Como Funciona a Engenharia Reversa

A engenharia reversa é uma técnica onde basicamente nós desconstruímos softwares para sabermos como eles funcionam. Existem outros tipos de engenharia reversa, como a de hardware, por exemplo, ou até mesmo de armas, como as usadas em tempos de guerra.

Basicamente, engenharia reversa é a técnica para entender como um trecho de código funciona sem possuir seu código aplicável em diversas áreas da tecnologia como análise de malware, reimplementação de software e protocolos, correção de bugs, análise de vulnerabilidades, adição e alteração de recursos no software, proteções anti-pirataria, etc.

Quando um programa tradicional é construído, o resultado final é um arquivo executável que possui uma série de instruções em código de máquina para que o processador de determinada arquitetura possa executar. Com a ajuda de softwares específicos, profissionais com conhecimento dessa linguagem (em nosso caso, Assembly) podem entender como o programa funciona e, assim, estudá-lo ou até fazer alterações no mesmo.

Dentro do Linux, abra o terminal e verifique se os programas vim, gcc e make estão instalados.

Abra o vim no terminal mesmo e crie um programa simples em C com esse código:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   printf("Olá, Mundo!\n");

   return 0;
}
```

Compile ele digitando *make programa* (sem o .c mesmo) e aguarde. Após isso execute ele digitando *./programa*.

Programas compilados, como os feitos em C, utilizam uma função comum argumento para, por exemplo, imprimir um texto, como a printf, que é uma função de um código de uma biblioteca nativa do C.

Para verificar estas bibliotecas, digite o comando *ldd programa*.

Agora instale o programa ltrace (digitando *sudo apt-get install ltrace*), que usaremos para vermos como é a chamada de biblioteca do programa, para isso digite *ltrace ./programa*. No exemplo ele poderá mostrar a função puts, que seria o printf simples do C.

Para entendermos como funciona, o executável criado chamou a biblioteca do C, que chamou o kernel, este que realmente escreveu a mensagem na tela. Ou seja, quem escreve a mensagem realmente não é o executável nem a biblioteca, isso é função apenas do kernel e de suas syscalls (chamadas de sistema).

Para ver essas bibliotecas e todas as syscalls que ele chama no kernel, usamos *strace ./programa* (se não tiver instale ele). A syscall chamada pra escrever é a write.

O processador ele precisa entender uma linguagem específica, já que ele trabalha com entradas e saídas de dados. A linguagem é a Assembly, já que os dados quando entra, ele faz as operações e gera uma saída. Veja por exemplo, o nosso programa é compilado para um processador específico (como o 32 bits ou 64 bits). Para ver esses dados do programa, digite *file programa*.

Em outras palavras, essa diferença de arquitetura de processadores que faz com que determinados programas não rode em determinados processadores, além da diferença de sistemas operacionais, já que o Linux fala com o kernel de um modo diferente do Windows, que fala diferente do Mac e assim por diante.

Todo arquivo com sequência de bytes são chamados de binários, por exemplo, o código-fonte que criamos em C será entendido pelo compilador as instruções e transformará o código num binário executável, ou seja, que o processador e o sistema operacional entende, no nosso exemplo, para Linux de 32 bits. Se rodarmos o comando cat no programa aparecerá um monte de caracteres imcompreensível por nós.

Para montarmos os programas usamos assemblers (que é o compilador) e para desmontarmos usamos disassemblers, um exemplo de disassembler é o objdump, que podemos digitar algo como *objdump -d programa | less*.

Dessa forma, sabendo como os bits se comportam, podemos alterar o programa para que ele funcione diferente, mudando as instruções das informações para o processador. Isso dá pra fazer com vários tipos de programas, de várias plataformas.

Podemos entender por exemplo, que um arquivo de 32 bits tem 4 bytes. Isso será falado mais pra frente.

Da mesma forma, linguagens interpretadas como o Python, tem o interpretador em C (no caso do Linux, está em /usr/bin/python), que chamará a biblioteca que chamará a syscall no kernel e executará o programa.

Sistemas de Numeração

Podemos comparar o computador (ou melhor, o seu microprocessador) como uma calculadora gigante, já que a base deles é com números.

A forma que os humanos usam para contar é o sistema decimal, com dez números (0, 1, 2, 3, 4, 5, 6, 7, 8 e 9), antes da existência de algarismos e da escrita, era usada uma referência, como por exemplo, pedrinhas pra contar ovelhas.

Basicamente, os números são símbolos, e podemos usar outros símbolos para contar.

No decimal, ao chegar no 9, adicionamos o 1 ao lado do 0 (ficando um 10) e passamos a usar dois algarismos para representar os números, até chegar em 99, aí adicionamos mais um ficando 100 e assim por diante, sempre que estourar a possibilidades. Basicamente, o sistema decimal que todos conhecemos é esse.

Podemos fazer algumas dessas contagens diretamente no terminal do Linux, assim:

echo {0..9}

echo {10..99}

Existem várias bases numéricas além da decimal, dentro da computação usamos o binário, o hexadecimal e o octal, com bases em 2, 16 e 8, respectivamente.

O sistema binário, com dois algarismos (0 e 1), é usado em eletrônica por poder representar dois estados (como aberto e fechado, desligado e ligado, true e false). A contagem funciona da mesma forma, com as regras de estourar o limite com apenas esses números, indo em 0, 1, 10, 11, 100, 101, 110, 111, etc.

Outro sistema usado é o octal, usado por exemplo, em permissões de arquivos para Linux, com oito números (0, 1, 2, 3, 4, 5, 6 e 7), funcionando da mesma forma.

Podemos criar nossas próprias bases inclusive com outros símbolos, por exemplo:

M = 0
B = 1
I = 2
N = 3

Contagem a partir de 0 nesse sistema:

M, B, I, N, BM, BI, BN, IM, IB, II, IN…

Outro sistema muito usado é o hexadecimal, com 16 dígitos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F), nesse sistema são pegos as seis primeiras letras do alfabeto, mas eles são considerados números. A contagem é da mesma forma, ao terminar as possibilidades, adiciona um número à esquerda e continua.

Abriremos o Python do nosso sistema (independente de qual seja) e digitamos esses comandos:

print(10) # Retorna 10 decimal.

print(0b10) # Retorna 2 (10 é lido como binário e retorna 2 em decimal).

print(0o10) # Retorna 8 (10 é lido como octal e retorna 8 em decimal).

print(0x10) # Retorna 16 (10 é lido como hexadecimal e retorna 16 em decimal).

Treine como exemplo, as contagens de 0 a 10 em decimal convertido em binário, 0 a 20 em decimal convertido em hexadecimal, e 0 a 15 em decimal convertido para octal.

Arquivos

Na prática, qualquer sequência de bits são considerados arquivos, representados de forma binário. Um arquivo como 10101010 seria um arquivo de um byte (que tem 8 bits). Todo arquivo que tem no computador tem conteúdo binário, seja executável, texto, música, foto, etc.

Num arquivo, os bytes são armazenados um do lado do outro, sem espaço nem nada. O menor arquivo que conseguimos criar é o de um byte.

Um arquivo com 11111111 11111111 11111111 seria representado como FF FF FF em hexadecimal. Podemos ver os arquivos em bytes digitando *hd nomedoarquivo*, que mostrará em hexadecimal, podemos usar *od nomedoarquivo* para ver em octal. Ambos mostram também os dados binários.

Cada byte tem 256 possibilidades possíveis, de 0 a 255 (2^8), representado em hexadecimal de 0 a FF e em binário de 00000000 a 11111111.

Lembrando que a extensão não quer dizer nada pra definir o tipo de arquivo, não é a extensão, só serve pro sistema operacional saber com qual programa abrirá ele. O que define um tipo de arquivo é o cabeçalho, a estrutura dele (mimetype). No Linux podemos usar *file nomedoarquivo* para vermos essas informações.

Podemos pegar apenas uma parte dos bytes do arquivo, por exemplo *hd -n32 /bin/ls*, que pegará apenas os 32 primeiros bytes do arquivo especificado. Observe esse código abaixo:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  a0 6a 00 00 00 00 00 00  |..>......j......|
00000020
```

No caso acima, exibimos 32 bytes (20 em hexadecimal), e cada linha terá 16 bytes, o primeiro é o byte 0, depois o 1 até chegar no F. O ELF é o formato dos executáveis do Linux (não confunda com extensão).

Para ver os dados sobre o arquivo no Linux, digite *stat nomedoarquivo*.

Como exercício, treine vendo os bytes de arquivos e suas posições (por exemplo, 11A e 4B9) usando o comando hd.

Arquivos Binários

Como sabemos, a extensão do arquivo não quer dizer nada em alguns sistemas, e sim os tipos dele que definem isso (mimetypes). O que influencia nisso é o conteúdo dele.

Pra começar, crie um arquivo de texto qualquer usando o comando *echo Teste»arquivo.txt*. Depois dê um *hd arquivo.txt* para vermos pelo visualizador hexa os bytes do arquivo, esse é o conteúdo real desse arquivo. Para vermos os bytes do arquivo, digite *wc -c arquivo.txt*, podemos dar um *ls -l* também. Dessa forma sabemos que todos os arquivos são binários.

Só que muitas vezes, quando falamos de arquivos binários, estamos nos referindo a programas (arquivos executáveis).

Como exemplo, pegaremos novamente aquele código em C e daremos o comando *hd arquivo.c* para ver o conteúdo hexa dele. Podemos ver que ele também é identificado por bytes e também é binário, mas ele é apenas um código-fonte que o computador não entende como um programa, e sim como texto. Compile o programa com o comando *gcc arquivo.c -o arquivo -no-pie* e dê o comando *hd arquivo* nesse executável, este sim chamado de binário por ser um arquivo executável. Veremos que tem muito mais bytes que o código.

PS: Por considerarmos mesmo os arquivos binários os executáveis, que estes se localizam na pasta /bin, no Linux.

Os programas (executáveis) são basicamente divididos em duas sessões principais, a .data (dados do programa) e a .text (o código, podendo ser .code em alguns casos). Isso é visto muito em códigos em Assembly. Até mesmo nos bytes isso é separado, mesmo que não percebamos.

Dentro de um programa compilado podemos encontrar os dados utilizados para criar ele, e o código dele, daí podemos alterar esses mesmos códigos para modificarmos o programa. Vamos supor que mudamos um byte de 50 para 51, ele escreverá outra coisa. Mesmo não sendo recomendado, podemos editar por um editor de texto como o Vi mesmo.

Strings de Texto

As strings de texto de um arquivo dizem muito sobre ele, por isso é importante conhecer isso. Os textos basicamente não existem pro computador, já que o que ele entende mesmo é números binários, para tornar um texto legível pra nós tem que haver uma relação entre isso.

Por exemplo, crie um arquivo de texto digitando *echo Alguma coisa»teste.txt*, e vendo o conteúdo em bytes dele digitando *hd teste.txt*, o verdadeiro conteúdo do arquivo são esses bytes mostrados em hexa, podemos ver por exemplo, 6d representando um “m”, a frase é representada cada letra por um byte.

Ao dar um comando cat no arquivo, ele já interpretará como um texto e exibirá o conteúdo do texto. Podemos ver os caracteres e sua representação em bytes digitando *man ascii* no terminal do Linux. Podemos digitar *echo -c \\x44* para ver que isso é interpretado como um D. Esses caracteres ASCII são usados largamente em vários casos, como os caracteres alt do Windows.

Digitando *file arquivo.txt* podemos ver que o arquivo é identificado como ASCII text.

Voltando ao programa em C que fizemos, e dê o comando cat no código-fonte para vermos o conteúdo do texto, e depois dê o comando hd no código-fonte para vermos o conteúdo em bytes dele. Sabendo disso, podemos analisar um código-fonte e vermos quantas puladas de linha existem nele, procurando os bytes 0a dentro do programa, e ao lado dele é representado por um ponto, já que o ponto representa caracteres não gráficos (atenção que o ponto é exibido mas é representaod por 2e).

Dê o comando hd no programa compilado, podemos ver que dentro dele temos algumas strings mais ou menos compreensíveis no meio de outros não compreensíveis. Se alterarmos um desses bytes, o programa terá outro comportamento. Para encontrarmos todas as strings dentro de um arquivo e suas posições, digitamos *strings -t x arquivo* (ele não é 100% preciso, mas encontra a maioria delas). Por padrão, ele não mostra strings com 3 ou menos caracteres, para ver elas digitamos *strings -n 3 -t x arquivo*. Podemos ver as funções de print do C digitando *man isprint* que chega se os caracteres são imprimíveis.

Faça um programa simples em C que realize um cálculo, por exemplo:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int i = 65;
   i = i + 2;

   printf("%d\n", i);
```

```
    return 0;
}
```

Compile o programa e dê o comando hd no executável.

O valor 65 estará como um byte 65 (não o número 65 decimal, mas 65 em hexa), que é o mesmo representando de m, podemos digitar no terminal *strings -n 1 programa | grep ^m* para procurar no programa todas as strings que começam com m. Faça o mesmo, trocando o m pelo o, já que um dos o é o 67.

Faça um novo programa em C com esse código:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *fp = fopen(argv[1], "rb");
    unsigned char byte;

    while(fread(&byte, sizeof(byte), 1, fp)) {
        printf("%c ", byte);
    }

    printf("\n");
    fclose(fp);

    return 0;
}
```

Esse programa pega um arquivo como argumento e lê de byte em byte esse arquivo e imprime na tela. Compile ele e rode o programa com algum arquivo de texto, pode até ser o código-fonte dele mesmo, por exemplo *./programa programa.c*.

Como exercício, edite esse programa que só imprime os bytes de caracteres legíveis (que começa em 20 e termina em 7E). Use com uma condição if, como essa:

```
while(fread(&byte, sizeof(byte), 1, fp)) {
    if(byte >= 0x20 && byte <= 0x7e) {
        printf("%c ", byte);
    }
}
```

Dessa forma, com esse programa, conseguimos ver apenas a parte legível de, por exemplo, programas executáveis compilados.

Para exercitar, pegue apenas os caracteres maiúsculos e só os minúsculos, por exemplo.

Executável PE – Apresentação

Os executáveis no Windows seguem um formato específico, existem vários, mas o mais conhecido é o PE. Esse formato é definido pela Microsoft, e pode rodar em várias arquiteturas do Windows.

Abrindo um executável no Hex Workshop, podemos ver como o arquivo é por dentro. Basicamente um arquivo PE começa com o cabeçalho MZ do DOS.

Os dois primeiros bytes (4D e 5A) são a representação em ASCII do MZ, que é o que faz esse arquivo um executável PE. Se editarmos um desses dois bytes, o programa será alterado e não funcionará corretamente.

Abra o programa crackme, e procure a posição 3C e olhe os quatro bytes (no caso aqui é 3C até 3F), onde encontramos uma DWORD de 4 bytes, que indica outra parte que está o cabeçalho PE, se tiver por exemplo 00 01 00 00, inverta os bytes, ficando 00 00 01 00, ou seja, 100 em hexadecimal, que é 256 no decimal. Vá até a posição 100 (ou a especificada no hexa) e pegue os bytes de 100 a 103, deverá ter algo como 00 10 09 00, isso é a assinatura do programa.

Quando um programa é executado, ele lê as informações do cabeçalho, vai até o offset 3C e lê esse ponteiro, que é a posição onde está a assinatura, se alterarmos qualquer um desses bytes, o programa será alterado e não funcionará.

Agora usaremos o READPE para isso, após descompactar a pasta, abra o CMD, entre nela e digite *readpe C:\Users\Usuario\nomedoexecutavel.exe*, para vermos apenas os dados do cabeçalho MZ, digite *readpe -h dos executavel.exe*. Depois digite *readpe -h coff executavel.exe* para ver as informações sobre o cabeçalho PE offset. No coff mostra dados como a data que ele foi compilado, o tipo de máquina que ele foi compilado.

No Hex Workshop, iremos até a posição da DWORD (no caso a 100), e iremos até a posição que virá após a 0103 (ou seja, a 104 e 105). E veremos os bytes 4C 01, invertendo fica 01 4C, e o 14C significa binário para máquinas i386 (ou seja, 32x).

Podemos gerar uma saída XML do programa, digitando *readpe --format xml -h coff nomedoexecutavel»dados.xml*.

Abra o Detected It Easy, coloque o caminho do executável pra escanear, e em PE ele mostrará todos os dados igual ao readpe, olhe todas as opções dentro do PE. Ele também pode editar arquivos executáveis.

Dentro do PEStudio, podemos também ver os dados do arquivo da mesma forma, em file-header. Inclusive ele também verifica se o arquivo tem malwares.

No PE Tools, podemos abrir um executável indo em Tools e PE Editor, e em File Header temos os dados sobre o programa também.

No Detected It Easy, em PE e em Sections, podemos ver as sessões dos executáveis, como a .data e o .text. Se o PE pode ser analogado como uma cômoda, as sessões são as gavetas.

Dentro das sections, podemos clicar em flags, em edit header e em characteristics, e vemos as persmissões dos arquivos (no caso, ler e executar).

No PE, em Address of Entry Point, podemos ver o código Assembly do programa, ou seja, o programa desassemblado.

Executável PE – Seções e Endereçamento

Abrindo o programinha crackme, vemos que ele tem um desafio de login e senha para ser quebrado. Vamos abrir esse programinha no Detected It Easy.

Novamente falando do cabeçalho PE, vá nessa opção no DIE e vá em Dos Header, ali vemos que os programas PE começam com o mesmo cabeçalho do DOS (que é 5A4D). Se vermos o mesmo programa no Hex Workshop, veremos que ele começa com 4D 5A, que são os mesmos bytes, representando MZ. Lembrando que eles ficam numa ordem reversa, como podemos ver os outros bytes do programa, como o 0050, que no Hex aparece 50 00, compare todos os bytes em ambos os programas.

O último campo do programa é de 4 bytes, que são os números 00000100 que aparecem no DIE. No editor Hex, encontramos esses mesmos bytes nas posições de 3C e 3F (ou aproximada), que terá 00 01 00 00. Esse é o final do cabeçalho DOS. Do lado aparece o número que ele seria no decimal. Esse campo indica onde está a assinatura PE, então iremos até a posição 100, que terá os bytes 50 45 00 00.

No PE, vamos em NT Headers, onde está a assinatura PE. Veremos que estará em File Header, em Machine, o número 014C, veremos então no Hex que logo após a assinatura PE, tem justamente os bytes 4C 01, isso é o início do cabeçalho do arquivo (cabeçalho coff), que é a máquina que ele vai rodar. O 014c é os sistemas 32 bits (8664 seria os sistemas 64 bits). Outros campos importantes são o NumberOfSessions e o TimeDateStamp.

Quando todos os cabeçalhos do File Header acabam, ele passa para o Optional Header, ali tem outros campos como o Magic e o AddressOfEntryPoint, que é onde o binário vai começar a executar, e o ImageBase que é um padrãozinho para binários no Windows.

Em File Headers, vamos em NumberOfSessions, as sessões são como as gavetas de uma cômoda, e cada sessão é colocado um tipo de dados diferentes. Cada cabeçalho das sessões tem campos diferentes pra cada uma. O binário mesmo está da flags pra trás.

No Hex, podemos procurar o cabeçalho CODE procurando o texto dele ao lado, onde deverá estar aproximadamente nas posições entre 1F8 e 1FF, são reservados sempre 8 bytes para o nome. Logo após ele, teremos o endereço dela, representado por 00 10, que no DIE teremos o 1000 em CODE V.size. Logo após vem o DATA, da mesma forma. Todas essas sessões cada uma tem uma função. Essas sessões são definidas pelo compilador, que podem variar, mas o comum é ter o CODE (a codificação), o DATA (para dados), etc.

Quando compilamos um programa em C, por exemplo, as variáveis ficariam numa sessão de dados, o if e o printf seriam na sessão de código, já que eles gerarão o código Assembly que faz essa chamada. Assim entendemos melhor pra quê servem as sessões. Normalmente as sessões de dados são dados que inclusive podem ser escritos, podemos por exemplo alterar um ponteiro num programa em C, por exemplo.

Vamos supor esse simples programa em C:

```
int a = 1;

char s[] = "um nome qualquer";

if(a > 4) {
    printf("%s\n", s);
}
```

s[2] = 'c';

Essa alteração fará ser escrito “umcnome qualquer”.

Quando o loader carrega um binário no Windows, ele pega essas sessões e as lê, e faz um mapeamento delas em memória, ao mapear elas, ele define as flags que ela vai ter, e estas flags tem permissões que vão funcionar em memória. Na flag do CODE, por exemplo, podemos clicar em Flags com o botão direito e ir em Edit Header, e depois em Characteristics, ali podemos ver todas as permissões que essa flag tem, como as de leitura e execução. As permissões de memória todas começam com MEM. Olhando a flag de DATA, nas suas características terá as permissões de leitura e gravação (o que permite coisas como alterar um dado no programa). Temos outras sessões também como a .rsrc, que contém os arquivos internos do programa, como só de ícones, representados por bytes em hexadecimal (podemos ver isso clicando no .rsrc com o botão direito e indo em HEX, e podemos fazer um DUMP da mesma, da mesma forma). No começo do DIE, podemos ir em Resource, que mostrará todos os arquivos do programa, como ícones.

Voltando no DIE em PE, em NT Headers e Optional Header, vamos em AddressOfEntryPoint, que aponta para um endereço que está naquela sessão, veremos que ele inicia em 1000, em Sections, veremos que em V.address em CODE terá também 1000. Essa é a primeira sessão a ser executada.

No começo do DIE, o ImageBase seria o arquivo em si quando está em memória, e ele irá para o endereço 400000 (padrão dos executáveis 32 bits, mas esse é só um padrão que ele ocuparia caso nenhum outro programa o estivesse usando). A soma do EntryPoint (1000) com o ImageBase (400000), poderemos ver no Assembly dele 401000 (clicando na setinha ao lado em EntryPoint), nesse endereço teremos os bytes 6A00, e sua interpretação em Assembly.

Win32 API

No DIE, em import, podemos ver as DLLs das quais o programa depende para funcionar. Duas DLLs são bastante utilizadas no Windows, a KERNEL32.dll (presente em todos os binários PE, ela que chama as funções do kernel para executar uma ação), e a USER32.dll (que tem funções “mais a cara” do usuário, que gera padrões de janelas, etc.). Resumindo, a API do Windows é o conjunto de funções que são oferecidas por determinadas DLLs. Podemos ler mais sobre as APIs, inclusive para componentes específicos como buttons e checkbox, e as DLLs das quais eles dependem.

São graças as DLLs que os programas em Windows ficam sempre com a “mesma carinha”, diferente do Linux que não segue um padrão.

Vamos supor esse programa em C:

```
#include <windows.h>

int main() {
    MessageBox(NULL, "Testando a Windows API", "Info", 2);

    return 0;
}
```

Ele gerará uma janela com três botões (abortar, tentar, ignorar). No lugar do número podemos colocar também as constantes da biblioteca.

No programa em C, uma MessageBox é chamada diretamente, em alguma linguagem como C#, Delphi ou Java, ele chama também essa biblioteca em C.

Imports Table

Tanto no Windows quanto no Linux, existem uma coisa chamada biblioteca compartilhada (no Windows são as DLL e no Linux geralmente são o SO e o LD). Quando escrevemos, por exemplo, um programa em C, a função interna printf chama uma biblioteca que permite a impressão do conteúdo no programa.

O Windows já disponibiliza uma API que tem as funções para criar as janelas dos programas, por isso elas sempre ficam iguaizinhas.

Quando analisamos um programa (como aquele crackme) no DIE, podemos ver que na janela dele tem um botão escrito imports, onde vemos todas as DLL dele, onde cada uma delas faz uma função diferente, essas DLLs são presentes nativamente no sistema. Podemos ver em PE e Directiories, onde ficam os diretórios de dados (no PE o padrão é 16). O Export é mais usado em DLLs que exportam função para serem utilizadas, depois vem o Import, que é uma tabela onde todos os nomes de funções e DLLs estão. Se faltar uma DLL o programa não funcionará, já que antes do programa ser carregado ele procura no sistema elas.

Sabendo de quais DLLs ele depende, podemos descobrir o que o programa faz. Isso vemos no DIE, abaixo, ao clicar nas DLLs, onde vemos as funções que cada uma faz.

No Detected it Easy, podemos ir em Import e ver as DLLs das quais o programa depende.

Agora, baixe o programa. Abra o executável dentro desse programa. Podemos ver as funções do programa dentro do debug, onde estão coisas como as DLL que ele depende, etc. Dê o run no programa para isso.

PS: Atenção ao ver o debug para vermos se estamos olhando o programa executável em si ou alguma DLL que ele depende.

Ao fazer a engenharia reversa num programa, nós procuramos apenas a parte que queremos alterar, e não o programa inteiro. Por exemplo: Se o programa tem uma chamada de erro, olhamos e alteramos somente essa parte.

Podemos pesquisar o que uma função do programa faz no Google, que deve retornar, por exemplo, a de MessageBox. Podemos alterar os números do endereço para, por exemplo, mudar o MessageBox de Warning para Information, por exemplo.

Executável ELF – Apresentação

Enquanto os executáveis do Windows geralmente são PE, os executáveis em Linux e sistemas Unix em geral (BSD, Mac OS, etc.) são os ELF.

Se dermos o comando *file* num executável no Linux, podemos ver os dados desse programa. Veja por exemplo *file /bin/ls*.

Dê o comando *view /usr/include/elf.h* para vermos as definições (macros, estruturas, etc.) de um programa ELF. Isso é um arquivo header de C.

Podemos ver por exemplo, as diferenças de definição de programas de 32 e 64 bits.

No começo desse cabeçalho, vemos as definições dos bytes, assim:

```
#define EI_MAG0       0          /* File identification byte 0 index */
#define ELFMAG0       0x7f       /* Magic number byte 0 */

#define EI_MAG1       1          /* File identification byte 1 index */
#define ELFMAG1       'E'        /* Magic number byte 1 */

#define EI_MAG2       2          /* File identification byte 2 index */
#define ELFMAG2       'L'        /* Magic number byte 2 */

#define EI_MAG3       3          /* File identification byte 3 index */
#define ELFMAG3       'F'        /* Magic number byte 3 */

/* Conglomeration of the identification bytes, for easy testing as a word.  */
#define ELFMAG        "\177ELF"
#define SELFMAG       4
```

Veja que as definições ELF estão lá, é isso que todo executável de sistemas Unix deverá ter. o *\177ELF* são os primeiros 4 bytes do programa.

Dê o comando *hd -n32 /bin/ls* para ver que os primeiros 4 bytes do programa ls são representados exatamente por um ponto e ELF (cuja representação em hexadecimal é 45 4c 46).

Agora observe essa linha do mesmo cabeçalho anterior:

```
#define EI_CLASS      4          /* File class byte index */
#define ELFCLASSNONE  0          /* Invalid class */
#define ELFCLASS32    1          /* 32-bit objects */
#define ELFCLASS64    2          /* 64-bit objects */
#define ELFCLASSNUM   3
```

Isso acima identifica a classificação do arquivo, cujo 0 é inválido, 1 é 32 bits e 2 é 64 bits. Um pouco mais abaixo podemos ver onde esses números estarão:

```
#define EI_DATA       5          /* Data encoding byte index */
#define ELFDATANONE   0          /* Invalid data encoding */
#define ELFDATA2LSB   1          /* 2's complement, little endian */
#define ELFDATA2MSB   2          /* 2's complement, big endian */
#define ELFDATANUM    3
```

Onde está comentado como complement, está o 2. Dando o comando hd no mesmo arquivo novamente, veremos que após os bytes referentes ao ELF, está justamente 02, o que identifica o programa como sendo de 64 bits.

Nessa linha, podemos ver as várias versões de Unix que o programa rodaria:

```
#define EI_OSABI      7          /* OS ABI identification */
#define ELFOSABI_NONE         0      /* UNIX System V ABI */
```

```
#define ELFOSABI_SYSV          0       /* Alias.  */
#define ELFOSABI_HPUX          1       /* HP-UX */
#define ELFOSABI_NETBSD        2       /* NetBSD.  */
#define ELFOSABI_GNU           3       /* Object uses GNU ELF extensions.  */
#define ELFOSABI_LINUX         ELFOSABI_GNU /* Compatibility alias.  */
#define ELFOSABI_SOLARIS       6       /* Sun Solaris.  */
#define ELFOSABI_AIX           7       /* IBM AIX.  */
#define ELFOSABI_IRIX          8       /* SGI Irix.  */
#define ELFOSABI_FREEBSD       9       /* FreeBSD.  */
#define ELFOSABI_TRU64         10      /* Compaq TRU64 UNIX.  */
#define ELFOSABI_MODESTO       11      /* Novell Modesto.  */
#define ELFOSABI_OPENBSD       12      /* OpenBSD.  */
#define ELFOSABI_ARM_AEABI     64      /* ARM EABI */
#define ELFOSABI_ARM           97      /* ARM */
#define ELFOSABI_STANDALONE    255     /* Standalone (embedded) application */
```

Observe essas estruturas:

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf64_Half    e_type;                 /* Object file type */
  Elf64_Half    e_machine;              /* Architecture */
  Elf64_Word    e_version;              /* Object file version */
  Elf64_Addr    e_entry;                /* Entry point virtual address */
  Elf64_Off     e_phoff;                /* Program header table file offset */
  Elf64_Off     e_shoff;                /* Section header table file offset */
  Elf64_Word    e_flags;                /* Processor-specific flags */
  Elf64_Half    e_ehsize;               /* ELF header size in bytes */
  Elf64_Half    e_phentsize;            /* Program header table entry size */
  Elf64_Half    e_phnum;                /* Program header table entry count */
  Elf64_Half    e_shentsize;            /* Section header table entry size */
  Elf64_Half    e_shnum;                /* Section header table entry count */
  Elf64_Half    e_shstrndx;             /* Section header string table index */
```

} Elf64_Ehdr;

O tamanho de e_type é 2 bytes, e aqui embaixo temos os dados referentes à ele:

/ Legal values for e_type (object file type). */*

```
#define ET_NONE     0          /* No file type */
#define ET_REL      1          /* Relocatable file */
#define ET_EXEC     2          /* Executable file */
#define ET_DYN      3          /* Shared object file */
#define ET_CORE     4          /* Core file */
#define ET_NUM      5          /* Number of defined types */
#define ET_LOOS     0xfe00     /* OS-specific range start */
#define ET_HIOS     0xfeff     /* OS-specific range end */
#define ET_LOPROC   0xff00     /* Processor-specific range start */
#define ET_HIPROC   0xffff     /* Processor-specific range end */
```

Na engenharia reversa, lidaremos mais com os programas que tem os bytes 2 (executáveis) e 3 (bibliotecas compartilhadas).

Quando damos o comando hd no arquivo /bin/ls, vemos algo do tipo:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  a0 6a 00 00 00 00 00 00  |..>......j......|
00000020
```

Cada linha (que seria como um array) tem 16 bytes, contados em hexadecimal (a posição 10, 20, etc. são na base 16). Observe que o primeiro byte da segunda linha (00000010), é 03, que identifica que o arquivo é uma biblioteca compartilhada, esse número poderia ser 02 que é o executável>

Em algumas versões do Debian e derivados (Ubuntu, Kali, Mint, etc.), para que o sistema tire proveito da randonização da memória, os programas do sistema tem o PIE ativado, por isso eles são biblioteca compartilhada, e nesses sistemas o compilador gcc/g++ gera por padrão esse citado, sendo necessário colocar -no-pie para gerar um executável. Para vermos isso, digite *gcc -v* e procure a linha –enable-default-pie.

Dando o comando file no programa, podemos ver se ele é executável ou biblioteca compartilhada, ou vendo pelo comando hd o byte especificado.

Nós vimos como se utiliza o readpe no Windows, ele já é instalado por padrão no Linux, então podemos utilizar ele também para analisar programas Windows dentro do Linux. Ele também tem instalado o readelf, que permite analisarmos executáveis ELF no Linux, digitando, por exemplo, *readelf -h programa*. Podemos ver mais dados do programa digitando *readelf -a programa | less*, como por exemplo os cabeçalhos em Assembly, e as flags. Podemos digitar também *strings* para vermos as strings dele, como os cabeçalhos em Assembly. Podemos ver os detalhes do programa digitando *ldd programa*.

Dando um hd no programa, podemos ver por exemplo, o Entry Point (endereço de ponto de entrada) dele, vamos supor que seja essa a saída dele:

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  f0 10 40 00 00 00 00 00  |..>.......@.....|
```

00000020

O Entry Point é os bytes de 18 até 1B, de trás pra frente, no caso acima foi 004010f0, podemos ver se é isso mesmo digitando *readelf -h programa* novamente e ver o endereço de ponto de entrada, que é 0x4010f0.

Execute o comando *hte programa* para vermos os bytes dele no terminal, podemos ver os cabeçalhos e outras opções nele, como para que arquitetura o programa foi feito, entre outras coisas. Por esse programa podemos mudar o executável, como no byte da posição 10, que mudando para 03 ele mudará para biblioteca compartilhada (mas não será mais executável de forma alguma). Para usar as opções como salvar e editar, use as teclas F (F1, F2, F3, F4, etc.).

Com isso podemos ver que, se um programa for corrompido, é porque alguém alterou algo no programa que fez ele parar de funcionar.

Executáveis ELF – Símbolos, PLT e GOT

Copie o repositório Git digitando *git clone https://github.com/mentebinaria/engrev.git*. Depois entre no diretório engrev e crc32, e compile o programa para 32 bits digitando *gcc -m32 *c -no-pie -o keygenme*.

Digite no terminal *hte keygenme*, onde podemos ver os bytes e dados em geral do programa. Falamos dos headers anteriormente (no Windows costumamos chamar de segmentos, no Linux como sessões). Digite F6 para vermos as sessões do programa.

O segmento do tipo phdr é onde está o programa header (entry 0). Os mais importantes no momento é os segmentos do tipo load, que serão carregados (mapeados) na memória, e dentro deles.

Podemos ver as sessões do Assembler do programa digitando F6 e section header.

A sessão PLT é como se fosse o import table do PE, que é onde ficam as chamadas para funções externas (de bibliotecas).

Vá na sessão .symtab, e depois na .dynsym (no F6, lá embaixo aparecerão), lá está as chamadas externas de funções, além de objetos e coisas de outros tipos. Algumas funções, como o printf e o puts, reconhecemos do C.

Podemos olhar os códigos-fonte do programa em C. São os mesmos na sessão .symtab.

Muitas das funções na .symtab são usadas apenas para fins de debug, e são locais (mas tem alguns externos), enquanto na .dymsyn só teremos as funções externas. Podemos digitar no terminal *gdb -q ./keygenme* e digitar *break main* para colocar um breakpoint na função main do programa.

Se dermos no terminal o comando *strip -s keygenme*, ele suprimará a função main, e não irá aparecer mais a sessão .symtab do program. Nesse caso, podemos usar o comando do gdb de novo e tentar dar um break no main, que dará erro por não estar mais no programa.

Compile o programa novamente, pois corrompemos o mesmo.

Volte aos sections headers e vá no .plt, que é onde estão os endereços das funções externas. Uma biblioteca dinâmica é carregada no sistema ao executar o programa, o printf, por exemplo, tá na

libc. Digite o comando *nm keygen | less* para vermos os símbolos do binário, onde veremos todas as bibliotecas das quais pertencem cada função.

Digite *gdb -q ./keygenme* e dentro do programa, digite *dissamble main*, onde vemos o código Assembler do main. Faça o mesmo com printf e com o endereço do mesmo.

Entre no hte novamente, clique F6 e vá em elf/image, onde está o programa dessassemblado. Digitando f8 podemos ver as funções em Assembler, incluindo o main. A sessão .got.plt é onde estará o endereço real da função printf.

Linux Syscalls

Syscalls no Linux são os equivalentes as API do Windows, mas funcionam de forma parecida.

Vamos usar o programa ltrace para nossas análises. Vamos supor esse programa em C:

```
#include <stdio.h>

int main() {
   printf("O número é: %d\n", 255);

   return 0;
}
```

Dê um make no programa pra compilar ele.

Quando um programa em C é executado, ele chama a função printf, que é nativa do C, presente em uma determinada biblioteca (podemos dar um ldd no programa para ver elas), mas quem exibirá a mensagem mesmo é o kernel. Dê um *ltrace ./hello* para ver quais funções foram executadas.

PS: O número 17 que aparecerá é o número de caracteres que o printf retorna. Para ver os retornos digite *man 3 printf*.

Dê o comando *ltrace ls* para vermos as funções que o programa ls chama. Dê o comando *ltrace -S ./hello* para ver as syscalls que as funções em C chamam. Um exemplo é a *SYS_write()*, syscall chamada pelo printf.

Resumindo, as syscalls são funções nativas do kernel.

Vamos refazer o programa em C, chamadn a função write ao invés do printf:

```
#include <unistd.h>

int main() {
   char *msg = "Oi\n";

   write(STDOUT_FILENO, msg, 3);

   return 0;
}
```

Mesmo usando outra função (write()), ele chama a mesma syscall SYS_write(), até porque a linguagem permanece sendo a mesma (C).

PS: Da mesma forma que acontece no Windows, ao usar uma linguagem como a Java (em qualquer sistema, Windows, Linux, Mac, BSD, etc.), por exemplo, a System.out.println() chama a printf do C que consequentemente chama a mesma syscall SYS_write.