COMPASS CTF Tutorial 6: Reverse Engineering

COMPASS CTF 教程【6】: 逆向工程

30016794 Zhao, Li (Research Assistant)

COMPuter And System Security Lab, Computer Science and Technology Department, College of Engineering (CE), SUSTech University.
南方科技大学 工学院 计算机科学与技术系 计算机与系统安全实验室

2023 年 8 月 15 日

逆向工程概述

逆向工程（Reverse engineering）是一种技术过程，即对一项目标产品进行逆向分析及研究，从而演绎并得出该产品的处理流程、组织结构、功能性能规格等设计要素，以制作出功能相近但不完全一样的产品的过程。在 CTF 中，逆向工程一般是指软件逆向工程，即对已经编译完成的可执行文件进行分析，研究程序的行为和算法，然后以此为依据，计算出出题人想隐藏的 flag。[1]

COMPASS CTF TEAM 计算机与系统安全实验室

逆向工程概述

一般，CTF 中的逆向工程题目形式为：程序接收用户的一个输入，并在程序中进行一系列校验算法，如通过校验则提示成功，此时的输入即 flag。这些校验算法可以是已经成熟的加解密方案，也可以是作者自创的某种算法。比如，一个小游戏将用户的输入作为游戏的操作步骤进行判断等。这类题目要求参赛者具备一定的算法能力、思维能力，甚至联想能力。

COMPASS CTF TEAM 计算机与系统安全实验室

可执行文件（ELF）

软件逆向工程分析的对象是程序，即一个或多个可执行文件。下面简单介绍可执行文件的形成过程、常见可执行文件类型，以便读者对它们有一个初步的认知。

可执行文件的形成过程（编译和链接）

对于刚刚接触这方面的萌新，形成一个正确的对可执行文件的理解和感觉是至关重要的。同样，作为人类文明一手创造的事物，可执行文件并不是如同变魔法一般直接生成的，而是经历了一系列的步骤。

绝大多数正常的可执行文件，都是由高级语言编译生成的。一般来说，编译时会发生这些流程：

COMPASS CTF TEAM 计算机与系统安全实验室

编译流程

1. 用户将一组用高级语言编写的源代码作为编译器输入。
2. 编译器解析输入，并为每个源代码文件产生对应的汇编代码。
3. 汇编器接收编译器生成的汇编代码，并继续执行汇编操作，将生成的每份机器代码临时存于各对象文件中。

COMPASS CTF TEAM 计算机与系统安全实验室

编译流程

4. 现在已经生成了多个对象文件，但是最后的目标是生成一个可执行文件。于是链接器参与其中，将分散的各对象文件相互连接，经过处理而融合成完整的程序。然后按照可执行文件的格式，填入各种指定程序运行环境的参数，最后形成一个完整的可执行文件。

COMPASS CTF TEAM 计算机与系统安全实验室

其他的编译过程

而在实际的环境中，由于需要考虑到生成的可执行文件的大小、可执行文件的运行性能、对信息的保护等原因，在每步过程中或多或少伴随着信息的丢失。例如，在编译阶段一般会丢弃掉源代码中的注释信息，在汇编时可能丢弃汇编代码中的 label（标签）名称，在链接时可能丢弃函数名、类型名等符号信息。
逆向则需要利用相关知识和经验，来还原其中的部分信息，进而还原全部或部分程序流程，从而实现分析者的各种目的。

COMPASS CTF TEAM 网络与系统安全实验室

不同格式的可执行文件

实际中，由于历史遗留问题和公司之间竞争等原因，上面介绍的每一步中产生的各种文件都会有多种文件格式。例如，Windows 系统使用的是 PE（Portable Executable）可执行文件，而 Linux 系统使用的是 ELF（Executable and Linkable Format）可执行文件。由于这两种可执行文件格式都是由 COFF（Common File Format）格式发展而来的，因此文件结构中的各种概念非常相似。

COMPASS CTF TEAM 计算机与系统安全实验室

不同格式的可执行文件

PE 文件由 DOS 头、PE 文件头、节表及各节数据组成；同时，如果需要引用外部的动态链接库，则有导入表；如果自己可以提供函数给其他程序来动态链接（常见于 DLL 文件），则有导出表。
ELF 文件由 ELF 头、各节数据、节表、字符串段、符号表组成。

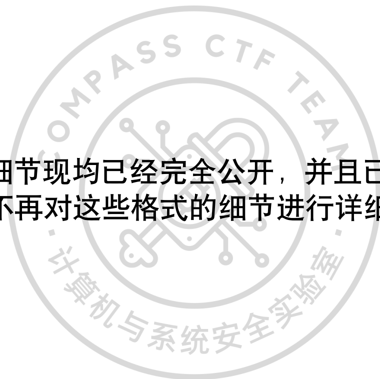
COMPASS CTF TEAM 计算机与系统安全实验室

不同格式的可执行文件

节（Section）是程序中各部分的逻辑划分，一般有特定名称，如.text 或.code 代表代码节、.data 代表数据节等。在运行时，可执行文件的各节会被加载到内存的各位置，为了方便管理和节省开销，一个或多个节会被映射到一个段（Segment）中。段的划分是根据这部分内存需要的权限（读、写、执行）来进行的。如果在相应的段内进行了非法操作，如在只能读取和执行的代码段进行了写操作，则会产生段错误(Segmentation Fault)。

COMPASS CTF TEAM 计算机与系统安全实验室

不同格式的可执行文件



PE[2]和 ELF[3]的基本格式细节现均已经完全公开，并且已经有大量的成熟工具可对其进行解析与修改，在此不再对这些格式的细节进行详细讲解，请感兴趣的读者自行查阅相关资料。

汇编语言基本知识

逆向者在解析文件后，面对的是一大片机器代码，而机器代码是由汇编语言直接生成的，因此逆向者需要对汇编有基本的认识才可以展开后续工作。
下面介绍汇编语言的重点概念，方便快速理解汇编语言。

COMPASS CTF TEAM 计算机与系统安全实验室

寄存器、内存和寻址

寄存器（Register）是 CPU 的组成部分，是有限存储容量的高速存储部件，用来暂存指令、数据和地址。一般的 IA-32（Intel Architecture，32-bit）即 x86 架构的处理器中包含以下在指令中显式可见的寄存器：

- 通用寄存器 EAX、EBX、ECX、EDX、ESI、EDI。
- 栈顶指针寄存器 ESP、栈底指针寄存器 EBP。
- 指令计数器 EIP（保存下一条即将执行的指令的地址）。
- 段寄存器 CS、DS、SS、ES、FS、GS。

寄存器、内存和寻址

对于 x86-64 架构，在以上这些寄存器的基础上，将前缀的 E 改成 R，以标记 64 位，同时增加了 R8 ~ R15 这 8 个通用寄存器。另外，对于 16 位的情况，则将前缀 E 全部去掉。16 位时，对于寄存器的使用有一定限制，由于现在已不是主流，故不再赘述。对于通用寄存器，程序可以全部使用，也可以只使用一部分。使用寄存器不同部分时对应的助记符见图-1。其中，R8 ~ R15 进行拆分时的命名规则为 R8d（低 32 位）、R8w（低 16 位）和 R8b（低 8 位）。

COMPASS CTF TEAM 计算机与系统安全实验室

寄存器、内存和寻址

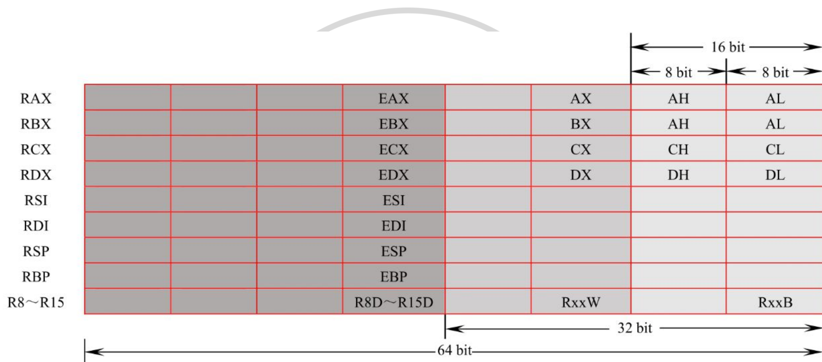


图: 寄存器的命名

标志寄存器

CPU 中还存在一个标志寄存器，其中的每位表示对应标志位的值，常用的标志位如下。

- AF：辅助进位标志（Auxiliary Carry Flag），当运算结果在第 3 位进位的时候置 1。
- PF：奇偶校验标志（Parity Flag），当运算结果的最低有效字节有偶数个 1 时置 1。
- SF：符号标志（Sign Flag），有符号整形的符号位为 1 时置 1，代表这是一个负数。

标志寄存器

- ZF：零标志（Zero Flag），当运算结果为全零时置 1。
- OF：溢出标志（Overflow Flag），运算结果在被操作数是有符号数且溢出时置 1。
- CF：进位标志（Carry Flag），运算结果向最高位以上进位时置 1，用来判断无符号数的溢出。

COMPASS CTF TEAM 计算机与系统安全实验室

寻址方式

CPU 不仅可对寄存器进行操作，还可对内存单元进行操作，因此存在多种不同的寻址方式。表-1 给出了 CPU 的不同寻址方式、示例及对应的操作对象。

寻址方式	示　例	操作对象
立即寻址	1000h	1000h 这个数字
直接寻址	[1000h]	内存 1000h 地址的单元
寄存器寻址	RAX	RAX 这个寄存器
寄存器间接寻址	[RAX]	以 RAX 中存的数作为地址的内存单元
基址寻址	[RBP+10h]	将 RBP 中的数作为基址，加上 10h，访问这个地址的内存单元
变址寻址	[RDI+10h]	将 RDI 作为变址寄存器，将其中的数字加上 10h，访问这个地址的内存单元
基址加变址寻址	[RBX+RSI+10h]	逻辑同上

图: 寻址方式列表

寻址方式

不难看出，“[]”相当于 C 语言中的“*”运算符（间接访问）。
在 x86/x64 架构中，寄存器间接寻址、基址寻址、变址寻址、基址加变址寻址这 4 种寻址方式在实现的功能方面几乎相同，但语义上是有区别的。在 16 位时代，这 4 种寻址方式不可混用，在现代编译器中，编译器会根据语义和优化选择合适的寻址方式，对于 CTF 参赛者来说，只需稍作了解即可。

COMPASS CTF TEAM 计算机与系统安全实验室

x86/x64 汇编语言

x86/x64 汇编语言存在 Intel、AT&T 两种显示/书写风格，我们将统一采用 Intel 风格。
什么是机器码？什么是汇编语言？机器码是在 CPU 上直接执行的二进制指令，而汇编语言是机器语言的一种助记符，汇编语言与机器码是一一对应的。机器码根据 CPU 架构的不同而不同，CTF 和平时最常见的 CPU 架构是 x86 和 x86-64（x64）。

COMPASS CTF TEAM 计算机与系统安全实验室

x86/x64 汇编语言

x86/x64 汇编指令的基本格式如下：

```
操作码 [操作数 1] [操作数 2]
```

x86/x64 汇编语言

其中，操作数的存在与否及形式由操作码的类型决定。由于篇幅限制，我们无法面面俱到地叙述各种指令的格式及功能，表-2 给出了几种常用指令的形式、功能和对应的高级语言写法。入门阶段的 CTF 参赛者并不需要掌握如何流畅地编写汇编语言程序，只需掌握下面介绍的常见指令，并在遇到这些常见的汇编指令时可以读懂即可。

COMPASS CTF TEAM 计算机与系统安全实验室

常用指令

指令类型	操作码	指令示例	对应作用
数据传送指令	mov	mov rax, rbx	rax = rbx
		mov qword ptr [rdi], rax	*(rdi) = rax
取地址指令	lea	lea rax, [rsi]	rax = & *(rsi)
算术运算指令	add	add rax, rbx	rax += rbx
		add qword ptr [rdi], rax	*(rdi) += rax
	sub	sub rax, rbx	rax -= rbx
逻辑运算指令	and	and rax, rbx	rax &= rbx
	xor	xor rax, rbx	rax ^= rbx
函数调用指令	call	call 0x401000	执行 0x40100 地址的函数
函数返回指令	ret	ret	函数返回
比较指令	cmp	cmp rax, rbx	根据 rax 与 rbx 比较的结果改变标志位
无条件跳转指令	jmp	jmp 0x401000	跳到 0x401000 地址执行
栈操作指令	push	push rax	将 rax 的值压入栈中
	pop	pop rax	从栈上弹出一个元素放入 rax

图: 几种常用指令的形式

条件跳转

汇编语言中的条件跳转指令有很多，它们会根据标志位的情况进行条件跳转。在条件跳转指令前往往存在用于比较的 cmp 指令，会根据比较结果对标志位进行相应设置（对标志位的影响等同于 sub 指令）。
表-3 给出了常见的条件跳转指令，以及所依据的 cmp 和标志位的情况。

COMPASS CTF TEAM 计算机与系统安全实验室

条件跳转

指　令	全　称	cmp a, b 条件	flag 条件
jz/je	jump if zero/equal	a = b	ZF = 1
jnz/jne	jump if not zero/equal	a != b	ZF = 0
jb/jnae/jc	jump if below/not above or equal/carry	a > b，有符号数	CF = 1
ja/jnbe	jump if above/not below or equal	a < b，有符号数	
jna/jbe	jump if not above/below or equal	a <= b，有符号数	
jnc/jnb/jae	jump if not carry/not below/above or equal	a >= b，有符号数	CF = 0
jg/jnle	jump if greater/not less or equal	a > b，无符号数	
jge/jnl	jump if greater or equal/not less	a >= b，无符号数	
jl/jnge	jump if less/not greater or equal	a < b，无符号数	
jle/jng	jump if less or equal/not greater	a <= b，无符号数	
jo	jump if overflow		OF = 1
js	jump if signed		SF = 1

图: 常见的条件跳转指令

反汇编

高级语言往往需要复杂的编译过程，汇编过程则只是直接翻译汇编语句为对应的机器代码，并直接将各条语句相邻地放在一起。因此，我们可以轻易地将机器代码翻译回汇编语言，这样的过程即**反汇编**。

COMPASS CTF TEAM 计算机与系统安全实验室

反汇编

正如前文中提到的，汇编过程同样是有着信息丢失的。虽然我们可以轻易地解析并还原给定指令的内容，但是我们必须知道哪些数据是机器代码，才可以相应地对它进行解析。冯·诺依曼架构模糊了代码与数据的区别界限，在代码节中可能穿插跳转表、常量池（ARM）、普通常量数据，甚至恶意的干扰数据等。所以，简单、直接地一条条连续地向下解析指令往往会出现问题。我们需要知道正确的指令的起始位置（如 label，中文译为“标签”，用来表示程序的一个位置，方便跳转、取地址时引用）来指引反汇编工具正确解析代码。

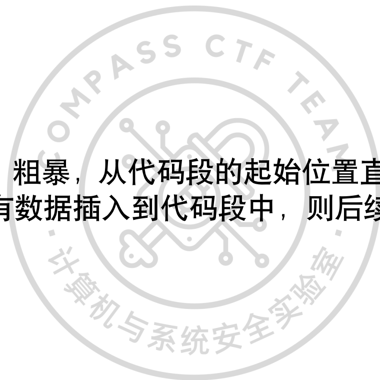
反汇编的算法

在汇编过程中，**label 信息会丢失**。因为 label 用于标识跳转位置，它决定着程序执行时可能执行到的位置，即汇编语句的起始位置。所以，还原出正确的 label 信息对于正确还原程序执行流程至关重要。

尽管有信息丢失，我们仍然可以通过一些算法成功还原程序的流程。下面介绍两种已知的算法：**线性扫描反汇编算法**和**递归下降反汇编算法**。

COMPASS CTF TEAM

线性扫描反汇编算法



线性扫描反汇编算法简单、粗暴，从代码段的起始位置直接一个接一个地解析指令，直至结束。其缺点是一旦有数据插入到代码段中，则后续的所有反汇编结果是错误的、无用的。

递归下降反汇编算法

递归下降反汇编算法则是人们在发现线性扫描反汇编算法的种种问题后创造的一种新算法，不是简单地解析指令并显示，而是尝试推测执行每条指令后程序将如何执行。例如，普通指令在执行后将直接执行下一条，无条件跳转指令会立即跳到目标位置，函数调用指令会临时跳出再返回继续执行，返回指令则会终止当前的执行流程，条件跳转指令则可能分出两条路径，在不同的条件下走向不同的位置。引擎先将一些已知的模式（pattern）匹配到起始位置，再根据指令的执行模式，逐个对程序执行情况进行跟踪，最后将程序完全反汇编。

调用约定

随着软件规模增大，开发人员不断增多，函数之间的关系同步变得越来越复杂，如果每个开发人员使用不同的规则传递函数参数，则程序往往会出现各种匪夷所思的错误，程序的维护开支会变得非常大。为此，在编译器出现后，人们为编译器创立了一些规定各函数之间的参数传递的约定，称为**调用约定**。常见的调用约定有以下几种。

COMPASS CTF TEAM 计算机与系统安全实验室

x86 32 位架构的调用约定

- __cdecl：参数从右向左依次压入栈中，调用完毕，由调用者负责将这些压入的参数清理掉，返回值置于 EAX 中。绝大多数 x86 平台的 C 语言程序都在使用这种约定。
- __stdcall：参数同样从右向左依次压入栈中，调用完毕，由被调用者负责清理压入的参数，返回值同样置于 EAX 中。Windows 的很多 API 都是用这种方式提供的。
- __thiscall：为类方法专门优化的调用约定，将类方法的 this 指针放在 ECX 寄存器中，然后将其余参数压入栈中。
- __fastcall：为加速调用而生的调用约定，将第 1 个参数放在 ECX 中，将第 2 个参数放在 EDX 中，然后将后续的参数从右至左压入栈中。

x86 64 位架构的调用约定

- Microsoft x64 位（x86-64）调用约定：在 Windows 上使用，依次将前 4 个参数放入 RDI、RSI、RDX、RCX 这 4 个寄存器，然后将剩下的参数从右至左压入栈中。
- SystemV x64 调用约定：在 Linux、MacOS 上使用，比 Microsoft 的版本多了两个寄存器，使用 RDI、RSI、RDX、RCX、R8、R9 这 6 个寄存器传递前 6 个参数，剩下的从右至左压栈。

COMPASS CTF TEAM 计算机与系统安全实验室

局部变量

写程序的时候，程序员经常会使用局部变量。但是在汇编中只有寄存器、栈、可写区段、堆，函数的局部变量该存在哪里呢？需要注意的是，局部变量有“易失性”：一旦函数返回，则所有局部变量会失效。考虑到这种特性，人们将局部变量存放在栈上，在每次函数被调用时，程序从栈上分配一段空间，作为存储局部变量的区域。每个函数在被调用的时候都会产生这样的局部变量的区域、存储返回地址的区域和参数的区域，见图-2。程序一层层地深入调用函数，每个函数自己的区域就一层层地叠在栈上。

局部变量

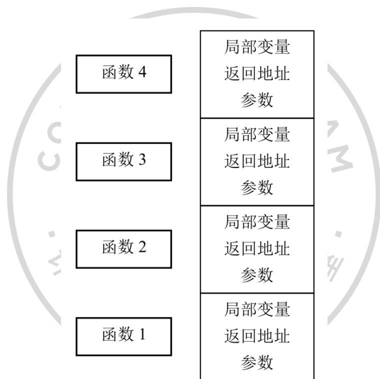


图: 栈帧

栈帧

人们把每个函数自己的这一片区域称为帧，由于这些帧都在栈上，所以又被称为栈帧。然而，栈的内存区域并不一定是固定的，而且随着每次调用的路径不同，栈帧的位置也会不同，那么如何才能正确引用局部变量呢？
虽然栈的内容随着进栈和出栈会一直不断变化，但是一个函数中每个局部变量相对于该函数栈帧的偏移都是固定的。所以可以引入一个寄存器来专门存储当前栈帧的位置，即 ebp，称为帧指针。程序在函数初始化阶段赋值 ebp 为栈帧中间的某个位置，这样可以用 ebp 引用所有的局部变量。由于上一层的父函数也要使用 ebp，因此要在函数开始时先保存 ebp，再赋值 ebp 为自己的栈帧的值，这样的流程在汇编代码中便是经典的组合：

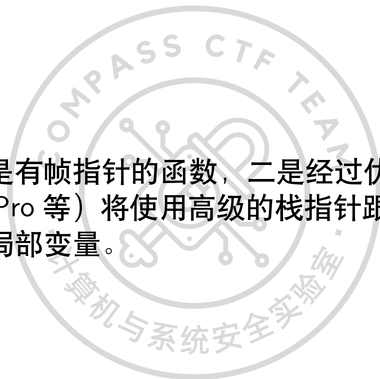
```
push    ebp
mov     ebp, esp
```

栈帧与调用链

现在每个函数的栈帧便由局部变量、父栈帧的值、返回地址、参数四部分构成。可以看出，ebp 在初始化后实际上指向的是父栈帧地址的存储位置。因此，*ebp 形成了一个链表，代表一层层的函数调用链。
随着编译技术的发展，编译器也可以通过跟踪计算每个指令执行时栈的位置，从而直接越过 ebp，而使用栈指针 esp 来引用局部变量。这样可以节省每次保存 ebp 时需要的时间，并增加了一个通用寄存器，从而提高了程序性能。

COMPASS CTF TEAM

栈帧与调用链



于是现在有两种函数：一是有帧指针的函数，二是经过优化后没有帧指针的函数。现代的分析工具（如 IDA Pro 等）将使用高级的栈指针跟踪方法来针对性地处理这两种函数，从而正确处理局部变量。

IDA Pro

IDA（Interactive DisAssembler）Pro（以下简称 IDA）是一款强大的可执行文件分析工具，可以对包括但不限于 x86/x64、ARM、MIPS 等架构，PE、ELF 等格式的可执行文件进行静态分析和动态调试。IDA 集成了 Hex-Rays Decompiler，提供了从汇编语言到 C 语言伪代码的反编译功能，可以极大地减少分析程序时的工作量，其界面见图-3 和图-4。

COMPASS CTF TEAM 计算机与系统安全实验室

IDA Pro

图: 静态分析

IDA Pro

File Edit Jump Search View Debugger Options Windows Help

No debugger

Library function | Regular function | Instruction | Data | Unexplored | External symbol

Functions window | IDA View-A | Pseudocode-A | Hex View-1

Function name	Segment
__mingw_invalidParameterHandler	.text
pre_c_init	.text
pre_cpp_init	.text
__tmainCRTStartup	.text
WinMainCRTStartup	.text
mainCRTStartup	.text
verify_password	.text
main	.text
_decode_pointer	.text
_encode_pointer	.text
_setargv	.text
__mingw_raise_matherr	.text
__mingw_setusermatherr	.text
_matherr	.text
__report_error	.text
__write_memory_part_0	.text
_pei386_runtime_relocator	.text
__mingw_SEH_error_handler	.text
__mingw_init_ehandler	.text
fpreset	.text
__do_global_dtors	.text
__do_global_ctors	.text
__main	.text
__security_init_cookie	.text
__report_gsfailure	.text
__dyn_tls_dtor	.text

```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char v4; // [rsp+20h] [rbp-60h]
  int v5; // [rsp+42Ch] [rbp+3ACh]

  _main();
  v5 = 0;
  while ( 1 )
  {
    printf("please input password:     ");
    scanf("%s", &v4);
    v5 = verify_password(&v4);
    if ( !v5 )
      break;
    puts("incorrect password!");
  }
  puts("Congratulation! You have passed the verification!");
  return 0;
}
```

图: 从汇编语言到 C 语言伪代码的反编译

OllyDbg 和 x64dbg

OllyDbg 是 Windows 32 位环境下一款优秀的调试器，最强大的功能是可扩展性，许多开发者为其开发了具备各种功能的插件，能够绕过许多软件保护措施。但 OllyDbg 在 64 位环境下已经不能使用，许多人因此转而使用了 x64dbg。OllyDbg 和 x64dbg 的界面见图-5 和图-6。

COMPASS CTF TEAM 计算机与系统安全实验室

OllyDbg 和 x64dbg

图: OllyDbg 的界面

OllyDbg 和 x64dbg

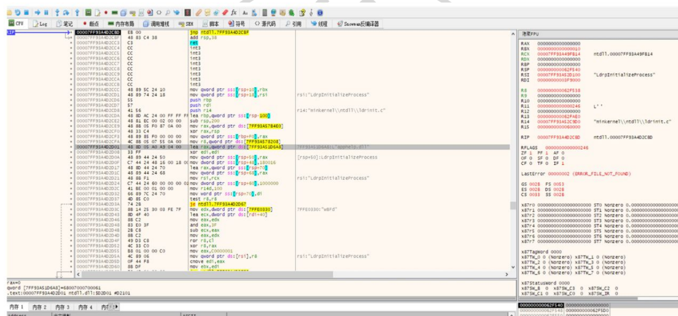


图: x64dbg 的界面

GNU Binary Utilities

GNU Binary Utilities（binutils）是 GNU 提供的二进制文件分析工具链，包含的工具见表-4。图-7 和图-8 为 binutils 中工具的简单应用例子。

GNU Binary Utilities

命令	功能	命令	功能
as	汇编器	nm	显示目标文件内的符号
ld	链接器	objcopy	复制目标文件，过程中可以修改
gprof	性能分析工具程序	objdump	显示目标文件的相关信息，亦可反汇编
addr2line	从目标文件的虚拟地址获取文件的行号或符号	ranlib	产生静态库的索引
ar	可以对静态库进行创建、修改和取出操作	readelf	显示 ELF 文件的内容
c++filt	解码 C++语言的符号	size	列出总体和 Section 的大小
dlltool	创建 Windows 动态库	strings	列出任何二进制的可显示字符串
gold	另一种链接器	strip	从目标文件中移除符号
nlmconv	可以转换成 NetWare Loadable Module 目标文件格式	windmc	产生 Windows 消息资源
		windres	Windows 资源编译器

图: 工具列表

GNU Binary Utilities

```
acdxvfsvd@manjaro  /home/acdxvfsvd  readelf -S /bin/ls
There are 25 section headers, starting at offset 0x21368:

节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标   链接   信息   对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         00000000000002a8  000002a8
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             00000000000002c4  000002c4
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             00000000000002e4  000002e4
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000000308  00000308
       00000000000000c8  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000000003d0  000003d0
       0000000000000c48  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000001018  00001018
       00000000000005ca  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           00000000000015e2  000015e2
```

图: 读取 section 标记

GNU Binary Utilities

```
acdxvfsvd@manjaro  /home/acdxvfsvd  objdump -d --stop-address=0x2
100 /bin/cat

/bin/cat：     文件格式 elf64-x86-64


Disassembly of section .init:

0000000000002000 <.init>:
    2000:       f3 0f 1e fa             endbr64
    2004:       48 83 ec 08             sub    $0x8,%rsp
    2008:       48 8b 05 19 7f 00 00    mov    0x7f19(%rip),%rax
  # 9f28 <__gmon_start__>
    200f:       48 85 c0                test   %rax,%rax
    2012:       74 02                   je     2016 <__progname@@GLIBC
_2.2.5-0x80aa>
    2014:       ff d0                   callq  *%rax
    2016:       48 83 c4 08             add    $0x8,%rsp
    201a:       c3                      retq
```

图: 导出汇编代码

GDB

GDB（GNU Debugger）是 GNU 提供的一款命令行调试器，拥有强大的调试功能，并且对于含有调试符号的程序支持源码级调试，同时支持使用 Python 语言编写扩展，一般用到的扩展插件为 gdb-peda、gef 或 pwndbg。图-9 为 GDB 启动时的提示信息，图-10 为使用 gef 插件时的命令行界面。

COMPASS CTF TEAM 计算机与系统安全实验室

GDB

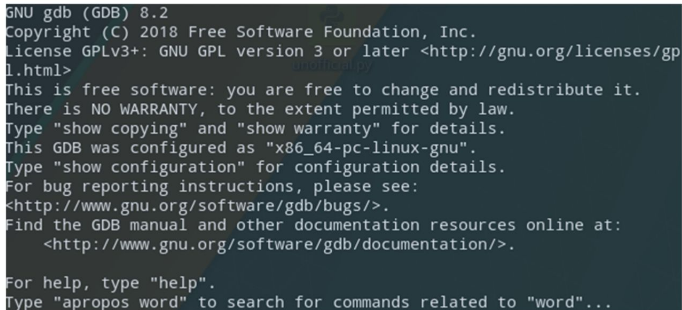


图: GDB 的界面

GDB

```
Program received signal SIGINT, Interrupt.
0x00007ffff7c7552d in pselect () from /usr/lib/libc.so.6
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────── registers ────
$rax   : 0xfffffffffffffdfe
$rbx   : 0x0
$rcx   : 0x00007ffff7c7552d  →  0x7b77fffff0003d48 ("H="?)
$rdx   : 0x0
$rsp   : 0x00007fffffffca00  →  0x0000000000000400
$rbp   : 0x00007ffff7f969d0  →  0x0000000000000000
$rsi   : 0x00007fffffffcb10  →  0x0000000000000001
$rdi   : 0x1
$rip   : 0x00007ffff7c7552d  →  0x7b77fffff0003d48 ("H="?)
$r8    : 0x0
$r9    : 0x00007fffffffca40  →  0x00007fffffffca90  →  0x0000000000000
000
$r10   : 0x0
$r11   : 0x246
```

图: 使用 gef 插件

GDB

```
$r12   : 0x00007ffff7d40860  →  0x00000000fbad2088
$r13   : 0x00007fffffffcb10  →  0x0000000000000001
$r14   : 0x00007fffffffca90  →  0x0000000000000000
$r15   : 0x00007fffffffca8f  →  0x0000000000000000
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overf
low resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x000
0
──────────────────────────────────────────────────────────── stack ────
0x00007fffffffca00│+0x0000: 0x0000000000000400    ← $rsp
0x00007fffffffca08│+0x0008: 0x0000000000000008
0x00007fffffffca10│+0x0010: 0x000055555588ef70  →  0x2024342e342d6873
("sh-4.4$"?)
0x00007fffffffca18│+0x0018: 0x000055555587d2a0  →  0x0000000000000000
0x00007fffffffca20│+0x0020: 0x000055555588ef78  →  0x0101010101010100
0x00007fffffffca28│+0x0028: 0x0000000000000008
0x00007fffffffca30│+0x0030: 0x000055555587d2a0  →  0x0000000000000000
0x00007fffffffca38│+0x0038: 0x00007ffff7d73090  →  <update_line+1856>
```

图: 使用 gef 插件

参考文献

[1] Nu1L. 从 0 到 1： CTFer 成长之路 [EB/OL]. https://book.douban.com/subject/35200558/.

[2] Portable Executable[EB/OL]. 2023. https://en.wikipedia.org/wiki/Portable_Executable.

[3] Executable and Linkable Format[EB/OL]. 2023. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

COMPASS CTF TEAM