# Relatório Prática 2 – Disassembly

Reinaldo Kaminski Neto e Matheus Berbel Barusso

## I. Introdução

A partir do conteúdo aprendido em sala de aula, no contexto da programação em Assembly [1], o presente laboratório tem como objetivo colocar em prática os conhecimentos adquiridos para investigar o funcionamento interno de um executável, utilizando a técnica de *disassembly*. Por meio desse processo, conseguimos compreender tanto o fluxo geral do programa quanto o seu comportamento em nível mais baixo, analisando detalhes como o uso dos registradores, a manipulação da memória e outras interações com a arquitetura da máquina. Essa prática é essencial para a análise de risco e segurança, pois permite enxergar o que ocorre *under the hood* do programa, tornando evidente o que antes era opaco a partir apenas do código-fonte ou da execução superficial [2].

Para a execução desta atividade, foi selecionado o executável número nove, e a ferramenta utilizada para o disassembly foi o *Interactive Disassembler* (IDA), em sua versão gratuita. Embora o IDA tenha sido a escolha para este trabalho, outras ferramentas igualmente competentes poderiam ter sido utilizadas, como o Ghidra, da NSA, que também oferece recursos avançados para a engenharia reversa.

.

## II. Materiais e Métodos

O programa analisado segue o seguinte fluxo: A função `WinMain` é o ponto de entrada e imprime algumas mensagens informativas na tela, utilizando chamadas para a função `printf`. Depois dessas mensagens iniciais, ele exibe o prompt “Informe a Senha:” e usa a função `scanf` para capturar a entrada do usuário em um buffer.

Durante a análise do programa, foi identificado que a função `algcifra` implementa uma cifra de César[3], deslocando cada caractere digitado pelo usuário em $+3$ no valor ASCII antes de compará-lo com a senha armazenada. Essa operação é feita por meio da instrução:

```
add bl, 3
```

Além disso, foi realizada a inspeção do hexdump gerado pelo Disassembler, que revelou a string cifrada usada para a verificação. Como mostra o trecho a seguir:

```
004030A0  63 65 6E 74 72 61 6C 3B  20 44 65 73 63 6F 62 72  central;·Descobr
004030B0  69 72 20 61 20 73 65 6E  68 61 2E 0D 0A 00 46 6F  ir·a·senha....Fo
004030C0  72 6D 61 74 6F 20 64 6F  20 45 78 65 63 75 74 61  rmato·do·Executa
004030D0  76 65 6C 3A 20 50 45 20  28 57 69 6E 64 6F 77 73  vel:·PE·(Windows
004030E0  29 0D 0A 00 49 6E 66 6F  72 6D 65 20 61 20 53 65  )...Informe·a·Se
004030F0  6E 68 61 3A 0D 0A 00 79  72 71 71 68 78 70 64 71  nha:...yrqqhxpdq
00403100  71 00 00 00 00 00 00 00  00 00 25 73 00 00 00 00  q.........%s....
```

Fig. 1. Hexadump do executável

Nessa região, foi possível identificar claramente a sequência `yrqqhxpdqq`, que corresponde à senha cifrada.

Sabendo que a comparação é feita entre essa string e a senha digitada após a cifra, basta subtrair 3 do valor ASCII de cada caractere para recuperar a senha correta:

| Caractere cifrado | ASCII | -3 | Novo caractere |
|---|---|---|---|
| y | 121 | 118 | v |
| r | 114 | 111 | o |
| q | 113 | 110 | n |
| q | 113 | 110 | n |
| h | 104 | 101 | e |
| x | 120 | 117 | u |
| p | 112 | 109 | m |
| d | 100 | 97 | a |
| q | 113 | 110 | n |
| q | 113 | 110 | n |

TABLE I
Decodificação da senha "`YRQQHXPDQQ`"

Assim, a senha correta que precisa ser digitada pelo usuário para que o programa aceite a entrada é "`vonneumann`". Conforme requisitado pelo enunciado, um programa ”equivalente” foi escrito em C para funcionar de maneira similar ao executável fornecido:

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

#define MAX 100

int main()
{
    printf("Organizacao e Arquitetura de ...
        Computadores\n");
    printf("Trabalho pratico parte 2 - ...
        Disassembly\n");
    printf("Analizar e Escrever ...
        algoritmo/fluxograma correspondente ...
        ao algoritmo central; Descobrir a ...
        senha.\n");
    printf("Formato do Executavel: PE ...
        (Windows)\n");
    printf("Informe a Senha: ");

    int i, j;
    char senha_correta[] = "vonneumann";

    char senha[MAX];
    char comp[MAX];
    int ascii[MAX];

    scanf("%s", senha);
    for(i = 0; i < strlen(senha); i++) {
        ascii[i] = (senha[i] - 3) - 0;
        comp[i] = (char)ascii[i];
    }

    if (strcmp(senha_correta, comp) == 0) {
```

```
        printf("CERTA RESPOSTA!!!!! ...
            PARABENS!");
        return 0;
    }
    else {
        printf("Resposta Errada! Tente ...
            novamente... ");
    }
    return 0;
}
```

Listing 1. Programa equivalente em C

## III. Resultados e Discussão

A partir das análises do código e da realização do disassembly, foi possível compreender o fluxo do programa e o ponto em que a verificação da senha ocorre. Após o estudo do trecho , descobriu-se que a senha correta foi devidamente encontrada e decifrada com sucesso. Entretanto, se o objetivo do usuário fosse ”burlar” o programa para aceitar qualquer senha, foi realizado um patch para que isso ocorra.

No executável analisado, o desvio entre a senha correta e incorreta é feito utilizando a instrução:

```
jns short ERRQ1
```

A instrução `jns` (*Jump if Not Sign*), em assembly x86, realiza o salto condicional caso o bit de sinal do registrador esteja zerado, ou seja, quando o resultado da operação anterior não for negativo. Dessa forma, o programa seguia para o label `ERRQ1` apenas quando o sinal da operação fosse positivo. Por outro lado, a instrução `jz` (*Jump if Zero*) realiza o salto apenas quando o resultado da operação anterior for zero, independentemente do sinal.

Assim, ao alterar a instrução `jns` para `jz`, passamos a desviar o programa para o fluxo correto sempre que a comparação resultar em zero — eliminando a verificação real da senha. Dessa forma, qualquer senha inserida pelo usuário será considerada válida.

A modificação foi simples e direta: no *hexaview* do binário, localizamos o par de bytes que representa a instrução `jns short ERRQ1`, que é `75 0D`. Então, basta editar esse valor para `74 0D`, que corresponde à instrução `jz short ERRQ1`. Após essa modificação, o binário foi salvo e executado novamente. Como esperado, a tela de acerto foi exibida independentemente da senha digitada.

Como resultado, conseguimos controlar o fluxo da aplicação e garantir que a autenticação, ou a falta dela, fosse sempre bem-sucedida.

## IV. Conclusão

Com o proposto pelo projeto alcançado, pudemos utilizar o conhecimento aprendido em sala de aula para entendermos melhor o processo de análise de um executável por meio do processo de Disassembly. Nessa análise foram aplicados conceitos aprendidos de forma teórica, como a lógica do funcionamento da linguagem assembly para a compreensão do que acontece no código visualizado no software IDA, e conceitos aprendidos de maneira prática em laboratório, conceitos como a possibilidade de análise do HEX Dump

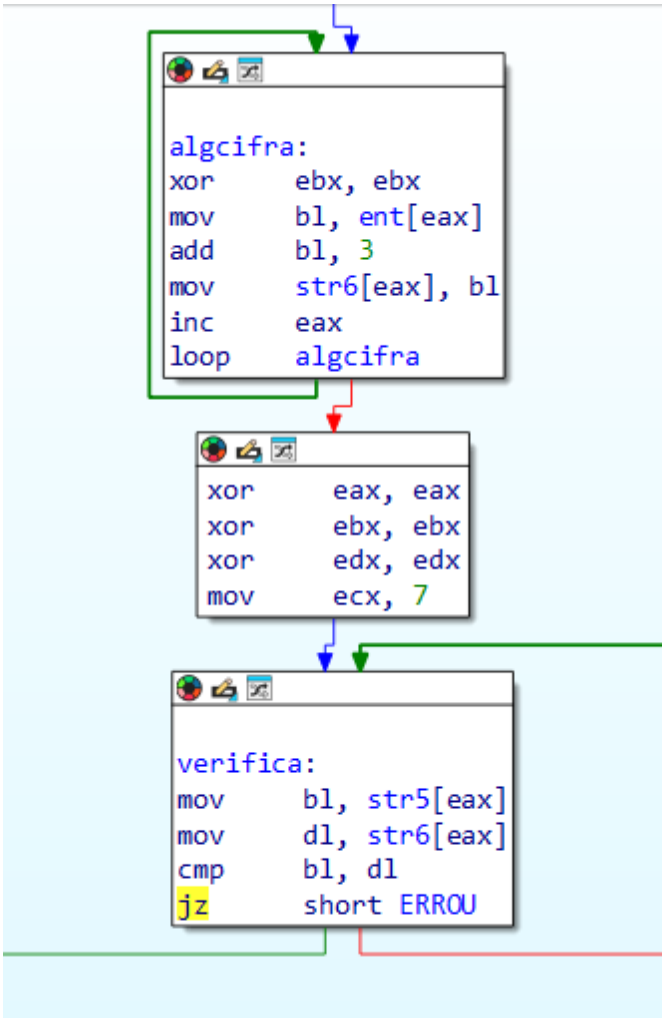


Fig. 2. Substituição de `jns` por `jz` no binário


Fig. 3. Tela de sucesso exibida após o patch

do executável e a possibilidade de realizarmos um patch no programa, fazendo alterações das instruções do código assembly por meio da alteração dos valores de Bytes. Com isso, confirmamos a importância e as possibilidades do uso do método de Desassembly para entender o funcionamento de um executável, permitindo inclusive a análise para fins de cyber-segurança, a fim de evitar o uso de softwares com Vírus e Malwares embutidos nele. Nesse trabalho pudemos também aplicar a realização de um Patch, para burlarmos o funcionamento original de um programa.

## REFERENCES

[1] A. A. Giron. (2025) Prática 3 – interactive disassembler (ida). Prática realizada em Laboratório. [Online]. Available: https://moodle.utfpr.edu.br/pluginfile.php/3074017/mod_resource/content/0/pratica3_IDA.pdf

[2] R. Pannain, F. H. Behrens, and D. Piva Júnior, *Organização básica de computadores e linguagem de montagem*, 1st ed. Rio de Janeiro: Elsevier, 2012.

[3] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1996, seção 2.1 “Substitution Ciphers”, incluindo Cifra de César.