第3章 逆向工程分析

本章进入系统安全领域的学习，围绕逆向工程相关技术，介绍软件安全的基本知识，并通过实践案例的方式阐述逆向工程分析的方法。

本章学习目标：

- 学会逆向工程关键技术。
- 运用 IDA、Ghidra、GDB 等工具进行逆向分析。
- 学会针对对称密码和非对称密码等算法的逆向技术。
- 熟悉二进制代码保护与混淆、符号执行、约束求解、软件加固脱壳技术。

通过本章的介绍和实践，希望可以帮助读者快速熟悉软件安全逆向工程，在逆向分析中还原软件结构，及时发现其中的安全漏洞，提升软件安全意识和系统安全实践技能。

3.1 逆向工程基础

逆向工程(Reverse Engineering)也叫反向工程，通过逆向分析程序获取或猜测其相关的实现代码。绝大多数应用程序作为公司的商业机密，其源代码并不会随意公开。如果希望获取这些程序的原始设计思路或代码实现逻辑，唯一可行的方案就是逆向分析。总体上，软件逆向分析主要是指对软件的结构、程序设计的流程、程序设计的加密算法以及相关功能实现代码进行逆向分析与拆解。

逆向分析技术包含静态分析与动态分析。在深入学习逆向工程之前，先介绍逆向分析所需的主要工具。静态分析工具主要有 Ghidra 和 IDA Pro，动态分析工具主要有 GDB 和 x96dbg。

3.1.1 逆向分析工具 Ghidra

Ghidra 是由美国国家安全局(National Security Agency，NSA)研究部门开发的软件逆向工程套件，支持对各种系统平台(包括 Windows、macOS 和 Linux)代码的分析，具有反汇编、汇编、反编译等功能。

Ghidra 安装可遵循官网的指导步骤。安装完成后，进入 GhidraInstallDir 目录，运行 GhidraRun.bat(Windows)或 GhidraRun(Linux 或 macOS)，即可在 GUI 模式下启动 Ghidra。Ghidra 按项目进行管理，使用者需要先创建一个项目，之后就可以使用 Import File 功能导入需要反编译的文件。Ghidra 在加载完反编译文件后，会显示该文件的基础信息，例如架

构、大小、MD5 值等。由于 Ghidra 基于 Java 开发，会花较长的时间分析，效率低于 C/C++ 编写的 IDA。

下面通过演示某大赛真题来阐述 Ghidra 功能，目标程序为一个 JPG 文件。使用二进制文件编辑工具 010Editor 或 WinHex 打开该文件，图 3-1 显示文件下方区域含有未知填充块。根据文件结构可知，PK 是压缩包的文件头。因此，可以用 binwalk 进行分离。binwalk 是一款快速且易用的、用于逆向工程分析和提取固件映像的工具。

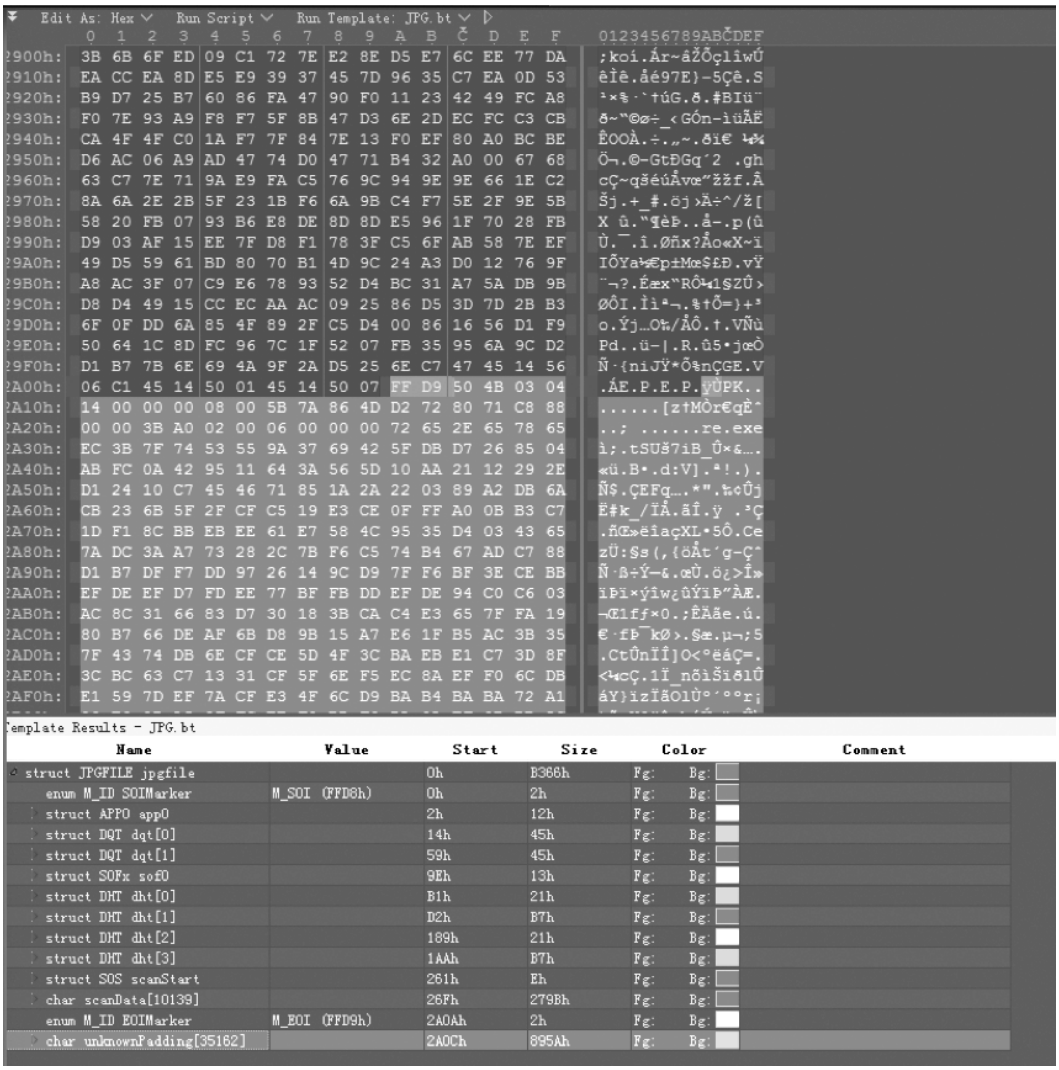

图 3-1　010Editor 查看程序结构

读者可自行下载 binwalk 安装文件。

binwalk 遵循标准的 Python 安装过程，即运行 setup.py 文件。安装完成后，运行命令分离文件，结果如图 3-2 所示。

```
binwalk -Me RE_Cirno.jpg
```

解压分离后的文件夹，得到可执行文件 re.exe。在命令行中运行 re.exe，查看提示信

图 3-2　文件分离结果

息。根据图 3-3 显示的运行结果，可以推测需要对获得的字符串进行反转，然后由栅栏密码对反转后的字符串进行解密，其中参数为 9，即每组字符数设置为 9。

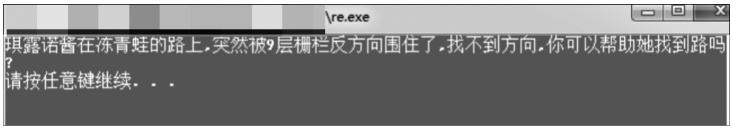

图 3-3　命令行运行目标文件

接着在 Ghidra 新建项目，将 re.exe 导入其中进行分析，如图 3-4 所示。

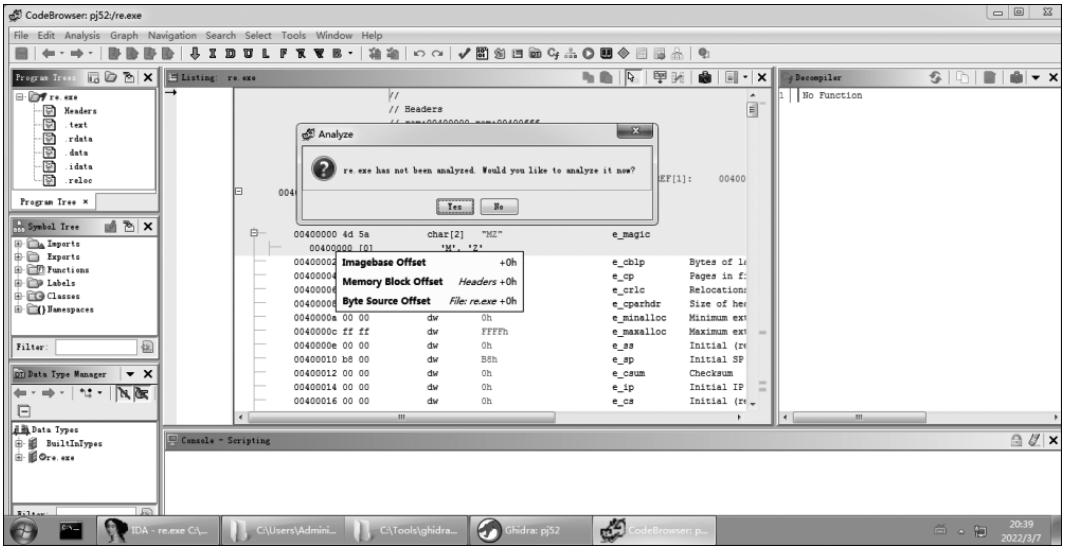

图 3-4　在 Ghidra 中分析目标文件

由于运行程序有按任意键继续的提示，猜测程序使用了 system("pause")函数。因此可以对 Ghidra 反编译的文件进行字符串搜索。在 Ghidra 上方的选项栏中找到 Search 按钮，单击 For Strings 按钮，在选项中查找 pause，验证是否调用 system("pause")函数，如图 3-5 所示。

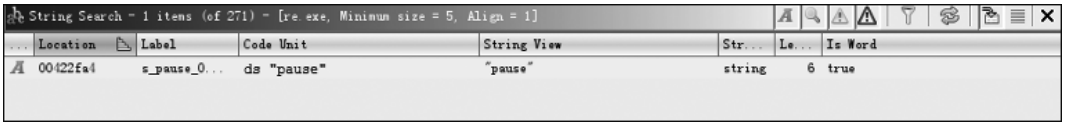

图 3-5　在 Ghidra 中找到目标文件关键字符串

通过单击右边的交叉引用功能，即 XREF 区域，找到调用该功能的关键函数，如图 3-6 和图 3-7 所示。

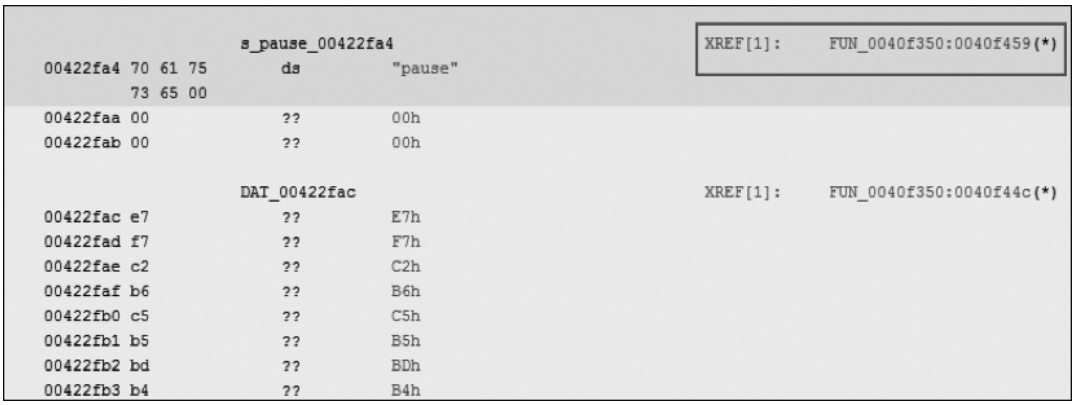

图 3-6　在 Ghidra 中找到关键函数

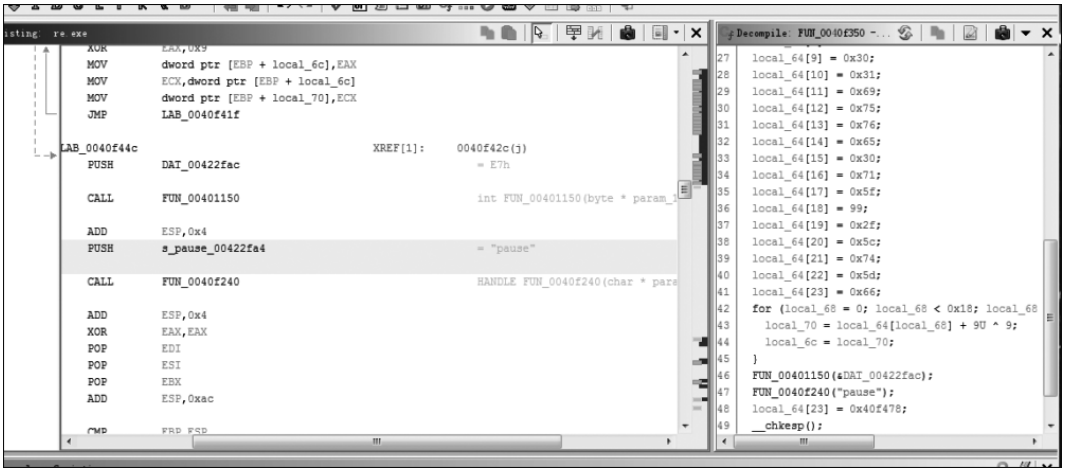

图 3-7　在 Ghidra 中找到关键函数及对应伪代码

继续单击 Windows 按钮，在子菜单中单击 Function Graph，可以看到该函数的图形化显示界面，如图 3-8 所示。

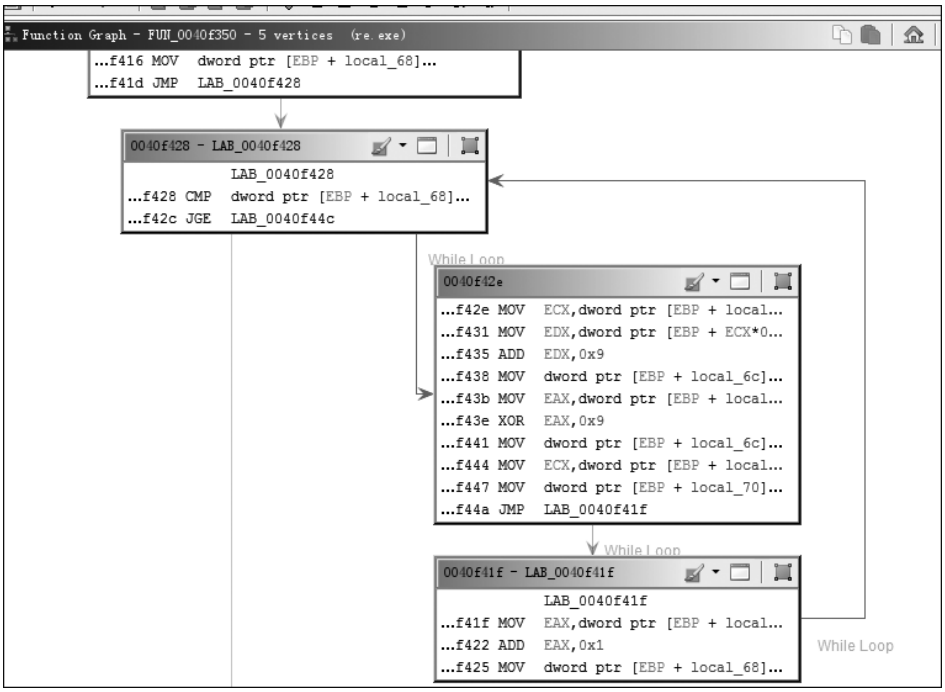

图 3-8　Ghidra 生成的关键函数调用逻辑关系

在右侧的反编译窗口，将关键的代码段复制下来，如下所示。

```
uint local_70;
uint local_6c;
int local_68;
int local_64[24];
local_64[0] = 0x73;
local_64[1] = 0x5e;
local_64[2] = 0x61;
local_64[3] = 0x72;
local_64[4] = 0x67;
local_64[5] = 0x2f;
local_64[6] = 0x6b;
local_64[7] = 0x72;
local_64[8] = 0x41;
local_64[9] = 0x30;
local_64[10] = 0x31;
local_64[11] = 0x69;
local_64[12] = 0x75;
local_64[13] = 0x76;
local_64[14] = 0x65;
local_64[15] = 0x30;
local_64[16] = 0x71;
local_64[17] = 0x5f;
local_64[18] = 99;
local_64[19] = 0x2f;
local_64[20] = 0x5c;
local_64[21] = 0x74;
local_64[22] = 0x5d;
local_64[23] = 0x66;
local_68 = 0;
```

```
while (local_68 < 0x18) {
  local_70 = local_64[local_68] + 9U ^ 9;
  local_68 = local_68 + 1;
  local_6c = local_70;
}
```

其伪代码逻辑清楚，只需要简单修改代码，重新编译运行即可还原该程序算法，还原代码脚本如下所示。

```
a=[115,94,97,114,103,47,107,114,65,48,49,105,117,118,101,48,113,95,99,47,92,116,93,102]
res=''
for i in range(len(a)):
  res+=chr((a[i]+9)^9)
print res
reversed_res = res[::-1]
print reversed_res
```

运行脚本，两次打印分别得到以下字符串：

```
uncry1}rC03{wvg0sae1ltof
fotl1eas0gvw{30Cr}1yrcnu
```

最后，读者可以在网上搜索一个在线网站来求解栅栏密码(关键词可以是：栅栏密码加密解密)，设置每组字符数为 9，得到最终结果 flag{C1rno1sv3rycute0w0}。

3.1.2 静态分析工具 IDA Pro

IDA(Interactive Disassembler)是一款交互式反汇编工具，官方网站提供的 IDA 安装包已经扩展到了多个操作系统平台，包括 Windows、macOS、Linux。基于不同的授权模式，IDA 提供专业版和免费版两种不同版本。免费版本仅包含基本的处理器加载模块和 Windows 系统下常见的可执行文件分析模块。专业版(Pro 版本)包含了所有平台的处理器信息，支持几乎所有的二进制格式，并且支持 Java、.NET、MIPS、ARM、DLL 等文件格式。

IDA 提供了强大的交互功能，用户可以通过编写自己的加载器和脚本来指导 IDA 分析未知格式文件。如果用户有未知硬件平台的处理器指令集信息，还可以编写基于特定处理器的文件分析模块。本节继续使用上一节的案例演示 IDA 的使用方法。

首先介绍 IDA 对主界面各个区域的功能，如图 3-9 所示。

(1) 工具栏。

工具栏(Toolbar)包含了文件分析最常用的一些工具，通过菜单中的“View→Toolbar”可以添加或者删除工具栏的按钮，通过拖拽功能可以将按钮放置到自己喜欢的位置。

(2) 导航带。

导航带(Overview navigator)以线性方式显示了当前加载文件的地址信息。默认情况下导航带会覆盖整个地址空间。在导航区域中单击鼠标右键，可以进行放大或缩小，以便进行代码区域定位。通过拖动导航带两侧的箭头，可以快速更换反汇编窗口显示其他地址空间的代码。窗口左侧显示了不同颜色对应的数据类型，可以快速得知当前光标所在地址的数据类型。IDA 菜单中“Options→Colors”的 Navigation bar 选项提供了导航带默认数据类型的颜色设置功能。

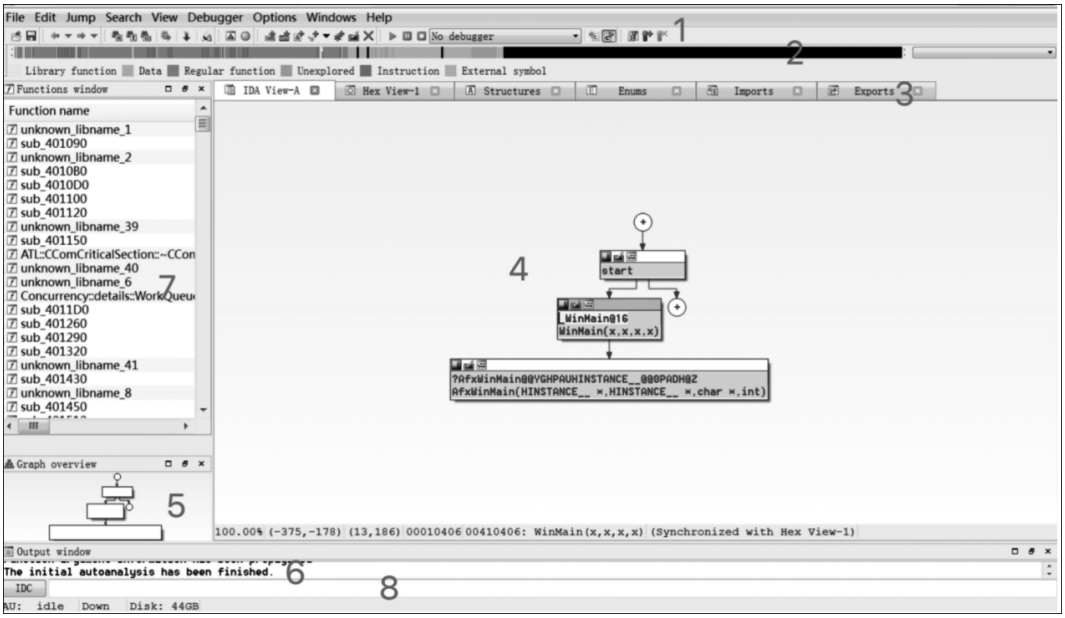

图 3-9　IDA 的主要窗口

(3) 标签栏。

标签栏(Tabs)提供了当前已经打开的子窗口标签。通过单击子窗口标题,可以在多个视图中快速切换。图 3-9 显示的子窗口包括 IDA View-A(反汇编窗口,当打开多个反汇编窗口,会依次按字母序号命名,例如 IDA View-B、IDA View-C)、Hex View-A(十六进制窗口,同 IDA view 窗口一样,当打开多个窗口会依次按字母序号命名)、Structures(结构体窗口)、Enums(枚举窗口)、Imports(输入表窗口)、Exports(输出表窗口)。如果需要其他参考窗口信息可以通过菜单“View→Open Subviews”功能打开新的参考窗口。

(4) 反汇编窗口。

反汇编窗口(Disassembly View)是主要的数据展示窗口,提供了两种不同的显示风格——图形视图和文字视图。通过图形视图可以快速分析程序的流程以及函数对于程序流程的影响。当两种视图激活之后,可以通过空格键在两种视图中快速切换。

(5) 图形全局视图。

当图形视图激活时,窗口仅显示了部分图形,这时 IDA 就会激活图形全局视图(Graph overview),通过在该窗口单击并拖拽鼠标平移设计图面,可以在 IDA View-A 中快速定位代码,有助于用户加深对程序整体流程的认识。激活文字视图时,该窗口将自动隐藏。

(6) 消息窗口。

消息窗口(Message Window)又称日志窗口,用于显示 IDA 在分析文件过程中执行的一些操作,或者显示 IDA 在分析过程汇总时出现的错误信息。如果运行 IDC 脚本,脚本的日志输出同样会在该窗口中显示。

(7) 函数窗口

如果安装了 Hex-Rays 插件,函数窗口(Functions Window)将显示当前已经识别的或者插件认为可能是函数的一些数据,包括函数所在的区段、地址等信息。如果没有安装插

件，这个区域显示名称(Names) 和字符串窗口(Strings)。

(8) 命令窗口。

命令窗口(IDC)用于执行简单的命令或者命令序列，以及显示命令执行结果和执行错误信息。

下面演示 IDA 逆向分析操作流程。将 re.exe 拖入 IDA 中，加载完成后，单击左侧函数窗口中的_main 函数，如图 3-10 所示。

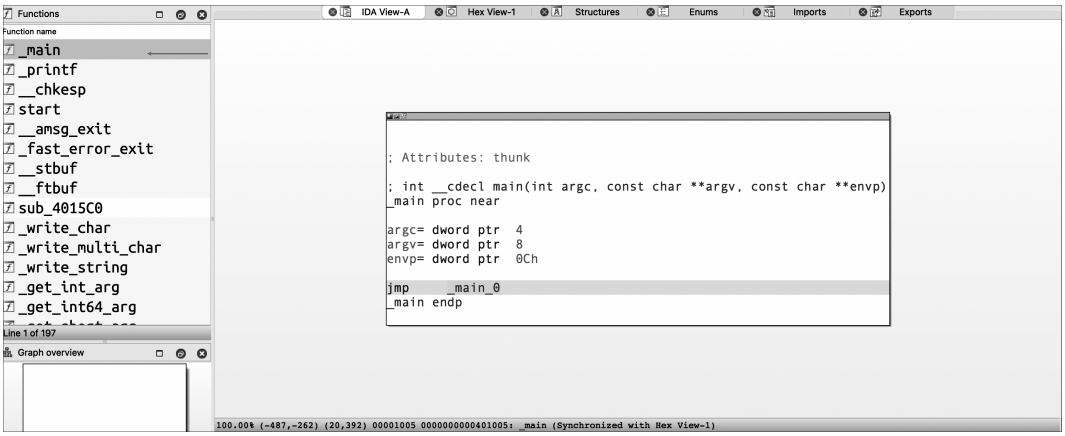

图 3-10　单击 IDA 主要窗口中的_main 函数

继续单击_main_0 函数，进入主要函数逻辑区域，如图 3-11 所示。

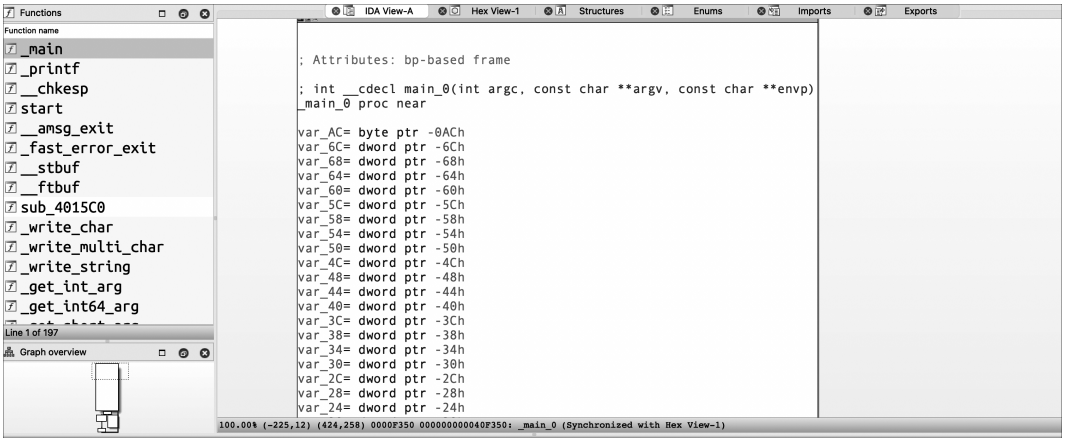

图 3-11　IDA 主窗口中 main 函数汇编界面

单击空格键将其转换为汇编模式，如图 3-12 所示。

在 IDA Pro 中可以使用 F5 功能进行伪代码的转换。如果没有 Pro 版本的 IDA，则无法使用 F5 功能，但可以结合汇编语言以及 Ghidra 中生成的伪代码同步查看，如图 3-13 所示。

接着便可以结合伪代码和汇编代码，对目标程序进行逆向分析，获取程序结构、设计流程等信息。分析结束后，一般选择不保存相关数据选项，直接退出程序。

根据两种逆向工具的分析结果，简单做一个比较。首先，观察 Ghidra 的逆向结果，图 3-7

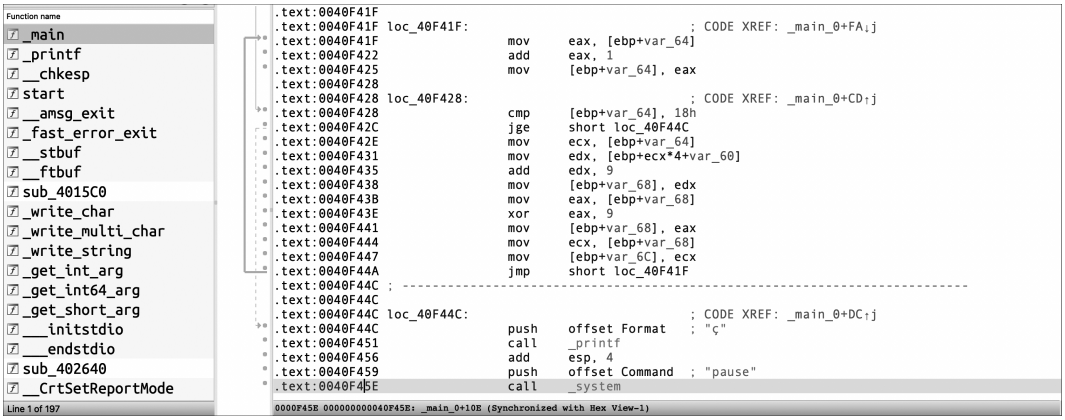

图 3-12　IDA 主窗口中汇编代码界面

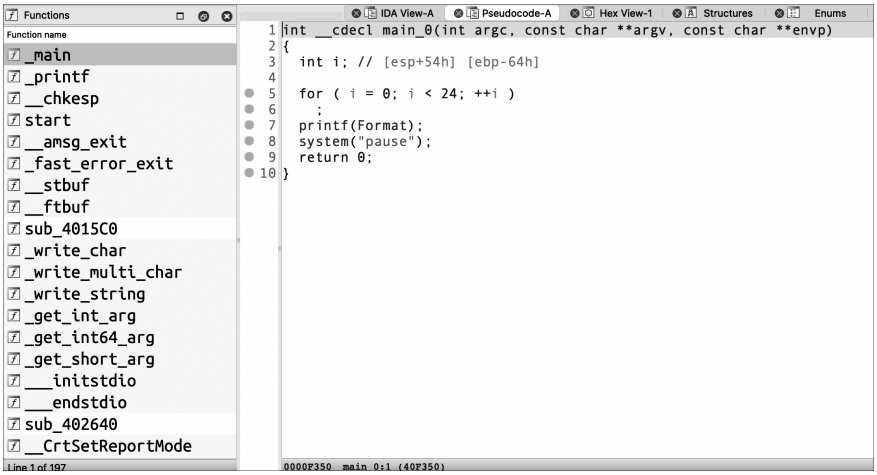

图 3-13　程序伪代码

显示了生成的伪代码，清晰地还原了程序逻辑。接着，观察 IDA 的逆向结果，图 3-12 为生成的汇编代码，在其地址 0x0040F43E 对应的汇编指令为“xor eax,9”，但在图 3-13 中并未体现异或操作。

Ghidra 和 IDA 作为目前两种最流行的静态逆向工具，各有所长。Ghidra 查看、定位反编译后的代码更接近源代码，不过其处理某些混淆后代码的能力还有所欠缺。IDA 的功能更加完善，界面更为友好，性能优于基于 Java 开发的 Ghidra。因此，逆向分析时结合多种工具进行交叉分析，有助于提升分析的正确性和效率。

3.1.3　动态分析工具 GDB

GDB 是 GNU 开发工具系列中的一个重量级产品，它是一个功能强大的调试器，既支持多种硬件平台，也支持多种程序语言；既可以用于本地调试，也可以用于远程调试；既支持符号调试，也支持指令级的反汇编调试。通过使用 GDB 可以完成以下工作：

- 启动程序，指定任何会影响其行为的条件；

- 让程序在特定的条件下暂停；
- 检查程序何时暂停，以及暂停时发生了什么事情；
- 改变程序状态或执行流程。

除此之外，GDB 还具有一些特色功能，例如命令自动补全功能、命令行编辑功能、面向对象语言支持（如 C++）、多线程支持等。

GDB 的安装非常简单，在 Ubuntu 下使用 apt-get install gdb 即可完成。在终端使用命令 gdb 即可启动调试。

GDB 的命令非常多，根据功能特点分类，包含断点类命令（Breakpoints）、数据类命令（Data）和文件类命令（Files）等。一个命令类中包含了功能相近的一组命令集合。GDB 提供了 help 命令帮助初学者了解所有命令类列表，用户可以使用 help 指定相应的命令类来列出该类型下所有命令的简短说明。除此之外，GDB 还能执行 shell 命令。

GDB 有两种退出方式：quit 命令和 Ctrl＋D 快捷键。需要注意的是，在 GDB 命令行中使用 Ctrl＋C 快捷键并不会使 GDB 退出，而只会中断正在执行的被调试程序。

下面开始通过实例演示 GDB 的功能和使用方法，实例来源于某场大型网络安全竞赛中的真题。查看文件的概要信息，包括文件类型、文件是 32 位还是 64 位、文件运行的基本情况。使用 010Editor 打开文件查看其二进制信息，可以判断是一个 ELF 文件，如图 3-14 所示。

图 3-14　文件类型

使用 file hero 查看文件信息，可知 hero 是一个 ELF 64-bit 的执行程序，那么就能使用 IDA64 查看分析目标程序，如图 3-15 所示。

```
root@whoami:~/demo# file hero
hero: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld
-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=5d43a21f3afe482b78a41a29648a070d01c0c2d9, no
t stripped
root@whoami:~/demo#
```

图 3-15　查看文件位数

将文件复制到 Linux 系统中，在其所在目录下使用命令./hero 运行，收集程序结构、分支条件等相关信息，如图 3-16 所示。

可以看到，程序运行后用户需要输入一个对应的功能选项。选择不同的标号，程序将会有不同的结果分支：选择 1 挑战 slime，选择 2 挑战 boss，选择 3 会进入商店花费一定的 coin 升级战斗力。

```
root@whoami:~/demo# ./hero
Day 0 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:1
A slime is refreshing , its Combat Effectiveness is 146
You die!root@whoami:~/demo# ./hero
Day 0 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:2
The dragon appears, its Combat Effectiveness is 1000000.
You die!root@whoami:~/demo#
```

图 3-16　文件运行情况

获取文件概要信息后，即可开始对其进行逆向分析。将程序放入 IDA64 中，得到程序的整体控制流程图，包括程序各个功能点及其函数分支，如图 3-17 和图 3-18 所示。

图 3-17　文件的程序逻辑结构

```
1  __int64 slime()
2  {
3    int v0; // ebx
4    int v2; // [rsp+Ch] [rbp-14h]
5
6    v0 = eff - 64;
7    v2 = v0 + rand() % 128;
8    if ( v2 == eff )
9      ++v2;
10   printf("A slime is refreshing , its Combat Effectiveness is %d\n", (unsigned int)v2);
11   if ( v2 > eff )
12   {
13     printf("You die!");
14     exit(1);
15   }
16   puts("The slime is beated, you get 1 coin.");
17   return (unsigned int)++coin;
18 }
```

图 3-18　文件的 slime 函数

接着按照相同的方法反编译 boss 函数的伪代码。图 3-19 显示，boss 函数有三条龙，战斗力分别是 1000000、3000000、5000000，需要打败三条龙才能拿到最终的 flag。因此，解题思路为修改变量 eff 的值，让其满足打败三条龙的条件。

```
__int64 boss()
{
  __int64 result; // rax

  if ( bossexist <= 2 )
  {
    if ( bossexist == 2 )
    {
      puts("The dragon appears, its Combat Effectiveness is 3000000.");
      if ( eff <= 3000000 )
      {
        printf("You die!");
        exit(1);
      }
      puts("The dragon is beated! flag is partly decrypted...");
      --bossexist;
      return decrypt2();
    }
    else
    {
      result = (unsigned int)bossexist;
      if ( bossexist <= 1 )
      {
        puts("The dragon appears, its Combat Effectiveness is 5000000.");
        if ( eff <= 5000000 )
        {
          printf("You die!");
          exit(1);
        }
        puts("The dragon is beated! combining flag and print...");
        --bossexist;
        return decrypt3();
      }
    }
  }
  else
  {
    puts("The dragon appears, its Combat Effectiveness is 1000000.");
    if ( eff <= 1000000 )
    {
      printf("You die!");
      exit(1);
    }
    puts("The dragon is beated! flag is partly decrypted...");
    --bossexist;
    return decrypt1();
  }
  return result;
}
```

图 3-19　文件的 boss 函数

使用 IDA 切换到汇编代码，boss 函数有三个关键比较，分别在地址 0x4015f9、0x40165c、0x4016bc 处。可以得知，程序首先与 eax 的值进行比较，然后根据比较结果决定是否跳转。因此，在调试过程改变 eax 的值，即可控制程序流程，从而打败三条龙，如图 3-20～图 3-22 所示。

```
.text:00000000004015F9                  cmp     eax, 0F4240h
.text:00000000004015FE                  jle     short loc_401628
.text:0000000000401600
.text:0000000000401600 loc_401600:                              ; DATA XREF:
.text:0000000000401600                  mov     edi, offset aTheDragonIsBea ;
.text:0000000000401605                  call    _puts
000015F9 00000000004015F9: boss+1F (Synchronized with Hex View-1)
```

图 3-20　boss 函数的第一个关键判断指令

```
.text:000000000040165C                  cmp     eax, 2DC6C0h
.text:0000000000401661                  jle     short loc_401688
.text:0000000000401663                  mov     edi, offset aTheDragonIsBea ;
.text:0000000000401668                  call    _puts
0000165C 000000000040165C: boss+82 (Synchronized with Hex View-1)
```

图 3-21　boss 函数的第二个关键判断指令

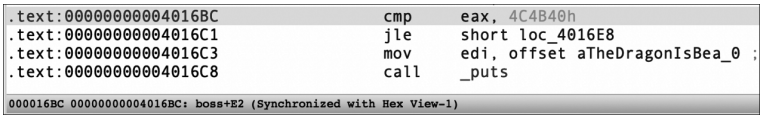

图 3-22　boss 函数的第三个关键判断指令

在终端使用命令 gdb hero 启动调试，使用命令 b * 0x4015F9、b * 0x40165C、b * 0x4016BC 在三个比较语句的位置打上断点，如图 3-23 所示(注：本书中安装了 gdb 的插件 pwndbg，读者可以自行安装)。

```
------- tip of the day (disable with set show-tips off) -------
Pwndbg resolves kernel memory maps by parsing page tables (default
stub command (use set kernel-vmmap-via-page-tables off for that)
pwndbg> b *0x4015F9
Breakpoint 1 at 0x4015f9
pwndbg> b *0x40165C
Breakpoint 2 at 0x40165c
pwndbg> b *0x4016BC
Breakpoint 3 at 0x4016bc
```

图 3-23　关键语句添加断点

执行命令 r 运行程序，选择 2，进入 boss 函数的第一个断点处，如图 3-24 所示。

```
pwndbg> r
Starting program: /root/demo/hero
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Day 0 , You want to:
+----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+----------------------+
Input the number of your chioce:2
The dragon appears, its Combat Effectiveness is 1000000.
```

图 3-24　运行程序至断点 1

在第一个断点处使用命令 set $eax=0x4c4b44 改变 eax 寄存器的值。根据图 3-19 反汇编逻辑，eax 的值只要大于 cmp 语句的第二操作数，程序就不会发生跳转，可顺利执行到 decrypt 函数，如图 3-25 所示。

```
pwndbg> set $eax = 0x4c4b44
pwndbg> info r
rax            0x4c4b44            5000004
rbx            0x0                 0
rcx            0x7ffff7e9aa37      140737352673847
rdx            0x1                 1
rsi            0x1                 1
rdi            0x7ffff7fa1a70      140737353751152
rbp            0x7fffffffe3b0      0x7fffffffe3b0
rsp            0x7fffffffe3b0      0x7fffffffe3b0
r8             0x7ffff7fa1a70      140737353751152
r9             0x0                 0
r10            0x7ffff7f44ac0      140737353370304
r11            0x246               582
r12            0x7fffffffe518      140737488348440
r13            0x401a40            4201024
r14            0x0                 0
r15            0x7ffff7ffd040      140737354125376
rip            0x4015f9            0x4015f9 <boss+31>
eflags         0x202               [ IF ]
```

图 3-25　设置寄存器 eax 的值 1

使用命令 info r 查看寄存器值，发现寄存器 rax 的值已经发生了改变。特别的，64 位寄

存器中的低 32 位可延用 32 位寄存器名，如 rax 的低 32 位可用 eax 表示，该部分内容可详见第 5 章寄存器的介绍。接着使用 c 命令继续运行至下一个断点，如图 3-26 所示。

```
pwndbg> c
Continuing.
The dragon is beated! flag is partly decrypted...
Day 1 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:2
The dragon appears, its Combat Effectiveness is 3000000.

Breakpoint 2, 0x000000000040165c in boss ()
```

图 3-26　继续运行程序至断点 2

程序运行之后，发现已经完成了第一个 boss 程序的验证，因此继续使用 c 命令执行程序并修改 eax 寄存器中的值，使用 set $eax＝0x4c4b44 命令，如图 3-27 所示。

```
pwndbg> set $eax = 0x4c4b44
pwndbg> info r
rax            0x4c4b44            5000004
rbx            0x0                 0
rcx            0x7ffff7e9aa37      140737352673847
rdx            0x1                 1
rsi            0x1                 1
rdi            0x7ffff7fa1a70      140737353751152
rbp            0x7fffffffe3b0      0x7fffffffe3b0
rsp            0x7fffffffe3b0      0x7fffffffe3b0
r8             0x7ffff7fa1a70      140737353751152
```

图 3-27　设置寄存器 eax 的值 2

继续使用命令 c 运行程序至下一个断点，如图 3-28 所示。

```
pwndbg> c
Continuing.
The dragon is beated! flag is partly decrypted...
Day 2 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:2
The dragon appears, its Combat Effectiveness is 5000000.

Breakpoint 3, 0x00000000004016bc in boss ()
```

图 3-28　继续运行程序至断点 3

继续使用命令 set $eax＝0x4c4b44 设置寄存器的值，如图 3-29 所示。

```
pwndbg> set $eax = 0x4c4b44
pwndbg> info f
Stack level 0, frame at 0x7fffffffe3c0:
 rip = 0x4016bc in boss; saved rip = 0x4018b9
 called by frame at 0x7fffffffe3f0
 Arglist at 0x7fffffffe3b0, args:
 Locals at 0x7fffffffe3b0, Previous frame's sp is 0x7fffffffe3c0
 Saved registers:
  rbp at 0x7fffffffe3b0, rip at 0x7fffffffe3b8
pwndbg> info r
rax            0x4c4b44            5000004
rbx            0x0                 0
rcx            0x7ffff7e9aa37      140737352673847
rdx            0x1                 1
rsi            0x1                 1
rdi            0x7ffff7fa1a70      140737353751152
rbp            0x7fffffffe3b0      0x7fffffffe3b0
```

图 3-29　设置寄存器 eax 的值 3

继续执行程序，获得最终结果，如图 3-30 所示。

```
pwndbg> c
Continuing.
The dragon is beated! combining flag and print...
flag{0259-6430-726f077b-5959-15baa412c83b}[Inferior 1
```

图 3-30　获得 flag

回顾上述分析过程：首先，定位到 slime 函数和 boss 函数，发现在 boss 函数中出现了 flag；接着，分析程序分支，执行命令 r 动态调试程序，使用 set $eax＝0x4c4b44 在断点 1 处改变条件判断语句的结果，在断点 2、断点 3 处做相同的操作；最后，使用 c 进行继续运行程序，一直选择 2 操作，不停地挑战 boss，挑战 3 次成功，获得最终结果 flag{0259-6430-726f077b-5959-15baa412c83b}。

继续观察 store 函数，发现还存在另一种解题方法——整数溢出。使用 F5 反编译 store 函数，如图 3-31 所示。注意到 store 函数通过 scanf 方式接收输入并比较，如果 scanf 输入变量发生溢出，则可能改变下一条判断语句的执行结果，进而改变程序的执行流程。

```
unsigned __int64 store()
{
  unsigned int v1; // [rsp+0h] [rbp-10h] BYREF
  int v2; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  puts("2 coins to upgrade 1 point of Combat Effectiveness");
  printf("Your coins:%d\n", (unsigned int)coin);
  printf("Your Combat Effectiveness is :%d\n", (unsigned int)eff);
  printf("input the points of you want to upgrade:");
  __isoc99_scanf("%d", &v1);
  if ( (int)v1 > 0 )
  {
    v2 = coin - 2 * v1;
    if ( v2 < 0 )
    {
      puts("Your coins is not enough.");
    }
    else
    {
      eff += v1;
      coin = v2;
      printf("Your Combat Effectiveness upgraded %d points, now it is %d points.\n", v1, (unsigned int)eff);
      printf("Your coins:%d\n", (unsigned int)coin);
    }
  }
  else
  {
    puts("please input a positive number.");
  }
  return __readfsqword(0x28u) ^ v3;
}
```

图 3-31　文件的 store 函数

int 类型可表示的十进制数据范围是－2147483648 至 2147483647，当 v1＞2147483647 或 v1＜－2147483648 时会造成整数溢出。v1＞2147483647 溢出后就会变为负数；相反，当 v1＜－2147483648 溢出后就会变为正数。故当 v1≤2147483647 时，能够进入第一个判断语句，有机会执行充值提升攻击力。进一步，当 v2（即 coin－2 * v1）＜－2147483648 时，会发生整数溢出变为正数，可执行第二个判断语句的条件分支 eff ＋＝ v1、coin＝v2。

综合 v1 和 v2 需满足的条件，运行程序，设置 v1＝1073741828、coin＝5，如图 3-32 所示。由于 v1＞0，则可执行 v2＝（coin－2 * v1）的赋值语句，发生溢出后可得 v2＝2147483645。由于 v2＞0，可执行 coin＝v2 的赋值语句，得到 coin＝2147483645，满足打败三条龙的条件。继续执行，最后得到 flag{0259-6430-726f077b-5959-bf477a78c83b}。

```
root@whoami:~/demo# ./hero
Day 0 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:3
2 coins to upgrade 1 point of Combat Effectiveness
Your coins:5
Your Combat Effectiveness is :100
input the points of you want to upgrade:1073741828
Your Combat Effectiveness upgraded 1073741828 points, now it is 1073741928 points.
Your coins:2147483645
Day 1 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:2
The dragon appears, its Combat Effectiveness is 1000000.
The dragon is beated! flag is partly decrypted...
Day 2 , You want to:
+-----------------------+
| 1.battle with slime.  |
| 2.battle with boss.   |
| 3.go to the shop.     |
+-----------------------+
Input the number of your chioce:2
The dragon appears, its Combat Effectiveness is 3000000.
The dragon is beated! flag is partly decrypted...
Day 3 , You want to:
```

图 3-32　运行程序并发生整数溢出

注意,本案例中的 flag 存在多解,读者可自行分析其多解的原因。

3.2　逆向脱壳分析

UPX(the Ultimate Packer for eXecutables)脱壳是逆向工程中一项重要的技术,本节围绕 UPX 壳,阐述软件加壳原理、ESP 定律脱壳法等内容。

3.2.1　软件加壳原理

加壳是利用特殊的算法,对 EXE、DLL 文件里的资源进行压缩、加密的过程,是保护文件的常用手段。加壳后的程序可以直接运行,但要经过脱壳才可以查看源代码。

UPX 是一款先进的可执行程序文件压缩器(压缩壳),其工作原理主要是压缩和实时解压。压缩包括两方面:在程序的开头或者其他合适的地方插入一段代码;将程序的其他区段压缩。压缩也可以叫作加密,因为压缩后的程序比较难看懂。较之未压缩的代码,压缩后的代码程序本身变小了,有利于程序的传输。程序执行时由压缩插入的代码完成实时解压,不会影响程序的执行效率。

UPX 加壳操作非常简单,可以直接在官网下载安装。针对不同平台架构,选择对应的 UPX 壳版本安装即可。安装完成后,对要加壳的程序使用命令 UPX,即可进行加壳操作。

3.2.2　ESP 定律脱壳法

针对压缩壳,可以使用命令 upx -d 自动脱壳。然而,针对某些“魔改”UPX 壳,使用命令并不能成功脱壳,因此要采用 ESP 定律手动脱壳。

硬件断点是 ESP 脱壳定律的关键技巧，是由硬件提供的调试寄存器组，用户可以对这些硬件寄存器设置相应的值，然后让硬件断在需要下断点的地址。硬件断点的触发条件有四种：访问、写入、I/O 以及读写。ESP 定律主要运用堆栈平衡的属性，关于栈的详细介绍可参考本书第 5 章。对于一个加壳程序，在进行自解密或者自解压时，会将当前寄存器的状态使用命令 pushad 入栈。相应地，在解压结束后，会使用命令 popad 将保存的寄存器状态出栈。当寄存器出栈时，原有的壳代码会恢复，触发 ESP 访问硬件断点，再继续单步运行就非常容易找到程序真正的入口点。

下面演示使用 x64dbg 调试器对 64 位程序进行 ESP 定律脱壳调试。32 位程序调试使用 x32dbg 或 OllyDbg，脱壳过程同 64 位程序。将目标程序拖入 x64dbg，如图 3-33 所示。

图 3-33　使用 x64dbg 打开目标程序

由于 Windows 系统特性，此时属于 Windows 的 ntdll 阶段，因此需要执行下一步操作，继续单击“运行”按钮直到找到 push 指令（即壳压入栈中代码的区域）。继续向下单步执行，如图 3-34 所示。单击右侧 RSP 寄存器，使用其“在内存窗口中转到”功能转到内存 1。

在 RSP 所指向地址中，从内存 1 窗口界面显示的起始位置开始随意选取一段内存区域，设置硬件访问断点，图 3-35 显示选择了 4 字节内存区域。继续执行程序，当该内容再次被硬件访问会被断下，如图 3-36 所示。硬件断点在第一条语句处断下。

继续使用 F8 键向下单步运行直到 jmp 指令，其所指向地址即为程序真正的入口点（OEP），如图 3-37 所示。

使用 F4 键运行到 jmp 所指位置，再使用 F8 键单步调试，如图 3-38 所示。

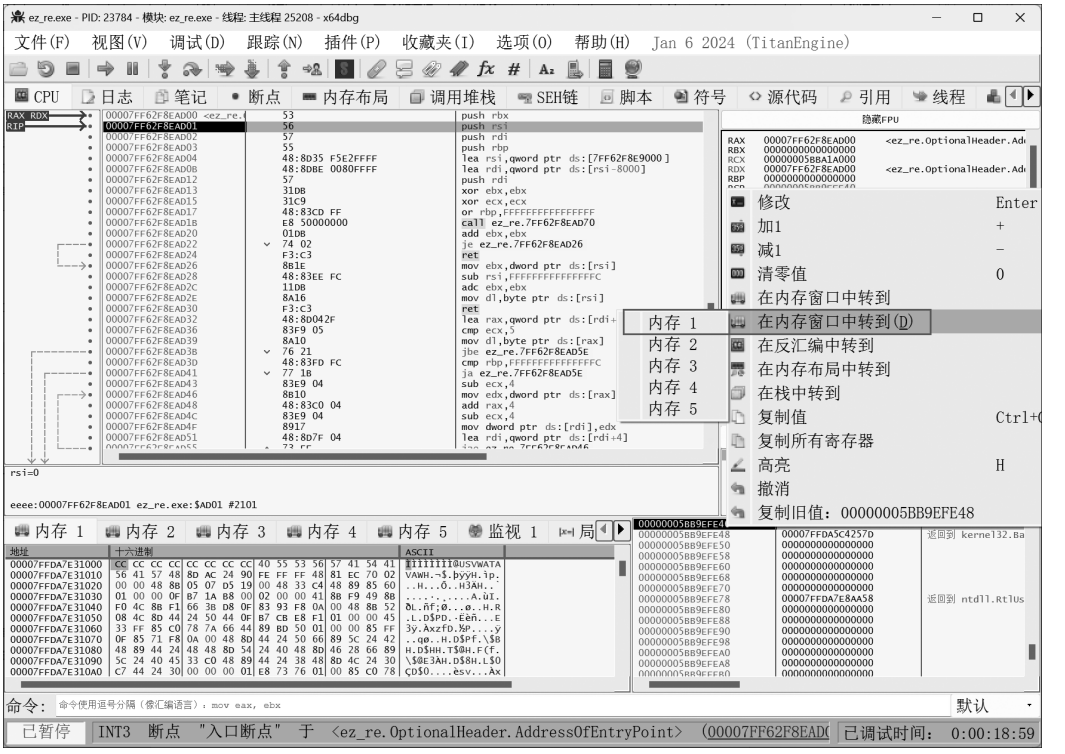

图 3-34　在内存窗口跟随

图 3-35　在程序中设置硬件访问断点

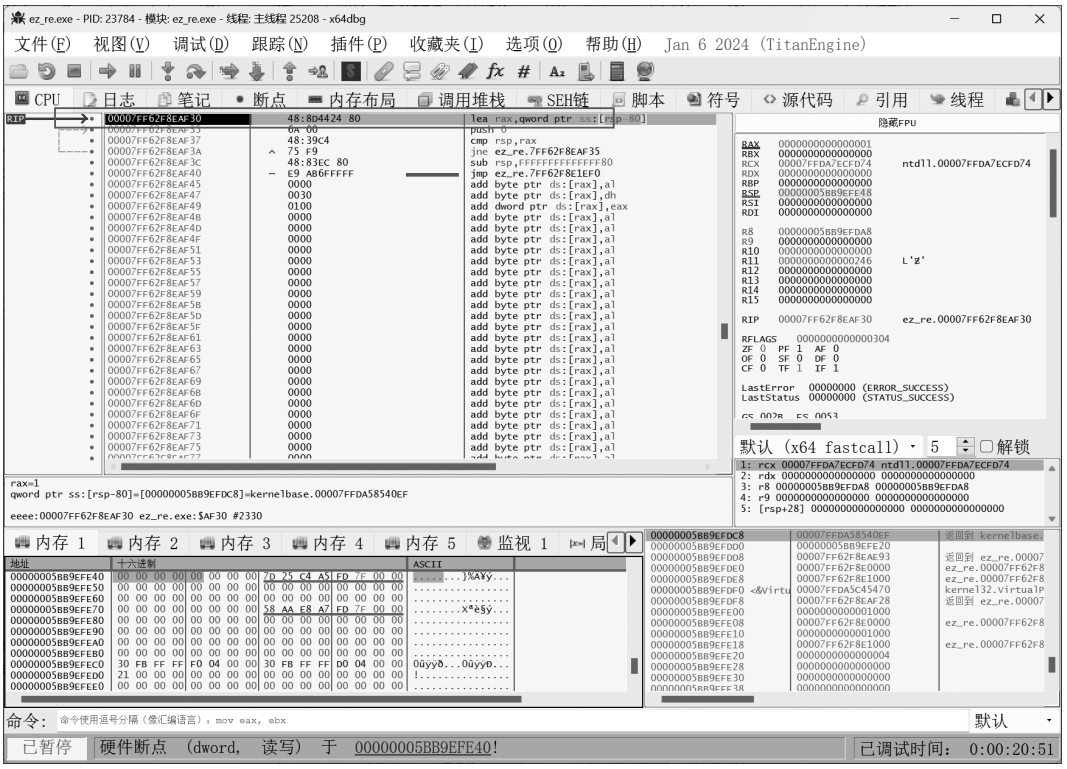

图 3-36　硬件访问地址断点位置

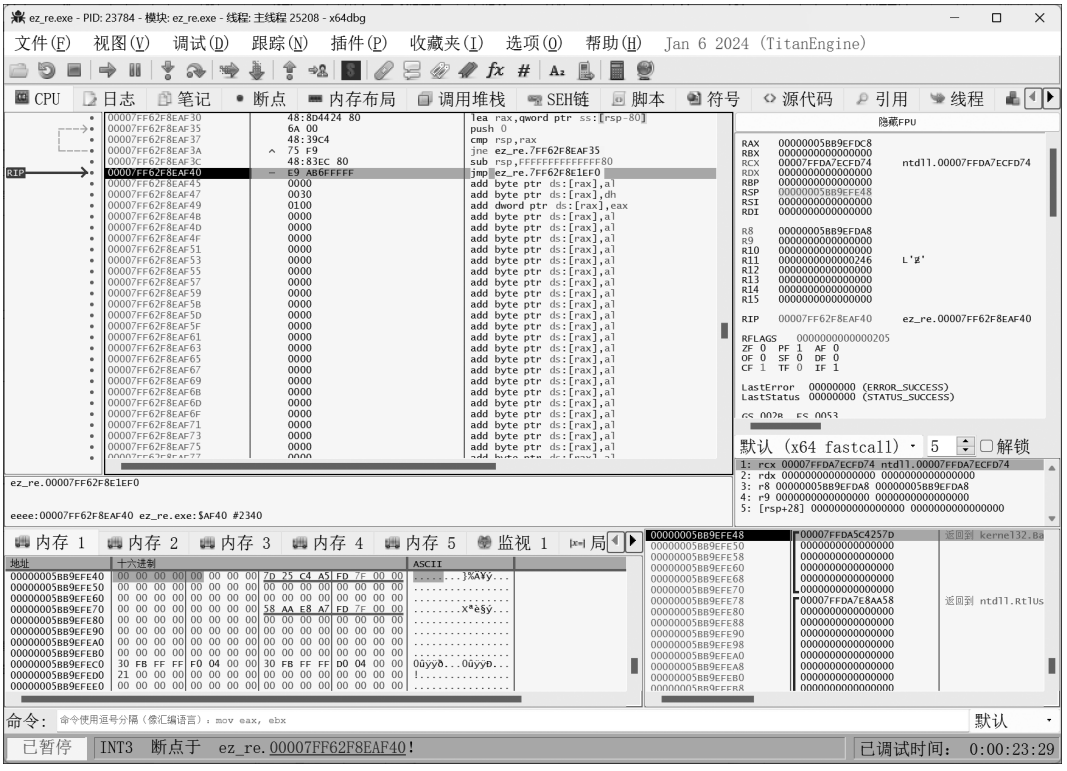

图 3-37　使用 x64dbg 脱壳找到程序的 OEP

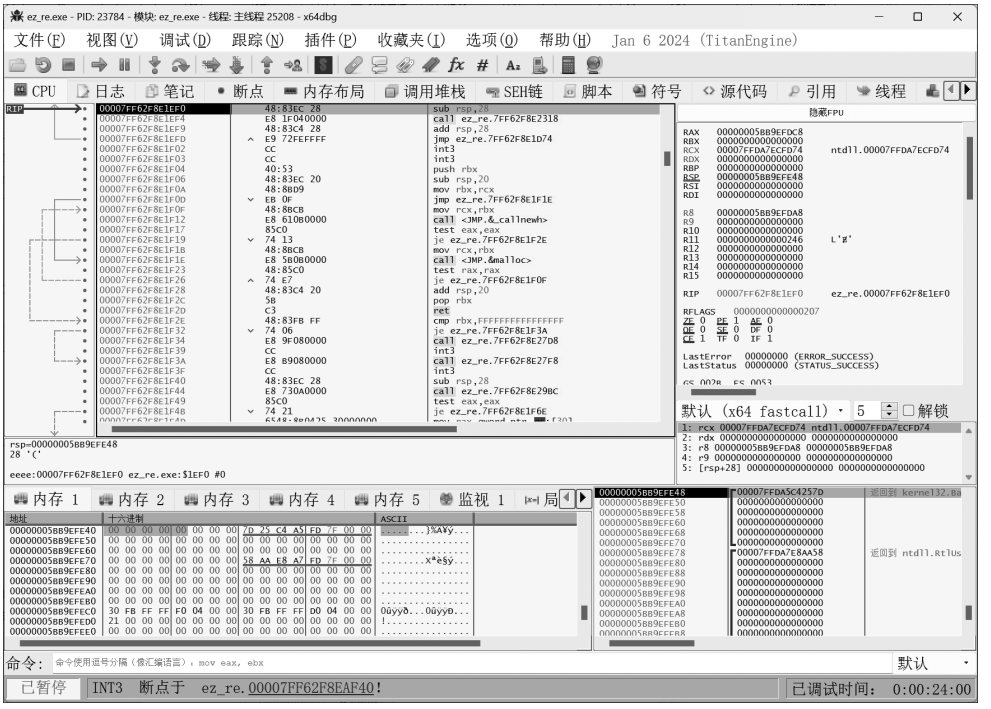

图 3-38　单步调试找到程序真正的 OEP

该位置为程序真正的入口点，随后单击“插件”，单击 Scylla 插件。首先，进行 IAT Autosearch、Get Imports 操作，如图 3-39 所示。

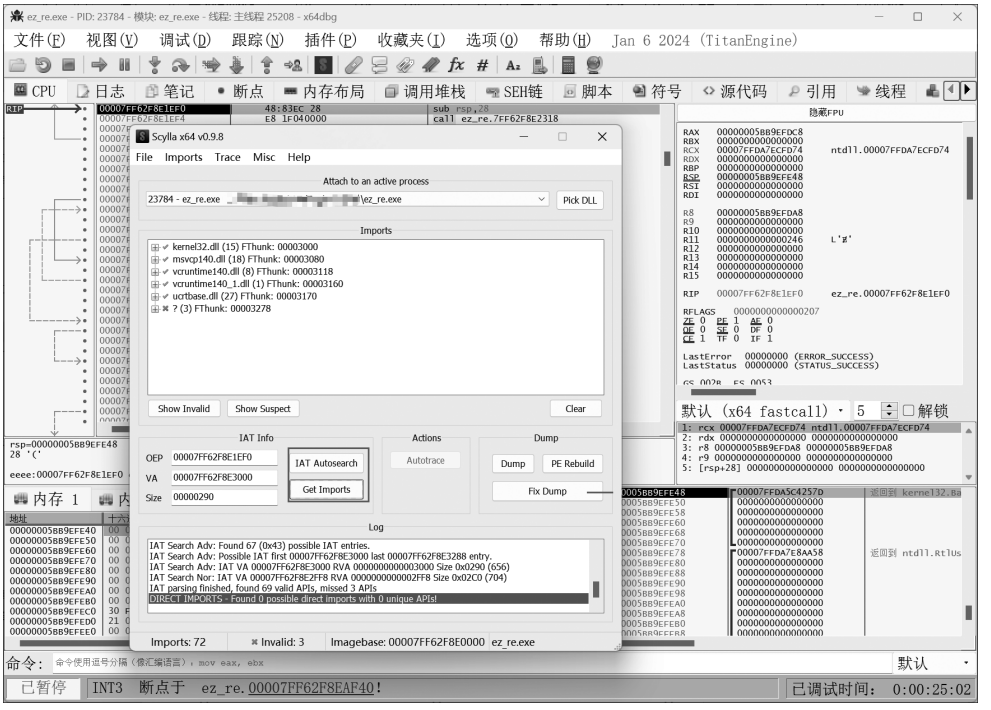

图 3-39　使用插件修复目标程序