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。

¥	Edit	As:	Hex	\sim	Run	Ser	ipt	\sim	Run	Tem	plat	e: JI	PG. b	t 🗸	₽								
																				DEF			
2900		3B	6B	6F	ED		C1	72	7E	E2	8E	D5	E7	6C	EE	77	DA.	;koi.Ár	~âŽÕçi	lîwÚ			
2910		EA	СС	EA	8D	E5	E9	39	37	45		96	35	C7	EA	0D	53	êÌê.åé9′	7E}-50	Çê.S			
2920		В9	D7	25	Β7		86	FA	47	90	FO	11	23	42	49	FC	A8	²×%·`†ú(G.ð.#3	BIü"			
2930		FO	7E	93	Α9	F8	F7	5F	8B	47	DЗ	6E	2D	EC	FC	СЗ	CB	ð~``©ø÷_4	< GÓn−i	ìüÃË			
2940		CA	4F	4F	C0	1A	F7	7F	84	7E	13	FO	EF	80	A0	BC	BE	Ê00À.÷.,	"~.ðiŧ	E 4%			
2950		D6	AC	06	Α9	AD	47	74	D0	47	71	Β4	32	A0		67	68	Ö¬.©-Gti	ÐGqʻ2	.gh			
2960		63	C7	7E	71	9A	Ε9	FA	C5	76	9C	94	9E	9E	66	1E	C2	cÇ~qšéú	Åvœ″ž	žf.Â			
2970		8A	6A	2E	2B	5F	23	1B	F6	6A	9B	C4	F7	5E	2F	9E	5B	Šj.+_#.0	öj≯Ä÷'	^/ž[
2980		58		FB	07	93	В6	E8	DE	8D	8D	E5	96	1F		28	FB	Xû.‴¶êi	Þå-	.p(û			
2990		D9	03	AF	15	EE	7F	D8	F1		ЗF	C5	6F	AB	58	7E	EF	Ù. [—] .î.øi	ñx?Åo	ĸX~ï			
29A0		49	D5	59	61	ВD	80		B1	4D	9C	24	AЗ	DO	12	76	9F	_IÕYa¥€p:	±MœŞ£i	Ð.vŸ			
29B0		A8	AC	3F	07	C9	E6		93	52	D4	BC	31	Α7	5A	DB	9B	~¬?.Éæx'	₩RÔ 4 1	ŞZÛ≻			
59C0		D8	D4	49	15	cc	EC	AA	AC	09	25	86	D5	ЗD		2B	B3	ØÔI.Ììª	¬.%†Õ	= } + 3			
29D0		6F	0F	DD	6A	85	4F	89	2F	C5	D4	00	86	16	56	D1	F9	o.Ýj0‰	/ĂÔ.†	.VÑù			
29E0		50	64	1C	8D	FC	96	7C	1F	52	07	FB	35	95	6A	9C	D2	Pdü-	.R.û5	•jœÒ			
29F0		D1	В7	7B	6E	69	4A	9F	2A	D5	25	6E	C7	47	45	14	56	Ñ {niJŸ	*Õ%nÇ(GE.V			
2A00		06	C1	45	14	50	01	45	14	50	07	FF		50		03	04	.ÁE.P.E	.P.yÙ	РК			
2A10		14	00	00	00	08	00	5B	7 A	86	4D	D2	72	80	71	C8	88	[3	z†MÒr(EqÈ^			
2A20		00			A0			06									65	;		.exe			
2A30		EC	ЗB	7F	74	53	55	9A	37	69	42	5F	DB	D7	26	85	04	ì;.tSUš	7iB_Ü	× &			
2A40		AB	FC	0A	42	95	11	64	ЗA	56	5D	10	AA	21	12	29	2E	≪ü.B∙.d	:V].*	!.).			
2A50		D1	24	10	C7	45	46	71	85	1A	2A	22	03	89	A 2	DB	6A	N\$. ÇEFq.		‱¢Ŭj			
2A60		СВ	23	6B	5F	2F	CF	C5	19	E3	CE	OF	FF	A0		B3	C7	E#k_/IA	.ãI.ÿ	`. 3Č			
2A70		1D	F1	8C	BB	EB	EE	61	E7	58	4C	95	35	D4	03	43	65	.ñŒ»ëîa	çXL•5(D.Ce			
2A80		7A	DC	3A	A7	73	28	2C	7B	F6	C5	74	B4	67	AD	C7	88	zU:§s(,	{öAt'	⊒-č.			
2A90		D1	B7	DF	F7	DD	97	26	14	9C	D9	7F	F6	BF	3E	CE	BB	N·B÷Y—&	.œU.ö	«I<3			
PAAC)h:	EF	DE	EF	D7	FD	EE	77	BF	FB	DD	EF	DE	94	C0	C6	03	î bî×ýîw,	¿üYïÞ'	"AE.			
PABC)h:	AC	8C	31	66	83	D7		18		CA	C4	E3	65	7F	FA	19		.;EAä	e.ů.			
PAC0	h:	80	B7	66	DE	AF	6B	D8	9B	15	A7	E6	1F	B5	AC	3B	35	€ fÞ kØ	>.§æ.;	μ¬;5			
PADO	h:	7F	43	74	DB	6E	CF	CE	5D	41	30	BA	EB	E1	C7	3D	81	.CtUnII	JO<°ea	aç=.			
PAEO)h:	30	BC	63	C7	13	31	CF	5F	6E	F5	EC	8A	EF	FO		DB	<4cÇ.11	nois	1810			
ZAFU	n:	E1	59	7D	EF	'/A	CF	E3	41	6C	D9	BA 	B4	BA	BA		A1	ay}izla	0100.0	ri			
Cempl	ate 1	Resul	lts –	- JP	G. bt																		
			Na	ne						Val	ue			Ste	ur t		Siz	e C	olor		Comme	nt	
≥ st	ruct	JPG	FILE	jpg	file								Oh			B	366h	Fg:	Bg:				
	enum	M_I	D SO	Iller	ker			M_S	50I	(FFD8	ih)		Oh			21	h	Fg:	Bg:				
2000	stru	ict A	PPO	appC									2h			1:	Zh	Fg:	Bg:				
>		ict I	QT d	lqt [C									141			4	5h	Fg:	Bg:				
>	stru	iet I	6 TQI	lqt [1									591	n.		4	5h	Fg:	Bg:				
>	stru	iet S	OFx	soft									9E)			1	3h	Fg:	Bg:				
1	5 1 2 11	10.0	ram 🛛 🖓	1753									- 31				L b	Ka.	8.2				

struct DQT dqt[1]		59h	45h	Fg:	Bg:
struct SOFx sof0		9Eh	13h	Fg:	Bg:
struct DHT dht[0]		B1h	21h	Fg:	Bg:
struct DHT dht[1]		D2h	B7h	Fg:	Bg: 📃
struct DHT dht[2]		189h	21h	Fg:	Bg:
struct DHT dht[3]		1 AAh	B7h	Fg:	Bg:
struct SOS scanStart		261h	Eh	Fg:	Bg: 📃
char scanData[10139]		26Fh	279Bh	Fg:	Bg: 🔤
enum M_ID EOIMarker	M_EOI (FFD9h)	2A0Ah		Fg:	Bg:
char unknownPadding[35162]		2A0Ch	895Ah	Fg:	Bg:

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

	kali	@kali:~/卜载/attachment _ □ :
文件 动作 编	辑 查看 帮助	文件(F) 编辑(E) 视图(V) 转到(G) 帮助
can Time: Target File: D5 Checksum: Tignatures:	2022-03-07 20:3 /home/kali/下载 5ad8668b8bcd9ac 411	34:09 /52/file/RE_Cirno.jpg J5b9e0944063aa4d33
ECIMAL	HEXADECIMAL	DESCRIPTION
		桌面 RE_Cirno.jpg
0764 ressed size: 5904	0×0 0×2A0C 35016, uncompres 0×B350	JPEG image data, JFIF standard 1.01 Zip archive data, at least v2.0 to extract, com ssed size: 172091, name: re.exe End of Zip archive, footer length: 22
can Time: arget File: D5 Checksum: ignatures:	2022-03-07 20:3 /home/kali/下 载 6df009ab420867a 411	34:09 /attachment/_RE_Cirno.jpg.extracted/re.exe 99248befca5f829bb3
ECIMAL	HEXADECIMAL	DESCRIPTION
304524	0×0	Microsoft executable, portable (PE)
—(kali⊛kali —\$	i)-[~/下载/attac l	nment]

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。

CodeBrowser: pj52:/re.exe		
File Edit Analysis Graph Nav	wigation Search Select Tools Window Help	
	▶ ↓ IDULFXXB- (油油) ♡ ~ / 節创四面 G 击 O 图 ◆ □ □ 品 ●	
Program Trees 🖬 🕼 🏠 🗙	🖃 Listing: re.exe 🐂 🐘 💽 🕫 📓 🗐 🔹 🗙 Sylecospiler	🌮 🕞 📑 👘 🖛 🗙
Crosses Construction Constr	I do nalyze I do Function I do Function	
←·→· *** N.8x	0040014 00 00 dw 0h e_ip Initial IP	
Θ	UU4UUULE UU UU UU UU E_CS IDILAI (P	
A Data Types ⊕-∰ BuiltInTypes ⊕-∰ Ore.exe	Canalar - Scripting	, @ 2/ ×
Riller B		20.30
IDA - re	re.exe C1 C1.Users\Admini C1.Tools\ghidra 🕢 Ghidra: pj52 🛃 CodeBrowser. p	

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。

a	b String Se	arch - 1 items (of 2	71) - [re.exe, Minimum size = 5, J	Align = 1]		A Q		7 8	∄≣≯
	. Location	🖹 Label	Code Unit	String View	Str	Le	Is Word		
1	[00422fa4	s_pause_0	ds "pause"	"pause"	string	6	true		



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。

	s_pause_00422	fa4	<pre>XREF[1]: FUN_0040f350:0040f459(*)</pre>
00422fa4 70 61 75	ds	"pause"	
73 65 00			
00422faa 00	22	00h	
00422fab 00	22	00h	
	DAT_00422fac		<pre>XREF[1]: FUN_0040f350:0040f44c(*)</pre>
00422fac e7	??	E7h	
00422fad f7	22	F7h	
00422fae c2	22	C2h	
00422faf b6	22	B6h	
00422fb0 c5	??	C5h	
00422fb1 b5	??	B5h	
00422fb2 bd	22	BDh	
00422fb3 b4	22	B4h	

图 3-6 在 Ghidra 中找到关键函数

A 7 70						4) -			
isting: re	e. exe				- 🖳 🕅 I	8 -	×	ş Decompile: FUN_0040£350 🌮 📄 🔯 🗮 🔫	x
1	XUR	EAX, UX9					2	2 10001 64[9] - 0720.	
	MOM	dword ptr [EBP + local_6c],EAX						/ 100al_04[5] = 0x50,	
	MOM	ECX, dword ptr [EBP + local_6c]					4	0 local_00[10] = 0x51;	
	VOM	dword ptr [EBP + local_70], ECK						9 10Cal_64[11] = 0x69;	
i L	JMP	LAB_0040f41f						U local_64[12] = 0x75;	
							3	1 local_64[13] = 0x76;	
	AB_0040f44c		XREF[1]:	0040f42c(j)			3	2 local_64[14] = 0x65;	
	PUSH	DAT_00422fac		= E7h				3 local_64[15] = 0x30;	
							3 B	4 local_64[16] = 0x71;	
	CALL	FUN_00401150		int FUN_00	401150 (byte 4	* param_1	3	5 local_64[17] = 0x5f;	
		_					3	6 local_64[18] = 99;	
	ADD	ESP,0x4					3	7 local_64[19] = 0x2f;	
	PUSH	s pause 00422fa4		= "pause"			3	<pre>8 local_64[20] = 0x5c;</pre>	
							3	<pre>9 local_64[21] = 0x74;</pre>	
	CALL	FUN 0040f240		HANDLE FUN	0040f240(ch	ar * pars	4	<pre>0 local_64[22] = 0x5d;</pre>	
							4	<pre>1 local_64[23] = 0x66;</pre>	
	ADD	ESP.0x4					4	<pre>2 for (local_68 = 0; local_68 < 0x18; local_6</pre>	8
	XOR	FAX FAX					4	<pre>3 local_70 = local_64[local_68] + 9U ^ 9;</pre>	
	POP	EDI					- 4	<pre>4 local_6c = local_70;</pre>	
	POP	FST						5 }	
	DOD	FRV					4	6 FUN_00401150 (#DAT_00422fac);	
	TOP	EDA DESA					4	7 FUN_0040f240("pause");	
	ADD	LSF, URAC					E 4	8 local_64[23] = 0x40f478;	
	CHID	PDD PCD				-	- 4	9 chkesp();	-
	4					•		(•
							_		

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 $64[1] = 0 \times 5e;$ $local_64[2] = 0x61;$ $local_{64[3]} = 0x72;$ $local_{64}[4] = 0x67;$ $local_64[5] = 0x2f;$ local $64[6] = 0 \times 6b;$ $local_{64}[7] = 0x72;$ $local_64[8] = 0x41;$ $local_64[9] = 0x30;$ local $64[10] = 0 \times 31;$ local $64[11] = 0 \times 69;$ local 64[12] = 0x75;local 64[13] = 0x76;local $64[14] = 0 \times 65;$ $local_{64}[15] = 0x30;$ local $64[16] = 0 \times 71;$ local $64[17] = 0 \times 5f;$ local 64[18] = 99; local 64[19] = 0x2f;10cal 64[20] = 0x5c;local 64[21] = 0x74;10cal 64[22] = 0x5d;local $64[23] = 0 \times 66;$ local 68 = 0;

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

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{wvg0saelltof
fotlleas0gvw{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。

🕖 Functions 🗆 💿 😒	S IDA View-A S Hex View-1 S A Structures S I Enums S M Imports S A Exports
Function name	
🗷 _main	
☑_printf	; Attributes: bp-based frame
	; intcdecl main_0(int argc, const char **argv, const char **envp)
✓ start	_main_0 proc near
	var AC= byte ptr -0ACb
	var_6C= dword ptr -6Ch
<pre>Z stbuf</pre>	var_68= dword ptr -68h
ftbuf	var_b4= dword ptr -b4n
Z sub 401500	var_oo= dword ptr -och
A write char	var_58= dword ptr -58h
	var_54= dword ptr -54h
∠ _write_multi_char	var_50= dword ptr -50h
☑ _write_string	var_4c= dword ptr -4ch
	var_so and ptr son
	var_40= dword ptr -40h
The set also at any	var_3C= dword ptr -3Ch
Line 1 of 197	Var_38= dword ptr -38h
🚊 Graph overview 🗆 🗿 🔇	var_34= dword ptr - 34n
	var_2C= dword ptr -2Ch
	var 28= dword ptr -28h
	var_24= dword ptr -24h
100.00% (-225,12) (424,258) 0000F350 000000000040F350: _main_0 (Synchronized with Hex View-1)

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

Function name	.text:0040F41F	
7 main	.text:0040F41F loc_40F41F: ; CODE XREF: _main_0+FAij	
∃_priptf	text:0040F41F mov eax [ebp+var_64]	
	toxt.0040F422 duu cdx, 1	
<pre></pre>	text:0040F425 milliov [ebp+var_o4], eax	
🗹 start	text:0040F428 loc 40F428 · CODE XRFE · main 0+CD+i	
7 amsg exit	text:0040F428 cmp [ebp+var 64]. 18h	
	- text:0040F42C jge short loc 40F44C	
I _rast_error_exit	text:0040F42E mov ecx. [ebp+var 64]	
☑stbuf	.text:0040F431 mov edx, [ebp+ecx*4+var 60]	
7 ftbuf	edx, 9	
Z cub 401500	text:0040F438 mov [ebp+var_68], edx	
Z SUD_4015C0	text:0040F43B mov eax, [ebp+var 68]	
团_write_char	• .text:0040F43E xor eax, 9	
🗷 write multi char	• .text:0040F441 mov [ebp+var_68], eax	
Z weite steine	.text:0040F444 mov ecx, [ebp+var_68]	
	• .text:0040F447 mov [ebp+var_6C], ecx	
	text:0040F44A jmp short loc_40F41F	
🗇 get int64 arg	.text:0040F44C ;	
Z set chest are	.text:0040F44C	
	.text:0040F44C loc_40F44C: ; CODE XREF: _main_0+DC↑j	
🗹initstdio	text:0040F44C push offset Format ; "ç"	
Image: Provide the second s	.text:0040F451 call _printf	
T cub 402640	.text:0040F456 add esp, 4	
Z SUD_402040	.text:0040F459 push offset Command ; "pause"	
<pre>了CrtSetReportMode</pre>	text:0040F45E Call _system	
Line 1 of 107	0000F45E 000000000A0F45E. main 0+10E (Sunchronized with Hey View-1)	



F Functions		0 0			0	IDA View-A	🕲 📳 Pseudo	code-A	🕲 🖸 Hex View-	1 🛛 🕲 🛋 Structures	0 🗄	Enums
Function name			1 1	int_	_cdecl	main_0(i	int argc,	const	char **ar	gv, const cha	**en	vp)
🗹 _main				2 { 3 int	· i : //	[esp+54]	1] [ebp-6	4h1				
☑ _printf			4	1		feeb eu	1 Level e					
☑chkesp				for	(i =	0; i < 2	24; ++i)					
🗹 start				pr	ntf(Fo	rmat);						
🗷amsg_exit			• 8	sys	stem("p	ause");						
<pre>Image: Content Co</pre>	:		10	rei	urn 0;							
团stbuf												
☑ftbuf												
🗷 sub_4015C0												
🗷 _write_char												
🗷 _write_multi_cha	эг											
🗷 _write_string												
☑ _get_int64_arg												
_get_short_arg												
🗷initstdio												
🗷endstdio												
🗷 sub_402640												
<pre>CrtSetReportMo </pre>	ode											
Line 1 of 197				0000F3	50 main (:1 (40F350)						

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。

0000h:	7F	45	4C	46	02	01	01	00	00	00	00	00	00	00	00	00	.ELF
0010h:	02	00	3E	00	01	00	00	00	70	08	40	00	00	00	00	00	>p.@
0020h:	40	00	00	00	00	00	00	00	A0	43	00	00	00	00	00	00	@ C
0030h:	00	00	00	00	40	00	38	00	09	00	40	00	1F	00	1C	00	@.8@
0040h:	06	00	00	00	05	00	00	00	40	00	00	00	00	00	00	00	
0050h:	40	00	40	00	00	00	00	00	40	00	40	00	00	00	00	00	0.00.0
0060h:	F8	01	00	00	00	00	00	00	F8	01	00	00	00	00	00	00	øø
0070h:	08	00	00	00	00	00	00	00	03	00	00	00	04	00	00	00	
0080h:	38	02	00	00	00	00	00	00	38	02	40	00	00	00	00	00	88.0
0090h:	38	02	40	00	00	00	00	00	1C	00	00	00	00	00	00	00	8.@
00A0h:	1C	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00	
00B0h:	01	00	00	00	05	00	00	00	00	00	00	00	00	00	00	00	
OOCOh:	00	00	40	00	00	00	00	00	00	00	40	00	00	00	00	00	
00D0h:	1C	24	00	00	00	00	00	00	1C	24	00	00	00	00	00	00	.\$\$
00E0h:	00	00	20	00	00	00	00	00	01	00	00	00	06	00	00	00	
OOFOh:	10	2E	00	00	00	00	00	00	10	2E	60	00	00	00	00	00	`
0100h:	10	2E	60	00	00	00	00	00	10	06	00	00	00	00	00	00	`
0110h:	A8	06	00	00	00	00	00	00	00	00	20	00	00	00	00	00	
0120h:	02	00	00	00	06	00	00	00	28	2E	00	00	00	00	00	00	(

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
rootawhoami:~/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 6, 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:
<pre>1.battle with slime. 2.battle with boss. 3.go to the shop. +</pre>
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。

: Attributes: bp-based frame public boss boss proc near :unwind (push rbp mov rbp, rsp mov eax, cs:bossexist cmp eax, 2 ile short loc 401641			
edi, offset aTheDragonAppea : "The dragon appears, its Combat Effect _puts Eax, cs:eff eax, 6#4240h short Loc_401628	v" loc_401641: mov eax.cs:bossex1st cmp eax.z jnc short loc_401641:		
Bast fast loc_401628: mov mov edi, offset aYouDie : "You die!" mov eax, off call _purintf mov edi, 1 call _ext			
	<pre>def = = = = = = = = = = = = = = = = = = =</pre>		
<pre>i. offset aTheDragonIsBea : "The dragon is beated! flag is partly de" uts x, cs:bossexist x, 1 bossexist, eax x, 0</pre>	add edf, offset aTheDragonIsBea; "The dragon is beated! flag is partly de" add add		



1	_int64 slime()
2	{
3	int v0; // ebx
4	<pre>int v2; // [rsp+Ch] [rbp-14h]</pre>
5	
6	v0 = eff - 64;
• 7	$v^2 = v^0 + rand() \% 128;$
• 8	if (v2 == eff)
• 9	++v2;
• 10	<pre>printf("A slime is refreshing , its Combat Effectiveness is %d\n", (unsigned int)v2);</pre>
• 11	if (v2 > eff)
12	{
• 13	<pre>printf("You die!");</pre>
• 14	exit(1);
15	}
• 16	<pre>puts("The slime is beated, you get 1 coin.");</pre>
• 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。

```
1
   2
     {
         int64 result; // rax
   3
   4
•
       if ( bossexist <= 2 )
   5
   6
       {
•
         if ( bossexist == 2 )
  7
   8
9
           puts("The dragon appears, its Combat Effectiveness is 3000000.");
if ( eff <= 3000000 )</pre>
10
  11
           {
• 12
              printf("You die!");
• 13
              exit(1);
  14
           }
15
           puts("The dragon is beated! flag is partly decrypted...");
16
            --bossexist:
0 17
           return decrypt2();
 18
  19
         else
  20
         {
0 21
            result = (unsigned int)bossexist;
0 22
           if ( bossexist <= 1 )
  23
           {
24
              puts("The dragon appears, its Combat Effectiveness is 5000000.");
0 25
              if ( eff <= 5000000 )
 26
              {
               printf("You die!");
27
28
                exit(1);
 29
              }
30
              puts("The dragon is beated! combining flag and print...");

31
32

              --bossexist;
              return decrypt3();
 33
           1
  34
35
         }
       }
  36
       élse
  37
       {
         puts("The dragon appears, its Combat Effectiveness is 1000000.");
if ( eff <= 1000000 )</pre>
38
• 39
 40
         {
           printf("You die!");
• 41
• 42
           exit(1);
 43
• 44
         puts("The dragon is beated! flag is partly decrypted...");
• 45
          --bossexist;
46
         return decrypt1();
 47
• 48
       return result;
• 49 }
```

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:0000000004015F9		cmp	eax, 0F4240h
.text:00000000004015FE		jle	short loc_401628
.text:0000000000401600		-	
.text:0000000000401600 1	oc 401600:		: DATA XREF:
.text:0000000000401600		mov	edi. offset aTheDragonIsBea :
text:0000000000401605		call	nuts
		curr	_pacs
000015F9 0000000004015F9: boss+1	IF (Synchronized with	Hex View-1)	
图 3-20	hoss 函数的	第一个:	关键判断指今
EI 0 10	DODD EI XAIJ	// / /	
<pre>.text:00000000040165C</pre>		cmp	eax, 2DC6C0h
toxt:00000000000000000000000000000000000		110	
		lie	short loc 401688
<pre>.text:0000000000401663</pre>		mov	short loc_401688 edi. offset aTheDragonIsBea :
<pre>.text:0000000000401001 .text:0000000000401663 .text:00000000000401668</pre>		mov call	short loc_401688 edi, offset aTheDragonIsBea ; puts
<pre>.text:0000000000401663 .text:0000000000401663</pre>		mov call	<pre>short loc_401688 edi, offset aTheDragonIsBea ; _puts</pre>

图 3-21 boss 函数的第二个关键判断指令

.text:00000000004016BC	cmp	eax, 4C4B40h
.text:00000000004016C1	jle	short loc_4016E8
.text:00000000004016C3	mov	edi, offset aTheDragonIsBea_0 ;
.text:00000000004016C8	call	puts
000016BC 000000000000016BC, boggtE2 (Sunchronized with	How Wiew-1)	

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
<pre>stub command (use set kernel-vmmap-via-page-tables</pre>	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> pwndbg>	set \$eax = 0x4c4b44 info r	
rax	0x4c4b44	5000004
rbx	0×0	Θ
rcx	0x7ffff7e9aa37	140737352673847
rdx	0x1	1
rsi	0x1	1
rdi	0x7ffff7fa1a70	140737353751152
rbp	0x7fffffffe3b0	0x7fffffffe3b0
rsp	0x7fffffffe3b0	0x7fffffffe3b0
r8	0x7ffff7fa1a70	140737353751152
r9	0×0	Θ
r10	0x7ffff7f44ac0	140737353370304
r11	0x246	582
r12	0x7fffffffe518	140737488348440
r13	0x401a40	4201024
r14	0×0	Θ
r15	0x7ffff7ffd040	140737354125376
rip	0x4015f9	0x4015f9 <boss+31></boss+31>
eflags	0x202	[IF]

图 3-25 设置寄存器 eax 的值 1

使用命令 infor 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
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, 0x0000000000040165c 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	
hmungs	THIO L	
rax	0x4c4b44	5000004
rbx	0×0	Θ
rcx	0x7ffff7e9aa37	140737352673847
rdx	0x1	1
rsi	0×1	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.
Trout the number of your chiece,2
The dragon appears, its Combat Effectiveness is 5000000.
Breakpoint 3 0x0000000000000000000000000000000000

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。

<pre>pwndbg> set \$ea pwndbg> info f</pre>	ax = 0x4c4b44		
Stack level 0,	frame at 0x7fffffff	e3c0:	
rip = 0x4016b	c in boss; saved rip	= 0x4018b9	
called by fram	<pre>me at 0x7fffffffe3f0</pre>		
Arglist at 0x	7fffffffe3b0, args:		
Locals at 0x7	fffffffe3b0, Previou	s frame's sp is 0	x7fffffffe3c0
Saved register	rs:		
rbp at 0x7fffffffe3b0, rip at 0x7fffffffe3b8			
pwndbg> info r			
rax	0x4c4b44	5000004	
rbx	0×0	0	
rcx	0x7ffff7e9aa37	140737352673847	
rdx	0x1	1	
rsi	0x1	1	
rdi	0x7tttt7ta1a70	140737353751152	
rbp	0x7fffffffe3b0	0x7tttttte3b0	

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。



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。



图 3-31 文件的 store 函数

int 类型可表示的十进制数据范围是 $-2147483648 \cong 2147483647$,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-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 3-37 使用 x64dbg 脱壳找到程序的 OEP

兼 ez_re.exe - PID: 23784 - 模块: ez_re.exe - 线和	a: 主线程 25208 - x64dbg		- 🗆 X	
文件(E) 视图(V) 调试(D)	跟踪(N) 插件(P) 收藏夹	(I) 选项(Q) 帮助(H) Jan 6 20	024 (TitanEngine)	
🖻 🔊 🔳 🔶 II 💠 😒	1 1 - 2 5 2 5 2	A fx # A2 🖺 🗐 🥮		
■ CPU 2 日志 1 笔记 •	断点 💻 内存布局 🗊 调	用堆栈 🔤 SEH链 🔟 脚本 📓 符	号 ◇ 源代码 🔎 引用 🛸 线程 📕 ◀ ▶	
BIE 000007FE62F8ELEE0 00007FE62F8ELEE4	48:83EC 28 E8 1F040000	sub rsp.28 call ez_re.7FF62F8E2318	隐藏FPU	
O00077F627851F2 O0007F7627851F2 O	48:8264 28 CC CC CC CC CC CC CC CC CC C	<pre>add rep.28 ing ez.re.7F6262E81074 int3 unch rbx sub rbx sub rbx sub rbx sub rcx.rbx add rep.20 mov rcx.rbx add rep.20 mov rcx.rbx add rep.40 mov rcx.rbx ad</pre>	RAX 0000005888/FCC4 RAX 000007FDA7EC7674 RAX 00007FDA7EC7674 RAX 00007FDA7EC7674 RAP 000000000000 RAP 000000000000 RAP 00000058885FF488 RAD 000000000000 RAI 000000000000 RAI 000000000000 RAI 0000000000000 RI 0000000000000 RI 0000000000000 RI 0000000000000 RI 0000000000000 RI 00000000000000 RI 000000000000000 RI 000000000000000000000000000000000000	
00007FF62F8E1F49 00007FF62F8E1F4B	85c0 74 21	test eax,eax je ez_re.7FF62F8E1F6E	默认 (x64 fastcall) · 5 ↓□解锁	
• 000007776375871740	1 6548-860435 30000000	Lassr nos mused of a 1801	1: rcx 00007FFDA7ECFD74 ntd11.00007FFDA7ECFD74 2: rdx 000000000000000 00000000000000	
rsp-0000005895EF48 3; r8 0000005895EFA8 0000005895EFA8 0000005895EFA8 000000050895EFA8 000000000000000000000000000000000000				
eeee:00007FF62F8E1EF0 ez_re.exe:\$1EF0 #0			5: [PSp+28] 000000000000000000000000000000000000	
中存 1	\$\vec{4}{3}\$ \$\vec{m}{m}\$ \nlime{price 4} \$\vec{m}{m}\$ \nlime{price 5} \$\vec{1}{10}\$ \$\vec{1}{10}\$ \$\vec{1}{10}\$ \$\vec{1}{10}\$ \$\vec{1}{10}\$	● 监视 1 座 局 ● 3X4Y x*55 0000005886 00000005886 00000005886 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 0000000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 000000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 00000586 0000586 00000586 0000586	Construction Construction Con	
命令: ↔<@用显母分隔 (@汇编语言), nov eax, ebx 默认 ·				
已暂停 INT3 断点于 ez_re. 00007FF62F8EAF40! [] 已调试时间: 0:00:24:00				

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。

👷 ez.r.e.exe - PID: 23784 - 模块: ez.r.e.exe - 低能主线程 25208 - x64dbg ロ ×						
文件(E) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选	项(<u>0</u>) 帮助(<u>H</u>)	Jan 6 2024 (TitanEng	ine)			
🗁 🔊 🔳 🔿 🖩 🍷 💫 🐏 🎍 🛊 📲 🖉 层 🖉 🦧 fx 🕏	# A2 📙 🗐 🧐					
■ CPU 🕞 日志 🗈 笔记 • 断点 = 内存布局 🗊 调用堆栈	🖷 SEH链 🛛 脚本	: 📓 符号 🗠 源代码	♀引用 ≫线程 ▲◀▶			
OOD07FE02FSELEE0 48:83EC 28 sub rsp.2 • 00007FE62F8ELEF4 E8 1F040000 call ez.r	8 e.7FF62F8E2318		隐藏FPU			
• 00007 Scylla x64 v0.9.8	- 🗆 🗙	RAX 00000005BI RBX 000000000	89EFDC8			
* 00007F File Imports Trace Misc Help						
OO007r O0007r O0007r O0007r Attach to an active process	100007 100007 10007					
23784 - ez_re.exe	Pick DLL	RDI 000000000	39FEDA8			
00007 00007 Imports	• 00007r • 00007r Imports R10					
• 00007r • 00007r • ernel32.dll (15) FThunk: 00003000 • 00007r • msvcp140.dll (18) FThunk: 00003080	R12 00000000 R13 00000000	0000000 0000000				
00007f vcruntime140.dll (8) FThunk: 00003118 vcruntime140.1 t dll (1) FThunk: 00003160	R14 000000000 R15 000000000	000000				
00007r 00007	RIP 00007FF62	8E1EF0 ez_re.00007FF62F8E1EF0				
	ZE 0 PE 1 AE OF 0 SE 0 DF	0				
00007f 00007f		CE 1 TF 0 IF	1 2000 (canon cuccese)			
→• 00007f • 00007f	00007F 00007F					
00007F 00007F 00007F		GS 0028 ES 005	3			
O0007F Show Invalid Show Suspect	Clear	默认 (x64 f	astcall) · <u>5</u> 中解锁			
rsp=0000005889EFE48 IAT Info Actions Dump		1: rcx 0000/FFDA/ECFD/4 ntd11.0000/FFDA/ECFD/4 2: rdx 00000000000000000000000000000000000				
28 '(' OEP 00007FF62F8E1EF0 IAT Autosearch Autotrace	Dump PE Rebuild	4: r9 000000000 5: [rsp+28] 0000	000000 00000000000000000000000000000000			
eeee: 0000/FF62F8EIEF0 VA 00007FF62F8E3000 Get Imports	Fix Dump	0005BB9EFE48	▼ FDA5C4257D 1近回至(kerne132.Ba			
		00058B9EFE50 000000 0005BB9EFE58 000000	0000000000			
2021 Log 0000000058896FE40 00 0 000000058896FE40 00 0	0005889EFE60 000000 0005889EFE68 000000 0005889EFE68 0000000	00000000000				
000000058B9EFE60 00 0 IAT Search Adv: Pound 67 (0x43) possible IAT entries. 00000005B89EFE70 00 0 IAT Search Adv: Possible IAT first 00007FF62F8E3080 last 00007FF62F8E3288 entries.	0005889EFE70 0005889EFE78 0005889EFE78 000000	FDA7E8AA58 运回到 ntdll.RtlUs				
00000005889EFE80 00 0 IAT Search Adv: IAT VA 00007FF62F8E3000 RVA 00000000000000000000000000000000	0005889EFE88 000000 0005889EFE90 000000	000000000000000000000000000000000000000				
00000005889FEFEA0 00 0 DIFFECT IMPORTS - Found 0 possible direct imports with 0 unique APIs!	0005889EFE98 000000 0005889EFEA0 000000	0000000000				
00000005895FFE0 21 0 0000005889FFE0 0 0 0	0005889EFEA8 000000 0005889EFEB0 000000 0005889EFEB0 000000	0000000000				
Imports 72 × Invalid: 3 Imagebase: 00007Ff62F8E0000 ez re.exe						
命令: ■今使用当学期(使用其读声) Boy eas, ebx 默认 ・ 文化 ·						
已習仔 LN13 断点于 ez_re. 00007Ff62F8EAF40! 已调试时间: 0:00:25:02						

图 3-39 使用插件修复目标程序