Capítulo

4

Introdução à Engenharia Reversa de Aplicações Maliciosas em Ambientes Linux

Marcus Botacin[1], Lucas Galante[2], Otávio Silva[2], and Paulo Lício de Geus[2]

[1]Universidade Federal do Paraná (UFPR)
[2]Universidade Estadual de Campinas (UNICAMP)

Abstract

Malicious software have been evolving to look even more indubious to the targeted system. Moreover, they have been relying on obfuscation and anti-analysis techniques to avoid their behavior being discovered during their execution, thus making forensic procedures harder to be conducted on a proper manner. Reverse engineering is an useful procedure to handle such type of threat, pinpointing the best inspection paths to be followed by analysts, such as how to unpack a given sample or which protection techniques are implemented by a sample. In this course, we present reverse engineering techniques to analyze malicious software samples, thus introducing the reader to general malware handling procedures. The course introduces techniques both in userland as well as in the OS kernel, including dynamic application tracing, debugging and rootkit techniques.

Resumo

Aplicações maliciosas têm evoluído de forma a cada vez mais permanecerem ocultas no sistema alvo. Além disso, essas aplicações utilizam de técnicas de ofuscação e/ou anti-análise a fim de evitar a descoberta de seu comportamento durante a execução, impedindo que atividades de perícia forense computacional sejam realizadas adequadamente. A engenharia reversa é um processo útil na análise desse tipo de aplicação, uma vez que pode-se descobrir quais caminhos seguir em uma perícia, por exemplo, como desempacotar a aplicação ou que tipo de truques ela usa para evitar ser destrinchada. Neste minicurso, apresentamos técnicas de engenharia reversa para análise de aplicações maliciosas tomando como base o ambiente Linux, visando introduzir os participantes ao tema e capacitá-los no tratamento de aplicações maliciosas em geral. O curso aborda técnicas tanto em modo usuário quanto em modo kernel do sistema operacional, incluindo o traço dinâmico de aplicações, técnicas de depuração e de rootkits.

4.1. Introdução

A plataforma `Linux` tem se consolidado como uma das principais plataformas para o desenvolvimento de novas tecnologias no mundo contemporâneo. Atualmente, ela suporta desde ambientes de computação em nuvem (*cloud*) [Venezla 2012] até o sistema operacional para *smartphones* `Android` [Android 2017]. Contudo, ao mesmo tempo em que proporciona o desenvolvimento de novas tecnologias, a plataforma `Linux` torna-se alvo de ataques, passando a requerer o desenvolvimento de ferramentas de análise de binários [Afonso et al. 2015] e para procedimentos forenses [Vecchia and Coral 2014].

Um dos principais vetores de ataque contra a plataforma `Linux` são códigos maliciosos (*malicious software* ou *malware*), que são arquivos executáveis que apresentam deliberadamente intenções prejudiciais à sistemas computacionais [Skoudis and Zeltser 2003], o que pode incluir do vazamento de informações sensíveis do usuário ao controle total do sistema infectado. Na prática, ataques por *malware* contra a plataforma `Linux` são frequentes e causam prejuízos significativos. Em 2016, a *botnet* `Mirai` [Fruhlinger 2018] causou interrupções na conexão à Internet da costa Atlântica dos Estados Unidos. Em 2017, o *ransomware* `Erebus` [Z. Chang 2017] sequestrou os servidores de um provedor de conteúdo sul-coreano em troca de resgate.

Para se prevenir e combater ataques por *malware*, diversas técnicas de análise são empregadas, dentre as quais se destaca a engenharia reversa dos códigos maliciosos. A engenharia reversa consiste em inspecionar o binário malicioso de modo a compreender seu funcionamento, identificar seu alvo no sistema infectado e potencialmente criar mecanismos para a defesa contra este tipo de ameaça. As técnicas de engenharia reversa apresentam muitas estratégias comuns a diferentes plataformas, como a execução do binário desconhecido em um ambiente controlado (*sandbox*). Entretanto, diferentes plataformas apresentam especificidades quanto ao alvo da execução em *sandbox*. Como exemplo, enquanto na plataforma `Windows` exemplares de *malware* afetam o subsistema de chaves de registro [Botacin et al. 2018], na plataforma `Linux`, exemplares de *malware* afetam o subsistema de arquivo `/proc` [Galante et al. 2018], de modo que entender as particularidades das técnicas de engenharia reversa em cada plataforma é essencial para prover respostas adequadas aos ataques que estas sofrem.

Contudo, apesar dos impactos da infecção por *malware* na plataforma `Linux` serem reais e das técnicas de engenharia reversa poderem auxiliar no combate à este ataques, a literatura acadêmica tem dado pouca atenção a plataforma `Linux` e focado na plataforma `Windows`, dada sua concentração de mercado historicamente grande, até a ascensão recente da plataforma `Linux`. Frente a este cenário de uso crescente e técnicas de análise pouco difundidas em comparação a plataforma anteriormente dominante, vislumbramos a necessidade de se apresentar os conceitos de engenharia reversa com foco específico nesta plataforma, fato que motiva o desenvolvimento deste curso.

Objetivos. Este minicurso busca ser uma introdução à engenharia reversa de aplicações em ambientes `Linux`, com foco na análise de aplicações maliciosas, em especial as que implementam técnicas de evasão de procedimentos de análise. Durante o minicurso, os participantes desenvolverão habilidades básicas de engenharia reversa, como a caracterização de binários, a identificação das proteções usadas e alguns contornos (*bypass*) para tais proteções.

Sobre o curso. Este curso é proposto na modalidade majoritariamente prática, através de exercícios de análise de binário reais, mas contando com o devido suporte teórico para embasar as decisões tomadas pelos participantes. O curso ocorrerá no formato de uma competição `capture-the-flag`, na qual os participantes serão desafiados a resolver alguns desafios que levem à engenharia reversa do binário dado. Desta forma, este capítulo corresponde a parte teórica do curso, fornecendo subsídios para que os alunos realizem os desafios.

Audiência. Este curso se destina ao público que pretende iniciar seus estudos na área de análise de binários. Não são exigidos conhecimentos prévios na área de segurança, apenas familiaridade com o uso de sistemas `GNU/Linux` e seu terminal (`bash`), e conhecimentos básicos em alguma linguagem de programação. Acreditamos que o curso possa ser de interesse também para quem já possua familiaridade com as soluções de análise da plataforma `Linux`, como *debuggers*, dado que as técnicas de engenharia reversa são distintas das empregadas comumente na depuração de aplicações convencionais.

Materiais. Todo os binários a serem analisados durante o curso podem ser obtidos através do repositório: `https://github.com/marcusbotacin/Malware.Reverse.Intro`, de modo que os leitores possam acompanhar este capítulo mesmo após a realização do curso presencial. Todos os códigos são disponibilizados juntamente com arquivos de compilação (`Makefiles`), não requerendo que os participantes tenham conhecimentos avançados, como a compilação de *drivers* de *kernel* para executar as tarefas propostas.

Estrutura. Este capítulo é dividido da seguinte forma: Na Seção 4.2, apresentamos a estrutura básica dos binários ELF utilizados nas plataformas `Linux` e ferramentas básicas para a manipulação destes; Na Seção 4.3, introduzimos os primeiros conceitos de engenharia reversa e apresentamos as verificações iniciais a serem realizadas sobre arquivos binários sem a execução do mesmo (análises estáticas), como a identificação do tipo de arquivo inspecionado, a presença de bibliotecas, e sinais de ofuscação de código, de modo a se obter uma compreensão inicial dos possíveis comportamentos exibidos pelo binário; Na Seção 4.4, estendemos as análise de observações passivas para modificações ativas no binário (*binary patching*), de modo a contornar verificações implementadas pelas rotinas de anti-análise; Na Seção 4.5, estendemos as análises anteriormente descritas através da execução dos binários desconhecidos em um ambiente controlado (*sandbox*), permitindo, assim, a análise de exemplares ofuscados; Na Seção 4.6, tal qual realizado para as técnicas de análise estática, estendemos as técnicas dinâmicas de observações passivas para interferências ativas na execução dinâmica de código, permitindo, assim, contornar técnicas de anti-análise implementadas pelos binários; Apresentamos, também, como os atacantes se utilizam destas mesmas técnicas para implementar seus comportamentos evasivos, tais como *rootkits*; Na Seção 4.7, apresentamos abordagens complementares de análise, com focos em subsistemas específicos, como a monitoração do subsistema de arquivos para recuperação de arquivos deletados pelos exemplares maliciosos; Na Seção 4.8, estendemos os conceitos e soluções anteriormente apresentadas em modo usuário (*userland*) para o modo mais privilegiado (*kernel*), visando contornar mecanismos de anti-análise mais sofisticados; Na Seção 4.9, apresentamos conceitos básicos de monitoração de tráfego de rede; Na Seção 4.10, apresentamos nossas considerações finais.

4.2. Binários ELF: Definição e Manipulação

Diferentes plataformas representam seus aplicativos de diferentes maneiras, de modo que compreender a representação de um arquivo executável é essencial para a execução de procedimentos de investigação de binários desconhecidos e suspeitos. Nesta seção, introduzimos o tipo binário utilizado pelas plataformas `Linux` e ferramentas básicas para a manipulação destes.

4.2.1. A Estrutura dos Binários ELF

O *Executable and Linkable Format* (ELF) é o formato padrão de arquivos executáveis na plataforma `Linux` [O'Neill 2016] e estabelece uma estrutura a ser seguida por todos as aplicações da plataforma de modo que o sistema tome conhecimento, através dos campos padronizados, de como carregar o conteúdo do arquivo binário em memória e inicializar um processo. Um arquivo ELF, como definido em `elf.h`, possui a estrutura básica apresentada na Figura 4.1.

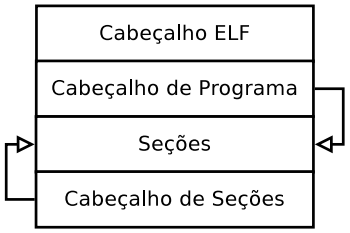

Figura 4.1. Arquivo ELF. Estrutura Básica.

O cabeçalho ELF, definido como mostrado no Código 4.1, é uma estrutura obrigatória e está sempre presente no deslocamento (*offset*) "0" de todo arquivo ELF. Ele apresenta as informações básicas do binário, como seu tipo (`e_type`)–podendo ser binários autocontidos ou bibliotecas—, a arquitetura do sistema alvo (`e_machine`), entre outras informações essenciais a execução.

```
1  #define EI_NIDENT 16
2  typedef struct {
3         unsigned char e_ident[EI_NIDENT];
4         uint16_t      e_type;
5         uint16_t      e_machine;
6         uint32_t      e_version;
7         ElfN_Addr     e_entry;
8         ElfN_Off      e_phoff;
9         ElfN_Off      e_shoff;
10        uint32_t      e_flags;
11        uint16_t      e_ehsize;
12        uint16_t      e_phentsize;
13        uint16_t      e_phnum;
14        uint16_t      e_shentsize;
15        uint16_t      e_shnum;
16        uint16_t      e_shstrndx;
17    } ElfN_Ehdr;
```

Código 4.1. Definição do cabeçalho de um arquivo ELF.

O cabeçalho de programa, tal qual definido no Código 4.2, armazena informações básicas de como as informações contidas no arquivo devem ser interpretadas (`p_type`), em qual posição as informações se localizam (`*_addr`) e qual o tamanho dos blocos contendo estas informações (`*sz`).

```
typedef struct {
       uint32_t   p_type;
       uint32_t   p_flags;
       Elf64_Off  p_offset;
       Elf64_Addr p_vaddr;
       Elf64_Addr p_paddr;
       uint64_t   p_filesz;
       uint64_t   p_memsz;
       uint64_t   p_align;
   } Elf64_Phdr;
```

Código 4.2. Definição do cabeçalho de programa de arquivo ELF.

Dentre os tipos que um arquivo ELF pode assumir, tal qual apontado pelo cabeçalho de programa, estão binários autocontidos e arquivos objetos, utilizado para a implementação de códigos em bibliotecas. A ligação de bibliotecas à binários pode ser realizada de duas maneiras [Simmonds 2015]: (i) estaticamente, em tempo de compilação (`gcc -static <arquivo>`); ou (ii) dinamicamente, em tempo de execução. Cada uma destas formas de ligação apresenta vantagens e desvantagens: Por um lado, ligações estáticas, por embutirem no binário todo o código necessário para a execução da aplicação, permitem que o binário seja executado em diversos sistemas sem a necessidade de se instalar qualquer dependência. Por outro lado, estas aumentam significativamente o tamanho final do binário gerado em comparação com binários ligados à bibliotecas dinâmicas.

O cabeçalho de seção de um arquivo ELF, como mostrado no Código 4.3, estabelece como o conteúdo da aplicação está disposto dentro do binário. Cada tipo de dado (instruções de máquina, variáveis estáticas, entre outros) é disposta em uma seção diferente, cada uma recebendo um nome (`sh_name`) e atributos (e.g., `sh_flags`) próprios.

```
typedef struct {
       uint32_t   sh_name;
       uint32_t   sh_type;
       uint64_t   sh_flags;
       Elf64_Addr sh_addr;
       Elf64_Off  sh_offset;
       uint64_t   sh_size;
       uint32_t   sh_link;
       uint32_t   sh_info;
       uint64_t   sh_addralign;
       uint64_t   sh_entsize;
   } Elf64_Shdr;
```

Código 4.3. Definição do cabeçalho de seção de um arquivo ELF.

O Código 4.4 exibe as seções de um um binário ELF típico: Instruções de máquina são armazenadas na seção `.text`; Variáveis inicializadas com e sem permissão de escrita são armazenados, respectivamente, nas seções `.data` e `.rodata`; Variáveis e outros dados dinâmicos são armazenados nas seções `*dyn`; Bibliotecas dinâmicas e outros apontadores carregados em tempo de execução são armazenados na região correspondente as seções do tipo `*symbols*`.

```
[Nr] Nome
[ 1] .interp            [ 2] .note.ABI-tag
[ 3] .note.gnu.build-i  [ 4] .gnu.hash
[ 5] .dynsym            [ 6] .dynstr
[ 7] .gnu.version       [ 8] .gnu.version_r
[ 9] .rela.dyn          [10] .rela.plt
[11] .init              [12] .plt
[13] .plt.got           [14] .text
[15] .fini              [16] .rodata
[17] .eh_frame_hdr      [18] .eh_frame
[19] .init_array        [20] .fini_array
[21] .jcr               [22] .dynamic
[23] .got               [24] .got.plt
[25] .data              [26] .bss
[27] .comment           [28] .shstrtab
[29] .symtab            [30] .strtab
```

Código 4.4. Exibição das seções de um arquivo ELF.

4.2.2. Manipulação Básica de Binários ELF

Uma vez que se compreenda a estrutura básica dos binários ELF, pode-se proceder a inspeção das seções e cabeçalhos dos binários de modo a identificar construções suspeitas. A plataforma `Linux` possui diversas soluções para a manipulação de binários ELF, incluindo a manipulação direta através dos cabeçalhos disponíveis para a linguagem C. Contudo, nesta seção, para fins didáticos, exemplificaremos a manipulação de binários ELF através da solução `pyelftools` [eliben 2017].

O Código 4.5 ilustra como listar o nome das seções de um binário ELF via `pyelftools`. Em análise de *malware*, este tipo de verificação é essencial para se identificar nomes de funções suspeitas, como os nomes deixados pela solução de empacotamento e ofuscação `UPX` (detalhado na seção 4.3). `Pyelftools` também permite ao analista verificar o tamanho das seções, cabeçalhos e valores dos apontadores. Este tipo de informação é útil pois muitos binários maliciosos acabam sendo malformados devido as rotinas de ofuscação e apresentam valores nulos nestes campos, o que pode servir como um indicador de infecção ou suspeição.

```
def process_file(filename):
    with open(filename, 'rb') as f:
        elffile = ELFFile(f)
        for section in elffile.iter_sections():
            print(section.name)
```

Código 4.5. Enumeração das seções de um arquivo ELF através da solução pyelftools.

4.3. Análise Estática

No contexto de engenharia reversa de aplicações maliciosas, referimos a análise estática como o conjunto de métodos utilizados para obter informações sobre os arquivos binários suspeitos sem a efetiva execução destes [Sikorski and Honig 2012]. No contexto de análise forense, este tipo de análise é usualmente a etapa inicial de um conjunto de procedimentos complexos realizados com o objetivo de se obter informações preliminares sobre o artefato em análise.

Através dos procedimentos de análise estática, um analista pode identificar o tipo do arquivo em análise e se bibliotecas de código são ligadas à este, permitindo a classificação do artefato. Um analista pode também inspecionar se o arquivo apresenta *strings* que possam prover informações forenses. Finalmente, um analista pode obter o código *assembly* para identificar, preliminarmente, possíveis comportamentos maliciosos ou técnicas de evasão. Apresentamos, a seguir, soluções e procedimentos para realizar estes passos de análise.

4.3.1. O utilitário e comando `file`

O comando *file* é frequentemente utilizado para o reconhecimento do tipo de dado contido em um arquivo. O Código 4.6 exemplifica a saída do comando para diferentes arquivos de entrada: um código na linguagem C, um diretório do sistema, e um arquivo `pdf`. É importante notar que a identificação do formato de arquivo pelo comando `file` é dado por uma base de dados própria [GNU 2018], sendo independente da extensão utilizada para nomear o arquivo.

```
1 >$ file hello.c
2 hello.c: ASCII text
3 >$ file Documents/
4 Documents/: directory
5 >$ file wticg.pdf
6 wticg.pdf: PDF document, version 1.4
```

Código 4.6. Comando file em diferentes arquivos.

O comando `file` é de especial utilidade para procedimentos de análise quando aplicados a arquivos ELF, como ilustrado pelo Código 4.7. Através deste, pode-se identificar, por exemplo: (i) que o arquivo é um binário executável (`ELF`) para arquiteturas de 64 *bits* (`x86-64`), como mostrado na linha 2; e que (ii) o arquivo se liga a bibliotecas de forma dinâmica, como mostrado na linha 3. A linha 4 exibe o *hash* (identificador único) do arquivo.

```
1 >$ file dynamic-malicious.bin
2 dynamic-malicious.bin: ELF 64-bit LSB executable, x86-64, version 1 (
    SYSV),
3 dynamically linked, interpreter /lib64/l, for GNU/Linux 2.6.32,
4 BuildID[sha1]=5e42df0d86c9df096eefb5fd99a703c479957fc8, not stripped
```

Código 4.7. Comando file em um arquivo ELF com ligação dinâmica

Identificar o tipo de ligação de um binário desconhecido é essencial para aumentar o entendimento sobre o funcionamento deste, uma vez que o tipo de ligação também influencia na realização dos procedimentos de análise. Um efeito colateral da ligação estática é que símbolos anteriormente exportados para auxiliar na resolução dos endereços das funções contidas nas bibliotecas agora são omitidos por não serem mais necessários, o que reduz a quantidade de informação obtida através dos procedimentos de análise estática. Este fato é frequentemente explorado por atacantes de forma a dificultar a engenharia reversa de suas aplicações maliciosas. Por exemplo, caso o mesmo arquivo ELF anteriormente apresentado fosse ligado estaticamente a alguma biblioteca, a saída do comando *file* seria como mostrado no Código 4.8, no qual a linha 3 exibe o tipo de ligação do arquivo e nenhuma informação sobre bibliotecas ou interpretadores é provida. Note ainda que, embora tendo sido compilado a partir do mesmo arquivo fonte, o *hash* do binário compilado (linha 4) difere do binário anterior. A recompilação de arquivos é uma estratégia frequentemente utilizada por atacantes para gerar `variantes` de arquivos maliciosos quando o binário original passa a ser detectado por soluções Antivírus (AVs).

```
1 >$ file static-malicious.bin
2 static-malicious.bin: ELF 64-bit LSB executable, x86-64, version 1 (
     GNU/Linux),
3 statically linked, for GNU/Linux 2.6.32,
4 BuildID[sha1]=a20285090944c95cc21aac376a87144b37657496, not stripped
```

Código 4.8. Comando file em arquivo ELF compilado com ligação estática

4.3.2. O utilitário e comando `LDD`

Quando da identificação de um binário utilizando-se ligação dinâmica, o analista deve proceder a identificação das bibliotecas ligadas de modo a melhor compreender o funcionamento do binário e, eventualmente, prever seu comportamento através dos tipos de bibliotecas ligadas (e.g., bibliotecas de rede sugerem que o *malware* possa ser um *downloader* [Grégio et al. 2015] ou que exfiltre informações sensíveis). Para tanto, pode-se utilizar o comando *List Dynamic Depedencies* (*ldd*), como exemplificado pelo Código 4.9.

```
1 >$ ldd dynamic-malicious.bin
2   linux-vdso.so.1 =>  (0x00007ffe987e6000)
3   libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f77a58b3000)
4   /lib64/ld-linux-x86-64.so.2 (0x00007f77a5c7d000)
```

Código 4.9. Comando ldd em arquivo ELF com ligação dinâmica

Caso o arquivo analisado tenha sido ligado estaticamente, por exemplo, por rotinas de ofuscação, o comando *ldd* informará ao analista não ser capaz de identificar nenhuma biblioteca, como exemplificado pelo Código 4.10.

```
1 >$ ldd static-malicious.bin
2   not a dynamic executable
```

Código 4.10. Comando ldd em arquivo ELF com ligação estática

4.3.3. O utilitário e comando `objdump`

Quando um binário apresenta ligação dinâmica, mais do que se identificar a quais bibliotecas este se liga, pode-se identificar quais funções presentes nestas bibliotecas são referenciadas pelo binário. Através da opção `-T`, o comando `objdump` permite analisar a tabela de símbolos dinâmicos dos binários fornecidos, incluindo os símbolos usados pelo ligador dinâmico, tal qual exemplificado pelo Código 4.11.

```
1  >$ objdump -T VirusShare_4e0ee9f1571107a015e63925626b562d
2
3  VirusShare_4e0ee9f1571107a015e63925626b562d:     file format elf32-
       i386
4
5  DYNAMIC SYMBOL TABLE:
6  080484d8      DF *UND*  0000003b  GLIBC_2.0   inet_addr
7  080484f8      DF *UND*  000000d8  GLIBC_2.0   __libc_start_main
8  08048508      DF *UND*  00000039  GLIBC_2.0   printf
9  08048518      DF *UND*  00000046  GLIBC_2.0   memcpy
10 08048528      DF *UND*  000001c8  GLIBC_2.0   gethostbyname
11 08048538      DF *UND*  000000e2  GLIBC_2.0   exit
12 08048548      DF *UND*  0000003e  GLIBC_2.0   atoi
13 08048558      DF *UND*  00000039  GLIBC_2.0   send
14 08048568      DF *UND*  0000000e  GLIBC_2.0   htons
15 08048578      DF *UND*  00000054  GLIBC_2.0   memset
16 08048588      DF *UND*  00000039  GLIBC_2.0   connect
17 08048598      DF *UND*  00000039  GLIBC_2.0   recv
18 080485a8      DF *UND*  00000034  GLIBC_2.0   sprintf
19 080485b8      DF *UND*  00000039  GLIBC_2.0   socket
```

Código 4.11. Saída do comando objdump exibindo a tabela de símbolos de um exemplar de malware indicando o uso de recursos de rede

A identificação das funções referenciadas pelo binário possibilita que o analista tenha uma primeira ideia dos possíveis comportamentos que o binário possa apresentar. No exemplo acima, identifica-se que este pode fazer significante uso de recursos de rede, dado as referências as funções `connect`, `send`, `recv` e `socket`, utilizadas para implementar diferentes tipos de comunicação entre diferentes máquinas (*hosts*) e/ou à Internet [Yocom et al. 2004].

Cientes da possibilidade de se identificar o comportamento do binário pelos símbolos das funções referenciadas, atacantes empregam diversas técnicas para confundir o analista, como: (i) referenciar muitas funções que não são executadas de fato, apenas para dificultar procedimentos de análise; (ii) ofuscar o código para que símbolos de bibliotecas sejam resolvidos apenas em tempo de execução; ou (iii) compilar bibliotecas e funções estaticamente, escondendo seus símbolos, como exemplificado pelo Código 4.12, já que a tabela de símbolos não precisa ser populada por bibliotecas ligadas estaticamente.

```
1  >$ objdump -T static-malicious.bin
2  static-malicious.bin:     file format elf64-x86-64
3  objdump: static-malicious.bin: not a dynamic object
4  DYNAMIC SYMBOL TABLE:
5  no symbols
```

Código 4.12. Saída do comando objdump indicando a ausência da tabela de símbolos para um binário ELF ligado estaticamente

Quando símbolos não estão presentes, pode-se tentar identificar construções maliciosas olhando diretamente para as instruções presentes no binário. Através da opção `-d`, o comando `objdump` permite realizar o *disassembly* da seção de instrução do binário, tal qual exemplificado pelo Código 4.13, no qual pode-se observar uma construção típica de binários que fazem uso de rede: (i) a tentativa de obtenção de um nome de rede (`gethostbyname`); (ii) a verificação do valor de retorno (`cmpl`); resultando no termino da execução (`exit`) em caso de erro (`perror`); ou (iv) na criação de um `socket` caso esta tenha sido bem sucedida.

```
1  >$ objdump -d VirusShare_4e0ee9f1571107a015e63925626b562d
2
3  0804876f <main>:
4  8048ced:       e8 36 f8 ff ff          call   8048528 <
       gethostbyname@plt>
5  8048cf2:       83 c4 10                add    $0x10,%esp
6  8048cf5:       89 85 e4 75 ff ff       mov    %eax,-0x8a1c(%ebp)
7  8048cfb:       83 bd e4 75 ff ff 00    cmpl   $0x0,-0x8a1c(%ebp)
8  8048d02:       75 1a                   jne    8048d1e <main+0x5af>
9  8048d04:       83 ec 0c                sub    $0xc,%esp
10 8048d0c:       e8 a7 f7 ff ff          call   80484b8 <perror@plt>
11 8048d11:       83 c4 10                add    $0x10,%esp
12 8048d14:       83 ec 0c                sub    $0xc,%esp
13 8048d19:       e8 1a f8 ff ff          call   8048538 <exit@plt>
14 8048d1e:       83 ec 04                sub    $0x4,%esp
15 8048d27:       e8 8c f8 ff ff          call   80485b8 <socket@plt>
```

Código 4.13. Saída do comando objdump ilustrando o dissassembly de um exemplar malicioso que faz uso de recursos de rede

Novamente, atacantes estão cientes da possibilidade de se identificar construções maliciosas através do *disassembly* das instruções do binário e empregam técnicas de anti-análise para dificultar a tarefa do analista. Entre as técnicas, destacam-se: (i) ofuscação do binário, fazendo com que as instruções maliciosas não sejam diretamente visíveis ao *disassembler*; (ii) técnicas de anti-*disassembly* [Branco et al. 2012], fazendo com o que o *disassembler* produza resultados incorretos; e (iii) a inserção de código morto (*dead-code*), código *assembly* sem qualquer utilidade real apenas para dificultar a tarefa de análise. Para superar estas limitações, o analista deve proceder para a análise dinâmica, a fim de observar o código efetivamente executado pelo binário.

4.3.4. O utilitário e comando `strings`

Além de tabelas de símbolos, a estrutura dos binários ELF carrega outras informações que podem ajudar a inferir o comportamento dos binários desconhecidos, como *strings* estáticas utilizadas pelo programador. *Strings* são alocadas na seção de dados estáticos do binário, cujos *bytes* podem ser interpretados como caracteres `ASCII`, como exemplificado na Figura 4.2.

As mesmas informações sobre *strings* contidas no exemplar podem ser obtidas através do comando *strings*, exibido no Código 4.14, que apresenta uma exibição mais facilitada para a interpretação por seres humanos. Pode-se notar a presença de diversas *strings* representando endereços IP, o que sugere a possibilidade deste exemplar fazer significante número de comunicações via rede. Em particular, o uso de formatadores de

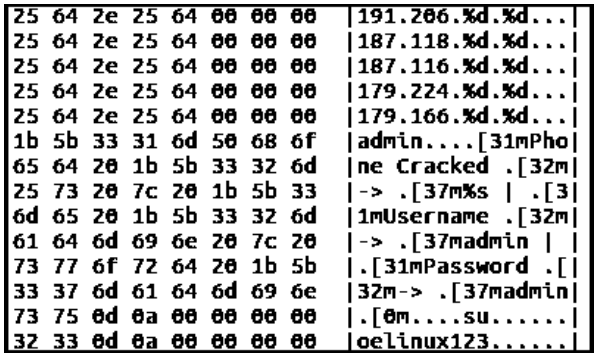

Figura 4.2. Comando hexdump -C em exemplar malicioso contendo *strings* suspeitas.

strings do tipo inteiro (`%d`), sugere que este exemplar pode ser um *scanner* [Galante et al. 2018, Cozzi et al. 2018] de rede, realizando tentativas contra todos os *hosts* possíveis dentro da sub-rede especificada. Tal hipótese é corroborada pelas demais *strings*, que ilustram diversas senhas que podem ser utilizadas em ataques de força bruta [Wang et al. 2018].

```
>$ strings
   a9a780c66ec18861e4881430202f62d6ceaba3187626d48ab241522f86a5c254
189.96.%d.%d            189.99.%d.%d
39.34.%d.%d             59.103.%d.%d
191.12.%d.%d            186.117.%d.%d
179.170.%d.%d           191.206.%d.%d
187.118.%d.%d           179.224.%d.%d
admin                   [31mPhone Cracked
[31mUsername            [31mPassword
[36mDevice Cracked      GETLOCALIP
My IP: %s               BOTKILL
Killing Bots            NETIS ON | OFF
[TELNET] SCANNER ON:%s  ICMP
HTTP                    STOP
ENDTHEBOTLOL            8.8.8.8
/proc/net/route         /etc/resolv.conf
```

Código 4.14. Saída do comando strings em exemplar malicioso indicando diversas strings suspeitas

Como nos casos anteriores, atacantes previnem-se de exibir suas *strings* através de: (i) geração de *strings* em tempo de execução; e (ii) ofuscação das *strings* estáticas, como exemplificado pelo Código 4.15.

```
>$ strings upx-file.bin
$Info: This file is packed with the UPX executable packer http://
   upx.sf.net $
dl%)
UPPH
q}Nsf
x`TJq&H
UPX!
```

Código 4.15. Saída do comando strings em arquivo empacotado pela solução UPX

4.3.5. O utilitário e comando `UPX`

Uma das formas mais comuns de se ofuscar códigos maliciosos é através do uso de empacotadores, capazes de embutir o arquivo original em sua estrutura e extraí-los em tempo de execução e dentre os quais a solução *Ultimate Packer for eXecutables* (*UPX*) [M.F.X.J. Oberhumer 2018] é a mais popular. O Código 4.16 ilustra o empacotamento de um arquivo com ligação estática via *UPX*.

```
>$ upx original.bin -o upx-file.bin
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep
    30th 2013

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     610309 ->    270256   44.28%  linux/elf386   1-upx

Packed 1 file.
```

Código 4.16. Empacotamento de um binário com ligação estática pela solução UPX

O binário empacotado não somente ofusca o conteúdo do binário malicioso original, mas também pode confundir soluções antivírus (AVs), já que estas frequentemente comparam o *hash* do arquivos suspeitos contra bases de *hashes* de binários conhecidos [Koret and Bachaalany 2015] e a compressão por `UPX` resulta em um binário com *hash* distinto do original, como exemplificado pelo Código 4.17. Este tipo de estratégia é deliberadamente utilizada por atacantes não apenas utilizando-se de compressores como `UPX`, mas diversas ferramentas, notadamente *crypters* [Tasiopoulos and Katsikas 2014].

```
b9c93c4cf9f0848f03834614c6f91759e4e068c0  original.bin
3a5d19d3372f4d8e05599da542bfa161a14a4a32  upx-file.bin
```

Código 4.17. Diferença no sha1sum dos binários original e empacotado

Felizmente, a ferramenta *UPX* permite desempacotar o arquivo empacotado como exibido no Código 4.18, de modo a retornar o arquivo ao estado original, permitido, assim, a análise do exemplar não ofuscado. Contudo, este é um caso particular, visto que muitos empacotadores requerem procedimentos complexos para que os binários gerados por estes possam ser desempacotados [Cheng et al. 2018, Coogan et al. 2009].

```
>$ upx -d upx-file.bin
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep
    30th 2013
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     610311 <-    270256   44.28%  netbsd/elf386  1-upx
Unpacked 1 file.
```

Código 4.18. Desempacotamento de um binário empacotado via UPX

4.4. Modificações em Arquivos Binários

Procedimentos de análise estática, como mostrado na Seção 4.3, são úteis para se obter informações dos binários de modo passivo, isto é, sem interferir na estrutura do binário em questão. Apesar de efetivo em muitos casos, muitas construções maliciosas, como técnicas de ofuscação, podem impedir a obtenção de dados significativos para a caracterização do comportamento do binário como malicioso ou benigno. Nestes casos, o analista pode optar por alterar a estrutura do binário de modo a exibir informações ocultas. Pode-se, por exemplo, contornar as rotinas de proteção e/ou ofuscação de modo a se inspecionar o real comportamento do binário. Apresenta-se, a seguir, uma estratégia de modificação de binários (*binary patching*) útil para estes casos.

4.4.1. *Binary Patching*

A técnica de *binary patching* consiste em alterar diretamente os *bytes* do binário sob análise de modo que este revele algum comportamento de interesse do analista. Embora a alteração possa ocorrer em qualquer seção do binário, esta é usualmente empregada sob as instruções de modo a contornar alguma rotina de verificação implementada pelo atacante. Por exemplo, o exemplar ilustrado pelo Código 4.19 requer que uma senha seja digitada para que a execução continue. Caso o analista não seja capaz de descobrir a senha por outros modos, a eliminação desta rotina de verificação é o único modo de se observar o progresso da execução deste binário.

```
1 int password()
2   scanf("%[^\n]s",string);
3   if(strcmp(string,PASS)==0)
4     malicious();
5   else
6     exit(0);
```

Código 4.19. Exemplo de código de um exemplar que exibe seu comportamento malicioso apenas quando a string de inicialização é corretamente definida.

```
400692      call        wrapper_601030_400520
400697      test        eax, eax
400699      jnz         loc_4006a5
40069b      mov         edi, strz_Oh_Yeah__40075a
4006a0      call        wrapper_601018_4004f0
4006a5
```

Figura 4.3. *Disassembly* da função verificadora. A função verificadora é invocada (linha 1) e seu valor de retorno é testado (linha 2) de modo a determinar se a execução deve proceder (linha 4) ou não (linha 3).

Técnicas de *binary patching* são independentes de ferramentas, já que são diretamente aplicadas ao binário alvo. Contudo, para fins didáticos, nos referiremos aqui a solução *HT Editor* (*hte*) [S. Weyergraf 2015]. Navegando pelo *hte* através de `F6 ->`

`elf/image` e `F8 -> main`, obtém-se a visão (*view*) ilustrada pela Figura 4.3. Nesta, identifica-se a sequência de instruções que implementa o comportamento exibido pelo Código 4.3: *call* para chamar a função *strcmp* comparar as *strings*; *test* para realizar a comparação do valor de retorno; e *jnz* para mudança no fluxo de execução caso o valor de retorno não seja o esperado, ou seja, senha incorreta.

Uma das formas de se forçar o prosseguimento da execução mesmo sem o conhecimento da senha requerida é eliminar a instrução *jnz*, de modo que o programa sempre execute o trecho seguinte. Na prática, contudo, não é possível meramente eliminar uma instrução, pois isto corromperia a estrutura do binário e desalinharia as regiões de memória. Desta forma, uma alternativa viável é substituir a instrução por uma de tamanho equivalente mas que não execute nenhuma ação concreta, como, por exemplo, a instrução *nop* (*no operation*). Para o exemplo apresentado, como a instrução *jnz* ocupa 2 *bytes* de memória, duas instruções *nop* de 1 byte cada são requeridas. Esta substituição pode ser realizada no *hte* via navegação (*F4*) até a instrução *jnz* e substituir pelo código `0x90` (`nop` duas vezes, como exibido na Figura 4.4. O binário alterado pode ser salvo (*F2*) e executado normalmente.

```
400692      call        wrapper_601030_400520
400697      test        eax, eax
400699      nop
40069a      nop
40069b      mov         edi, strz_Oh_Yeah__40075a
4006a0      call        wrapper_601018_4004f0
```

Figura 4.4. *Disassembly* da função verificadora. A função verificadora pode ser modificada de modo que a rotina de verificação seja eliminada.

4.4.2. Verificações de integridade

Assim como atacantes estão cientes de que seus códigos serão inspecionados por procedimentos passivos de análise estática, estes também estão cientes de que analistas tentarão modificar os binários distribuídos. Como contramedida, atacantes implementam rotinas de verificação de integridade, de modo a impedir ou dificultar a modificação das funções de verificação implementadas nos binários maliciosos. Um tipo popular de verificação é o calculo do *Cyclic Redundancy Check* (*CRC*) [Peterson and Brown 1961]. O Código 4.20 ilustra uma versão modificada do exemplo anterior na qual a função *passwd*, responsável por verificar a senha, possui um *CRC* calculado previamente (*MYCRC*), o qual também é verificado durante a execução, de modo que uma alteração na função devido a *binary patching* resulte no encerramento da aplicação.

```
1 int main()
2     crc32 = Crc32_ComputeBuf(0,
          passwd,33);
3     if(crc32!=MYCRC)
4         exit(0);
5     passwd(pass);
```

Código 4.20. Exemplo de código de verificação de CRC32 para proteger a função verificadora de strings.

Para realizar análise do exemplar com verificação de integridade via *CRC*, deve-se, agora, realizar *binary patching* em duas seções do exemplar: (i) na função *passwd* para ignorar o resultado da verificação da senha, tal qual no caso anterior; e (ii) na rotina de verificação de integridade da função *passwd* via `CRC`. Através do uso do *hte*, pode-se realizar *binary patching* das duas seções; a primeira tal qual demonstrado anteriormente, e a segunda tal qual exibido na Figura 4.5, onde a instrução *jz* foi substituída pela instrução *jnz* de modo a se evitar o término da execução mesmo com a falha de verificação de integridade da função modificada. Resultado semelhante pode ser obtido de forma dinâmica, como mostrado na Seção 4.6.

```
4007fd      call        Crc32_ComputeBuf
400802      mov         [rbp-    ], rax
400809      cmp         qword ptr [rbp-    ],
400814      jnz         loc_400820
400816      mov         edi,
40081b      call        wrapper_602050_400680
```

Figura 4.5. *Disassembly* da região ao redor da função verificadora.. Rotina de verificação do CRC32 é invocada (linha 1) e resulta no término da execução (linha 4) caso a função original tenha sido alterada.

4.5. Traços de Execução

A análise estática permite ao analista obter informações preliminares sobre o possível comportamento do binário suspeito e, assim, traçar um plano de investigação concreto. Contudo, devido as limitações discutidas na Seção 4.3, a confirmação das suspeitas se dá apenas através do emprego de métodos de análise dinâmica, com a execução do artefato suspeito em um ambiente controlado, denominado *sandbox* [Sikorski and Honig 2012]. Este ambiente deve limitar a ação do *malware* de modo a evitar que uma infecção se propague. No âmbito deste curso, recomenda-se executar os experimentos em uma máquina virtual desconectada da Internet.

O produto mais comum da execução de um binário em uma *sandbox* é um traço de execução: uma lista sequencial de ações realizadas pelo binário. As ações podem incluir chamadas de sistema (*syscalls*), de aplicações (API), de instruções de máquina executadas, entre outros. Nesta seção, descrevemos soluções para obtenção dos três tipos de traço de execução anteriormente referidos.

Contudo, assim como atacantes implementam técnicas de ofuscação para dificultar procedimentos de análise estática, atacantes também implementam técnicas de anti-análise para evadir a execução em *sandboxes*. Apesar de cumprir papel semelhante, a estratégia de evasão se difere da de ofuscação pois enquanto a última apenas mascara os dados, a segunda tenta impedir o procedimento de análise em si. A seguir, detalha-se estratégias para a obtenção de traços de execução de aplicações e para o contorno de técnicas de evasão de análise.

É importante notar que a plataforma Linux possui diversas soluções para *tracing* de binários (uma lista completa pode ser encontrada no *survey* [Gebai and Dagenais

2018]) e que, por questões didáticas, limitamo-nos a apresentar exemplos de utilização das soluções mais populares e que detalhes de todas as chamadas de sistemas apresentadas nos traços de execução podem ser consultados via *manpages* [Kernel.org 2017].

4.5.1. A ferramenta e o comando `strace`

Uma das soluções mais populares para a obtenção de traços de execução a nível de chamadas de sistema (*syscall*) é o *strace* (*system call tracer*), que reporta, sequencialmente, as chamadas de sistemas executadas em conjunto dos argumentos passados para estas e dos valores de retorno destas.

O Código 4.21 ilustra um processo realizando a chamada de sistema *open*, com `strace` reportando o *pid* do processo (`11047`), os parâmetros da chamada (`/etc/passwd`) e o valor de retorno (`-1`). Neste exemplo, realizado por um exemplar malicioso, o processo tenta acessar o arquivo `passwd`, que contém informação das senhas de usuários, um tipo de ação característica de exemplares maliciosos que tentam realizar roubo de credenciais. Para este caso específico, o valor de retorno `-1` indica que o acesso a este arquivo foi negado, o que era esperado dado que, em sistemas típicos, o arquivo `passwd` só pode ser acessado com permissões de super-usuário (*root*).

```
1 [pid 11047] open("/etc/passwd", O_WRONLY|O_CREAT|O_APPEND, 0666) = -1
      EACCES (Permission denied)
```

Código 4.21. Exemplo de registro de chamada de sistema via strace de uma tentativa de acesso a credenciais do usuário (/etc/shadow).

Enquanto uteis para identificar padrões de acesso e comportamentos maliciosos (uma lista detalhada de comportamentos maliciosos é apresentada no artigo [Grégio et al. 2012]), traços de execução, por conterem todas as chamadas que um binário executa, apresentam muitas informações que não são essenciais aos procedimentos de análise, como rotinas de inicialização e de *clean up*, causando poluição dos arquivos de *log*. O Código 4.22 ilustra um exemplar malicioso sendo inicializado (carregando bibliotecas via `access`) e, posteriormente, já próximo ao termino de sua execução, exibindo seu código de *usage*, no caso, requisitando os endereços IP e porta para executar corretamente, visto que este exemplar realiza ataques via rede. Observa-se, ao final do traço, a chamada de sistema para o término do processo: `exit_group` com valor menos um, dado que a execução esperada (com parâmetros corretos) falhou.

```
1  >$ strace ./malware.bin
2  execve("./malware.bin", ["./malware.bin"], [/* 25 vars */]) = 0
3  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
4  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
5  ...
6  write(2, "Invalid␣parameters!\n", 20Invalid parameters!)   = 20
7  write(1, "DNS␣Flooder␣v1.3\nUsage:␣./malwar"..., 244DNS Flooder v1.3
8  Usage: ./malware.bin <target IP/hostname> <port to hit> <reflection
       file (format: IP DOMAIN>> <number threads to use> <time (optional
       )>
9  exit_group(-1)                          = ?
10 +++ exited with 255 +++
```

Código 4.22. Traço de chamadas de sistema de exemplar malicioso exibindo rotinas de carregamento e de finalização

Uma vantagem significativa do `strace` sobre outros métodos de análise é que os traços de execução obtidos por este independem do tipo de ligação (estática ou dinâmica) entre binários e bibliotecas. Desta forma, ainda que o atacante tenha conseguido esconder as funções referenciadas em tempos de compilação, estas aparecerão nos traços dinâmicos caso sejam invocadas. Portanto, a única observação divergente entre traços de execução de binários estáticos e dinâmicos é a presença de diversas funções de carregamento de bibliotecas executadas pelas rotinas de inicialização da aplicação, tal qual ilustrado pelo Código 4.23.

```
1  strace 0
       c4f1bf21f943331f9952a594fa37a868708a9458180c2a156a08ae9a9a1d29f
2  execve("./malware.bin", ["./malware.bin"], [/* 25 vars */]) = 0
3  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
       directory)
4  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
       directory)
5  open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
       directory)
7  open("/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
8  access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
       directory)
9  open("/usr/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC)
       = 3
10 open("/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
11 open("/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
12 open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
13 rt_sigaction(SIGINT, {0x408de1, [INT], SA_RESTORER|SA_RESTART, 0
       x7fbc2ac834b0}, {SIG_DFL, [], 0}, 8) = 0
14 rt_sigaction(SIGILL, {0x408de1, [ILL], SA_RESTORER|SA_RESTART, 0
       x7fbc2ac834b0}, {SIG_DFL, [], 0}, 8) = 0
```

Código 4.23. Traço de chamadas de sistema de binário carregando diversas bibliotecas

Embora o *strace* não apresente restrições quanto a análise de binários compilados estaticamente, atacantes podem dificultar o trabalho do analista ao dividir as ações maliciosas em múltiplos processos. Nestes casos, caso o analista obtenha um traço apenas do processo inicial, não observará os comportamentos associados aos demais processos. Contudo, ao "seguir" os processos filhos criados pelo processo inicial, os comportamentos podem ser observados normalmente.

O Código 4.24 ilustra o funcionamento de um exemplar malicioso que realiza a chamada de sistema *clone* (equivalente a *fork*), que cria um processo filho, idêntico ao processo pai, mas com identificador (*pid*) diferente. O processo filho começa sua execução do ponto de chamada do processo pai e com os mesmos valores de variáveis. Contudo, por ser um processo independente, ações diferentes do processo pai podem ser executadas com o prosseguimento da execução. A análise do traço de exemplo permite identificar tais diferenças, pois, após realizar *clone*, o processo pai entra em espera *wait* para que o filho assuma a execução. O filho, que carrega o conteúdo malicioso (*payload*), por sua vez, evade a análise por detectar o ambiente *ptrace*, como será discutido em detalhes no futuro. O rastreio de ações do processo filho pode ser executado inicializando a ferramenta *strace* com o argumento `-f`.

```
1  >$ strace -f
       d73a9a8dbd1a4a73048d973457eaa854fe19c689bcd8f87395d65ae8b1d7587d
2  execve("./malware.bin", ["./malware.bin"], [/* 25 vars */]) = 0
3  brk(NULL)                                = 0x240b000
4  access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or
       directory)
5  access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or
       directory)
6  ...
7  clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|
       SIGCHLD, child_tidptr=0x7fbc2bce09d0) = 11045
8  wait4(-1, strace: Process 11045 attached
9   <unfinished ...>
10 [pid 11045] ptrace(PTRACE_TRACEME, 0, NULL, NULL) = -1 EPERM (
       Operation not permitted)
11 [pid 11045] execve("./malware.bin", ["./malware.bin", "2", "3", "4"],
        [/* 25 vars */]) = 0
12 ...
```

Código 4.24. Traço de chamadas de sistema de exemplar malicioso com chamada fork e evasão de execução por parte do processo filho

Outra técnica de evasão de análise que também afeta o *strace* é o *delay* na execução de chamadas, comumente implementado através de chamada da função *sleep* por um tempo suficientemente longo para que a análise termine antes dos comportamentos maliciosos serem exibidos [Grégio et al. 2012], seja durante uma sessão de depuração ou durante a execução em um ambiente de análise automatizado limitado por um *timeout*. O Código 4.25 ilustra a ocorrência deste tipo de técnica de evasão durante um traço de execução de um processo malicioso. Nota-se que nenhuma outra chamada de sistema é reportada após a invocação da função *nanosleep* dado que esta não retornou dentro do período de análise deste binário. Um método para mitigar os efeitos deste tipo de técnica de evasão é discutido na Seção 4.6.2.

```
1  >$ strace
       a4332ab4b8f8e2f04fef7c40c103ab570f42011ba41b3caaa03029b8928cba2e
2  [pid 11046] waitpid(11084, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}],
       0) = 11084
3  [pid 11046] munmap(0xf6f0a000, 4096)     = 0
4  [pid 11046] ioctl(1, SIOCGIFFLAGS, {ifr_name="ens3", ifr_flags=IFF_UP
       |IFF_BROADCAST|IFF_RUNNING|IFF_MULTICAST}) = 0
5  [pid 11046] gettimeofday({312344432895491, -17819621742608320}, NULL)
        = 0
6  [pid 11046] nanosleep({429496729600000000, 577756380723720712},  <
       unfinished ...>
```

Código 4.25. Traço de chamadas de sistema de exemplar malicioso que realiza uma chamada sleep com tempo suficientemente longo para causar o término da execução em muitos sistemas de análise

Finalmente, uma outra forma de evadir análise baseadas em `strace` é explorar os efeitos colaterais das decisões de projeto para a construção da própria ferramenta `strace`, que é baseada no *framework* `ptrace` [Kernel.org 2017], uma poderosa solução de monitoração nativa da plataforma `Linux` (implementações baseadas em `ptrace` são

discutidas na Seção 4.5.3). `Ptrace` estabelece um *lock* quando anexado a um processo, de modo que nenhum outro processo possa se anexar ao mesmo processo sendo analisado. Naturalmente, ao se analisar um binário usando `strace`, este *lock* é estabelecido pela solução de *tracing*.

O estabelecimento deste *lock* implica na possibilidade de detecção do ambiente `ptrace` por parte de binários maliciosos casos estes possam detectar que o *lock* tenha sido estabelecido para o seu próprio processo malicioso. Atacantes podem detectar o *lock* ao tentar estabelecer uma ligação `ptrace` consigo mesmo, pois esta falhará caso o *lock* já tenha sido estabelecido por algum processo anterior (no caso, o `strace`). Deste modo, atacantes podem evadir processos de análise quando da identificação da execução em uma solução de *tracing*. Uma possível implementação deste tipo de evasão é ilustrado no Código 4.26

```
1 int main(){
2   if(ptrace(PTRACE_TRACEME)==-1)
3      exit(0);
4   malicious()
```

Código 4.26. Evasão de análise por strace através da autoverificação da presença do framework ptrace.

Caso este código seja executado em `strace`, a análise será encerrada, como exemplificado pelo Código 4.27. Contudo, sua execução em ambientes nativos, como as máquinas das vítimas/usuários, o código executa naturalmente e exibe seus comportamentos maliciosos. Este tipo de técnica de evasão pode ser contornada através da modificação do fluxo de execução, como descrito na Seção 4.6.

```
1 >$ strace ./ptrace
2 ...(ommitted initialization system calls)
3 ptrace(PTRACE_TRACEME, 18446744073049584056, 0
     x7ffdd8a9b1c8, NULL) = -1 EPERM (Operation
     not permitted)
4 exit_group(0)                           = ?
5 +++ exited with 0 +++
```

Código 4.27. Traço de chamadas de sistema de arquivo com evasão de análise via chamada ptrace

4.5.2. Ltrace

A solução de análise de traço de execução *ltrace* é o análogo da solução *strace* para a obtenção de traços de execução a nível de chamadas de funções (APIs). O traço a nível de API é útil para determinar comportamentos maliciosos em mais alto nível, uma vez que diferentes APIs (`printf, putc`) podem resultar na mesma chamada de sistema (`write`). O Código 4.28 exemplifica o traço de um exemplar malicioso que carrega o *exploit DirtyCow* [Oster 2019] para escalar privilégios e copiar o arquivo de senhas *passwd*.

```
>$ ltrace
   b0148c40166b2b6cea71e1f00100a05087d2a492f21e008e30c594cf91ecf9f1.ltr

[pid 11043] __libc_start_main(0x400ab2, 1, 0x7ffc53304f98, 0x400d00 <
   unfinished ...>
[pid 11043] printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"..., "/usr/bin/
   passwd") = 308
[pid 11043] puts("DirtyCow␣root␣privilege␣escalati"...) = 35
[pid 11043] printf("Backing␣up␣%s␣to␣/tmp/bak\n", "/usr/bin/passwd")
   = 39
[pid 11043] asprintf(0x7ffc53304ea0, 0x400f1e, 0x6020c0, 0x7fffffe5)
   = 27
[pid 11043] system("cp␣/usr/bin/passwd␣/tmp/bak" <no return ...>
...
```

Código 4.28. Traço de chamadas de funções de exemplar malicioso executando um exploit para a vulnerabilidade DirtyCow

Tal qual `strace`, análises baseadas em `ltrace` devem "seguir" processos filhos (`fork`) para evitar evasão. Similarmente, `ltrace` também é vulnerável a evasão por longos *delays* e possivelmente por detecção de *locks* do tipo `ptrace`. Além disso, atacantes também exploram outro fator derivado das decisões de projeto para a implementação de `ltrace`: dado que `ltrace` instrumenta as APIs monitoradas através da alteração da tabela de símbolos, este só pode lidar com binários ligados dinamicamente. Em caso de binários estáticos, `ltrace` exibe um erro, como ilustrado no Código 4.29.

```
>$ ltrace ./static-malicious.bin
Couldn't␣find␣.dynsym␣or␣.dynstr␣in␣"/proc/25332/exe"
```

Código 4.29. Traço de chamadas de funções reportando falha ao analisar um binário estaticamente ligada via ltrace

4.5.3. Ptrace

Soluções como *strace* baseiam seu funcionamento no *framework ptrace*, que fornece uma interface clara para o monitoramento de aplicações. Através da chamada `ptrace`, tal qual mostrada no Código 4.30, um processo (`monitor`) pode monitorar um outro processo (`monitorado`) especificado através de um *Process IDentifier* (PID). Em arquiteturas de monitoramento típicas, tais quais as implementadas por `strace` e `ltrace`, o processo monitor invoca `ptrace` sobre um processo filho, o binário monitorado.

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr,
   void *data);
```

Código 4.30. Protótipo para a chamada ptrace

`Ptrace` é usado como suporte para a construção de diferentes tipos de monitores, de *traces* como os apresentados a *debuggers*, como o GDB [Alves et al. 2017]. Isto é possível devido as diversas capacidades de monitoração providos por este, especificadas pela *enum ptrace_request*, e que incluem: obtenção dos valores nos registrador do processador (`PTRACE_GETREGS`), de valores e argumentos de chamadas de sistema (`PTRACE_SYSCALL`), e até mesmo execução *single-step* (`PTRACE_SINGLESTEP`).

Consulte o repositório indicado na introdução deste capítulo para a obtenção de exemplos de como programar usando `ptrace`.

Considerando as capacidades de monitoração providas por `ptrace` e as capacidades de desenvolvimento da plataforma `Linux`, analistas podem desenvolver suas próprias soluções de análise, customizando-as de acordo com suas necessidades. Uma solução de análise interessante de ser desenvolvida a partir de `ptrace` é um *instruction tracer* (`itracer`), através das capacidades de execução e inspeção *single-step*, o que permite obter detalhes da execução de processos com informações de baixo nível. O Código 4.31 ilustra o traço de execução de um arquivo malicioso quando este realiza mudanças do fluxo de execução entre o binário monitorado (`itrace`) e uma biblioteca ligada (`ld-2.23.so`). Os endereços providos permitem ao analista identificar com exatidão os pontos de troca de contexto.

```
1  >$ itrace
       f5d5557da51fb1ac8da1a84fc9a6783496cadf2be438110995dbebe6ae0337d4
2  RIP: 0x400c63 | /home/SBSeg/itrace 0x400000 0x403000
3  RIP: 0x400c6a | /home/SBSeg/itrace 0x400000 0x403000
4  RIP: 0x7f446f862c17 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
5  RIP: 0x7f446f862c1a | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
6  RIP: 0x7f446f862c1e | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
7  RIP: 0x7f446f862c20 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
8  RIP: 0x7f446f862c27 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
9  RIP: 0x7f446f862c2a | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
10 RIP: 0x7f446f862c2c | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
11 RIP: 0x7f446f862c30 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
12 RIP: 0x7f446f862c33 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
13 RIP: 0x401d84 | /home/SBSeg/itrace 0x400000 0x403000
14 RIP: 0x401d88 | /home/SBSeg/itrace 0x400000 0x403000
15 RIP: 0x401d8c | /home/SBSeg/itrace 0x400000 0x403000
16 RIP: 0x7f446f862c35 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
17 RIP: 0x7f446f862c38 | /lib/x86_64-linux-gnu/ld-2.23.so 0x7f446f852000
       0x7f446f878000
```

Código 4.31. Traço de execução de um exemplar malicioso a nível de instruções. Pode-se identificar os pontos exatos de troca de contexto entre a execução do código do binário e da biblioteca ligada.

4.6. Alteração de contexto

Tal qual abordagens estáticas podem ser caracterizadas como passivas (Seção 4.3) ou ativas (Seção 4.4), abordagens dinâmicas também podem envolver apenas a observação da execução (Seção 4.5) ou a modificação ativa dos fluxos de execução, como apresentado nesta seção. Técnicas de modificação de fluxo consistem em alterar o caminho de ex-

ecução natural da aplicação, fazendo, por exemplo, que um salto (*branch*) seja tomado mesmo quando as condições lógicas para tal não são satisfeitas ou invocando uma função fora do seu contexto. Este tipo de técnica é útil para contornar rotinas de anti-análise (escapando do seu salto) e descobrir rotinas ocultas (invocando funções não referenciadas naquele contexto). Apresentamos, a seguir, duas estratégias para a modificação de contexto com foco na análise de binários maliciosos.

4.6.1. *Debugging*

Uma das soluções mais populares para a depuração (*debugging*) de código em plataformas Linux é o GDB [Alves et al. 2017], que também pode ser usado para realizar a engenharia reversa de aplicações, tal qual exemplificado nesta seção. O nosso objetivo não é esgotar todas as possibilidades de uso da ferramenta, mas apresentar algumas possibilidades importantes ao leitor, de modo que este possa, futuramente, se aprofundar no assunto através da literatura especializada em GDB [Matloff and Salzman 2008] ou engenharia reversa [Wong 2018].

Uma das capacidades do GDB é realizar o *disassembly* do código binário, como exemplificado pelo Código 4.32. Embora a solução `objdump` mostrada na Seção 4.3 também seja capaz de realizar esta tarefa, o código *assembly* exibido pelo GDB se baseia na interpretação dos *bytes* do binário carregados na memória e está sujeito as modificações de contexto realizadas pelo analista.

```
1  (gdb) disassemble main
2  Dump of assembler code for function main:
3     0x0000000000400646 <+0>:     push   %rbp
4     0x0000000000400647 <+1>:     mov    %rsp,%rbp
5  => 0x000000000040064a <+4>:     sub    $0x110,%rsp
6     0x0000000000400651 <+11>:    mov    %fs:0x28,%rax
7     0x000000000040065a <+20>:    mov    %rax,-0x8(%rbp)
8     0x000000000040065e <+24>:    xor    %eax,%eax
9     0x0000000000400660 <+26>:    mov    $0x400744,%edi
10    0x0000000000400665 <+31>:    callq  0x4004f0 <
          puts@plt>
```

Código 4.32. Disassembly realizado pelo GDB.

Por se tratar de uma solução de inspeção dinâmica, o GDB permite que o analista suspenda a execução em determinados pontos de interesse, denominados *breakpoints* ou pontos de parada, como exemplificado pelo Código 4.33. Este recurso permite que o analista se foque nos dados ao redor da região de interesse e inspecione o programa no exato momento em que uma atividade suspeita está sendo executada pelo binário desconhecido.

```
1  (gdb) b main
2  Ponto de parada 1 at 0x40064a
3  (gdb) r
4  Starting program: /home/sbseg/malware
5  Breakpoint 1, 0x000000000040064a in main ()
```

Código 4.33. Definição de Breakpoints no GDB.

Uma das principais informações a serem obtidas quando um ponto de parada é atingido é a verificação dos valores armazenados nos registradores, como exibido no Código 4.34. Os valores armazenados nos registradores indicam o estado da aplicação, como os registradores de propósito geral (`rax-rdx` e `r8-r15`), *flags* de processamento que indicam se alguma operação resultou em zero, *overflow*, entre outros, e o endereço da instrução sendo executado (`rip`). Note que, neste caso, o endereço apontado por `rip` é o endereço do ponto de parada.

```
(gdb) info registers
rax 0x400646            rbx    0x0
rcx 0x0                 rdx    0x7fffffffdd78
rsi 0x7fffffffdd68      rdi    0x1
rbp 0x7fffffffdc80      rsp    0x7fffffffdc80
r8  0x400730            r9     0x7ffff7de7ac0
r10 0x846               r11    0x7ffff7a2d740
r12 0x400550            r13    0x7fffffffdd60
r14 0x0                 r15    0x0
rip 0x40064a <main+4>   eflags 0x246 [ PF ZF IF ]
```

Código 4.34. Obtenção dos valores atuais nos registradores do programa em execução através do GDB.

Os locais de análise não precisam ser limitado pelos pontos de parada, de modo que um analista pode avançar a execução de forma granular a partir destes para melhor inspecionar uma região de interesse. Uma forma típica de se inspecionar uma aplicação é realizar sua execução passo a passo (*step-by-step*). O Código 4.35 ilustra a execução passo a passo a partir do ponto de parada anteriormente estabelecido através do comando `stepi`, exibindo a instrução executada em cada etapa através da definição `display/i $pc`.

```
(gdb) display/i $pc
Breakpoint 1, 0x000000000040064a in main ()
1: x/i $pc
=> 0x40064a <main+4>:   sub    $0x110,%rsp
(gdb) stepi
0x0000000000400651 in main ()
1: x/i $pc
=> 0x400651 <main+11>:  mov    %fs:0x28,%rax
(gdb)
0x000000000040065a in main ()
1: x/i $pc
=> 0x40065a <main+20>:  mov    %rax,-0x8(%rbp)
(gdb)
0x000000000040065e in main ()
1: x/i $pc
=> 0x40065e <main+24>:  xor    %eax,%eax
```

Código 4.35. Avanço de execução passo a passo.

Fazendo uso dos recursos anteriormente apresentados, analistas podem identificar uma região de interesse, como um ponto de evasão, e modificá-lo de forma a dar

prosseguimento ao processo de análise. Por exemplo, no caso da identificação de uma rotina de evasão implementada através de uma construção do tipo `if-else`, que resulta em instruções de desvio (*branch*) quando interpretadas em `assembly`, o analista pode inverter o caminho originalmente tomado pela aplicação, de modo que não se execute a rotina de evasão mas sim as rotinas maliciosas. Dado que instruções de desvio são controladas pelo registrador de *flags*, a modificação do caminho executado pode se dar pela modificação das *flags* armazenadas neste registrador. O Código 4.36 ilustra um trecho de uma sessão de depuração na qual o registrador de *flags* é alterado, respectivamente, para incluir e remover a flag zero (`ZF`), representada pelo *byte* `0x40`. Consulte a referência [C-Jump 2017] para obter informações sobre a representação de todas as demais *flags*.

```
(gdb) set $eflags|=0x40
(gdb) info registers eflags
eflags         0x246    [ PF ZF IF ]
(gdb) set $eflags|=~0x40
(gdb) info registers eflags
eflags         0x54fd7  [ CF PF AF ZF SF TF IF DF OF NT RF AC ]
```

Código 4.36. Alteração do registrador de flags via GDB.

Além de valores em registrador, a depuração via GDB também permite a alteração de valores em memória, o que permite ao analista fazer uso de diversas técnicas, incluindo *binary patching*, como mostrado na Seção 4.4. O Código 4.37 ilustra um trecho de uma sessão de depuração que realiza o *patch* do binário tal qual mostrado na Seção 4.4. Este tipo de alteração em memória é muito útil para a realização de testes sem a necessidade de se salvar o binário modificado.

```
(gdb) disassemble main
Dump of assembler code for function passwd:
   0x0000000000400692 <+76>:    callq  0x400520 <strcmp@plt>
   0x0000000000400697 <+81>:    test   %eax,%eax
   0x0000000000400699 <+83>:    jne    0x4006a5 <main+95>
(gdb) set {int}0x0000000000400699 = 0x90
(gdb) set {int}0x000000000040069a = 0x90
Dump of assembler code for function passwd:
   0x0000000000400692 <+76>:    callq  0x400520 <strcmp@plt>
   0x0000000000400697 <+81>:    test   %eax,%eax
   0x0000000000400699 <+83>:    nop
   0x000000000040069a <+84>:    nop
```

Código 4.37. Binary Patching via GDB.

As capacidades de alteração de fluxo do GDB não se limitam a alteração dos valores de contexto das aplicações analisadas, mas também pode se dar pela invocação de funções de maneira independente do contexto, de modo que analistas possam descobrir funções ocultas dentro de binários, uma prática muito comum em exemplares maliciosos. O Código 4.38 ilustra a chamada da função verificadora de senha exibida na Seção 4.4 de modo independente das verificações de integridade, o que possibilita a execução da mesma sem a necessidade de se realizar um *patch* do binário. Enquanto o valor de retorno negativo para a primeira chamada indica que a verificação falhou, o valor de retorno positivo para a segunda chamada indica que a senha foi identificada.

```
(gdb) call passwd("MYPASS")
$1 = -7
(gdb) call passwd("The Pass")
Oh Yeah!
$2 = 9
```

Código 4.38. Invocação de função independente do contexto através do GDB.

4.6.2. Injeção de Código com `LD_PRELOAD`

Um recurso de modificação de fluxo de execução nativamente presente na plataforma Linux é o *Linux dynamic linker preload* (*LD_PRELOAD*). que permite que uma biblioteca especificada pelo usuário seja chamada com preferência sobre as bibliotecas originalmente referenciadas pelo binário, tais como a *libc*, de modo que funções presentes na biblioteca injetada que possuam a mesma assinatura (nome, parâmetros e retorno) serão preferencialmente invocadas pelo binário em execução, como exemplificado pelo Código 4.39. A técnica de injeção de código via `LD_PRELOAD` é utilizada tanto por atacantes como por analistas, como descrito a seguir. As assinaturas das funções referidas nesta seção podem ser encontradas nas *manpages* [Kernel.org 2017].

```
LD_PRELOAD=./library.so ./binary.bin
```

Código 4.39. Exemplo de chamada LD_PRELOAD na qual a biblioteca substitui funções originalmente referenciadas pelo binário

4.6.2.1. Contornando anti-análise com `LD_PRELOAD`

Na Seção 4.5.1, apresentamos alguns desafios de análise dinâmica de exemplares evasivos, dentre os quais o *delay* de execução via longas chamadas da função *sleep*. A injeção de código via *LD_PRELOAD* pode ser utilizada para contornar esse tipo de técnica de anti-análise através da substituição da função original por uma função modificada que ignore o tempo estipulado pelo atacante e retorne imediatamente, como exemplificado pelo Código 4.40.

```
unsigned int sleep(unsigned int seconds){
  return 0;
}
```

Código 4.40. Implementação alternativa da função sleep para contornar a evasão de análise via longos delays

Devido a precedência de execução da biblioteca injetada em relação a biblioteca original, a função *sleep* modificada será invocada pelo *malware* e a execução deste procederá normalmente, possibilitando a análise deste.

4.6.2.2. *Logging* usando `LD_PRELOAD`

A modificação do fluxo de execução através de *LD_PRELOAD* pode ser utilizada para se implementar mecanismos de *log* customizados, de forma semelhante a `strace` e `ltrace`, através da substituição das funções originais por funções que registrem a chamada da função. Para isso, deve se estabelecer uma função `trampolim` que realize este registro e, em seguida, retorne a execução para a função originalmente referenciada.

A Figura 4.6 exibe o fluxo de execução original de invocação de uma função a partir de um processo não monitorado, realizando a busca do endereço da função a ser executada na tabela de símbolos.

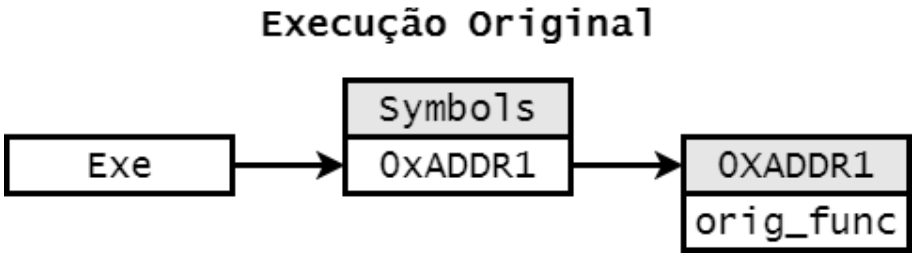

Figura 4.6. *Logging* com LD_PRELOAD. Fluxo original de execução através da invocação da chamada de função originalmente definida na tabela de símbolos.

A Figura 4.7, por sua vez, exibe o fluxo de execução alterado devido a injeção de código via *LD_PRELOAD*. Neste caso, uma função (trampolim) com a mesma assinatura da original é invocada previamente e realiza o registro das informações de *log*. Após o registro, a execução é direcionada para o endereço original para que a função monitorada realize a operação requisitada pelo binário.

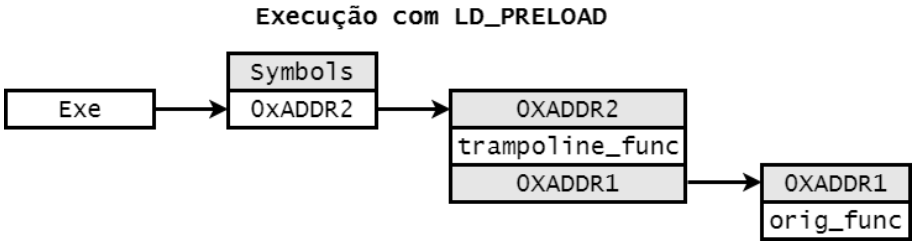

Figura 4.7. *Logging* com LD_PRELOAD. Fluxo de execução alterado após a modificação da tabela de símbolos por *LD_PRELOAD* para a invocação de uma função trampolim.

Ao contrário do caso da função *sleep*, em que ignorar o comportamento original da função é desejável para se mitigar evasões por *delay* da execução, no caso da implementação de um mecanismo de *logging*, deseja-se que a função original seja executada. Caso contrário, a execução do binário seria prejudicada. Desta forma, deve-se proceder a invocação da função original após o registro da operação, tal como exemplificado pelo Código 4.41. Note que esta estratégia requer que (i) a função trampolim identifique e armazene o endereço da função original e que (ii) a função original seja invocada com os mesmos parâmetros passados para a função trampolim.

```
typedef int (*orig_puts_f_type)(const char *str);

int puts(const char *str){
    FILE *fd = fopen("log","a+");
    orig_fprintf_f_type orig_fprintf;
    orig_fprintf = (orig_fprintf_f_type)dlsym(RTLD_NEXT,"fprintf");
    orig_fprintf(fd,str);
    fclose(fd);

    orig_puts_f_type orig_puts;
    orig_puts = (orig_puts_f_type)dlsym(RTLD_NEXT,"puts");
    return orig_puts(str);
}
```

Código 4.41. Código da função trampolim injetada via LD_PRELOAD para registrar a invocação da função. Após o registro, deve-se invocar a função original para garantir a realização da operação

4.6.2.3. *Rootkits* baseados em `LD_PRELOAD`

A capacidade de injeção de código provida pelo mecanismo *LD_PRELOAD* é explorada não apenas por analistas mas também por atacantes, sendo observada frequentemente na prática através de *rootkits*, um tipo de *malware* utilizado para esconder ou camuflar processos maliciosos do usuário [Beegle 2007], por exemplo, no caso de mineradores de cripto moedas [Remillano 2018].

O Código 4.42 exibe uma prova de conceito de um *rootkit* injetado via `LD_PRELOAD` na ferramenta *list directory contents* (*ls*), de modo a esconder um arquivo malicioso (*HIDDEN*) durante a leitura do diretório, de modo que um usuário seja incapaz de localizá-lo.

```
typedef struct dirent* (*orig_readdir_type)(DIR *dirp);
  struct dirent *readdir(DIR *dirp){
  struct dirent* valueOfReturn;
  orig_readdir_type orig_readdir;
  orig_readdir = (orig_readdir_type)dlsym(RTLD_NEXT,"readdir");
  valueOfReturn = orig_readdir(dirp);
  if(strcmp(HIDDEN,valueOfReturn->d_name) == 0)
    return NULL;
  return valueOfReturn;
}
```

Código 4.42. Exemplo de rootkit capaz de impedir a listagem de um arquivo.

4.6.2.4. Detectando injeções com `LD_PRELOAD`

Embora bastante poderoso, a injeção de código por `LD_PRELOAD` também pode ser contornada por atacantes, seja pela utilização de ligação estática, gerando, assim, um binário com tabela de símbolos vazia, impossibilitando a instrumentação, seja pela detecção direta da injeção de código no processo em execução, como exemplificado pelo Código 4.43. Este código detecta, através de execução (*getenv* ou */etc*), se as bibliotecas

de *LD_PRELOAD* estão presentes. Como contramedida, analistas podem se utilizar de instrumentação a nível de *kernel*, como discutido na Seção 4.8.

```
1  int main()
2  {
3      if(getenv("LD_PRELOAD"))
4          printf("LD_PRELOAD detected through getenv()\n");
5      else
6          printf("Environment is clean\n");
7      if(open("/etc/ld.so.preload", O_RDONLY) > 0)
8          printf("/etc/ld.so.preload detected through open()\n");
9      else
10         printf("/etc/ld.so.preload is not present\n");
11     return 0;
12 }
```

Código 4.43. Detecção da injeção de código via LD_PRELOAD

4.7. Demais Mecanismos de Monitoração em Modo Usuário

Além das soluções de análise especificamente voltadas a inspeção, monitoração e análise de binários, tais quais as anteriormente apresentadas, a plataforma `Linux` apresenta outras soluções que podem ser utilizadas para fins de análise em cenários específicos. Apresentamos, a seguir, o uso de monitores do sistema de arquivos e de mecanismos de *log* para fins de identificação de comportamentos maliciosos.

4.7.1. Monitoração do Sistema de Arquivos

Embora não sejam especificamente focados na análise e engenharia reversa de exemplares suspeitos, alguns mecanismos de monitoração podem ser empregados para este fim quando da operação em contextos específicos. Por exemplo, a monitoração do sistema de arquivos pode indicar atividades suspeitas relacionadas a criação e remoção de arquivos. Este tipo de monitoração pode ser utilizada, por exemplo, para a identificação de exemplares maliciosos pertencentes a famílias de *ransomware*, como o *Erebus* [Z. Chang 2017], que realizam acessos sequenciais aos arquivos do sistema para encriptá-los e demandar resgate.

A monitoração de ações no sistema de arquivo pode ser monitorada pelo *framework* `inotify` [McCutchan 2005], que permite a um administrador do sistema colocar vigias (*watches*) em arquivos e/ou diretórios de interesse. O Código 4.44 apresenta *logs* de monitoração via `inotify` de um diretório afetado pela execução do `Erebus`. Nota-se que o exemplar criar cópias criptografadas dos arquivos originais, sendo estes deletados. O uso da *API inotify* alerta o sistema ou usuário que a configurou em tempo real, de modo que se possa tomar uma possível ação de bloqueio ou de registro do evento em um *log* para posterior análise forense.

```
CREATE event: /home/sbseg/myfile.1
DELETE event: /home/sbseg/myfile
CREATE event: /home/sbseg/myfile2.1
DELETE event: /home/sbseg/myfile2
CREATE event: /home/sbseg/myfile3.1
DELETE event: /home/sbseg/myfile3
```

Código 4.44. Monitoração do diretório sbseg/ via inotify durante a execução do ransomware Erebus.

4.7.2. Mecanismos de *Logging*

Além da monitoração direta do sistema de arquivos via `inotify`, outros mecanismos de *log* presentes na plataforma `Linux` podem ser utilizado para monitorar os múltiplos subsistemas, o que pode incluir chamadas de sistema, comandos de usuário, eventos de segurança e acesso de rede. Uma das soluções que possibilita esta monitoração é o *Linux Audit system* (`Audit`) [Jahoda et al. 2017], sendo totalmente configurável a partir de regras. O Código 4.45 exemplifica o estabelecimento de regras para registrar acessos aos arquivos `passwd` e `shadow`.

```
auditctl -w /etc/passwd -p r
auditctl -w /etc/shadow -p r
```

Código 4.45. Definição de regra do Audit para monitoração da leitura dos arquivos passwd e shadow.

O `Audit` não provê mecanismos ativos para o bloqueio de ataques, se limitando a registrar os eventos ocorridos num sistema em um arquivo de *log* para consulta posterior. Apesar disto, este tipo de monitoração pode indicar claramente a ocorrência de diversos ataques. O Código 4.46, por exemplo, ilustra o *log* gerado pelo *audit* quando da monitoração da execução de um binário explorando a vulnerabilidade *DirtyCow* [Oster 2019]. Nota-se que, após realizar a exploração, o binário acessa os arquivos *passwd* e *shadow*, para a atribuição de permissão de super-usuário, e eleva seus próprios privilégios para *root* através do comando *su*, registrado através da invocação desta chamada de sistema.

```
type=PATH msg=audit(1565115273.377:158): item=0 name="/etc/passwd"
    inode=2105354 dev=08:01 mode=0100644 ouid=0 ogid=0 rdev=00:00
    nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000
    cap_fe=0 cap_fver=0
type=PATH msg=audit(1565115273.377:159): item=0 name="/etc/shadow"
    inode=2105378 dev=08:01 mode=0100644 ouid=0 ogid=0 rdev=00:00
    nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000
    cap_fe=0 cap_fver=0
type=SYSCALL msg=audit(1565115273.377:159): arch=c000003e syscall=2
    success=yes exit=5 a0=7f0a44c46c50 a1=80000 a2=1b6 a3=80000 items
    =1 ppid=2604 pid=4308 auid=4294967295 uid=1000 gid=1000 euid=0
    suid=0 fsuid=0 egid=1000 sgid=1000 fsgid=1000 tty=pts2 ses
    =4294967295 comm="su" exe="/home/sbseg/dcow" key=(null)
```

Código 4.46. Log do Audit coletado durante um ataque DirtyCow.

4.8. Mecanismos para Monitoração em *Kernel*

Quando atacantes desenvolvem exemplares de *malware* avançados a ponto de evadir os principais mecanismos de monitoração a nível de usuário, analistas devem recorrer a mecanismos de monitoração que operem em nível mais privilegiado que o objeto analisado, como o *kernel*, de modo a evitarem serem descobertos e subvertidos [Rossow et al. 2012]. Nesta seção, apresentamos os fundamentos da programação de módulos de *kernel* e da implementação de mecanismos para interposição de rotinas através destes. O objetivo desta seção não é ser um guia exaustivo para a modificação do *kernel*, mas sim uma introdução ao tema com foco em monitoração de aplicações. Informações detalhadas sobre a programação em *kernel* podem ser obtidas consultando a literatura especializada [Corbet et al. 2005].

4.8.1. Módulos de *Kernel*

Um *Loadable Kernel Module (LKM)* é um mecanismo para adicionar ou remover funcionalidades do *kernel* `Linux` em tempo de execução, sendo comumente usados por *drivers* de dispositivos, permitindo que o *kernel* se comunique com dispositivos de *hardware* sem possuir conhecimento prévios sobre o funcionamento destes. Ademais, LKMs também são frequentemente utilizados para implementar sistemas de arquivos e protocolos de rede, além de novas *features* no *kernel*, como *frameworks* de segurança [AppArmor 2019, Corporation 2015, Spengler 2019, Morris 2013b]. O uso de LKMs é vantajoso pois elimina a necessidade de se recompilar o *kernel* a cada alteração, reduz o tamanho total do código do *kernel* e também a complexidade das estruturas de controle responsáveis por gerenciar as novas funcionalidades. O código 4.8.1 apresenta o código fonte de um módulo simples, que utiliza das estruturas básicas do *kernel* para executar código em *kernel*. No caso, são exibidas mensagens de depuração no carregamento e descarregamento do módulo.

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  MODULE_LICENSE("GPL");
5
6  static int __init lkm_init()
7  {
8    printk(KERN_INFO "Kernel!\n");
9    return 0;
10 }
11 static void __exit lkm_exit()
12 {
13   printk(KERN_INFO "Tchau!\n");
14 }
15
16 module_init(lkm_init);
17 module_exit(lkm_exit);
```

4.8.2. Suporte a Chamadas de Sistema

Em última instância, as chamadas de API originadas em espaço de usuário são entregues ao *kernel* na forma de chamadas de sistema (*system calls*). O *kernel* `Linux` mantém uma

tabela de apontadores que fazem referência às chamadas de sistema e exporta estes endereços para o espaço de usuário. Essa tabela de apontadores (*syscall table*) em conjunto com as *syscalls* formam a *syscall layer* do *kernel* (ou *syscall subsystem*). Detalhes da interação dos dados entre nível de usuário e *kernel* podem ser encontrados na documentação interativa do *kernel* [The Linux Kernel doc. 2017].

Tal qual apresentado para o mecanismo `LD_PRELOAD`, a tabela de chamadas de sistemas também pode ser modificada para a implementação de funções trampolim. O uso deste tipo de técnica em nível de *kernel* é denominada *syscall hooking* e é implementado através dos módulos LKM anteriormente apresentados. Dado que a tabela de chamadas de sistema é global, a implementação deste tipo de técnica em *kernel* se mostra vantajosa pois dispensa a necessidade de se realizar o *trace* de processos individuais em modo usuário, sendo, portanto, uma opção para a análise de exemplares evasivos. De forma análoga aos casos anteriores, este tipo de técnica também pode ser utilizada por atacantes para a implementação de *rootkits* de modo *kernel*.

4.8.2.1. Substituição da Tabela de Chamadas de Sistema

De um modo geral, as etapas requiridas para efetuar *syscall hooking* são as seguintes:

1. Localizar, na memória do *kernel*, a tabela de *syscalls*.
2. Localizar o deslocamento (*offset*) do apontador da tabela de *syscalls* para a função que executa a *syscall* originalmente presente no *kernel* e que será substituída.
3. Atribuir permissão de escrita ao segmento de memória que contém a tabela.
4. Substituir o apontador da *syscall* de interesse originalmente presente na tabela de *syscall* pelo apontador para a nova *syscall* contendo a função de interposição.
5. Remover a permissão de escrita da tabela.

Localizando a tabela de chamadas de sistema Em versões recentes do *kernel* `Linux`, como mecanismo de proteção, a tabela de chamadas de sistema não é mais exportada nativamente. Desta forma, seu endereço deve ser obtido de maneira alternativa. Para se localizar a tabela de *syscall*, usualmente promove-se uma varredura de uma região da memória conhecida até se localizar o símbolo para o código de uma *syscall*, a partir da qual pode-se resolver dinamicamente o símbolo da tabela de *syscalls* que lhe aponta. Alternativamente, pode-se também realizar o *parsing* do arquivo `/boot/System.map-{versão-kernel}`, que armazena todos os símbolos do *kernel* para fins de *debug*. Para fins didáticos, nos limitaremos a exemplificar apenas a primeira abordagem.

Em procedimentos de varredura, o programador inicialmente considera o endereço de um símbolo do *kernel* localizado próximo ou dentro da página de memória da *syscall table*, como *sys_open*. A partir deste, realiza a leitura de sequências de 4 ou 8 *bytes* (para ponteiros de 32 ou 64 *bits*) e compara os valores destes com os valores de apontadores para *syscalls* conhecidas até que estes coincidam. Tendo identificado esta região como sendo referente a página que armazena a tabela de chamadas de sistema, pode-se obter

o endereço desta considerando os cinco ponteiros anteriores a *sys_close* (contando com *sys_newstat*).

Uma implementação típica deste tipo de abordagem utiliza como ponteiro inicial o símbolo *loops_per_jiffy*, que se encontra sempre anterior ao apontador para a tabela de *syscall* na lista de símbolos do *kernel* e promove iteração de uma palavra do processador por vez, usando o número de indexação de uma das *syscalls*, e.g., o macro *__NR_close* para o símbolo de *sys_close*, como uma lista de ponteiros. Caso este ponteiro esteja apontando para o mesmo endereço de memória (*entry point*) de uma *syscall*, a *syscall table* foi encontrada. O Código 4.47 exemplifica esta abordagem.

```
1  unsigned long ptr;
2  unsigned long *p;
3  static long (*sys_close) (unsigned int fd)=NULL;
4
5  sys_close=(void *)kallsyms_lookup_name("sys_close");
6  if (!sys_close)
7  {
8    printk(KERN_DEBUG "[HOOK] Symbol sys_close not found\n");
9    return NULL;
10 }
11
12 for (ptr = (unsigned long)sys_close;
13   ptr < (unsigned long)&loops_per_jiffy;
14   ptr += sizeof(void *))
15   {
16     p = (unsigned long *)ptr;
17     if (p[__NR_close] == (unsigned long)sys_close)
18       {
19         printk(KERN_DEBUG "[HOOK] Found the sys_call_table!!!\n");
20         return (unsigned long **)p;
21       }
22   }
23 return NULL;
24 }
```

Código 4.47. Função find_sys_call_table() utilizada para identificar o endereço da tabela de chamadas de sistema.

Escrevendo na tabela de chamadas de sistema Após a localização do endereço da tabela de chamadas de sistema, deve-se proceder a substituição do apontador contida nesta pelo apontador para a função de interceptação desejada. Contudo, para fins de segurança, desde a versão *2.6.12* do *kernel*, a região de memória contendo a tabela é marcada como *read-only* através do *bit* `Protected Mode Enable` no registrador *CR0* das arquiteturas Intel, o que requer que esta proteção seja desligada através da atribuição de permissão de escrita para a página de memória contendo a tabela de modo a possibilitar a escrita do novo apontador.

Nas arquiteturas *i386* e *x86-64* da Intel, a proteção e gerência das áreas de memória do *kernel* são controladas pelos registradores *Control register (CR0-CR2)* do processador. Ao final do *boot* do *kernel*, assim que o subsistema de *syscalls* termina o carregamento e antes do subsistema de módulos iniciar, o *kernel* trocar o registrador *CR0* para *write-protect*, evitando que a *CPU* consiga escrever em áreas de memória marcadas como

leitura. A alteração da permissão de escrita é realizada através da escrita do *bit* "0" na posição "16" do registrador *CR0*. Para facilitar a manipulação de registradores, o *kernel* exporta alguns símbolos: *CR0_WP*, uma palavra do processador com o 16º *bit* como "1"; *write_cr0*, função que encapsula as operações dependentes de arquitetura para escrita diretamente no registrador `CR0`; *read_cr0*, função que encapsula operações para a leitura da palavra no registrador `CR0`.

O Código 4.48 ilustra um excerto do código de inicialização do módulo responsável por implementar o *hook* da *syscall sys_uname*, incluindo a troca da permissão da tabela de *syscall*, a escrita da *syscall* de interposição, a resolução dinâmica do *set_memory_rw* e a localização da tabela de *syscall*.

```
1  int ret; unsigned long cr0;
2
3  syscall_table = (void **)find_sys_call_table();
4  if (!syscall_table){
5    printk(KERN_DEBUG "[HOOK] Trying sys_call_table symbol\n");
6    syscall_table=(void **)kallsyms_lookup_name("sys_call_table");
7    if (!syscall_table)
8    {
9      printk(KERN_DEBUG "[HOOK] Cannot find the sys_call_table address\
           n");
10     return -EINVAL;
11   }
12 }
13
14 cr0 = read_cr0();
15 write_cr0(cr0 & (~CR0_WP));
16 do_set_memory_rw = (void *)kallsyms_lookup_name("set_memory_rw");
17 do_set_memory_ro = (void *)kallsyms_lookup_name("set_memory_ro");
18
19 if (do_set_memory_rw == NULL)
20 {
21   printk(KERN_DEBUG "[HOOK] Symbol not found: 'set_memory_rw'\n");
22   return -EINVAL;
23 }
24
25 ret = do_set_memory_rw(PAGE_ALIGN((unsigned long)syscall_table),1);
26 if(ret){
27   printk(KERN_DEBUG
28   "[HOOK] Cannot set the memory to rw (%d) at addr 0x%16lx\n",
29   ret, PAGE_ALIGN((unsigned long)syscall_table));
30   return -EINVAL;
31 }else
32   printk(KERN_DEBUG "[HOOK] Syscall Table page set to rw\n");
33
34 orig_sys_uname = syscall_table[__NR_uname];
35 syscall_table[__NR_uname] = hook_sys_uname;
36 write_cr0(cr0);
37 return 0;
38 }
```

Código 4.48. Função __init do módulo de kernel que implementa o hook da syscall sys_uname.

O vetores *orig_sys* e *hook_sys_uname*, armazenam, respectivamente, os apontadores para a *syscall* originalmente presente no *kernel* e para a função de interposição. O escaneamento de endereços (linha 8) é executado nos casos em que a versão do *kernel* é superior a *4.17* e, portanto, os símbolos das *syscalls* não são exportados. Nas linhas 38 e 39, o índice `NR_uname` é um inteiro referente a posição da *syscall* a ser substituída na tabela de chamadas de sistema—neste caso *sys_uname*. Desta forma, toda vez que a *syscall* `uname` for invocada, a função do módulo *hook_sys_uname* será chamada.

A função de interposição da *syscall* deve ter o mesmo protótipo usado pela *syscall* original, além de seguir o mesmo acesso aos *buffers* do usuário feita pela *syscall*. A implementação das *syscalls* originalmente presentes no *kernel* se encontram em *"kernel_source/kernel/sys.c"*, sobre a macro *SYSCALL_DEFINEN*, onde ***N*** é o número de parâmetros da *syscall*. O Código 4.49 apresenta a função de interposição da *syscall* `uname`, onde o acesso aos *buffers* é feito respeitando o mapeamento do *kernel* entre o nível de usuário. A linha 6 passa a execução para a *syscall* original, cujo ponteiro foi salvo na linha 38 do Código 4.48, onde apenas o *buffer* `sysname` é alterado para, de forma didática, ilustrar a substituição da chamada original.

```
1  asmlinkage long hook_sys_uname(struct new_utsname __user *name){
2    char msg[5]="Hook\0";
3    struct new_utsname tmp;
4    orig_sys_uname(name);
5    if(!copy_from_user(&tmp,name,sizeof(tmp)))
6    {
7      memcpy(tmp.sysname,msg,5);
8      if(copy_to_user(name,&tmp,sizeof(tmp)))
9        printk(KERN_DEBUG "[HOOK] Can't write to user-buffer!\n");
10   }
11   else
12     printk(KERN_DEBUG "[HOOK] Can't copy user-buffer :(\n");
13   return 0;
14 }
```

Código 4.49. Função de interposição.

O Código 4.50 ilustra a substituição da *syscall* através da saída do utilitário `Driver Message` (dmesg).

```
1  uname -s; rmmod hook; dmesg; uname -s
2  [ 8001.652808] [HOOK] Symbol sys_close not found
3  [ 8001.652808] [HOOK] Trying sys_call_table symbol
4  [ 8001.656808] [HOOK] System call table at 0xffffffff8e6001e0
5  [ 8001.660797] [HOOK] Syscall Table page set to rw
6  [ 8001.660797] [HOOK] sys_uname: 0xffffffff8da93460 - hook_sys_uname:
       0xffffffffc0860000
7  [ 8003.016080] [HOOK] Inside hook_sys_uname
8  [ 8003.016080] [HOOK] Can't write to user-buffer!
9  [ 8073.156170] [HOOK] Inside hook_sys_uname
10 [ 8073.156170] [HOOK] uname->sysname: Linux
11 [ 8073.291740] [HOOK] released module
12 Linux
```

Código 4.50. Exemplo de saída do dmesg durante o carregamento do LKM implementando o hook de syscall.

É importante notar que o uso frequente de *hooking* de funções do *kernel* `Linux` por diversas soluções motivou a criação do *Linux Security Modules* [Morris 2013a] e do SystemTap [Makarov 2019] para unificar e padronizar os *hooks* em *callbacks*. Entretanto, sistemas independentes ao *kernel mainstream*, tais como o `GrSecurity` [Spengler 2019] e códigos maliciosos, ainda utilizam *hooks* explícitos pela simplicidade e compatibilidade com outras *features* do *kernel*, possibilitando um maior nas operações do *kernel*, principalmente com *hooks* no subsistema de *syscall*.

4.9. Monitoração de Tráfego de Rede

Além das interações com o sistema operacional, parte relevante dos comportamentos exibidos por exemplares maliciosos se refere a interação com outros computadores via rede, de modo que a análise do tráfego de rede entre a máquina de análise e a rede remota (e.g., Internet) pode revelar informações importantes para o entendimento do exemplar desconhecido.

Técnicas para análise do tráfego de rede são bem descritas na literatura [Combs 2012] e nosso objetivo não é apresentar um guia exaustivo delas, mas apresentar as especificidades das análises na plataforma `Linux`. Mais especificamente, apresentamos, nesta seção, abordagens para se analisar exemplares suspeitos utilizando-se do filtro de pacotes `iptables` já presente nos sistemas baseados em `Linux`. Novamente, nosso objetivo não é ser um guia exaustivo de uso desta solução, o que pode ser consultado na literatura [Purdy 2004], mas sim apresentar uma introdução ao assunto.

O filtro de pacotes `iptables` atua em três cadeias: (i) `INPUT`, para as conexões originadas externamentes e que se destinam a máquina em questão; (ii) `FORWARD`, para conexões originadas externamente e que se destinam a outras máquinas, sendo roteadas através da máquina em questão; e (iii) `OUTPUT`, para conexões originadas localmente e destinadas a máquinas externas. Para cada uma destas cadeias, pode-se definir políticas de segurança, como a permissão (`ACCEPT`) ou o bloqueio (`DROP`) das conexões, como ilustrado pelo Código 4.51.

```
1 iptables -A INPUT -j ACCEPT
2 iptables -A FORWARD -j ACCEPT
3 iptables -A OUTPUT -j DROP
```

Código 4.51. Regras básicas iptables. O tráfego de saída pode ser bloqueado de modo a evitar que infecções se propaguem a partir da sandbox.

O bloqueio das conexões de saída pode ser utilizado para a implementação de um ambiente isolado da rede, como a *sandbox* referida na Seção 4.5, de modo a permitir a execução de exemplares maliciosos na máquina local sem que estes exemplares infectem as máquinas vizinhas e propaguem suas infecções.

Além de políticas globais para toda a cadeira, as regras `iptables` podem ser definidas para portas e protocolos específicos, como exemplificado pelo Código 4.52, de modo a permitir maior flexibilidade de operação.

```
1 iptables -A INPUT -p tcp -j ACCEPT
2 iptables -A INPUT -p udp -j DROP
```

Código 4.52. Exemplos de políticas iptables. Regras podem ser definidas para cada protocolo.

Para fins de análise, a principal política de interesse é o estabelecimento de registros para as conexões, de modo que se possa investigar com quais endereços IP e em quais portas o código sob análise se comunica. O Código 4.53 ilustra o estabelecimento de políticas de *log* padrão para todas as cadeias.

```
1 iptables -A INPUT -j LOG
2 iptables -A FORWARD -j LOG
3 iptables -A OUTPUT -j LOG
```

Código 4.53. Estabelecimento de políticas de log utilizando-se iptables para fins de monitoração.

Através da política de *log*, pode-se identificar alguns comportamentos típicos de exemplares maliciosos. O Código 4.54 apresenta um excerto de *log* de um exemplar malicioso do tipó *scanner*, que escaneia toda a rede ao qual o computador está conectado, de maneira sequencial, visando a identificação de vulnerabilidades que permitam a propagação do conteúdo malicioso do *malware*.

```
1 May 13 13:21:49 lab kernel: [ 3610.320968] IN= OUT=ens3 SRC=192
    .168.122.5 DST=91.189.89.196
2 May 13 13:21:49 lab kernel: [ 3610.321356] IN= OUT=ens3 SRC=192
    .168.122.5 DST=91.189.89.197
3 May 13 13:21:49 lab kernel: [ 3610.321503] IN= OUT=ens3 SRC=192
    .168.122.5 DST=91.189.89.198
4 May 13 13:21:49 lab kernel: [ 3610.321633] IN= OUT=ens3 SRC=192
    .168.122.5 DST=91.189.89.199
```

Código 4.54. Log de rede exemplificando a atuação de um malware do tipo scanner.

4.10. Conclusão

Nesse minicurso, apresentamos os conceitos básicos da engenharia reversa de aplicações maliciosas e introduzimos o leitor as características particulares das soluções de análise para o ambiente Linux e também aos comportamentos maliciosos dos exemplares afetando esta plataforma. Acreditamos que a partir dos conhecimentos obtidos neste curso, os leitores estarão aptos a realizar procedimentos básicos de análise em artefatos desconhecidos e a projetar experimentos de análise de binário. Esperamos, assim, contribuir para o desenvolvimento das pesquisas dos leitores em análise de binários de maneira geral. Embora focado na plataforma `Linux`, acreditamos que os conhecimentos de análise aqui apresentados possam estimular o desenvolvimento dos leitores quanto a análise de

binários em múltiplas plataformas (*Unix-Like*). Por fim, recomendamos a leitura dos trabalhos referenciados para que o leitor obtenha uma compreensão mais detalhada dos conceitos apresentados neste capítulo. Em especial, recomendamos a leitura da bibliografia especializada na plataforma `Linux` [Galante et al. 2018, Cozzi et al. 2018, Damri and Vidyarthi 2016, Isohara et al. 2010, Duncan and Schreuders 2019] para uma melhor compreensão de como as soluções apresentadas podem ser empregadas nas múltiplas tarefas associadas a engenharia reversa e segurança computacional.

Agradecimentos. Os autores agradecem a: Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), em especial via Projeto FORTE - Forense Digital Tempestiva e Eficiente (Processo: 23038.007604/2014-69 - Edital 24/2014 - Programa Ciências Forenses); Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), em especial via Programa de Bolsas de Doutorado (processo número 164745/2017-3) e Programa de Bolsas de Iniciação Científica (processo número 138239/2017-7).

Referências

[Afonso et al. 2015] Afonso, V. M., de Amorim, M. F., Grégio, A. R. A., Junquera, G. B., and de Geus, P. L. (2015). Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17.

[Alves et al. 2017] Alves, P., Brobecker, J., Evans, D., and Zaretskii, E. (2017). Gdb: The gnu project debugger. `https://www.gnu.org/software/gdb/`.

[Android 2017] Android (2017). System and kernel security. `https://source.android.com/security/overview/kernel-security.html#linux-security`. Acessado em: Abril/2017.

[AppArmor 2019] AppArmor, W. (2019). Apparmor project wiki. `http://wiki.apparmor.net`.

[Beegle 2007] Beegle, L. E. (2007). Rootkits and their effects on information security. *Inf. Sys. Sec.*, 16(3):164–176.

[Botacin et al. 2018] Botacin, M. F., de Geus, P. L., and Grégio, A. R. A. (2018). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98.

[Branco et al. 2012] Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. `https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_WP.pdf`.

[C-Jump 2017] C-Jump (2017). 7. eflags individual bit flags. `http://www.c-jump.com/CIS77/ASM/Instructions/I77_0070_eflags_bits.htm`.

[Cheng et al. 2018] Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X., and Marion, J.-Y. (2018). Towards paving the way for large-scale windows malware analysis:

Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 395–411, New York, NY, USA. ACM.

[Combs 2012] Combs, G. (2012). *Wireshark Network Analysis (Second Edition): The Official Wireshark Certified Network Analyst Study Guide*. Laura Chappell University.

[Coogan et al. 2009] Coogan, K., Debray, S., Kaochar, T., and Townsend, G. (2009). Automatic static unpacking of malware binaries. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 167–176, Washington, DC, USA. IEEE Computer Society.

[Corbet et al. 2005] Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc.

[Corporation 2015] Corporation, N. D. (2015). About tomoyo linux. `http://tomoyo.osdn.jp/about.html.en`.

[Cozzi et al. 2018] Cozzi, Graziano, Fratantonio, and Balzarotti (2018). Understanding linux malware. `http://www.s3.eurecom.fr/~yanick/publications/2018_oakland_linuxmalware.pdf`.

[Damri and Vidyarthi 2016] Damri, G. and Vidyarthi, D. (2016). Automatic dynamic malware analysis techniques for linux environment. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 825–830.

[Duncan and Schreuders 2019] Duncan, R. and Schreuders, Z. C. (2019). Security implications of running windows software on a linux system using wine: a malware analysis study. *Journal of Computer Virology and Hacking Techniques*, 15(1):39–60.

[eliben 2017] eliben (2017). Parsing elf and dwarf in python. `https://github.com/eliben/pyelftools`.

[Fruhlinger 2018] Fruhlinger, J. (2018). The mirai botnet explained: How teen scammers and cctv cameras almost brought down the internet. `https://bit.ly/2Irz5e3`.

[Galante et al. 2018] Galante, L., Botacin, M., Grégio, A., and de Geus, P. L. (2018). Malicious linux binaries: A landscape. In *Anais Estendidos do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 213–222, Porto Alegre, RS, Brasil. SBC.

[Gebai and Dagenais 2018] Gebai, M. and Dagenais, M. R. (2018). Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Comput. Surv.*, 51(2):26:1–26:33.

[GNU 2018] GNU (2018). magic - linux man pages. `https://www.systutorials.com/docs/linux/man/5-magic/`.

[Grégio et al. 2012] Grégio, A. R. A., Afonso, V. M., Filho, D. S. F., de Geus, P. L., Jino, M., and dos Santos, R. D. C. (2012). Pinpointing malicious activities through network and system-level malware execution behavior. In Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A. M. A. C., Taniar, D., and Apduhan, B. O., editors, *Computational Science and Its Applications – ICCSA 2012*, pages 274–285, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Grégio et al. 2015] Grégio, A. R. A., Afonso, V. M., Filho, D. S. F., Geus, P. L. d., and Jino, M. (2015). Toward a Taxonomy of Malware Behaviors. *The Computer Journal*, 58(10):2758–2777.

[Isohara et al. 2010] Isohara, T., Takemori, K., Miyake, Y., Qu, N., and Perrig, A. (2010). Lsm-based secure system monitoring using kernel protection schemes. In *2010 International Conference on Availability, Reliability and Security*, pages 591–596.

[Jahoda et al. 2017] Jahoda, M., Krátký, R., Prpič, M., Čapek, T., Wadeley, S., Ruseva, Y., and Svoboda, M. (2017). A guide to securing red hat enterprise linux: System auditing. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing`.

[Kernel.org 2017] Kernel.org (2017). The linux man-pages projects. `https://www.kernel.org/doc/man-pages/`.

[Koret and Bachaalany 2015] Koret, J. and Bachaalany, E. (2015). *The Antivirus Hacker's Handbook*. Wiley Publishing, 1st edition.

[Makarov 2019] Makarov, S. (2019). Systemtap overview. `http://sourceware.org/systemtap`.

[Matloff and Salzman 2008] Matloff, N. and Salzman, P. J. (2008). *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Francisco, CA, USA.

[McCutchan 2005] McCutchan, J. (2005). inotify - monitoring filesystem events. `http://man7.org/linux/man-pages/man7/inotify.7.html`.

[M.F.X.J. Oberhumer 2018] M.F.X.J. Oberhumer, László Molnár, J. F. R. (2018). Upx the ultimate packer for executables. `https://upx.github.io/`.

[Morris 2013a] Morris, J. (2013a). Linux security module framework. `https://www.linux.com/learn/overview-linux-kernel-security-features`.

[Morris 2013b] Morris, J. (2013b). Selinux project wiki. `http://selinuxproject.org`.

[O'Neill 2016] O'Neill, R. E. (2016). *Learning Linux Binary Analysis*. Packt Publishing.

[Oster 2019] Oster, P. (2019). Cve-2016-5195. `https://nvd.nist.gov/vuln/detail/CVE-2016-5195`.

[Peterson and Brown 1961] Peterson, W. W. and Brown, D. T. (1961). Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235.

[Purdy 2004] Purdy, G. N. (2004). *Linux iptables Pocket Reference*. O'Reilly.

[Remillano 2018] Remillano, A. I. (2018). Cryptocurrency-mining malware targets linux systems, uses rootkit for stealth. `https://tinyurl.com/y3yyv5oo`.

[Rossow et al. 2012] Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., and Steen, M. v. (2012). Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 65–79, Washington, DC, USA. IEEE Computer Society.

[S. Weyergraf 2015] S. Weyergraf, S. B. (2015). Ht editor. `http://hte.sourceforge.net/index.html`.

[Sikorski and Honig 2012] Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

[Simmonds 2015] Simmonds, C. (2015). *Mastering Embedded Linux Programming*, chapter Linking with libraries: static and dynamic linking. Packt Publishing.

[Skoudis and Zeltser 2003] Skoudis, E. and Zeltser, L. (2003). *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Spengler 2019] Spengler, B. (2019). Grsecurity features. `https://grsecurity.net/features.php`.

[Tasiopoulos and Katsikas 2014] Tasiopoulos, V. G. and Katsikas, S. K. (2014). Bypassing antivirus detection with encryption. In *Proceedings of the 18th Panhellenic Conference on Informatics*, PCI '14, pages 16:1–16:2, New York, NY, USA. ACM.

[The Linux Kernel doc. 2017] The Linux Kernel doc. (2017). Linux system calls implementation. `https://linux-kernel-labs.github.io/master/lectures/syscalls.html`.

[Vecchia and Coral 2014] Vecchia, E. D. and Coral, L. (2014). Linux remote evidence colector – uma ferramenta de coleta de dados utilizando a metodologia live forensics. *Anais do SBSEG 2014*, pages 586–597.

[Venezla 2012] Venezla, P. (2012). A world without Linux: Where would Apache, Microsoft – even Apple be today? `http://www.infoworld.com/article/2616083/data-center/a-world-without-linux--where-would-apache--microsoft----even-apple-be-today-.html`. Acessado em: Abril/2017.

[Wang et al. 2018] Wang, D., Ming, J., Chen, T., Zhang, X., and Wang, C. (2018). Cracking iot device user account via brute-force attack to sms authentication code. In *Proceedings of the First Workshop on Radical and Experiential Security*, RESEC '18, pages 57–60, New York, NY, USA. ACM.

[Wong 2018] Wong, R. (2018). *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*. Packt.

[Yocom et al. 2004] Yocom, N., Turner, J., and Davis, K. (2004). *The Definitive Guide to Linux Network Programming (Expert's Voice)*. Apress.

[Z. Chang 2017] Z. Chang, G. Sison, J. J. (2017). Erebus resurfaces as linux ransomware. `https://tinyurl.com/y6qwxs3q`.