelas que foram úteis, e determine quantas funções de biblioteca foram reconhecidas;
3. Faça um breve relatório com suas conclusões sobre a aplicação de assinaturas neste caso específico.

Atividade 9.4 – Visualização gráfica

1. Tente identificar e analisar as funções básicas a partir do início (top-down); observe que uma única função chama diversas outras. Atenção para ela!
2. Identifique as funções chamadas a partir do *entry point* e pelo menos três chamadas a partir de “sub_5031E4”;
3. Se ainda houver tempo, identifique mais funções.

Atividade 9.5 – Análise de backtrace

1. Navegue até a área de funções importadas (em rosa no navegador);
2. Dê CTRL-X para abrir a janela de referências para uma função;
3. Identifique como o código faz a chamada à função importada, para identificar e analisar pelo menos cinco funções do programa.

Atividade 9.6 – Funções básicas

1. Comece pelas funções do *entry point* e por aquelas que utilizam funções importadas;
2. Lembre-se de comentar o código e renomear funções e variáveis (selecione o nome e use 'N');
3. Use F12, CTRL-SHIFT-F e CTRL-SHIFT-T para examinar o gráfico de fluxo e gráfico de chamadas das funções que analisar. Algumas vezes isso ajuda a entender o funcionamento do código;
4. Identifique pelo menos três funções básicas nas pontas do gráfico de chamada do programa.

10

Análise do worm MyDoom

objetivos

Aplicar o conhecimento adquirido durante em uma análise de uma ameaça real, descobrindo suas principais funcionalidades, assim como detectar funções escondidas dentro do código.

conceitos

Worm MyDoom, formas de propagação, infecção do sistema, detecção de backdoor, script IDC.

Exercício de nivelamento 1
Análise de backtrace

Como podemos identificar a função de criação de mutex de sistema utilizando análise de backtrace? Por onde começar?

Que forma este worm utiliza para se manter ativo no sistema após o boot?

Análise do MyDoom

O malware que estamos analisando é uma variante do MyDoom. Temos que responder a algumas perguntas:

- Como ele se propaga?
- Que dados ele acessa ou modifica?
- Como ele se mantém no sistema?
- Existe algum backdoor escondido?
- Que ações ele toma após invadir o sistema?
- Qual o impacto da presença do programa malicioso no computador do usuário?

Neste capítulo, vamos finalizar a análise do programa malicioso que iniciamos no capítulo anterior. Como você já deve ter identificado diversas funções nas atividades do Capítulo 9, usaremos esse conhecimento para identificar as características mais genéricas do programa.

O programa que analisamos é uma variante do worm MyDoom. Precisamos agora responder as perguntas propostas, identificando funções específicas que possam nos ajudar a respondê-las.

Como adotamos uma abordagem de analisar primeiramente as funções mais óbvias, temos agora uma parte das funções identificadas e uma visão geral de como estão distribuídos os dados e códigos do programa. O próximo passo é juntar todas essas funções em módulos maiores que descrevam o funcionamento completo do programa.

Para criar uma marca no IDA Pro, posicione o cursor na posição desejada e pressione Alt+M. Para navegar pelos marcadores, use Control+M.

Exercício de fixação 1
Aplicando os conhecimentos

Como poderíamos aplicar os conhecimentos adquiridos em uma análise completa de um artefato malicioso? Por onde começar?

Como ele se propaga?

- Procure por funções de acesso à rede ou de envio de e-mail.
- Procure funções de criação e leitura de arquivo.
- Veja nas strings se existe alguma dica e descubra funções para criar um spam.
- Existe algum acesso à rede?

Para responder a primeira pergunta, devemos primeiro pensar nos métodos de propagação que o programa poderia usar para se propagar: acesso a diretórios compartilhados, exploração de vulnerabilidades remotas, spam com o programa malicioso anexado, entre outros.

Depois de ter em mente possíveis métodos de propagação, o analista deve procurar pelas funções importadas que permitem acesso à rede ou criação e acesso a arquivos. Normalmente, procurar na lista de strings pode dar uma dica de quais arquivos são acessados, se o programa cria algum e-mail ou se acessa algum diretório ou compartilhamento remoto.

Examinando as strings do programa, podemos perceber que diversas delas indicam textos que provavelmente são usados em um e-mail. Fazendo o backtrace das chamadas a essas strings, você poderá encontrar as funções que criam o spam.

Para enviar o spam, o programa deve utilizar algum método de comunicação com a rede. Alternativas possíveis seriam o uso de um cliente SMTP interno ao programa, o uso do servidor de e-mail configurado no sistema ou de algum servidor aberto para relay.

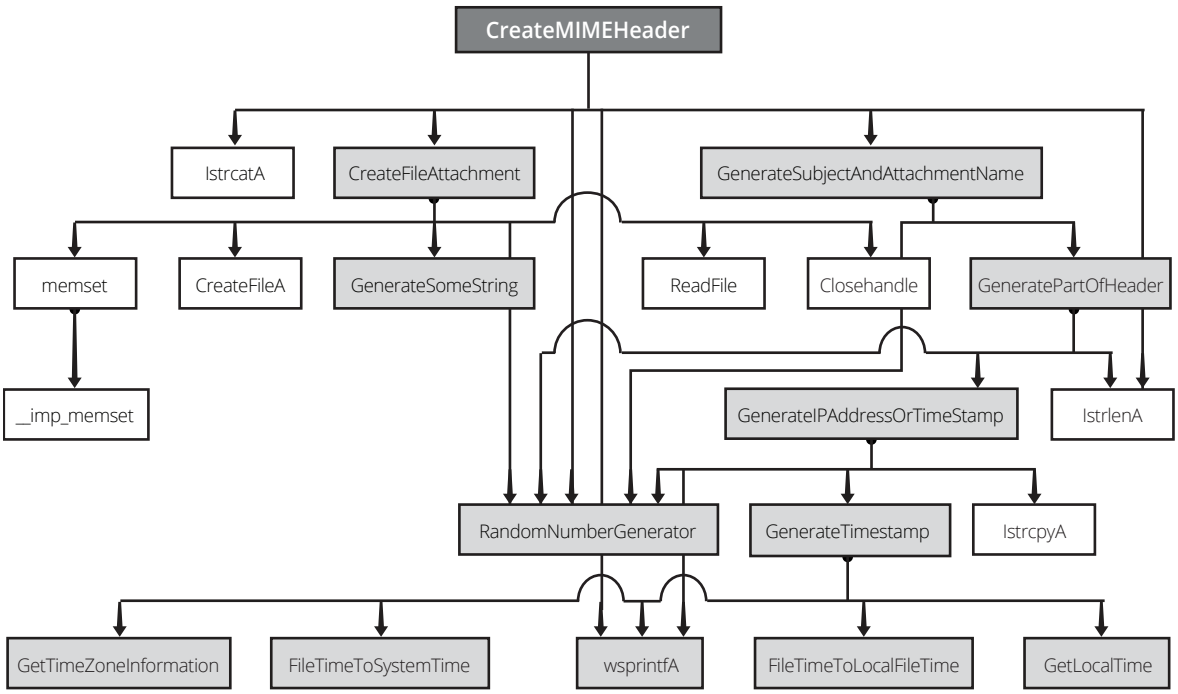


Figura 10.1 Módulo que cria o cabeçalho da mensagem MIME.

A figura anterior mostra o módulo que cria o cabeçalho da mensagem MIME que será enviada com o worm em anexo. As chamadas às funções importadas estão em cinza no gráfico. Neste gráfico podemos perceber duas partes principais, a que cria o arquivo anexado e a que gera o timestamp e os IPs que serão usados na criação do cabeçalho falso. Além disso, podemos ver que ambas as partes usam a função de geração de números randômicos que vimos anteriormente.

Como é a comunicação com o servidor de e-mail?

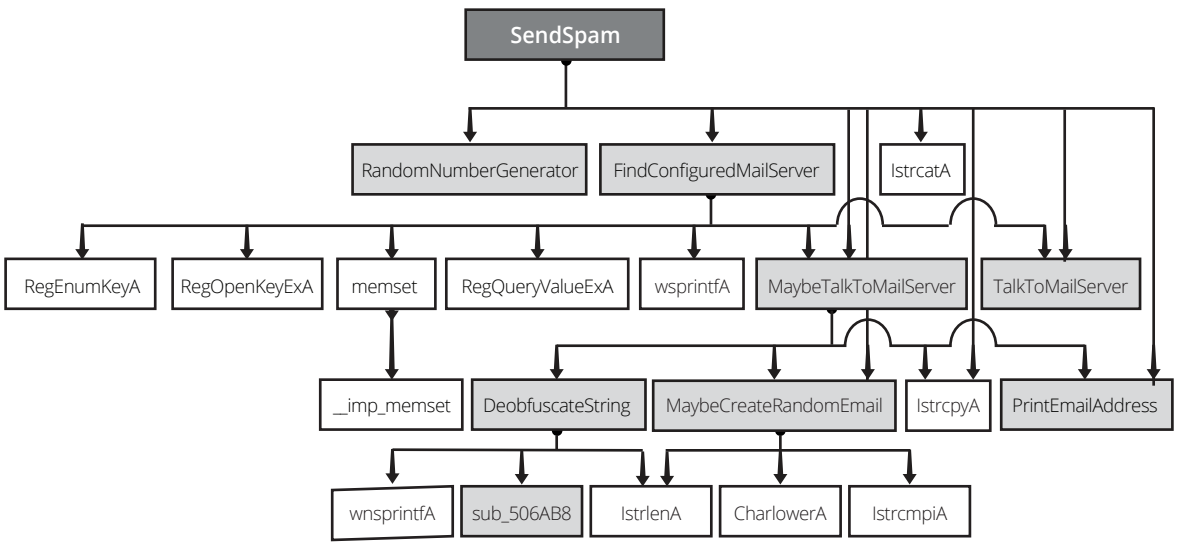


Figura 10.2 Funções não identificadas ou não compreendidas.

No caso do MyDoom, a propagação ocorre através do envio de e-mails para o servidor SMTP configurado no sistema. Inicialmente ele pesquisa o registro procurando pelas configurações de e-mail do Outlook, e depois se conecta ao servidor, enviando os comandos necessários.

Observe na figura que algumas funções não foram identificadas ou não foram compreendidas corretamente. Sugerimos o exercício de tentar corrigir essas funções e identificá-las corretamente.

Quais dados são acessados ou modificados?

- Verifique os arquivos que são criados.
- Quais chaves de registro são lidas/escritas/modificadas?
- O programa acessa mais algum arquivo no sistema?
  - FindFirstFileA()
  - FindNextFileA()
  - MapViewOfFile()
  - GetDriveType()
  - GetEnvironmentVariableA()
  - Entre outros.

Outra pergunta importante a responder, principalmente para auxiliar na recuperação da máquina invadida, é saber quais arquivos ou dados do sistema são modificados quando a máquina é infectada.

Para descobrir isso, analise as chamadas às funções de criação/acesso a arquivos, acesso ao registro, e outras funções de acesso a arquivos e diretórios.

As funções sugeridas acima mostram apenas uma parte das possíveis funções de acesso a arquivo que podem ser encontradas no programa.

- Outra pergunta importante a responder, principalmente para auxiliar na recuperação da máquina invadida, é saber quais arquivos ou dados do sistema são modificados quando a máquina é infectada.
- Para descobrir isso, analise as chamadas às funções de criação/acesso a arquivos, acesso ao registro, e outras funções de acesso a arquivos e diretórios.
- As funções sugeridas mostram apenas uma parte das possíveis funções de acesso a arquivo que podem ser encontradas no programa.

```
                                        ; sub_504AE2+D9↓p
                                        ; SearchforEmailinFiles+C5↓p
                                        ; CreateMapViewOfFile+EB↓p ___
                                        ; CloseHandle (import by name)
; HANDLE __stdcall CreateFileA(LPCSTR lpFileName,DWORD dwDesiredAccess, DWORD dwShareMode, LPSE
                extrn CreateFileA:dword ;
                                        ;
                                        ;
                                        ;
                                        ;
; BOOL __stdcall SetFileAttributesA(LPCST
                extrn SetFileAttributesA;
                                        ;
                                        ;
                                        ;
; int __stdcall lstrlenA(LPCSTR ipString)
                extrn lstrlenA:dword    ;
                                        ;
                                        ;
                                        ;
                                        ;
; DWORD __stdcall GetTempPathA(DWORD nBuf
                extrn GetTempPathA:dword
                                        ;
                                        ;
                                        ;
                                        ;
; UINT __stdcall GetWindowsDirectoryA(LPS
                extrn GetWindowsDirectory
```

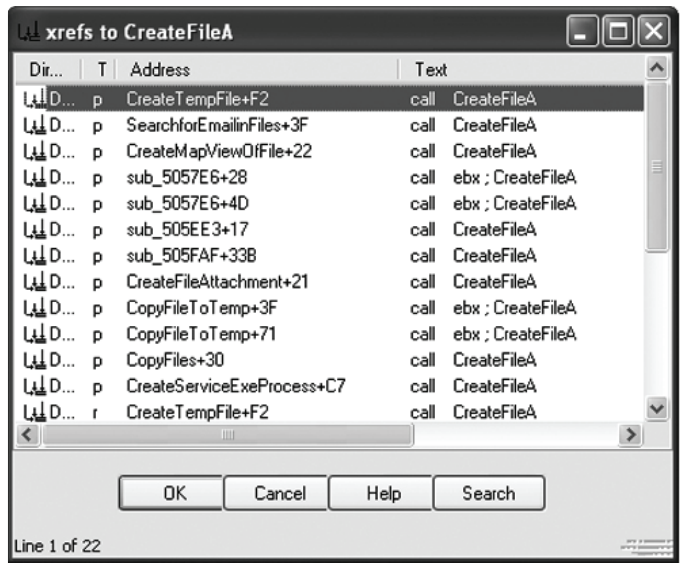


Figura 10.3 Chamadas à função CreateFileA.

Lembre-se de utilizar a funcionalidade de referências cruzadas do IDA para examinar as funções que fazem chamadas às funções importadas que podem ser usadas para criar ou modificar arquivos.

Veja no exemplo que existem muitas chamadas à *CreateFileA()*, o que pode ser uma indicação de que o programa tenta criar algum arquivo no sistema. Examinando as funções que fazem chamadas à *CreateFile()*, pode-se encontrar os arquivos que são criados.

Verifique chamadas a *RegOpenKeyExA()*, *RegSetValueExA()* e *RegQueryValueExA():*

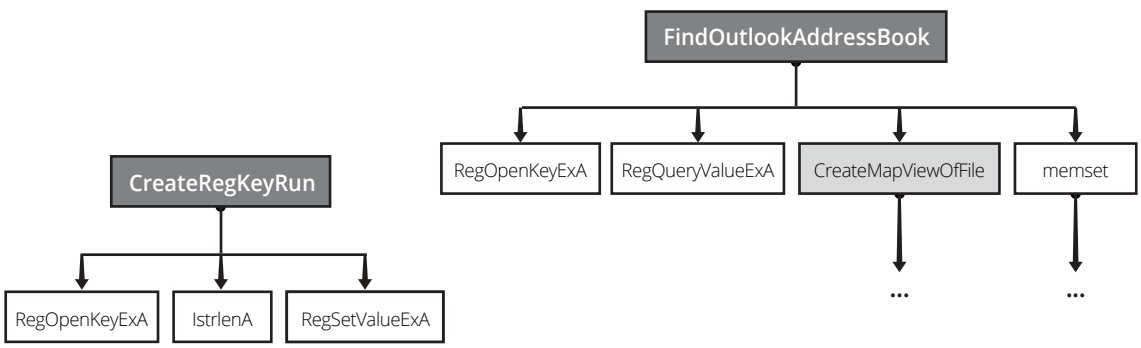


Figura 10.4 Função de pesquisa.

Aqui temos alguns exemplos de chamada a funções que acessam ou modificam dados no registro. Lembre-se de que as caixas mostradas em rosa nos gráficos de chamada são funções importadas de bibliotecas, e as em preto são funções do programa.

Podemos ver acima uma função que pesquisa o registro tentando encontrar o diretório onde o Outlook armazena a lista de contatos do usuário, e outra que cria uma chave de registro para executar automaticamente o programa malicioso no início do sistema.

> Verifique os parâmetros passados para chamadas a *FindFirstFileA()* para saber quais arquivos e diretórios são pesquisados.

A função *FindFirstFileA()* é usada quando um programa precisa listar os arquivos de um diretório. Após uma chamada a essa função, o programa geralmente realiza uma chamada em um loop à *FindNextFileA()* para encontrar os outros arquivos do diretório. Estas funções são geralmente chamadas quando o programa tenta procurar por algum arquivo específico.

No caso do nosso exemplo, o MyDoom tenta encontrar arquivos de catálogo de endereços, arquivos HTML e documentos de texto que possam conter endereços de e-mail que ele usará para enviar o binário anexado. Veja no exemplo uma parte do módulo que faz a pesquisa de catálogo de endereços e arquivos.

A seguir, precisamos descobrir como o programa analisado faz para se manter no sistema.

Como ele se mantém no sistema?

- O programa sobrevive ao boot?
- Quais processos são iniciados?
- Ele cria algum mutex*?*
- Como ele usa os recursos do sistema?

Na maioria dos casos, os programa maliciosos tentam instalar algum tipo de mecanismo para serem reiniciados caso o computador seja desligado.

Outra informação importante é descobrir se apenas o programa malicioso é executado no sistema, ou se ele de algum modo inicia outros processos também. É comum que programas maliciosos baixem e instalem outras ferramentas para comprometer ainda mais o sistema. Esta é inclusive uma fonte de renda para os autores dos programas, pois eles costumam vender essa funcionalidade para outros invasores que desejam instalar suas ferramentas no maior número possível de máquinas.

Para evitar que o mesmo programa infecte mais de uma vez a mesma máquina, alguns programas maliciosos costumam usar algum tipo de semáforo, ou mutex, para indicar que a máquina já foi comprometida. Se o mutex existir, significa que a máquina já foi comprometida e não precisa mais ser infectada.

E finalmente, veremos como identificar o tipo de impacto que o programa tem no sistema como um todo. Alguns vírus costumam sobrecarregar a máquina com um processamento muito agressivo, como ao executar um ataque de negação de serviços, ou enviar spam indiscriminadamente.

- O programa sobrevive ao boot?
  - Vimos que existe uma função que cria uma chave de registro em:
    - \Software\Microsoft\Windows\CurrentVersion\Run

- Outras formas de sobreviver ao boot seriam:
  - Instalar um serviço.
  - Salvar o executável em um diretório de inicialização.
  - Instalar uma extensão do shell do Explorer.

Vimos que o programa cria uma chave de registro para se reiniciar quando o sistema for desligado. Além desse método, um programa pode usar outras formas para se manter ativo após um reboot. Existem diversas chaves de registro que podem ser usadas para isso, conforme os exemplos seguintes:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Winlogon

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
RunOnce

HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\
RunService

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
RunService

HKCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run

HKLM\SOFTWARE\Microsoft\Active Setup\Installed Components\
```

Mais informações sobre essas chaves podem ser encontradas nos links:

- http://antivirus.about.com/od/windowsbasics/tp/autostartkeys.htm
- http://technet.microsoft.com/en-us/sysinternals/bb963902.aspx

Lembre-se de que um programa pode instalar um serviço ou usar um diretório de inicialização do Windows, como o diretório "Startup".

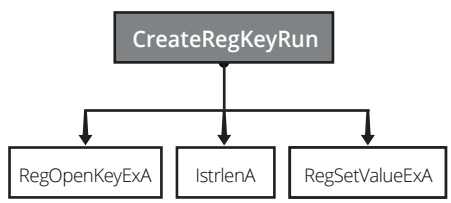


Figura 10.5
Criando uma chave de registro.

- Quais processos são iniciados?
- Examinando a função *CreateProcessA()* é possível encontrar o processo iniciado pelo programa.
- Verifique também possíveis ocorrências de *LoadLibraryA()* e *GetProcAddress()*, pois eles podem ser utilizados para iniciar processos utilizando DLLs.

Além da possibilidade de baixar outras ferramentas de ataque, alguns programas maliciosos podem também trazer embutidos outros programas ou DLLs que possam usar durante o comprometimento da máquina, como por exemplo um programa que instala um backdoor e seja independente do primeiro.

Para iniciar esses processos alternativos, o programa precisa executar uma chamada a alguma função como *CreateProcessA()* ou *LoadLibraryA(),* que permitem a execução de programas externos.

```
push    44h
lea     eax, [ebp+Dst]
pop     esi
push    esi             ; Size
push    ebx             ; Val
push    eax             ; Dst
call    memset

add     esp, 18h
lea     eax, [ebp+ProcessInformation]
mov     [ebp+Dst], esi
mov     [ebp+var_1C], 81h
push    eax             ; lpProcessInformation
lea     eax, [ebp+Dst]
push    eax             ; lpStartupInfo
push    ebx             ; lpCurrentDirectory
push    ebx             ; lpEnvironment
push    ebx             ; dwCreationFlags
push    1               ; bInheritHandles
push    ebx             ; lpThreadAttributes
push    eax, [ebp+CommandLine]
push    ebx             ; lpProcessAttributes
push    eax             ; lpCommandLine
push    ebx             ; lpApplicationName
mov     [ebp+var_18], bx
call    CreateProcessA  ;
```

Figura 10.6
Chamada à função CreateProcessA.

- Ele cria algum mutex*?*
  - Um mutex é um tipo de semáforo criado pelo programa para evitar infecções múltiplas.
  - Existem diversos tipos de *mutex:*
    - Arquivos.
    - Semáforos no kernel.
    - Chaves no registro.
- Procure por:
  - Ocorrências de *CreateMutexA()*.
  - Arquivos com tamanho nulo.
  - Chaves de registro inócuas.

Como vimos, um programa pode usar um *mutex* para evitar infectar um computador mais de uma vez. Existem basicamente três tipos de *mutex* que podem ser criados:

- **Mutex no kernel**: através de uma chamada à *CreateMutexA()* o programa cria um semáforo no sistema operacional. Este semáforo fica ativo enquanto o sistema não for reiniciado. Ao iniciar, o processo pode fazer uma tentativa de criar o *mutex* e, se o processo falhar, significa que já existe uma cópia do processo em memória.
- **Mutex por arquivo**: o programa cria um arquivo específico que indica que ele já foi executado no sistema. Se este arquivo existir, a máquina já foi comprometida. Normalmente estes arquivos não têm nenhum conteúdo e são criados em locais escondidos para evitar que sejam descobertos e apagados.
- **Mutex por chaves de registro**: além de criar um arquivo no sistema de arquivos, um processo pode criar chaves de registro para indicar sua execução prévia. Estas chaves costumam não ter nenhum efeito no funcionamento do sistema. Se o analista encontra a criação de uma chave de registro que aparentemente não tem nenhum impacto no sistema, esta pode estar sendo usada como mutex.

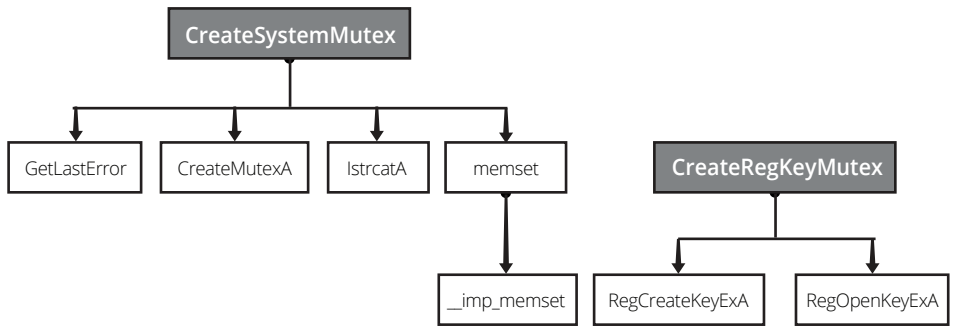


Figura 10.7
Mutex por chaves de registro.

Como ele usa os recursos do sistema?

- Alguns programas maliciosos costumam abusar dos recursos do sistema.
- Uma indicação está em chamadas a *CreateThread()*, principalmente se estiver dentro de loops.

Podemos ver no exemplo que o programa cria diversos threads no sistema dentro de um loop infinito, com um intervalo de um segundo entre cada criação. Isto pode não ser muito prejudicial ao sistema, mas se examinarmos a função chamada em cada thread, veremos que o sistema pode se tornar inutilizável dentro de pouco tempo.

A função em questão faz a procura de endereços de e-mail no sistema usando diversas técnicas, entre elas a pesquisa em arquivos de documentos. Isto pode causar uma grande sobrecarga no sistema pelas constantes leituras a disco.

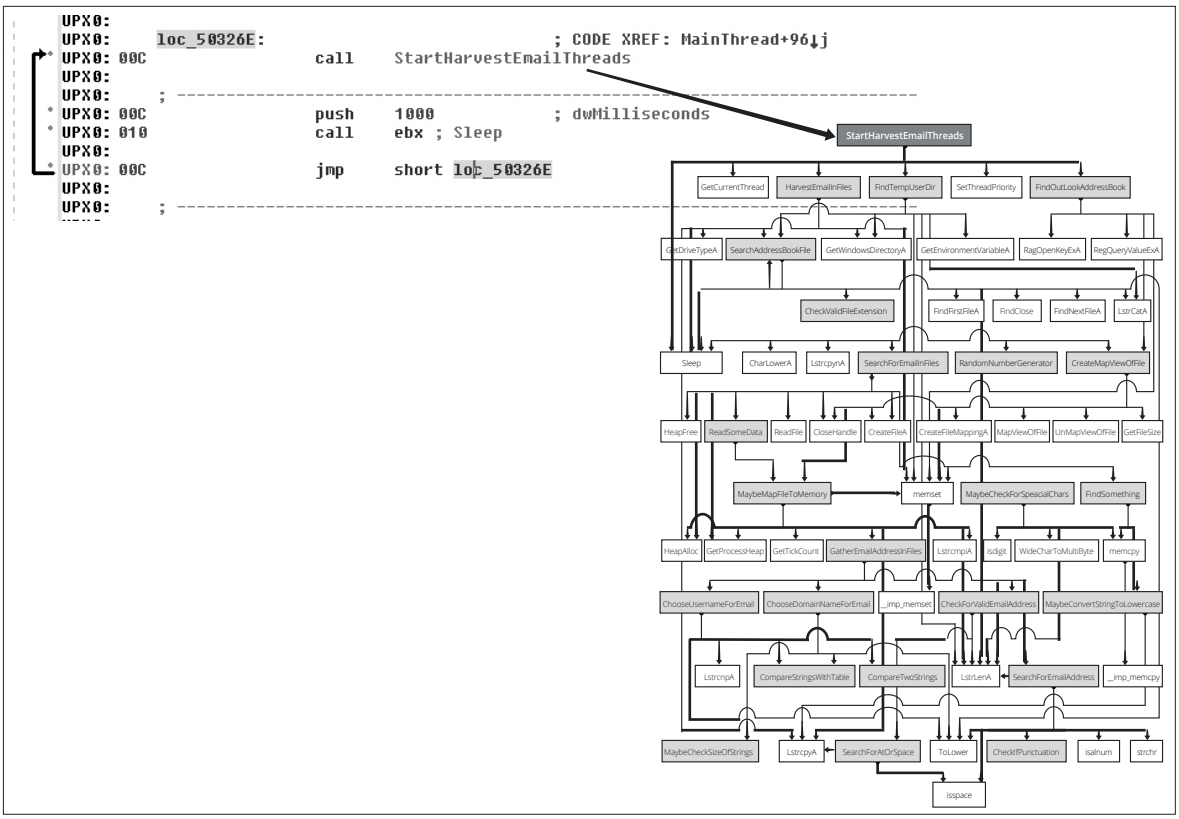


Figura 10.8
Criação de threads no sistema dentro de um loop infinito.

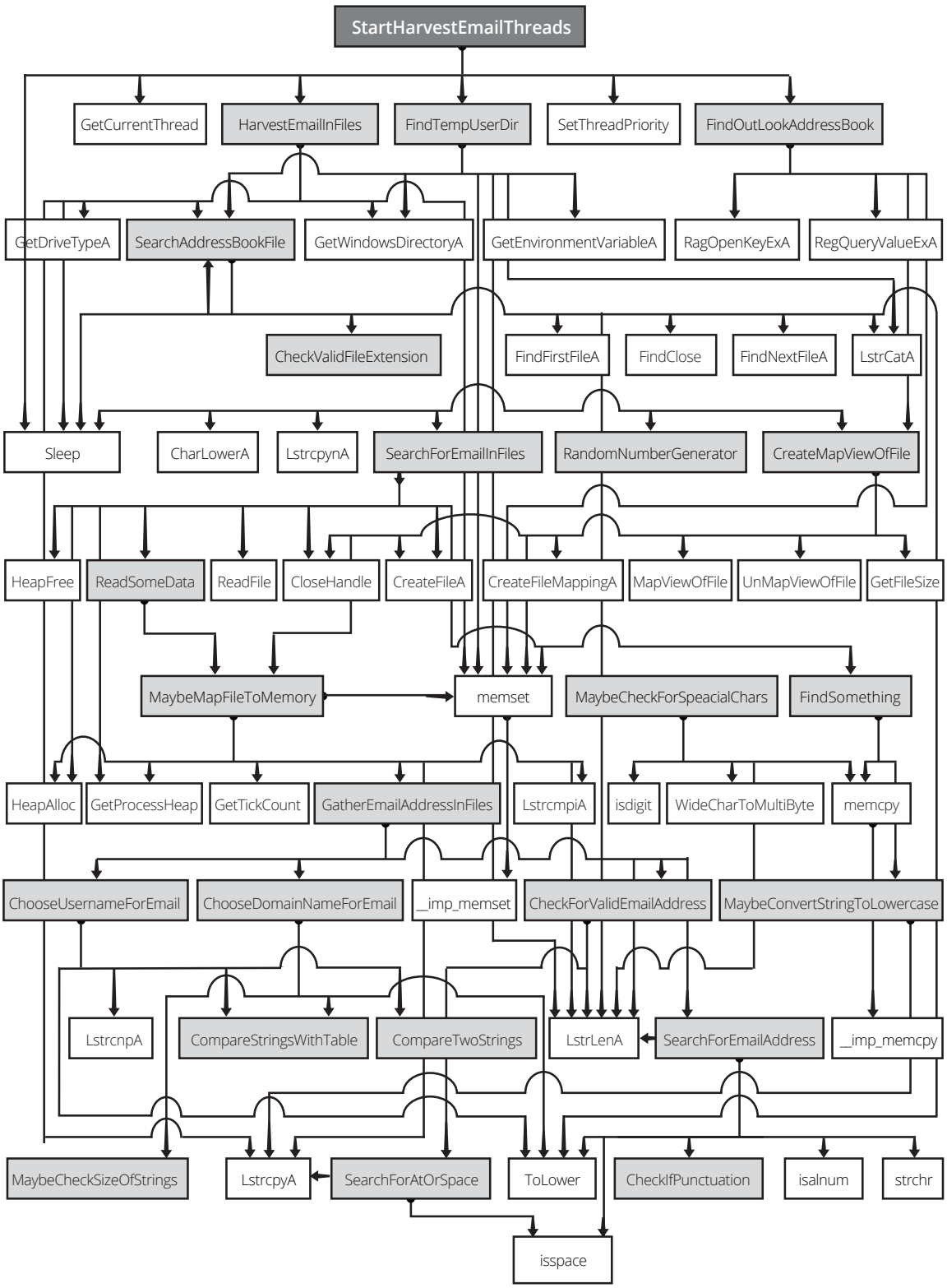


Figura 10.9
Criação de threads no sistema dentro de um loop infinito.

Exercício de fixação 2
Executável escondido

Há possibilidade de existir outro executável escondido dentro do código de um artefato malicioso? Como identificar este tipo de comportamento?

Existe algum backdoor escondido?

- O MyDoom se mantém ativo através de diversas técnicas, incluindo um backdoor.
- Procure por dados não identificados que sejam referenciados por código.
- Procure por uma função de decriptografia (XOR).
- Procure por chamadas a *CreateFile()/WriteFile()* e a *CreateProcessA().*

É muito comum que um programa malicioso traga junto consigo outros programas embutidos, que são extraídos e salvos no sistema infectado. Estes programas podem realizar funções diversas do programa original.

No caso do MyDoom, existe um programa embutido que é usado para abrir um backdoor na máquina para permitir o acesso remoto pelo autor da ferramenta após a máquina ter sido comprometida.

Para encontrar este tipo de informação, lembre-se de que um programa tem um tamanho razoável. Procure por uma grande quantidade de dados referenciados por um código que faça chamadas à *CreateFileA()* e *WriteFileA()*, usado para gravar o arquivo no disco.

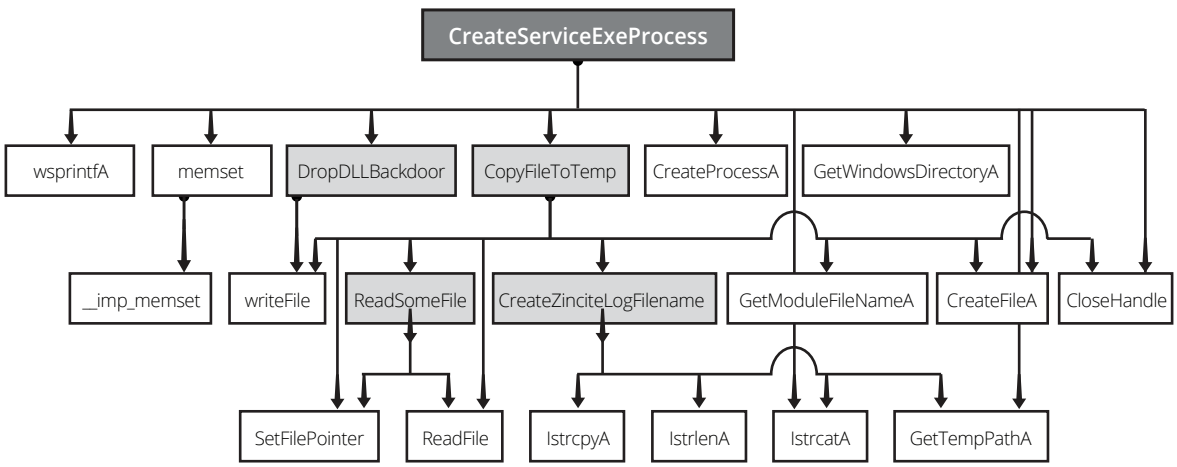


Figura 10.10
Abertura de um backdoor.

Conclusão

- Mesmo depois de tudo isso, ainda existem muitos dados não identificados no programa.
- Como última alternativa, é possível executar o programa dentro da máquina virtual e analisar o tratamento desses dados.
- Ainda existe outro binário embutido neste programa! Você consegue achá-lo?
- Estude as soluções sugeridas nas atividades e treine o máximo que puder.

O objetivo deste curso foi oferecer o conhecimento mínimo para analisar programas maliciosos. É importante lembrar sempre que somente o treino constante vai trazer a proficiência, e quanto mais variados forem os exemplos utilizados, melhor.

Mesmo no exemplo usado nos dois últimos capítulos, ainda existem muitas informações a serem identificadas, incluindo outro binário embutido.

- A proficiência em engenharia reversa só vem com a experiência.
- Treine sempre, com o maior número possível de executáveis.
- Procure nas referências e na internet por tutoriais em engenharia reversa e análises de programas maliciosos. Eles são uma boa fonte de treinamento.

Treinar os conceitos apresentados nesse curso e as soluções para as atividades sugeridas é um bom começo, mas o aluno deve procurar mais informações em outras fontes, como nos sites indicados na bibliografia.

Roteiro de Atividades 10

Atividade 10.1 – Como ele se propaga?

Após a identificação de funções do programa, usaremos essas informações para descrever as suas características mais gerais e seus métodos de ação, auxiliando na identificação da praga na rede e na criação de métodos de proteção contra novas infecções.

1. Procure por funções de acesso à rede ou de envio de e-mail;
2. Procure por funções de criação e leitura de arquivos;
3. Verifique nas strings se existe alguma dica;
4. Descubra funções para criar o spam;
5. Existe algum acesso à rede?

6. Como é a comunicação com o servidor de e-mail?

7. Tente agora identificar as funções que fazem a criação e o envio dos e-mails com o worm anexado. Tente identificar e analisar um mínimo de cinco funções e variáveis possíveis.
8. Use os comentários no código para marcar áreas importantes e identificar a lógica de criação do spam. Informações que podem ser muito úteis para o combate a essa praga em uma rede são os possíveis nomes dos arquivos anexados ou nomes de usuários para os quais o worm será enviado.

Atividade 10.2 – Que dados ele acessa ou modifica?

Analisando o código do programa:

1. Verifique os arquivos que são criados;
2. Quais chaves de registro são lidas/escritas/modificadas?
3. O programa acessa mais algum arquivo no sistema?
4. Examine as referências cruzadas às funções de acesso a arquivo, e analise as funções que as utilizam.
5. Que dados são acessados nos arquivos lidos pelo programa malicioso? Identifique as funções e analise o código de acordo com o que foi visto na sessão teórica.

Atividade 10.3 – Como ele se mantém no sistema?

Analisando o código do programa, responda:

1. O programa sobrevive ao boot?
2. Quais processos são iniciados? Tente identificar o nome dos processos.
3. Ele cria algum mutex*?* Quais semáforos são criados pelo programa malicioso?
4. Como ele usa os recursos do sistema? Identifique as funções que são chamadas dentro dos threads criados.

Atividade 10.4 – Existe algum backdoor escondido?

1. Descubra a área do programa que armazena o arquivo de backdoor.
2. Descubra e analise a função que faz a decriptografia dos dados.
3. Descubra como o processo é iniciado e onde ele é gravado.

Atividade 10.5 – Script IDC

Para essa atividade final, o aluno deve usar um programa em IDC para decriptografar a região do programa que contém o binário embutido, sem precisar executar o programa.

O IDC é uma ferramenta poderosa para auxiliar nesse tipo de situação. Como ensinar programação IDC não é o objetivo desse curso, e por si só já seria matéria para um curso completo. Um script para realizar essa conversão é fornecido no diretório de scripts IDC do IDA.

1. Utilizando o script IDC xor.idc, decodifique o binário embutido. Leia o script para entender seu funcionamento.
2. Modifique o script para salvar o código embutido em um arquivo, já decodificado.
3. Examine o script, entenda seu funcionamento e use-o para decriptografar o programa embutido.

Se você se sentir confortável, pode tentar modificar o programa para salvar o arquivo em disco após decodificá-lo. Veja os outros arquivos IDC para ter um exemplo de como salvar arquivos.

Após a decodificação, você poderá conferir a validade da operação examinando os dados decodificados para verificar se realmente trata-se de um executável.

Bibliografia

- Cartoon Hackers: http://hackerschool.org/DefconCTF/17/B300.html
- EAGLE, Chris. *The IDA Pro Book* – The Unofficial Guide to the World's Most Popular Disassembler.
- Free IDA Pro Binary Auditing Training Material for University Lectures: http://www.binary-auditing.com/
- GUILFANOV, Ilfak. *TLS callbacks*: http://www.hexblog.com/
- HINES, Eric S. *MyDoom.B Worm Analysis de Applied Watch Technologies*: http://isc.sans.org/presentations/MyDoom_B_Analysis.pdf
- Malware Analysis System: https://mwanalysis.org/ (antigo CWSandbox).
- Microsoft PE and COFF Specification: http://msdn.microsoft.com
- OpenRCE Open Reverse Code Engineering community: http://www.openrce.org/
- OS X ABI Mach-O File Format Reference. Mac Developer Library, 2009.
- PC Assembly Language Tutorial: http://www.drpaulcarter.com/cs/
- PIETREK, Matt. *A Crash Course on the Depths of Win32™ Structured Exception Handling.* Microsoft Systems Journal, 1997.
- _____. An In-Depth Look into the Win32 Portable Executable File Format, 2002.
- PORRAS et al. *An Analysis of Conficker's Logic and Rendezvous Points*, 2009.
- ROBERTS, Wesley. *New Conficker B++ Worm Variant on the Prowl*, 2009.
- Securith-forum.net: http://forums.security-forum.net/
- Threat Analyses de Secure Works: http://www.secureworks.com/cyber-threat-intelligence/threats/
- Vulnerabilities on Sophos: http://www.sophos.com/

esr.rnp.br

LIVRO DE APOIO AO CURSO

O curso apresenta técnicas de análise de malware para apoiar a investigação forense digital e a resposta a incidentes envolvendo programas mal-intencionados. O objetivo é fornecer aos administradores de TI habilidades práticas para a análise destes programas. São abordados os conceitos, procedimentos e ferramentas para a análise de um código malicioso, com o uso de uma ferramenta para a realização das atividades práticas, que consolidam o conhecimento teórico. São apresentados os comandos básicos de Assembly para que o aluno execute a engenharia reversa de worms que afetaram milhares de computadores. O aluno aprenderá as melhores práticas antiengenharia reversa, desenvolvendo competências para a criação de defesas mais eficazes contra códigos maliciosos.

Este livro inclui os roteiros das atividades práticas e o conteúdo dos slides apresentados em sala de aula, apoiando profissionais na disseminação deste conhecimento em suas organizações ou localidades de origem.

ISBN 978-85-63630-26-1