

Engenharia Reversa do ENCRI 3.30

Escrito por Marcos Velasco em 29/07/2026 – www.velasco.com.br

Do executável DOS protegido à reconstrução do código-fonte

Procedimento técnico, detalhado e reproduzível

Escopo legítimo: O procedimento deve ser aplicado apenas a arquivos próprios, programas para os quais exista autorização, preservação histórica, interoperabilidade, auditoria ou pesquisa permitida. O documento descreve análise estática e reconstrução técnica; não substitui a verificação jurídica aplicável ao software analisado.

1. Visão geral do processo

O trabalho foi dividido em duas grandes fases. A primeira teve como objetivo descobrir e reproduzir o mecanismo do ENCRI 3.30. A segunda teve como objetivo analisar o programa real depois da remoção da proteção.

Fase	Entrada	Objetivo	Saída
1. Preservação	Arquivo .COM original	Garantir integridade e repetibilidade	Cópia original + hashes
2. Primeira desmontagem	Arquivo protegido	Encontrar o stub e o algoritmo ENCRI	Mapa de offsets e pseudocódigo
3. Implementação	Algoritmo documentado	Criar o Encri33Decryptor	Executável C# de linha de comando
4. Remoção	Arquivo protegido	Retirar todas as camadas	Arquivo .COM limpo
5. Segunda desmontagem	Arquivo limpo	Analisar o programa verdadeiro	Assembly organizado e funções identificadas
6. Reconstrução	Assembly comentado	Gerar representação em alto nível	Código-fonte funcionalmente equivalente

2. Preparação do ambiente de análise

A análise deve ser realizada em um ambiente separado do computador principal. O arquivo original nunca deve ser alterado diretamente.

2.1 Ferramentas recomendadas

- Editor hexadecimal: HxD, ImHex ou 010 Editor.
- Disassembler de 16 bits: Ghidra, IDA, radare2, NDISASM ou ferramenta equivalente.
- Ambiente de execução: DOSBox-X, 86Box, PCem ou máquina virtual com DOS/FreeDOS.
- Depurador opcional: DEBUG do DOS, debugger integrado ao DOSBox-X ou Turbo Debugger.
- Compilador: Visual Studio ou compilador C# compatível com C# 7.3.

- Ferramenta de hash: PowerShell, CertUtil ou utilitário SHA-256.

2.2 Estrutura de pastas

```
C:\ENCRI_ANALISE\  
  00_ORIGINAL\  
    MONOCITO.COM  
  01_TRABALHO\  
    MONOCITO_TRABALHO.COM  
  02_DISASSEMBLY_PROTEGIDO\  
  03_ENCRI33DECRYPTOR\  
  04_DESCRIPTOR_GRAFADO\  
  05_DISASSEMBLY_LIMPO\  
  06_CODIGO_RECONSTRUIDO\  
  07_DOCUMENTACAO\
```

Essa separação evita misturar a amostra original, os resultados intermediários e os arquivos produzidos durante a reconstrução.

3. Preservação e identificação do arquivo original

3.1 Criar a cópia de trabalho

- Copie o arquivo recebido para a pasta 00_ORIGINAL.
- Marque a cópia original como somente leitura.
- Crie uma segunda cópia na pasta 01_TRABALHO.
- Faça toda a desmontagem e todos os testes sobre a cópia de trabalho.

3.2 Calcular hashes

No PowerShell:

```
Get-FileHash .\MONOCITO.COM -Algorithm SHA256  
Get-FileHash .\MONOCITO.COM -Algorithm MD5
```

Com o CertUtil:

```
certutil -hashfile MONOCITO.COM SHA256  
certutil -hashfile MONOCITO.COM MD5
```

Registre o nome, o tamanho e o hash. Depois da análise, esses valores permitem provar que o arquivo original permaneceu inalterado.

4. Compreender o formato DOS .COM antes de desmontar

Um arquivo .COM é uma imagem binária simples. O DOS cria o PSP no início do segmento e carrega o arquivo no offset 0100h. A CPU começa a executar em CS:0100h.

Offset no arquivo	Endereço em memória	Uso no ENCRI
0000h	0100h	Início do stub
0004h	0104h	Assinatura textual ENCRI
0100h	0200h	Quantidade de parágrafos

0102h	0202h	Valores iniciais de DL e DH
0250h	0350h	Início do bloco criptografado

Regra essencial: Para converter um endereço observado em memória para um offset do arquivo .COM, subtraia 0100h. Exemplo: memória 0350h corresponde ao offset 0250h do arquivo.

5. Inspeção hexadecimal inicial

5.1 Abrir o arquivo sem executá-lo

5. Abra MONOCITO_TRABALHO.COM no editor hexadecimal.
6. Examine os primeiros 640 bytes.
7. Procure textos ASCII legíveis.
8. Procure repetições ou blocos com alta aparência de entropia.
9. Anote os offsets de textos, palavras de 16 bits e possíveis inícios de código.

5.2 Localizar a assinatura

No formato analisado, o texto ENCRI aparece a partir do offset 0004h. Essa assinatura ajuda a reconhecer uma camada, mas não deve ser a única validação, porque um arquivo comum poderia conter a mesma palavra por coincidência.

```
Offset 0004h: 45 4E 43 52 49
ASCII:      E  N  C  R  I
```

5.3 Registrar os offsets candidatos

- 0004h: assinatura textual.
- 0100h: palavra little-endian usada como quantidade de parágrafos.
- 0102h: palavra usada para inicializar DL e DH.
- 0250h: início do fluxo criptografado.
- 0270h: tamanho mínimo necessário para conter as estruturas conhecidas.

6. Primeira desmontagem: analisar o arquivo protegido

6.1 Configurar corretamente o disassembler

- Arquitetura: Intel 8086/x86 de 16 bits.
- Modo: Real Mode.
- Formato: raw binary ou DOS COM.
- Base/endereço de carregamento: 0100h.
- Ponto de entrada: 0100h.

Uma base incorreta faz com que todos os endereços e referências sejam exibidos deslocados, dificultando a comparação com os offsets do arquivo.

6.2 Objetivo da primeira desmontagem

Nesta etapa ainda não se procura entender o programa MONOCITO. O objetivo é entender apenas o carregador ENCRI. O código real está criptografado e, portanto, será interpretado como instruções sem sentido se for desmontado diretamente.

6.3 Identificar o laço de descriptografia

A rotina relevante foi reduzida à sequência lógica abaixo:

```
AL = [SI]
AH = AL
AL = AL - DH - DL - 0Dh
DH = AL
DL = AH
[DI] = AL
SI = SI + 1
DI = DI + 1
CX = CX - 1
Repetir enquanto CX != 0
```

Interpretação detalhada:

Operação	Significado
AL = [SI]	Lê o byte criptografado apontado por SI.
AH = AL	Guarda uma cópia do byte criptografado, pois AL será modificado.
AL = AL - DH - DL - 0Dh	Remove do byte atual os dois valores do estado e a constante 13 decimal.
DH = AL	O byte descriptografado passa a participar da descriptografia do próximo byte.
DL = AH	O byte criptografado original também passa a participar do próximo cálculo.
[DI] = AL	Grava o byte descriptografado no destino.
LOOP	Decrementa CX e repete enquanto ainda houver bytes.

7. Converter o Assembly em algoritmo independente

7.1 Descobrir o tamanho do bloco

A palavra no offset 0100h é interpretada em little-endian e representa a quantidade de parágrafos. Cada parágrafo possui 16 bytes.

```
paragraphCount = ReadUInt16LE(data, 0x100)
decryptedLength = paragraphCount * 16
```

No código C#, a multiplicação por 16 foi escrita como deslocamento de quatro bits:

```
int decryptedLength = paragraphCount << 4;
```

7.2 Ler a chave inicial

```
byte dl = data[0x102];
byte dh = data[0x103];
```

A ordem é importante: no formato little-endian, o primeiro byte da word corresponde a DL e o segundo corresponde a DH.

7.3 Reproduzir a aritmética de 8 bits

```
int plain = encrypted - dh - d1 - 0x0D;  
byte decrypted = (byte)(plain & 0xFF);
```

O & 0xFF reproduz o estouro natural de um registrador de 8 bits. Se o resultado matemático for negativo, somente os oito bits inferiores permanecem.

7.4 Atualizar o estado

```
dh = decrypted;  
d1 = encrypted;
```

Essa atualização torna o algoritmo encadeado: um erro em um byte altera todos os bytes seguintes.

8. Criar o projeto Encr33Decryptor

8.1 Criar o projeto

10. Abra o Visual Studio.
11. Crie um projeto Aplicativo de Console em C#.
12. Use o nome Encr33Decryptor.
13. Selecione uma versão compatível com C# 7.3, caso essa seja a exigência do ambiente.
14. Substitua o Program.cs pelo código do descriptografador.
15. Compile inicialmente em Debug para facilitar testes; ao final, compile em Release.

8.2 Definir os offsets como constantes

```
private const int EncrHeaderTextOffset = 0x04;  
private const int LengthWordOffset = 0x100;  
private const int KeyWordOffset = 0x102;  
private const int EncryptedSourceOffset = 0x250;  
private const int MinimumEncrSize = 0x270;
```

Usar constantes nomeadas evita números mágicos espalhados pelo programa e facilita a comparação com as anotações do disassembly.

8.3 Implementar a leitura little-endian

```
private static int ReadUInt16LE(byte[] data, int offset)  
{  
    return data[offset] | (data[offset + 1] << 8);  
}
```

8.4 Implementar a comparação ASCII

```
private static bool AsciiEquals(byte[] data, int offset, string text)  
{  
    if (offset < 0 || data.Length < offset + text.Length)  
        return false;  
  
    for (int i = 0; i < text.Length; i++)  
    {
```

```

        if (data[offset + i] != (byte)text[i])
            return false;
    }

    return true;
}

```

Essa função verifica a assinatura sem depender de conversões de encoding ou criação de strings temporárias.

9. Validar uma camada antes de descriptografar

A função `IsEncr133Layer` deve rejeitar dados incompletos ou incompatíveis. Isso evita tentar ler posições inexistentes e também reduz falsos positivos.

16. Confirmar que o vetor não é nulo.
17. Confirmar que o tamanho é pelo menos 0270h bytes.
18. Confirmar que ENCRI existe no offset 0004h.
19. Ler a quantidade de parágrafos no offset 0100h.
20. Converter a quantidade para bytes.
21. Rejeitar tamanho zero ou negativo.
22. Confirmar que 0250h + tamanho não ultrapassa o final do arquivo.

```

private static bool IsEncr133Layer(byte[] data)
{
    if (data == null || data.Length < MinimumEncr1Size)
        return false;

    if (!AsciiEquals(data, Encr1HeaderTextOffset, "ENCRI"))
        return false;

    int paragraphCount = ReadUInt16LE(data, LengthWordOffset);
    int decryptedLength = paragraphCount << 4;

    if (decryptedLength <= 0)
        return false;

    if (EncryptedSourceOffset + decryptedLength > data.Length)
        return false;

    return true;
}

```

10. Implementar a remoção de uma camada

A função `DecryptOneLayer` recebe a camada completa e retorna somente o bloco descriptografado, sem copiar o stub de 0250h bytes.

```

private static byte[] DecryptOneLayer(byte[] data)
{
    int paragraphCount = ReadUInt16LE(data, LengthWordOffset);
    int decryptedLength = paragraphCount << 4;

    if (EncryptedSourceOffset + decryptedLength > data.Length)
        throw new InvalidDataException("Camada ENCRI inválida ou incompleta.");
}

```

```

byte dl = data[KeywordOffset];
byte dh = data[KeywordOffset + 1];
byte[] output = new byte[decryptedLength];

for (int i = 0; i < decryptedLength; i++)
{
    byte encrypted = data[EncryptedSourceOffset + i];
    int plain = encrypted - dh - dl - 0x0D;
    byte decrypted = (byte)(plain & 0xFF);
    dh = decrypted;
    dl = encrypted;
    output[i] = decrypted;
}

return output;
}

```

10.1 Conferência manual de um byte

Para validar a implementação, escolha um byte do bloco e reproduza manualmente a conta usando os valores correntes de DH e DL. Compare o resultado com o byte produzido pelo programa. Faça isso para os primeiros bytes, pois qualquer divergência logo no início indica offset, ordem da chave ou aritmética incorretos.

11. Detectar e remover várias camadas

O resultado da primeira camada pode ser outra camada ENCRI completa. Por isso o programa não deve executar a função apenas uma vez.

```

int layers = 0;

while (IsEncri33Layer(data))
{
    layers++;
    Console.WriteLine("Descriptografando camada " + layers + "...");
    data = DecryptOneLayer(data);
}

```

A cada repetição, o vetor data deixa de representar a camada anterior e passa a representar o conteúdo recém-descriptografado. No MONOCITO analisado, o processo continuou por 26 camadas.

Critério de parada: O laço termina quando o resultado já não possui uma estrutura ENCRI válida. Isso não depende de um número fixo de camadas e permite tratar outros arquivos gerados pelo mesmo protetor.

12. Compilar e usar o Encri33Decryptor

12.1 Compilar

23. Escolha a configuração Release.
24. Compile o projeto.
25. Localize Encri33Decryptor.exe na pasta bin\Release ou equivalente.
26. Copie o executável para a pasta de trabalho ou utilize o caminho completo.

12.2 Executar pela linha de comando

Encr133Decryptor.exe MONOCITO_TRABALHO.COM

Saída esperada:

```
ENCRI 3.30 Decryptor - linha de comando
-----
Descriptografando camada 1...
Descriptografando camada 2...
...
Descriptografando camada 26...

Concluído.
Camadas removidas: 26
Arquivo gerado: MONOCITO_TRABALHO_descriptografado.COM
Tamanho final: ... bytes
```

12.3 Interpretar os códigos de retorno

Código	Significado
0	Processo concluído.
1	Quantidade incorreta de argumentos.
2	Arquivo de entrada não encontrado.
3	Nenhuma camada ENCRI reconhecida.
10	Exceção durante leitura, processamento ou gravação.

13. Validar o arquivo descriptografado

13.1 Verificação estrutural

- O arquivo final deve ser menor que o protegido, porque os stubs foram descartados.
- A assinatura ENCRI não deve mais aparecer no início.
- O primeiro bloco deve formar instruções 8086 coerentes.
- Strings e tabelas do programa podem tornar-se visíveis.
- O tamanho final deve corresponder ao comprimento da última camada.

13.2 Verificação comportamental

27. Execute o arquivo original protegido em ambiente DOS isolado e registre seu comportamento.
28. Execute o arquivo descriptografado no mesmo ambiente.
29. Compare telas, entradas, saídas, arquivos acessados e encerramento.
30. Se houver diferença, volte à validação dos offsets e do algoritmo.

13.3 Não confundir descriptografia com reconstrução

Nesse ponto foi recuperado o executável real, mas ainda não existe código-fonte. O resultado continua sendo linguagem de máquina. A etapa seguinte é desmontar e interpretar o programa limpo.

14. Segunda desmontagem: analisar o programa real

14.1 Criar um novo projeto de análise

Não reutilize automaticamente todas as marcações feitas no projeto do arquivo protegido. Crie uma análise separada para o arquivo descriptografado, novamente com base 0100h e modo 16 bits.

14.2 Localizar a inicialização

- Primeiro JMP ou sequência de entrada.
- Configuração de DS e ES.
- Configuração de SS e SP, se existente.
- Inicialização de variáveis globais.
- Primeira chamada de função ou interrupção DOS.

14.3 Identificar chamadas ao DOS

AH	INT 21h	Interpretação típica
01h	Leitura de caractere	Entrada pelo teclado
02h	Escrita de caractere	Saída no console
09h	Exibir string terminada por \$	Mensagens do programa
0Ah	Entrada em buffer	Leitura de linha
3Dh	Abrir arquivo	Acesso a arquivo
3Eh	Fechar arquivo	Finalização do acesso
3Fh	Ler arquivo	Entrada de dados
40h	Gravar arquivo/dispositivo	Saída de dados
4Ch	Encerrar processo	Fim do programa

15. Transformar o disassembly em funções compreensíveis

15.1 Criar limites de funções

O disassembler pode não reconhecer corretamente todas as funções. Observe instruções CALL, destinos compartilhados, prólogos, epílogos e RET. Cada rotina identificada deve receber inicialmente um nome funcional provisório, por exemplo:

```
sub_1234 -> ExibirMenu
sub_14A0 -> LerOpcaoTeclado
sub_18F2 -> AbrirArquivoDados
sub_1B10 -> ProcessarRegistro
```

15.2 Criar nomes semânticos

Os nomes originais foram perdidos na compilação. Portanto, os novos nomes são hipóteses documentadas. Prefira nomes que descrevam o efeito observado, não nomes que afirmem algo ainda não comprovado.

15.3 Reconstruir estruturas de controle

```
Assembly:
    CMP AX, 0
    JE fim
    ...
    JMP inicio
```

```

fim:

Reconstrução:
    while (ax != 0)
    {
        ...
    }

```

Salto condicional podem representar if, while, do-while ou partes de um switch. O contexto e as referências de entrada precisam ser analisados antes de escolher a estrutura de alto nível.

16. Reconstruir dados, strings e tabelas

31. Marque como string as sequências ASCII realmente referenciadas pelo código.
32. Identifique strings terminadas por \$, zero ou comprimento explícito.
33. Localize tabelas acessadas por índice.
34. Diferencie bytes de dados de bytes que o disassembler interpretou incorretamente como código.
35. Documente o endereço, tamanho, formato e rotinas que usam cada estrutura.

Em executáveis .COM, código e dados compartilham o mesmo segmento. Por isso é comum o disassembler confundir tabelas com instruções ou instruções com dados.

17. Gerar pseudocódigo antes do código-fonte

A reconstrução deve ocorrer em camadas. Primeiro, produza pseudocódigo que preserve o comportamento, mesmo que ainda use variáveis genéricas.

```

funcao ProcessarEntrada():
    opcao = LerTecla()
    se opcao == '1':
        ExecutarOperacao1()
    senao se opcao == '2':
        ExecutarOperacao2()
    senao:
        ExibirErro()

```

Somente depois que o fluxo estiver validado, converta para C, C# ou outra linguagem escolhida.

18. Gerar código-fonte funcionalmente equivalente

18.1 Escolher a linguagem

C costuma representar mais diretamente ponteiros, segmentos e buffers. C# facilita organização, segurança de memória e manutenção, mas exige abstrações para comportamentos de baixo nível.

18.2 Preservar o comportamento antes de melhorar o projeto

- Primeiro reproduza entradas e saídas.
- Depois reproduza cálculos e formatos de dados.
- Só então reorganize classes, métodos e nomes.
- Não introduza otimizações enquanto o comportamento ainda não estiver validado.

18.3 Manter rastreabilidade

Cada função reconstruída deve indicar de qual intervalo de endereços veio. Exemplo:

```
// Reconstruído a partir de CS:14A0h até CS:151Ch
static int LerOpcaoTeclado()
{
    ...
}
```

Esse comentário permite voltar rapidamente ao Assembly quando surgir dúvida sobre uma tradução.

19. Limitações técnicas

- Não é possível recuperar comentários originais.
- Nomes originais de variáveis e funções normalmente não existem no binário.
- Tipos de dados precisam ser inferidos pelo uso.
- Otimizações do compilador podem fundir ou reorganizar funções.
- Macros, includes e organização dos arquivos-fonte são perdidos.
- O código reconstruído pode ser funcionalmente equivalente sem ser textualmente semelhante ao original.

20. Checklist completo

- Hash do original registrado.
- Cópia original preservada.
- Arquivo .COM configurado com base 0100h.
- Assinatura ENCRI confirmada.
- Offsets 0100h, 0102h e 0250h validados.
- Laço Assembly convertido para pseudocódigo.
- Aritmética de 8 bits reproduzida.
- Leitura little-endian validada.
- Encri33Decryptor compilado.
- Todas as camadas removidas.
- Arquivo final executado em ambiente isolado.
- Segunda desmontagem criada separadamente.
- Funções e dados identificados.
- Pseudocódigo produzido.
- Código-fonte reconstruído com rastreabilidade.
- Comportamento comparado com o executável original.

22. Conclusão

O processo completo não consiste simplesmente em abrir o arquivo em um disassembler. Primeiro foi necessário reconhecer que o início do executável era um stub do ENCRI 3.30. Depois, a rotina de descritografia foi estudada em Assembly, convertida em algoritmo independente e implementada no Encri33Decryptor. O descritografador removeu cada camada sucessivamente, produzindo o executável real. Somente então uma segunda desmontagem passou a revelar a lógica verdadeira do programa. A reconstrução do código-fonte foi realizada por identificação de

funções, dados, chamadas ao DOS, estruturas de controle e testes de equivalência. O resultado final é uma implementação tecnicamente documentada e funcionalmente equivalente, não uma recuperação literal do código-fonte original.