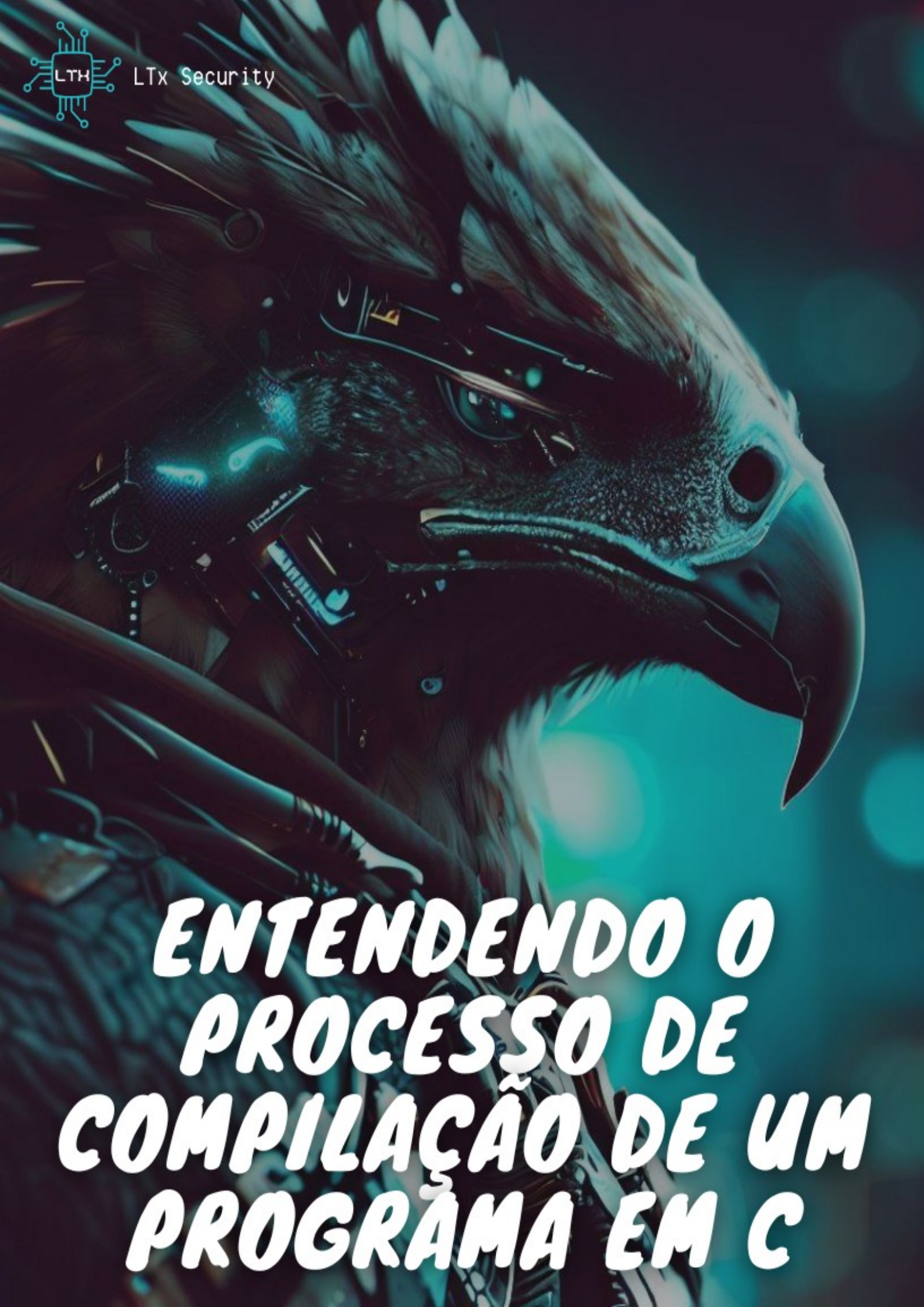
LTx Security
ENTENDENDO O PROCESSO DE COMPILAÇÃO DE UM PROGRAMA EM C

Processo de compilação

Ao produzir um executável a partir de um arquivo de código-fonte C, o processo de compilação na verdade passa por quatro estágios separados e cada um gera um novo arquivo:

- Pré-processamento - O pré-processador substitui todas as diretivas de pré-processamento no arquivo “.c” do código-fonte original pelo código da biblioteca que implementa essas diretivas. Por exemplo, a parte do código onde se encontram as diretivas #include <lib> será substituída pelo código da biblioteca que se deseja incluir. O arquivo gerado contendo as substituições têm formato de texto e geralmente possui uma extensão “.i”.

- Tradução - O compilador traduz as instruções de alto nível do arquivo “.i” em instruções de linguagem Assembly de baixo nível. O arquivo gerado contendo a tradução tem formato de texto e normalmente tem uma extensão “.s”.

- Assembling - O assembler converte as instruções de texto da linguagem Assembly criada no arquivo “.s” em código de máquina. O arquivo de objeto gerado contendo a conversão tem formato binário e geralmente tem uma extensão “.o”.

- Vinculação (Linking) - O linker combina um ou mais arquivos de objeto binário “.o” em um único arquivo executável. O arquivo gerado tem formato binário e geralmente tem uma extensão de arquivo “.exe”.

Estritamente falando, “compilação” descreve os três primeiros passos citados acima, que utilizam um único arquivo de código-fonte C e geram um único arquivo de objeto binário. Se em algum lugar do código-fonte do programa for encontrado erros de sintaxe, como um ponto e vírgula ausente ou um parêntese ausente, eles serão relatados pelo compilador e a compilação falhará.

O vinculador (linker), por outro lado, pode utilizar vários arquivos de objeto e gerar um único arquivo executável. Isso permite a criação de programas grandes a partir de arquivos de objetos modulares que podem conter funções reutilizáveis. Se o linker encontrar uma função com o mesmo nome, definida em mais de um arquivo de objeto, ele relatará um erro e o arquivo executável não será criado.

Normalmente, os arquivos temporários criados durante os estágios intermediários do processo de compilação são excluídos automaticamente, mas eles podem ser salvos incluindo a opção -save-temps no comando do compilador.

Vamos ver tudo isso acontecendo na prática.

Na prática

Para esse exemplo criei um código bem simples que somente mostra a mensagem “Hello World!” na tela.

```c
#include <stdio.h>

int main(){

    printf("Hello World!\n");

    return 0;

}
```

Então compilei o programa usando a opção -save-temps e como pode ser visto na imagem abaixo, todos os 4 arquivos foram criados.

```
C:\Users\teodo\C>gcc hello.c -save-temps -o hello.exe

C:\Users\teodo\C>dir
 O volume na unidade C é OS
 O Número de Série do Volume é 12A1-51B8

 Pasta de C:\Users\teodo\C

21/06/2023  12:01    <DIR>          .
21/06/2023  12:00    <DIR>          ..
21/06/2023  12:00                92 hello.c
21/06/2023  12:01           135.733 hello.exe
21/06/2023  12:01            79.109 hello.i
21/06/2023  12:01               882 hello.o
21/06/2023  12:01               565 hello.s
               5 arquivo(s)        216.381 bytes
               2 pasta(s)    3.421.704.192 bytes disponíveis
```

O primeiro arquivo criado foi o “hello.i”, nele é possível ver que as diretivas de pré-processamento foram substituídas. Então, no local do #include <stdio.h> foi inserido todo o código presente na biblioteca stdio.

```
  {
    return _vsnwprintf_s_l(_DstBuf, _DstSize, _MaxCount, _Format, ((void *)0), _ArgList);
  }
  static __attribute__ ((__unused__)) __inline__ __attribute__((__cdecl__)) int __attribute__((__cdecl__)) _snwprintf_s_l(wchar_t *_DstBuf, size_t _DstSize, size_t
ocale, ...)
  {
    __builtin_va_list _ArgList;
    int _Ret;
    __builtin_va_start(_ArgList, _Locale);
    _Ret = _vsnwprintf_s_l(_DstBuf, _DstSize, _MaxCount, _Format, _Locale, _ArgList);
    __builtin_va_end(_ArgList);
    return _Ret;
  }
  static __attribute__ ((__unused__)) __inline__ __attribute__((__cdecl__)) int __attribute__((__cdecl__)) _snwprintf_s(wchar_t *_DstBuf, size_t _DstSize, size_t _M
  {
    __builtin_va_list _ArgList;
    int _Ret;
    __builtin_va_start(_ArgList, _Format);
    _Ret = _vsnwprintf_s_l(_DstBuf, _DstSize, _MaxCount, _Format, ((void *)0), _ArgList);
    __builtin_va_end(_ArgList);
    return _Ret;
  }
# 820 "C:/Ruby32-x64/msys64/ucrt64/include/sec_api/stdio_s.h" 3




  __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__)) _wfopen_s(FILE **_File,const wchar_t *_Filename,const wchar_t *_Mode);
  __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__)) _wfreopen_s(FILE **_File,const wchar_t *_Filename,const wchar_t *_Mode,FILE *_OldFile);

  __attribute__ ((__dllimport__)) errno_t __attribute__((__cdecl__)) _wtmpnam_s(wchar_t *_DstBuf,size_t _SizeInWords);

# 870 "C:/Ruby32-x64/msys64/ucrt64/include/sec_api/stdio_s.h" 3
  __attribute__ ((__dllimport__)) size_t __attribute__((__cdecl__)) _fread_nolock_s(void *_DstBuf,size_t _DstSize,size_t _ElementSize,size_t _Count,FILE *_File);
# 1639 "C:/Ruby32-x64/msys64/ucrt64/include/stdio.h" 2 3
# 2 "hello.c" 2


# 3 "hello.c"
int main(){

    printf("Hello World!\n");

    return 0;

}
```

Agora que as substituições foram feitas, o próximo passo é realizar a tradução desse código para a linguagem assembly, então a partir do arquivo “hello.i” a tradução foi feita e o resultado foi salvo no arquivo “hello.s”.

```
C:\Users\teodo\C>type hello.s
        .file   "hello.c"
        .text
        .def    __main; .scl    2;      .type   32;     .endef
        .section .rdata,"dr"
.LC0:
        .ascii "Hello World!\0"
        .text
        .globl  main
        .def    main;   .scl    2;      .type   32;     .endef
        .seh_proc       main
main:
        pushq   %rbp
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        subq    $32, %rsp
        .seh_stackalloc 32
        .seh_endprologue
        call    __main
        leaq    .LC0(%rip), %rax
        movq    %rax, %rcx
        call    puts
        movl    $0, %eax
        addq    $32, %rsp
        popq    %rbp
        ret
        .seh_endproc
        .ident  "GCC: (Rev10, Built by MSYS2 project) 12.2.0"
        .def    puts;   .scl    2;      .type   32;     .endef
```

Com as instruções em assembly criadas, é a hora do assembler converter essas instruções para a linguagem de máquina, e o resultado é salvo no arquivo “hello.o”. Esse arquivo possui o formato binário, então, não é possível visualizar o seu conteúdo.

```
C:\Users\teodo\C>type hello.o
då-0¶♦.text0,0┤0♥ P`.data@P└.bssÇP└.rdata►\0@P@.xdata
l0@0@.pdata
x0Ê0♥@0@/40ä0@P@UHëÕHâý ÞHì♣Hë└Þ0Hâ─ ]┤ÉÉÉÉÉÉÉÉÉHello World!♥2♦♥0P'GCC: (Rev10, Built by MSYS2 project) 12.2.0  ↕♦►
♦↑‼♦♦♥♦♦♥
.xdata♦♥00hello.cmain0 00.text0♥0'♥.data0♥0.bss♥♥0.rdata♦♥0
.pdata♦♥0
♥♥0,__main 0puts 0
```

Agora que o arquivo objeto foi criado, basta o linker combinar o arquivo “hello.o” em um objeto executável, nesse exemplo o nome escolhido para esse executável foi “hello.exe”.

Após todo esse processo, temos como resultado um programa funcional.

```
C:\Users\teodo\C>hello.exe
Hello World!
```

LTx Security

Segurança Ofensiva