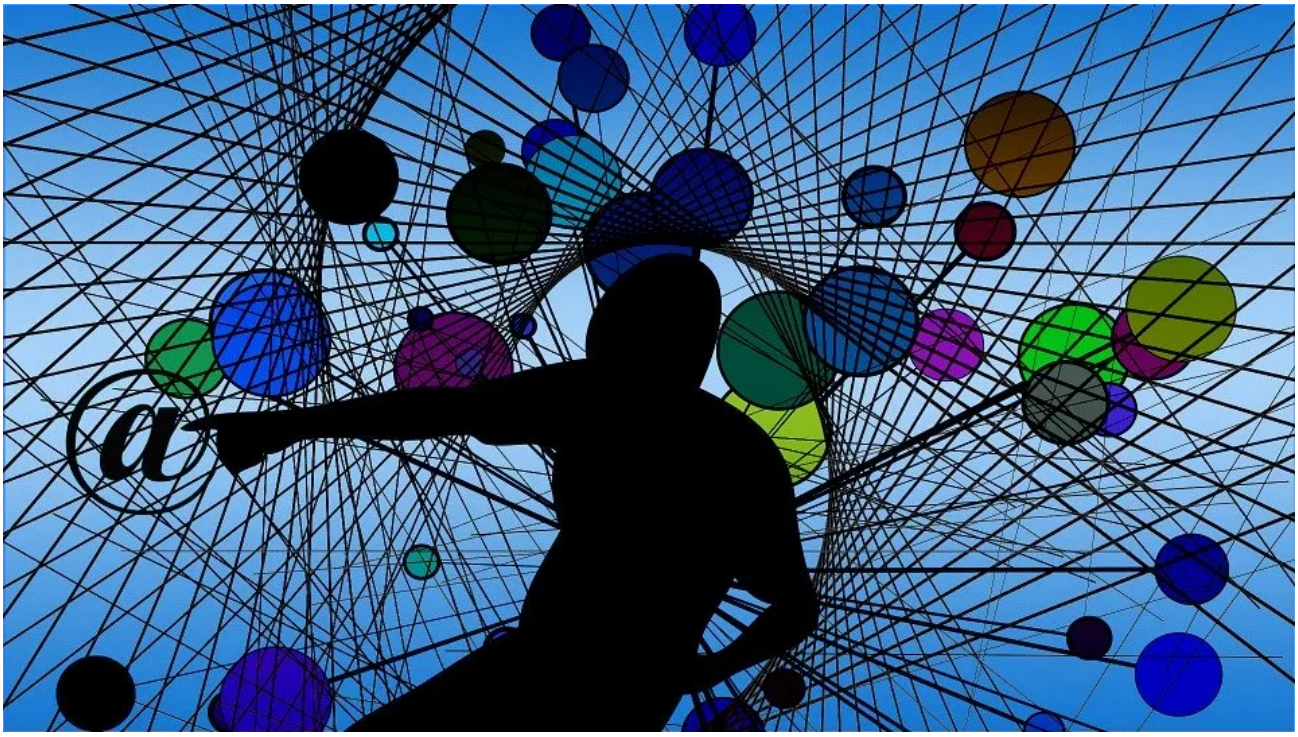
# 深入窥探动态链接

1Oin0 看雪学院 4月21日



本文为看雪论坛优秀文章
看雪论坛作者ID：1Oin0

## 0x00 前言

本文主要分析了在延迟绑定中，调用某函数之后如何找到正确的地址。文章中深入的分析了这个过程，并且分析完之后针对该链接介绍了一些攻击手法和程序所作的一些保护。

## 0x01 基础知识

### >>>> 动态链接

在动态链接方式实现以前，普遍采用静态链接的方式来生成可执行文件。如果一个程序使用了外部的库函数，那么整个库都会被直接编译到可执行文件中。

ELF 支持动态链接，这在处理共享库的时候就会非常高效。

当一个程序被加载进内存时，动态链接器会把需要的共享库加载并绑定到该进程的地址空间中。

随后在调用某个函数时，对该函数地址进行解析，以达到对该函数调用的目的。

## >>>> 两个表

### *1. PLT表(Procedure Linkage Table)*

**（1）简介**： 全局偏移表，在程序中以 .plt 节表示，该表处于代码段，每一个表项表示了一个与要重定位的函数相关的若干条指令，每个表项长度为 16 个字节，存储的是用于做延迟绑定的代码。

**（2）结构简介**：

```
PLT[0]  --> 与每个函数第一次链接相关指令
例：
0x4004c0:
0x4004c0:  ff 35 42 0b 20 00        push   QWORD PTR [rip+0x200b42]      // p
0x4004c6:  ff 25 44 0b 20 00        jmp    QWORD PTR [rip+0x200b44]      // j
0x4004cc:  0f 1f 40 00              nop    DWORD PTR [rax+0x0]
即：
    第一条指令为 push 一个值，该值为 GOT[1] 处存放的地址，
    第二条指令为 jmp 到一个地址执行，该值为 GOT[2] 处存放的地址

PLT[1]  --> 某个函数链接时所需要的指令,与 got 表一一对应
例：
0x4004d0 <__stack_chk_fail@plt>:
0x4004d0:  ff 25 42 0b 20 00        jmp    QWORD PTR [rip+0x200b42]  // jmp G
0x4004d6:  68 00 00 00 00           push   0x0                       // push
0x4004db:  e9 e0 ff ff ff           jmp    0x4004c0 <_init+0x20>     // jmp P
即：
    第一条指令为: jmp 到一个地址执行，该地址为对应 GOT 表项处存放的地址，在下文中会具
    第二条指令为: push 一个值，该值作用在下文提到
    第三个指令为: jmp 一个地址执行，其实该地址就是上边提到的 PLT[0] 的地址，
                也就是说接下来要执行 PLT[0] 中保存的两条指令
 .
```

```
.
.
```

### *2. GOT表(Global Offset Table)*

**（1）简介：** 过程连接表，在程序中以 .got.plt 表示，该表处于数据段，每一个表项存储的都是一个地址，每个表项长度是当前程序的对应需要寻址长度（32位程序：4字节，64位程序：8字节）。d_tag = DT_PLTGOT

**（2）结构简介**：

```
GOT[0]  --> 此处存放的是 .dynamic 的地址；该节(段)的作用会在下文讨论
GOT[1]  --> 此处存放的是 link_map 的地址；该结构也会在下文讨论
GOT[2]  --> 此处存放的是 dl_runtime_resolve 函数的地址
GOT[3]  --> 与 PLT[1] 对应，存放的是与该表项 (PLT[1]) 要解析的函数相关地址，
            由于延迟绑定的原因，开始未调用对应函数时该项存的是 PLT[1] 中第二条指令的
            当进行完一次延迟绑定之后存放的才是所要解析的函数的真实地址
GOT[4]  --> 与 PLT[2] 对应，所以存放的是与 PLT[2] 所解析的函数相关的地址
 .
 .
 .
```

### *3. 两个表之间的关系*

```
GOT[0]: .dynamic 地址                       PLT[0]: 与每个函数第一次链接相关指令
GOT[1]: link_map 地址
GOT[2]: dl_runtime_resolve 函数地址
GOT[3]  -->  PLT[1]    // 一一对应
GOT[4]  -->  PLT[2]    // 相互协同，作用于一个函数
GOT[5]  -->  PLT[3]    // 一个保存的是该函数所需要的延迟绑定的指令
GOT[6]  -->  PLT[4]    // 一个是保存个该函数链接所需要的地址
 .             .
 .             .
 .             .
```

## >>>> 一个段（节）三个节

在下面只对一些接下来要用到的结构体成员做一些中文解释。

### *1. .dynmic*

**（1）介绍**：因为在加载过程中，.dynmic 节整个以一个段的形式加载进内存，所以说在程序中的 .dynmic 节也就是运行后的 .dynmic 段。

该段主要与动态链接的整个过程有关，所以保存的是与动态链接相关信息，此处主要用于寻找与动态链接相关的其他节( .dynsym .dynstr .rela.plt 等节)。

该段保存了许多 Elf64_Dyn 结构，该数据结构保存了一些其他节的信息。下面展示该段所保存的数据结构。p_type = PT_DYNAMIC（值为 0x2）的段。

**（2）结构**：

```
// 该结构都有 64 位程序和 32 位程序的区别，不过大致结构相似，此处只讨论 64 位程序中的
// /usr/include/elf.h

 typedef struct
 {
   Elf64_Sxword  d_tag;                  /* Dynamic entry type */
                                          // d_tag 识别该结构体表示的哪一个节，通
   union
     {
       Elf64_Xword d_val;                /* Integer value */
                                         // 对应节的地址，用于存储该结构体表示下的
       Elf64_Addr d_ptr;                 /* Address value */
                                         // 一般于上一个字段表示的值相同，所以笔者
     } d_un;
 } Elf64_Dyn;
```

### *2. .dynsym*

**（1）介绍**：

动态符号表，存储着在动态链接中所需要的每个函数所对应的符号信息，每个结构体分别对

应一个符号 (函数) 。结构体数组。d_tag = DT_SYMTAB(值为 0x6) 的节。

**（2）结构：**

```
typedef struct
{
  Elf64_Word    st_name;          /* Symbol name (string tbl index) */
                                  // 保存着该函数函数名在 .dynstr 中的偏移，可以
  unsigned char st_info;          /* Symbol type and binding */
  unsigned char st_other;         /* Symbol visibility */
  Elf64_Section st_shndx;         /* Section index */
  Elf64_Addr    st_value;         /* Symbol value */
                                     // 如果这个符号被导出，则存有这个导出函数的虚
  Elf64_Xword   st_size;          /* Symbol size */
} Elf64_Sym;
```

### *3. .dynstr*

**（1）介绍**：动态字符串表，表中存放了一系列字符串，这些字符串代表了符号的名称，在此处可以看成函数名，以空字符作为终止符。

该结构是一个字符串数组。d_tag = DT_STRTAB(值为 0x5) 的节。

**（2）结构**：一个字符串数组

### *4. .rel.plt (.rela.plt)*

**（1）介绍**：重定位节，保存了重定位相关的信息，这些信息描述了如何在链接或者运行时，对 ELF 目标文件的某部分内容或者进程镜像进行补充或修改。

每个结构体也与某一个重定位的函数相关。结构体数组。d_tag = DT_REL(值为 0x11) / d_tag = DT_RELA(值为 0x7) 的节。

**（2）结构：**

```
typedef struct
{
  Elf64_Addr    r_offset;           /* Address */
```

```
                                        // 此处表示的是解析完的函数真实地址存放的位置
                                        // 即对应解析函数的 GOT 表项地址
  Elf64_Xword   r_info;                 /* Relocation type and symbol index */
                                        // 该结构主要用到高某位，表示索引，低位表示类
                                        // 例如：0x10000007 此处 1 表示索引，7 代表
                                        //每一个表项的第二条指令， PUSH 了一个索引，
                                        //也就是通过 PLT 中 PUSH 的索引找到当时解析
} Elf64_Rel;

//与上一结构体类似，只是不同编译环境下产生的不同结构，作用相同，就不再次讨论
 typedef struct
 {
   Elf32_Addr    r_offset;            /* Address */
   Elf32_Word    r_info;              /* Relocation type and symbol index */
   Elf32_Sword   r_addend;            /* Addend */
 } Elf32_Rela;
```

### >>>> 扩充结构体（在Full RELRO用到）

本节讨论的是在 Full RELRO 攻击中用到的结构，所以如果不打算研究该攻击手法可以跳过该节。d_tag = DT_DEBUG。

```
struct r_debug{  //由于并没有找到该结构体的定义，所以没有声明类型
    r_version
    r_map        //指向 link_map
    r_brk
    r_state
    r_ldbase
}
```

### >>>> link map 结构

#### *1. 简介*

保存着 Binary 里面所有信息的一个结构体，该结构体很大，内容丰富。

*2.主要字段*

```
l_next：链接着该程序所有用到的 libary
        上边提到的 GOT[1] 中保存的地址是第一层 link_map 中所表示的 libary，此时是指
        不过可以用 l_next 结构寻找下一层表示的 libary，以此来遍历程序中所用到的 liba
        并利用下边所提到的字段找到该层 libary 的名字、基地址、以及所有的 section 等信
l_name：表示 libary 的名字
l_addr：表示 libary 的基地址
l_info[x]：指向该 libary 下的 .dynamic。
           l_info[1] 指向 d_tag = 1 时所表示的 section ，所以可以改变 x 的值找到每
           在链接过程中 binary 中的 section 地址，以及 libary 中的地址都是通过此方法
```

## 0x02 链接过程

### >>>> 概括描述

完成延迟绑定的函数主要是 dl_runtime__resolve(link_map_obj, reloc_arg) ，该函数的第一个参数是一个 link_map 结构，第二个参数是一个重定位参数，即运行 PLT 中的代码时 PUSH 进栈中的参数。

该函数主要是调用一个 dl_fixup(link_map_obj, reloc_arg) 完成了主要功能。

参数一的主要作用是：获得重定位函数所在了 libary 的基地址，以及获取在 libary 中寻找需要定位函数时所需要的 Section (.dynstr .dynsym 等)。

第二个函数主要是确定需要解析的函数名，以及解析完之后写回的地址。
该过程可以先大概理解为，dl_fixup 函数通过 reloc_arg 参数确定当前正在解析的函数名。

之后，拿着这个函数名，再利用 link_map 结构找到 libary 中的 .dynsym .dynstr 。利用 .dynsym .dynstr 进行匹配。若匹配成功，则从 .dynsym 中获取该函数的函数地址。

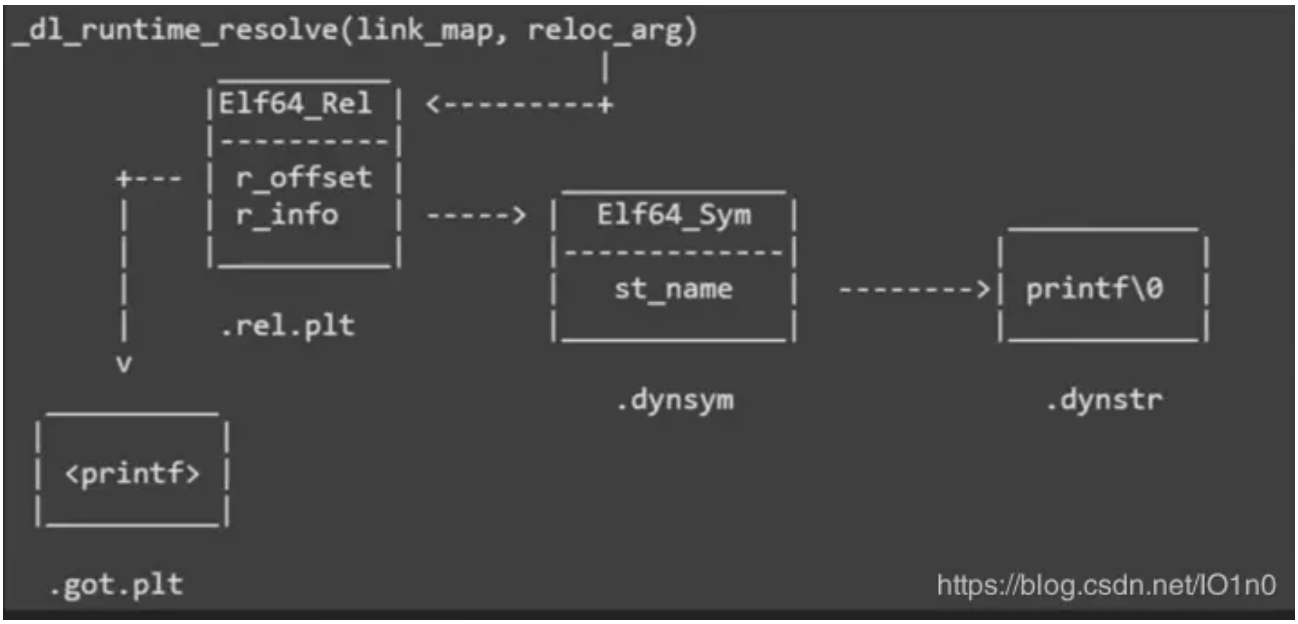
```
// 上边的详细过程
reloc_arg   -->   函数名 A

利用 link_map -->  l_info[x] 通过改变 x 的值，确定 .dynsym .dynstr
再用 .dynsym 与 .dynstr  对整个动态符号表 .dynstym 进行遍历，去匹配函数名 A
若 某一个 Elf64_Sym(符号) 的 st_name + .dynstr ==  A
则 该 Elf64_Sym 表示的符号即为函数 A

//  整个过程可以这样理解，不过真实情况使用的 Hash 方法去寻找的这个 Elf64_Sym(符号)
```

## >>>> 具体过程

**1. 调用某个函数后进入该函数的 PLT[x] ，在 PLT[x] 中 push 一个参数 reloc_arg 。**




【问题 1】通过这个 reloc_arg 可以干什么?

【答案 1】拿到这个 reloc_arg 后，链接器会通过该值找到对应函数的 Elf_Rel 结构，通过该结构的 r_info 变量中的偏移量找到对应函数的 Elf_Sym 结构，然后再通过 Elf_Sym 结构的 st_name 结合之前已经确定的 .dynstr 地址，通过 st_name + .dynstr 获得对应函数的函数名。

这就是拿到 reloc_arg 参数后链接器获得的信息，即知道了本次链接中的函数的函数名。（注：此处用到的 binary 中的 Elf_Rel Elf_Sym .dynstr 等地址都是通过 link_map->l_info[x] 的方式寻找的。）

**2. 在链接过程中 PLT[0] 会 push dl_runtime_resolve 函数的第二个参数 link_map。**

【问题 2】通过 link_map 我们能获得什么?

【答案 2】拿到这个变量后链接器会获得所要解析的函数的函数库(通过 link_map 的 l_next 字段)，然后拿到这个外部库之后 link_map 的 l_addr 字段会记录该库的基地址，然后链接器通过 new_hash 函数求出要链接函数的 hash（new_hash(st_name + .dynstr)），然后通过该 hash 和之前的保存值进行匹配，如果匹配上就获得了该函数在外部库的 Elf64_Sym 结构，然后通过该结构的 st_value 获取该函数在外部库里面的偏移，最后通过 st_value + l_addr 获取该函数的真实地址，最后通过 Elf64_Rel 的 r_offset 定位该函数在 GOT 中对应的地址，然后将最后结果写入该地址中。(其中有通过这两个参数共同获得的东西，不过为了便于理解就不再分开讨论。)

## 0x03 攻击

### >>>> 保护手段（RELRO）

**RELRO：重定位只读手段**

*1.无保护*

在这种模式下关于重定位并不进行任何保护。

*2.部分保护*

在这种模式下，一些段 (包括.dynamic) 在初始化后将会被标识为只读。

*3.完全保护*

在这种模式下，除了会开启部分保护外。惰性解析会被禁用（所有的导入符号将在开始时被解析，.got.plt 段会被完全初始化为目标函数的终地址，并被标记为只读）。

此外，既然惰性解析被禁用，GOT[1] 与 GOT[2] 条目将不会被初始化，存值为0。

### >>>> 对应攻击方法

动态装载器认为它接收到的参数都是值得信任的，因为它假设这些都是直接由 ELF 文件提供的或者是它自己在开始时初始化的。

然而，当一个攻击者能够修改这些数据时，这个假设就不成立 了。一些动态装载器(FreeBSD)会验证自己接收到的输入。然而，他们还是完全地信任控制结构 ，但这些也会可以轻易地破坏。

### *1. 无保护*

**原理**

动态装载器从 .rel.plt 中的 Elf_Rel 结构开始工作，顺着其中的下标找到 .dynsym 段中对应 Elf_Sym 结构的位置，并终使用它确定待解析符号的名称(在 .dynstr 段中的一段字符串)。

简单的调用任意函数的办法就是使用希望的函数的名称覆盖字符串表中的条目 ，然后再调用动态装载器，但这是不可能的，因为保存着动态符号字符串表的段，即.dynstr，是不可写的。

**过程**

动态装载器是从 .dynamic 段的 DT_STRTAB 条目中获得 .dynstr 段的地址的，而且 DT_STRTAB 条目的位置是已知的，默认情况下也可写。我们可以将这个条目的 d_val 域覆盖为 .bss 段。

这块内存区域上将会包含一段字符串，比如 system。到了这一步，攻击者需要选择一个已经存在的符号，它的偏移在伪造的字符串表中正好指向 system 的位置，接着调用其对应的符号解析重定位过程。可以通过将其重定位项的偏移压栈并跳转到 PLT0 实现。

**限制**

这种方式非常简单，但仅当二进制程序的 .dynamic 段可写时有效。对于使用部分或完全 RELRO 编译的二进制程序，需要使用更复杂的攻击。

### *2. 部分保护*

**原理**

_dl_runtime_resolve 函数的第二个参数是 Elf_Rel 条目在 .rel.plt 段中对应当前请求函数的偏移。动态装载器将这个值加上 .rel.plt 的基地址来得到目标 Elf_Rel 结构的绝对地址。

然而多数动态装载器实现不去检查重定位表的边界。这就表明如果一个大于 .rel.plt 的值传到 _dl_runtime_resolve 中，装载器将会认为特定的地址上的数据是一个 Elf_Rel 结构并使用它，即使那里已经超出了.rel .plt段的范围。

**过程**

计算一个新的 reloc_arg 参数，将 _dl_runtime_resolve 解析的位置劫持到一个可控内存。然后在那里构造一个 Elf_Rel 结构，并填写 r_offset 的值为一个可写的内存地址，将最后解析出的函数地址写在那里。

同时，r_info 也要修改成一个可控区域处。并在该区域伪造一个 Elf_Sym 结构，其中的 st_name 域，指向另一个可控区域，并在该处填写要伪造成的函数名(例：system)。

简而言之，该过程伪造了函数链接中所需要的所有结构(Elf_Sym Elf_Rel .dynstr)，通过控制 reloc_arg 指向到伪造的 Elf_Rel ，再通过Elf_Rel 中的 r_info 找到伪造的 Elf_Sym 最后通过 Elf_Sym 的 st_name 找到最终伪造后需要解析的函数(例：system)，解析完后通过 Elf_Rel 的 r_offset 写回到正确位置，达到劫持函数解析的目的，最终执行自己想要执行的函数。

**限制**

首先，Elf_Rel 的下标需要是正数，因为 r_info 域在 ELF 标准中规定是一个无符号整数。这就意味着在实际中这块可写的内存空间(例如.bss段)必须是位于 .dynsym 段之后。

这种情况总是满足的。另一个限制是 ELF 会使用的符号版本系统。在这种情况下，Elf_Rel 的 r_info 域不仅用作动态符号表中的下标，也用作符号版本表(.gnu.version段)中的下标。

**扩充方法**

可以通过修改指向程序那一层的 link_map，具体做法是把该层 link_map->l_info[DT_STRTAB]->st_value 的值劫持到一个我们可控的区域，然后在该区域填充伪

造函数，其实该方法也是通过修改 .dynstr 的方式实现攻击的手法。不过该方法必须有能够改写 st_value 值所需要的 gadget。

### *3. 完全保护*

**原理**

DT_DEBUG 条目的值是动态装载器在加载时设置好的，它指向一个 r_debug 类型的数据结构。

这个数据结构保存着调试 器用来标识动态装载器的基地址并拦截相应事件需要的信息。

此外，这个结构的 r_map 域保存着一个指向 link_map 链表头部的指针。

**过程**

攻击者使用 DT_DEBUG 这个动态条目来获取 r_debug 结构。接着，解引用 r_map 域从而得到主程序的 link_map 结构。然后像上边扩充方法那样破坏 l_info[DT_STRTAB]。

接着攻击者同样需要恢复 _dl_runtime_resolve 函数的指针，通过 link_map->l_next 获取其他链接库中用到的该函数。

具体获取手法是攻击者通过 l_info[DT_PLTGOT] 来获取对应的符号，然后通过 st_value 获取 .plt.got 节所在的地址，前面讨论过，该节的第三个偏移所存放的内容即为 _dl_runtime_resolve 函数地址。

不过在这一切都做好之后，还有一个问题值得关注，_dl_runtime_resolve 不仅仅会调用目标函数，还会尝试将它的地址写到 GOT 项中。因为完全 RELRO 保护下 GOT 是不可写的，所以程序就会崩溃。

不过我们可以通过伪造 link_map 中的 DT_JMPREL 动态条目来绕过这个问题。(具体操作方式和扩充方法中修改 .dynstr 类似)原本 DT_JMPREL 指向 .rel.dyn 段，我们将其改为一块可控区域，并那里写有一个 Elf_Rel 结构，并将其 r_offset 域指向一块可写的内存区域，其 r_info 指向我们的目标符号。至此，我们就完成了整个攻击过程。

## 0x04 参考

:) paper：sec15-paper-di-frederico

:) bilibili 视频 *(https://www.bilibili.com/video/av17482224)*

## 0x05 总结

第一次在论坛发文章，文章中可能有错误或不恰当的地方，如果有发现欢迎各位大佬批评指正，同时欢迎各位道友交流探讨。

- End -

**看雪ID：1Oin0**

https://bbs.pediy.com/user-864295.htm

*这里由看雪论坛 1Oin0 原创，转载请注明来自看雪社区。

推荐文章++++

* 为了理解反汇编引擎而写的X86 / X64反汇编引擎
* 捆绑包驱动锁首病毒分析
* **游戏逆向分析笔记
* 对宝马车载apps协议的逆向分析研究
* x86_64架构下的函数调用及栈帧原理

好书推荐



公众号ID：ikanxue

官方微博：看雪安全

商务合作：wsc@kanxue.com



戳“阅读原文”一起来充电吧！

阅读原文