

3.课程内容：综合实践三一补丁

- 1 补丁的概念
- 2 补丁工具
- 3 冷补丁
- 4 热补丁

(4) 三级--补丁-要求



- ①基础[T3-4-1]: 对二进制程序进行冷补丁, 修正程序错误 (初级)
- ②进阶[T3-4-2]: 对二进制程序进行热补丁, 不停机修正程序错误 (中、高级)
- ③编写**check**程序 (高级), 修复后的程序的运行, 进行验证, 可行性、有效性做对比、评价

(4) 补丁的概念

- 补丁是指衣服、被褥上为遮掩破洞而钉补上的小布块。现在也指对于大型软件系统(如微软操作系统)在使用过程中暴露的问题（一般由软件安全公司、黑客或病毒设计者发现）而发布的小程序。



pro11.msi 64位&32位 官方版

大小：6.1MB 时间：2018-04-20 星级：★★★★★

本站提供pro11.msi下载。pro11.msi是一款非常重要软件补丁文件，如果缺失或损坏将会造成office2003无法安装和使用，用户则需要下载这个文件到指定目录替换源文件，office2003才能正常使用。

[立即下载](#)

adobe cc 2015 破解补丁 64位&32位 中文版

大小：901KB 时间：2018-03-05 星级：★★★★★

本站提供adobe cc 2015 破解补丁下载。adobecc2015破解补丁是一款adobe软件非常重要的补丁组件。支持用户完美使用adobe系列软件，对电脑32位64位操作系统都支持，是用户使用adobe产品必备辅助工具。

[立即下载](#)

Idoo File Encryption Free v5.6 免费版

大小：3.11 MB 时间：2017-11-23 星级：★★★★★

Idoo File Encryption 注册版是最好的一键式文件加密、锁定软件，一键锁定文件、文件夹，禁止使用、拷贝、删除功能，文件加密采用256位AES加密，适合单个文件、邮件加密，加密后的文件，请牢记密码，因为目前技术很难破

[立即下载](#)

Idoo USB Encryption v3.0 中文版

大小：1.9 MB 时间：2017-11-23 星级：★★★★★

[立即下载](#)

(4) 补丁-类型

- “高危漏洞”的补丁，这些漏洞可能会被木马、病毒利用，应立即修复。--所有者
- 软件更新的补丁，用于修复一些流行软件的安全漏洞。---开发者
- 功能性更新补丁，主要用于更新系统或软件的功能，可根据需要选择性进行安装。---开发者
- 恶意修复补丁：修改正常合法文件变成执行非法工作---敌对方

(4) 补丁-角度——谁完成补丁

- 有源码——下载更新，
比如，微软每个月第二个星期二周期性发布-----
程序的开发者
- 无源码--二进制程序
 - 二进制程序所有者
补救措施
 - 第三方或安全公司
 - 敌对方

打补丁的方式

❑ 冷补丁-[T3-4-1]

- ❑ 被打补丁的程序以静态二进制文件的形式，采取相应的工具，对其进行修改
- ❑ 工具：二进制文件的编辑软件都可以

❑ 热补丁[T3-4-2]

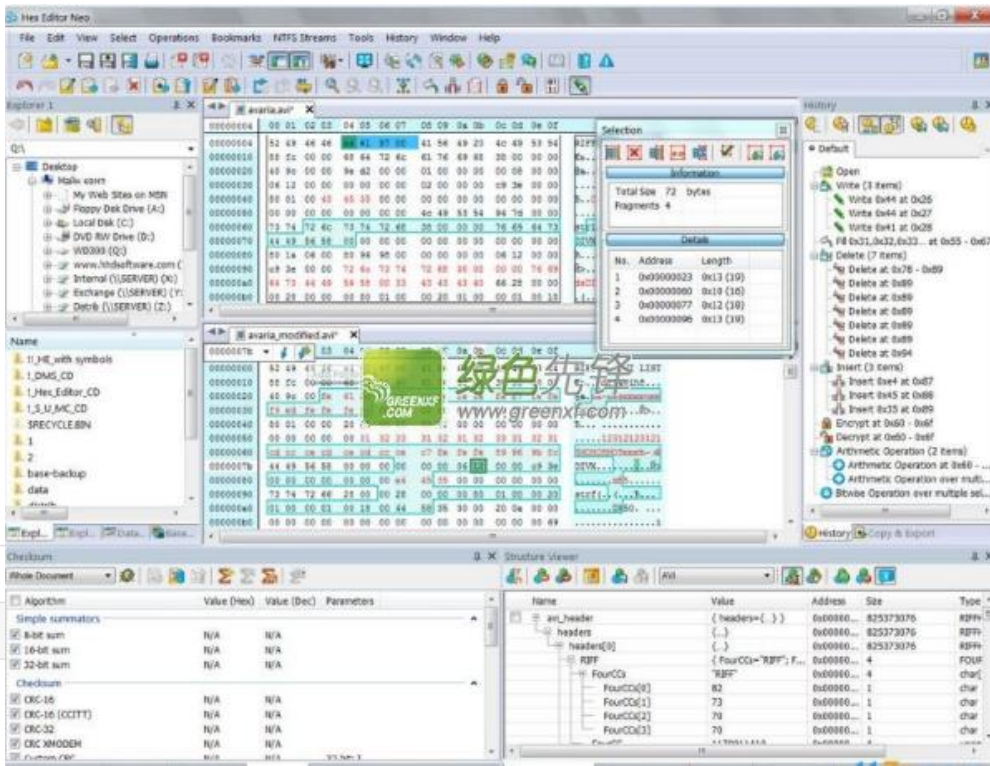
- ❑ 被打补丁的程序运行中打补丁
- ❑ 工具：内存管理软件，hook技术，进程管理软件

冷补丁[T3-4-1]二进制编辑工具

□ UltraEdit

十六进制编辑模式通常用于非ASCII文件，或二进制文件。这些文件一般都包含不可打印的字符，并且不是文本文件。

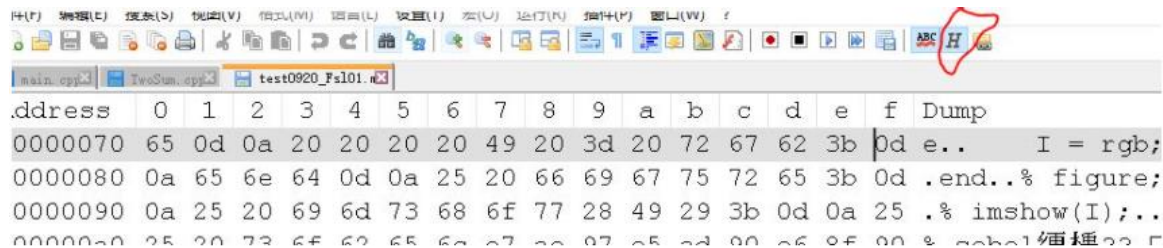
将屏幕范围分割成如下三个区域，其中文件偏移范围显示位于行首的字符相对于文件头部的字节偏移。



文件偏移:	十六进制表示	: ASCII 表示
0000000h:	30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35	:123456789012345

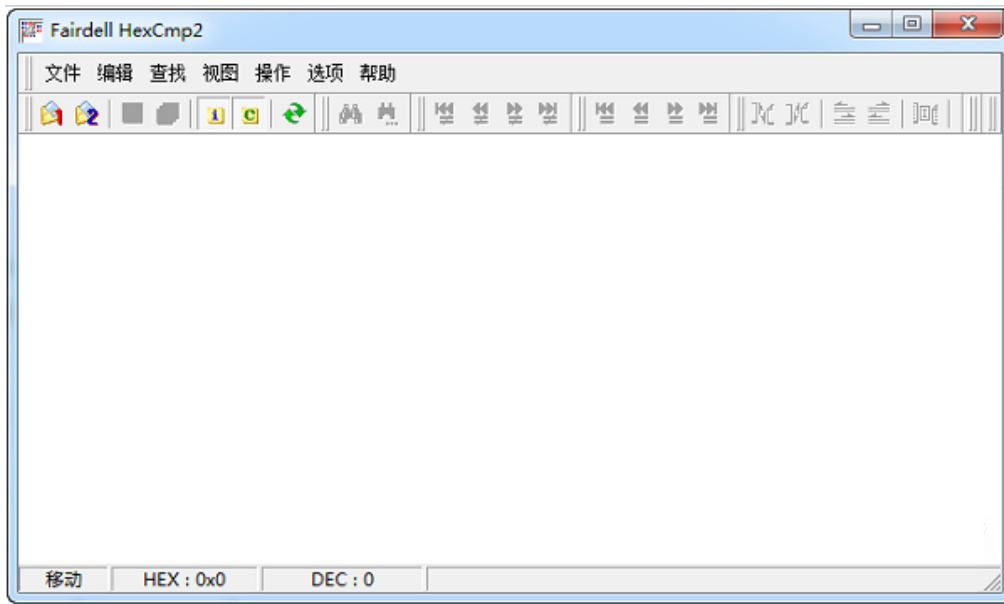
冷补丁二进制编辑工具

- ❑ **Notepad++**查看、编辑二进制文件
+ 安装附加组件**HexEditor**:



冷补丁二进制编辑工具

- 二进制文件比较编辑软件 (fairdell hexcmp) 是一款文件的比较编辑工具，具有实时同步文件传输以及数据快速分析的功能，支持进行文件比较、编辑、搜索和修复。该工具是进行二进制文件比较、编辑、搜索和修复的必备工具。

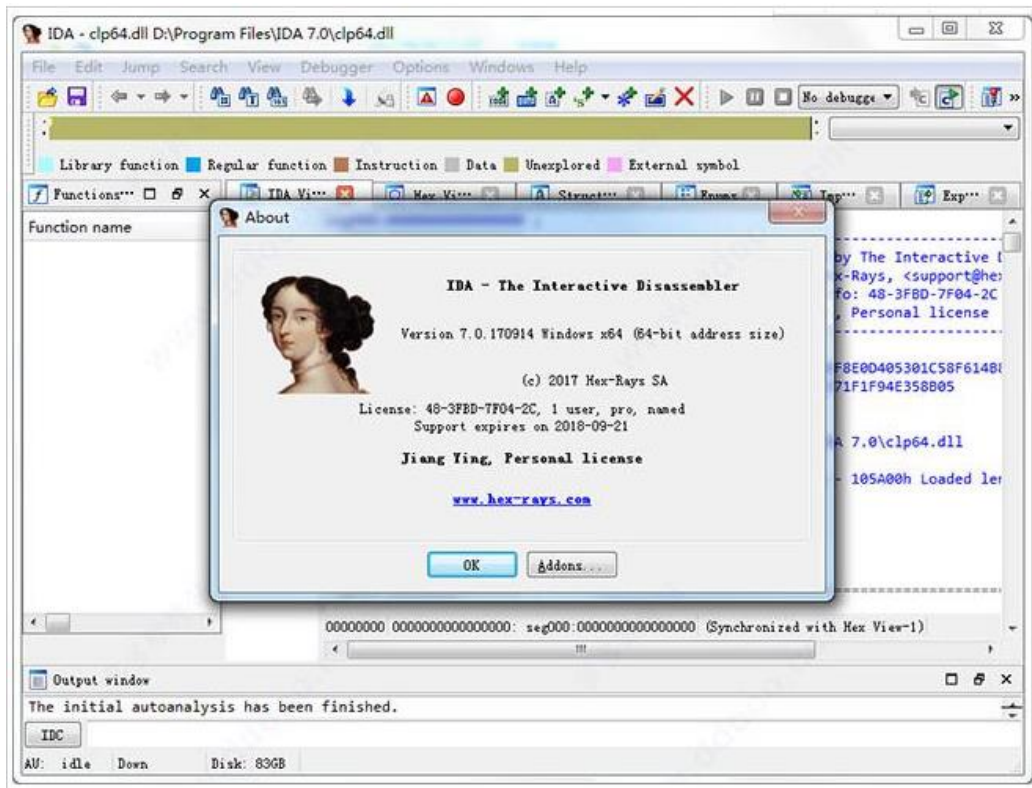


冷补丁二进制编辑工具

- **Linux** 下二进制文件编辑神器——**Ghex**（免费）
- **GHex** 是一个简单的二进制文件编辑器。它允许用户使用多级撤消/重做机制查看和编辑 **hex** 和 **ascii** 中的二进制文件。功能包括查找和替换功能，二进制，八进制，十进制和十六进制值之间的转换，以及使用另一种用户可配置的多文档界面概念，该概念允许用户使用多个视图编辑多个文档。

冷补丁二进制编辑工具

- IDA Pro是一款世界顶级的交互式反汇编工具，IDA Pro全名Interactive Disassembler Professional(交互式反汇编器专业版)，是Hex-Rays公司的旗舰产品，目前最新版为IDA Pro7.0。主要用在反汇编和动态调试等方面，支持对多种处理器的不同类型，可有执行模块进行反汇编处理，具有方便直观的操作界面，可以为用户呈现尽可能接近源代码的代码，减少了反汇编工作的难度，提高了效率。

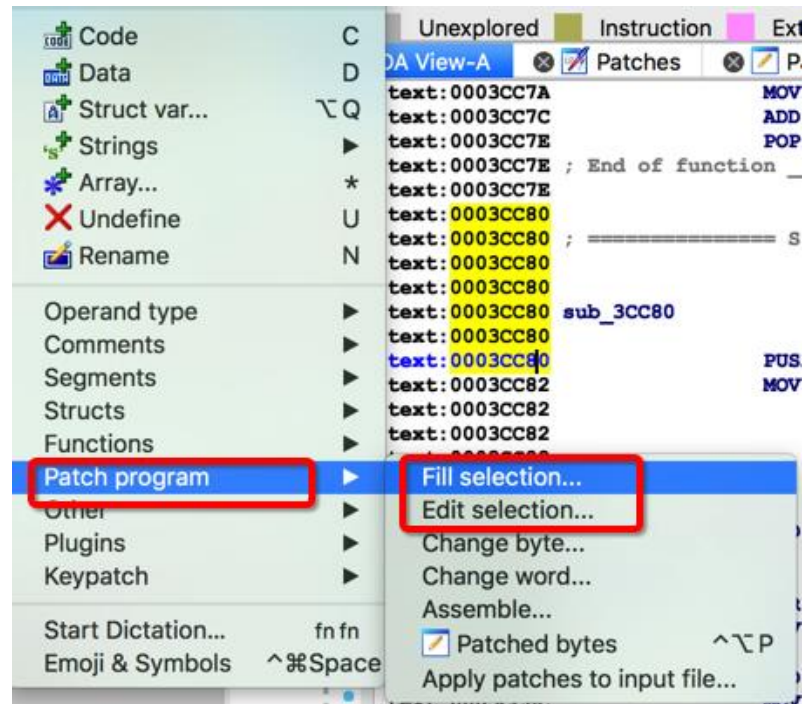


IDA Pro 功能特色

- 1、可编程性
IDA Pro 包含了一个由非常强大的类似于宏语言组成的完全开发环境，可用于执行简单到中等复杂的自动化任务。对于一些高级任务，开放式插件架构对外部开发人员是没有限制的，这样可以完善 IDA Pro 的功能。
- 2、交互性
IDA Pro 拥有完全的互动性，IDA 可以让分析师重写决策或者提供相应的线索。交互性是内置程序语言和开放式插件架构的最终要求。
- 3、调试器
IDA Pro 调试器补充了反汇编的静态分析功能：允许分析师通过代码一步一步来调查，调试器经常会绕过混淆，并得到一些能够对静态反汇编程序进行深入处理的数据，有些 IDA 调试器也可以运行在虚拟环境的应用上，这使得恶意软件分析更有成效。
- 4、反汇编
IDA Pro 可用的二进制程序的探索开发，也能确保代码的可读性，甚至在某些情况下和二进制文件产生的源代码非常相似。该程序图的代码可以为进一步的调查提供后期处理。

IDA Pro 功能特色

- 5. 补丁功能 ida-patcher , 支持的 CPU 架构:
support Arm, Arm64 (AArch64/Armv8), Hexagon, Mips, PowerPC, Sparc, SystemZ & X86 (include 16/32/64bit).
- 6. 支持的平台 Windows, MacOS, Linux



IDApatcher打补丁

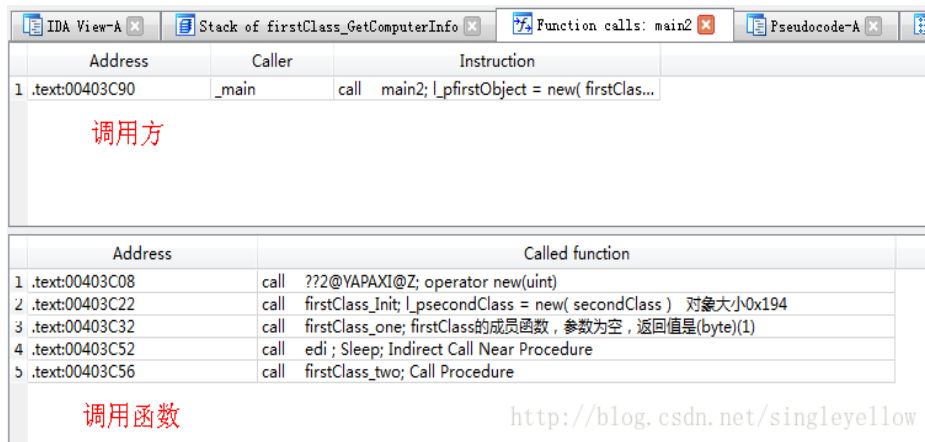
- 我们三级所讲的方式基本上为修改原程序二进制代码的方式，适用于**Windows95**之后的平台（**Win32**）的可执行文件都是**PE**格式，**.exe**、**.ocx**、**.dll**等都是常见的**PE**格式的文件映像，看一个文件是否为**PE**文件，不是看它的扩展名，而是看它的文件头中是否有**PE**文件头标示和具体的文件内容。

IDA Pro 图形接口：

- 文本模式：文本模式左侧部分被称为箭头窗口，显示了程序的非线性流程，实线标记的是无条件跳转，虚线标记了条件跳转，向上的箭头表示一个循环。
- 图形模式：图形模式中，箭头的颜色和方向显示程序的流程，红色表示一个条件跳转没有被采用，绿色表示这个条件跳转被采用，蓝色表示一个无条件跳转被采用，向上的箭头同样表示一个循环条件。
- IDA窗口分布：****函数窗口**：列举可执行文件中的所有函数，并显示每个函数的长度。这个窗口中每个函数关联了一些标志，如**L**代表此函数是库函数。
- 名称窗口：列举每个地址的名字，包括函数、命名代码、命名数据、字符串。
- 字符串窗口：显示所有字符串，默认显示长度超过**5**个字符的**ASCII**字符串。
- 导入表窗口：列举一个文件的所有导入函数。
- 导出表窗口：列举一个文件的所有导出函数，一般多用于分析**DLL**文件。
- 结构窗口：列举所有的活跃数据的结构布局。软件程序使用链接和交叉引用：常见的几个链接类型：子链接：一根函数开始的链接，如**printf**本地链接：跳转指令目的地址的链接，如**loc_40107E**偏移链接：内存偏移的链接****导航栏**：****导航栏**包括一个以颜色伪代号的被加载的二进制地址空间的线性视图

IDA-pro 函数调用地址查找

- 1 View->Open subviews->Function calls
显示出函数调用窗口，如下：



The screenshot shows the 'Function calls: main2' window in IDA Pro. It has two tabs: 'Callers' and 'Called functions'. The 'Callers' tab is active, showing a list of instructions that call the function 'main2'. The 'Called functions' tab is also visible, showing a list of functions called by 'main2'.

Address	Caller	Instruction
1 .text:00403C90	_main	call main2; L_pfirstObject = new(firstClass...

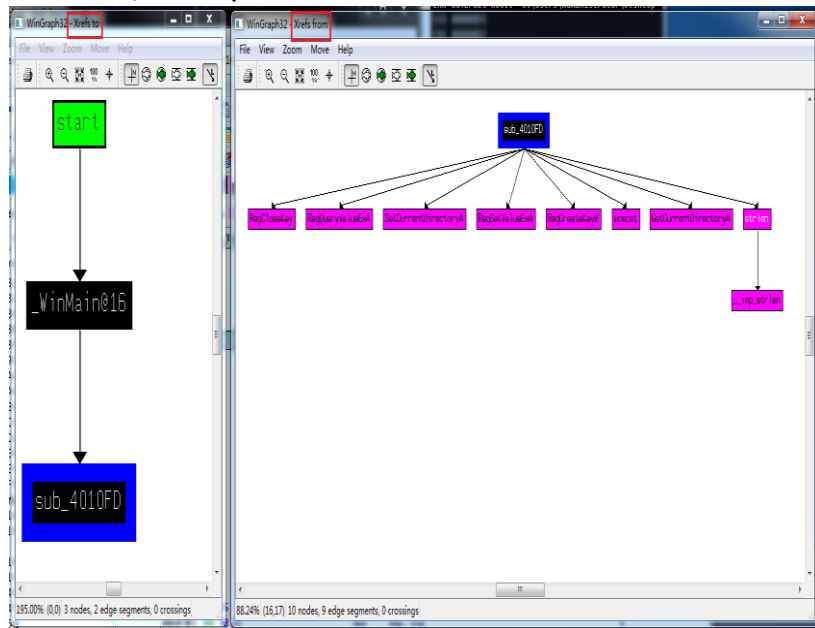
调用方

Address	Called function
1 .text:00403C08	call ??2@YAPAXI@Z; operator new(uint)
2 .text:00403C22	call firstClass_Init; L_psecondClass = new(secondClass) 对象大小0x194
3 .text:00403C32	call firstClass_one; firstClass的成员函数，参数为空，返回值是(byte)(1)
4 .text:00403C52	call edi ; Sleep; Indirect Call Near Procedure
5 .text:00403C56	call firstClass_two; Call Procedure

调用函数

<http://blog.csdn.net/singleyellow>

- 2 点击按钮 Display graph of xrefs from current identifier(从当前标识符绘制交叉引用图)



-附录IDApron-常见符号

- IDA图形视图会有执行流，Yes箭头默认为绿色，No箭头默认为红色，蓝色表示默认下一个执行块。我们可以在左侧查看代码的运行过程，按下空格键也可以直观地看到程序的图形视图。IDA View-A是反汇编窗口，
- HexView-A是十六进制格式显示的窗口，
- Imports是导入表（程序中调用到的外面的函数），
- Functions是函数表（这个程序中的函数），
- Structures是结构，
- Enums是枚举。
- 在反汇编窗口中大多是eax, ebx, ecx, edx, esi, edi, ebp, esp等。这些都是X86汇编语言中CPU上的通用寄存器的名称，是32位的寄存器。这些寄存器相当于C语言中的变量。
EAX 是“累加器”(accumulator)，它是很多加法乘法指令的缺省寄存器。
EBX 是“基地址”(base)寄存器，在内存寻址时存放基地址。
ECX 是计数器(counter)，是重复(REP)前缀指令和LOOP指令的内定计数器。
EDX 则总是被用来放整数除法产生的余数。
ESI/EDI 分别叫做“源/目标索引寄存器”(source/destination index)，因为在很多字符串操作指令中，DS:ESI指向源串，而ES:EDI指向目标串。
EBP 是“基址指针”(BASE POINTER)，它最经常被用作高级语言函数调用的“框架指针”(frame pointer)。
ESP 专门用作堆栈指针，被形象地称为栈顶指针，堆栈的顶部是地址小的区域，压入堆栈的数据越多，ESP也就越来越小。在32位平台上，ESP每次减少4字节。

基础[T3-4-1]: 对二进制程序进行冷补丁

---修正程序错误 (初级)

- 实验一: 简单修改二进制文件实现漏洞修补:

1、格式化字符串漏洞 `print_with_puts`

```
#include<stdio.h>

int main() {
    puts("test1");
    char s[20];
    scanf("%s", s);
    printf(s);
    return 0;
}
```

编译:

```
root@DESKTOP-HUI9I31:/# gcc print_with_puts.c -o print_with_puts
```

漏洞验证:

```
root@DESKTOP-HUI9I31:/# ./print_with_puts
test1
%%s%%s%%s%%s%%s%%s%%s%%s%%s%%s
段错误 (核心已转储)
```

格式化字符串漏洞发生的条件就是格式字符串要求输入的参数和实际提供的参数不匹配。

1、格式化字符串漏洞print_with_puts

- 在只有二进制文件而没有源代码的情况下，用IDA进行反编译，注意划线处：

.text:0000000000000745
.text:0000000000000746
.text:0000000000000749
.text:000000000000074D
.text:0000000000000756
.text:000000000000075A
.text:000000000000075C
.text:0000000000000763
.text:0000000000000768
.text:000000000000076C
.text:000000000000076F
.text:0000000000000776
.text:000000000000077B
.text:0000000000000780
.text:0000000000000784
.text:0000000000000787
.text:000000000000078C
.text:0000000000000791
.text:0000000000000796

代码区未显示

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
lea     rdi, s          ; "test1"
call    _puts
lea     rax, [rbp+format]
mov     rsi, rax
lea     rdi, aS          ; "%s"
mov     eax, 0
call    __isoc99_scanf
lea     rax, [rbp+format]
mov     rdi, rax          ; format
mov     eax, 0
call    _printf
mov     eax, 0
mov     rdx, [rbp+var_8]
```

跳转指引区

主要是三个区域：
地址区、OpCode
区（操作码区）、
反编译代码区

1、格式化字符串漏洞print_with_puts

- 观察到该程序中存在puts函数，且调用puts函数与调用printf函数的指令均为五个字节，想到可以将call _printf简单修改为call _puts，方法如下：

(1) 查看puts函数的地址：结果为0x0610

```
.plt:00000000000000610 _puts          proc near          ; CODE XREF: main+1E↓p
.plt:00000000000000610          jmp      cs:puts_ptr
.plt:00000000000000610 _puts          endp
```

(2) 选中调用printf指令的语句，通过鼠标右键或Ctrl+Alt+K快捷键调用IDA插件Keypatch:

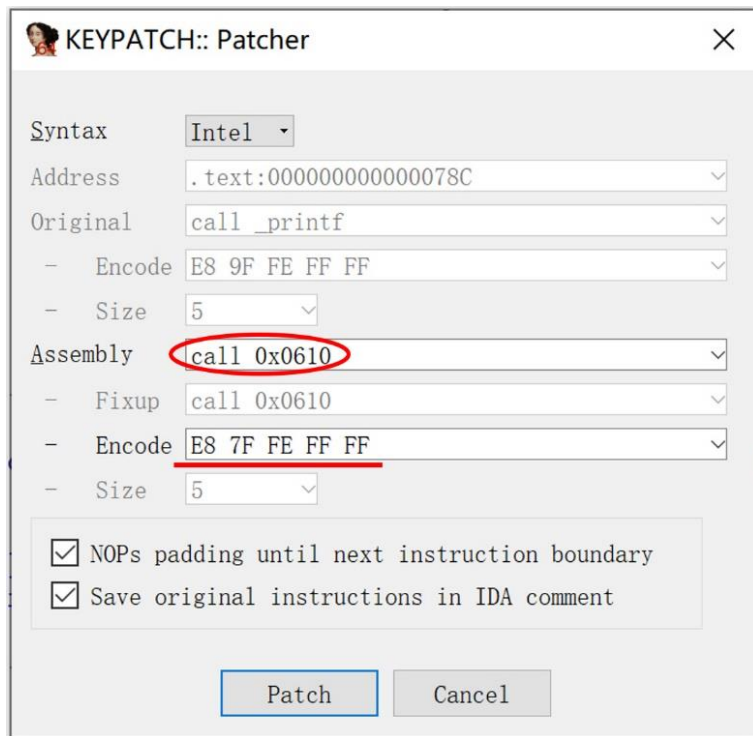
1、格式化字符串漏洞print_with_puts

```
.text:0000000000000780      lea     rax, [rbp+format]
.text:0000000000000784      mov     rdi, rax          ; format
.text:0000000000000787      mov     eax, 0
.text:000000000000078C      call    printf
.text:0000000000000791      mov     eax, 0
.text:0000000000000796      mov     rdx, 0
.text:000000000000079A      xor     rdx, rdx
.text:00000000000007A3      jz      short 00000000000007A5
.text:00000000000007A5      call    ___chkstk@100000000
.text:00000000000007AA      ; -----
.text:00000000000007AA      locret_7AA:
.text:00000000000007AB      leave   rax
.text:00000000000007AB      retn
.text:00000000000007AB      ; } // starts at 745
.text:00000000000007AB      main    endp
.text:00000000000007AB      ; -----
.text:00000000000007AC      align 10h
.text:00000000000007B0      ; ===== S U B R O
.text:00000000000007B0      ; =====
.text:00000000000007B0      ; void __libc_csu_init(void)
.text:00000000000007B0      public __libc_csu_init
.text:00000000000007B0      __libc_csu_init proc near
.text:00000000000007B0      ; __unwind {
```

0000078C 000000000000078C: main+47 (Synchronized with Hex View-1)

1、格式化字符串漏洞print_with_puts

(3) 将调用printf函数的语句修改为调用puts函数的语句，注意，由于Keypatch不能识别符号地址跳转，因此修改时不能使用call _puts这样的语句，而应该直接给定跳转地址，这也是第(1)步中必须准备好puts函数地址的原因：



1、格式化字符串漏洞print_with_puts补丁

□ Keypatch修改后的结果为：

```
.text:000000000000075A
.text:000000000000075C
.text:0000000000000763
.text:0000000000000768
.text:000000000000076C
.text:000000000000076F
.text:0000000000000776
.text:000000000000077B
.text:0000000000000780
.text:0000000000000784
.text:0000000000000787
.text:000000000000078C
.text:000000000000078C
.text:0000000000000791
.text:0000000000000796
```

```
xor     eax, eax
lea     rdi, s           ; "test1"
call    _puts
lea     rax, [rbp+format]
mov     rsi, rax
lea     rdi, aS          ; "%s"
mov     eax, 0
call    __isoc99_scanf
lea     rax, [rbp+format]
mov     rdi, rax         ; s
mov     eax, 0
call    _puts           ; Keypatch modified
                           ; call _printf

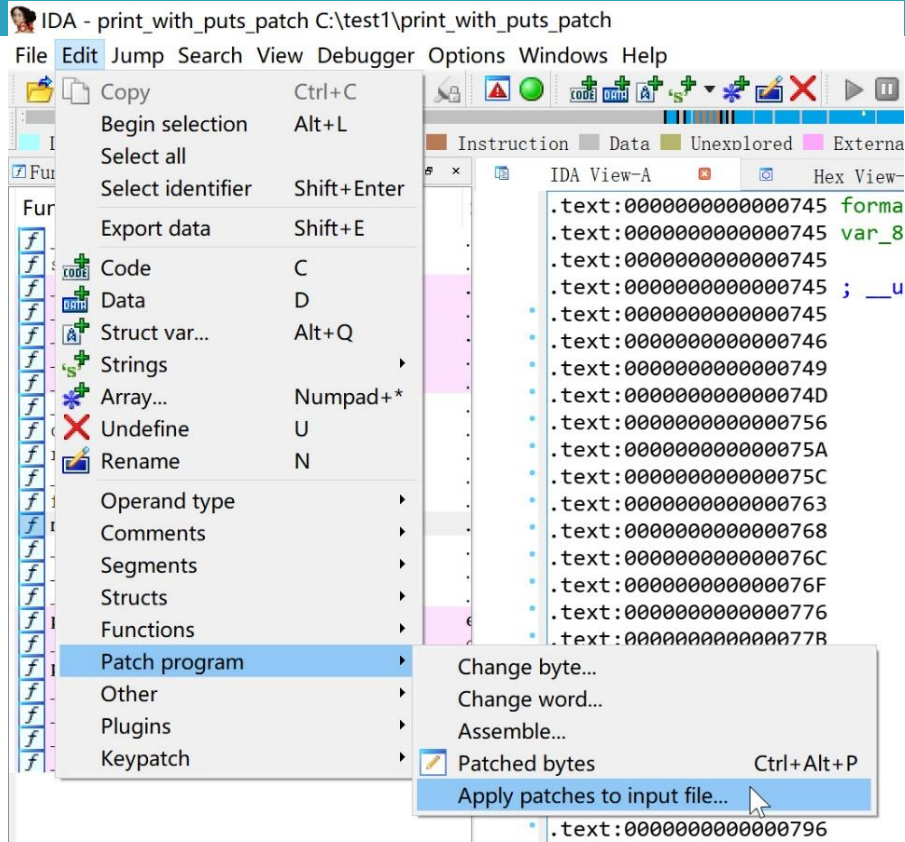
mov     eax, 0
mov     rdx, [rbp+var_8]
```


1、格式化字符串漏洞print_with_puts补丁

(4)通过IDA将Keypatch的修改结果保存到二进制文件:

补丁后:

```
root@DESKTOP-HUI9I31:/# ./print_with_puts_patch
test1
%s%s%s%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s
```



课后作业

□ 课后作业：

<http://https://222.20.126.111/login>

□ /按照作业的说明完成，提交即可

作业



安全综合实践III [课程讨论区](#)

主讲人: 韩兰胜 鲁宏伟 周威 余添龙 慕冬亮

时间: 2023-12-10 00:00:00 ~ 2024-01-22 00:00:00

章节	简介	类型	开放时间	答题截止时间	操作
第1章 逆向分析基础	PE文件格式、IDA Pro...	理论	2023-12-18 00:00:00	2024-01-10 00:00:00	开始学习 习题
第2章 Android程序逆向分析		理论	-	2024-01-31 00:00:00	开始学习 习题
第3章 Linux 内核空指针解引用利用		理论	-	2024-01-31 00:00:00	开始学习 习题
第4章 Linux内核释放后使用漏洞利用		理论	-	2024-01-31 00:00:00	开始学习 习题
第5章 漏洞的修复-补丁	补丁指的是为了修复软...	理论	-	2024-01-31 00:00:00	开始学习 习题
第6章 热补丁	热补丁是一种在程序运...	理论	-	2024-01-31 00:00:00	开始学习 习题

[illegible]

文件名	大小	操作
 fmt64	16.38K	 下载

B I H U S x² x₂ “ ” ⌋ ⌈ </> ↺ ↻

请作答

提交



1.

作答正确

讨论区

通过逆向了解程序的执行逻辑，按照要求修补漏洞
补丁前，程序的执行情况：

```
root@patch:/ctf/work # ./fmt64
Please input your student number:aaaaa
aaaaa
root@patch:/ctf/work # ./fmt64
Please input your student number:aaaaaaaaa
aaaaaaaaa
Your number:
root@patch:/ctf/work # ./fmt64
Please input your student number:%p%x
0x7ffde1a673d9a
```

补丁后程序的执行效果

要求Your number: 打印自己的学号

```
root@patch:/ctf/work # ./fmt64_patched
Please input your student number:aaaaa
aaaaa
root@patch:/ctf/work # ./fmt64_patched
Please input your student number:aaaaaaaaa
aaaaaaaaa
Your number:M22222222
root@patch:/ctf/work # ./fmt64_patched
Please input your student number:%p%x
%p%x
root@patch:/ctf/work #
```

答对了

习题附件：

文件名	大小	操作
 fmt64	16.38K	下载

B I H U S x² x₂ 1234

1234

1234

[T3-4-2]热补丁

- 热补丁是一种在程序运行时动态修补安全漏洞的技术，这种修补不需要重启操作系统或应用程序，因此能够大大增强系统的可用性。根据修补对象的不同，热补丁技术可以分为应用程序热补丁和系统内核热补丁两类，它们的技术原理是类似的，但具体的实现细节有所不同。

[T3-4-2]热补丁技术一般三个步骤

- 首先，对程序中存在的安全漏洞进行详细的分析，明确漏洞成因，在此基础上编写相应的代码，并编译出可动态加载的补丁文件。
- 其次，通过加载程序将第一步得到的补丁文件加载到目标程序的内存空间，对于同一个系统，加载程序可以是通用的，补丁文件则因安全漏洞而异。
- 最后，修改程序的执行流程，把存在安全漏洞的代码替换为新的代码，完成热补丁的修补。

[T3-4-2]热补丁

- 热补丁技术的关键在于补丁文件的加载和程序执行流程的修改，工程上通常借助钩子技术（**hook**）来实现。钩子技术通过拦截系统调用、消息或事件，得到对系统进程或消息的控制权，进而改变或增强程序的行为。主流操作系统，如**Windows**和**Linux**，都提供了**hook**的相应机制，并已被广泛运用到热补丁及代码调试等场景中。

A应用程序热补丁-Preload Hook

- ❑ **Preload Hook**是指利用操作系统对预加载（preload）的支持，将外部程序模块自动注入到指定的进程中的一种钩子技术——动态链接的程序有用，静态的无效，优先加载。
- ❑ **Preload Hook**有两种常见的用法：一种是配置环境变量 **LD_PRELOAD**（**SUID**或**SGID**位被置1，加载的时候会忽略 **LD_PRELOAD**），另一种是配置文件 **/etc/ld.so.preload**。对于配置环境变量 **LD_PRELOAD**，通过命令行指定 **LD_PRELOAD** 将仅影响当前新进程及其子进程，写入全局环境变量则将影响所有新进程，但新进程的父进程可以控制子进程的环境变量从而取消 **preload**。

样例

□ 初始程序源码

```
C original.c ×
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main()
5  {
6      puts("A sample of preload hook.");
7      sleep(2);
8      puts("The end.");
9      return 0;
10 }
```

□ 补丁源码

```
C patch.c ×
1  #include <stdio.h>
2
3  int sleep(int t)
4  {
5      puts("Your sleep() is hook by xxx.");
6  }
```

样例

补丁文件编译为：

```
root@DESKTOP-C6VM2Q8:/home/project/pre# gcc -m32 -fPIC --shared patch.c -o patch.so
```

通过命令行进行Preload Hook：

```
root@DESKTOP-C6VM2Q8:/home/project/pre# LD_PRELOAD=./patch.so ./original
A sample of preload hook.
Your sleep() is hook by xxx.
The end.
root@DESKTOP-C6VM2Q8:/home/project/pre#
```

可以看到已经成功地将系统的sleep函数修改成了自定义的功能。原理：loader在进行动态链接的时候，会将有相同符号名的符号覆盖成LD_PRELOAD指定的so文件中的符号。换句话说，可以用自己的so库中的函数替换原来库里有的函数，从而达到hook的目的。这和Windows下通过修改import table来hook API很类似。相比较之下，LD_PRELOAD更方便了，不用自己写代码了，系统的loader会直接调用。通过这个可以启动恶意代码，或劫持、监控软件等。

热补丁的完整实现

- 利用 **Preload Hook** 虽然可以进行补丁修补，但它还不能算是真正的热补丁，因为对于已经处于运行状态的应用程序，这种方法是无法生效的。真正的热补丁必须通过专门的加载程序，利用动态的 **hook** 机制来实现补丁文件的加载和程序执行流程的修改。下面以 **Linux** 系统为例，介绍加载程序的编写及热补丁的完整实现。

C original.c X

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <time.h>
4
5  int main()
6  {
7      while(1){
8          sleep(5);
9          printf("original: %ld\n", time(0));
10     }
11     return 0;
12 }
```

补丁源码

C patch.c X

```
1  #include <stdio.h>
2
3  int newprintf()
4  {
5      puts("My student number is xxx.");
6      return 0;
7  }
```

补丁文件编译

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# gcc -m32 -fPIC --shared patch.c -o patch.so
```

热补丁完整实现--第一步：关联

- Linux系统提供了一种专门用于程序调试的系统调用**ptrace**，热补丁的加载程序可以借助**ptrace**对运行状态的应用程序进行**hook**，并最终实现热补丁修补。具体分为以下五个步骤。
- 第一步：加载程序通过**ptrace**关联（**attach**）到需要修补的进程上，并将该进程的寄存器及内存数据保存下来，代码如下：

```
/* 关联到进程 */
void ptrace_attach(int pid)
{
    if(ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0) {
        perror("ptrace_attach");
        exit(-1);
    }

    waitpid(pid, NULL, /*WUNTRACED*/0);

    ptrace_readreg(pid, &oldregs);
}
```

第二步-定位：ELF文件及link_map

- 找到需要补丁的可执行程序**的elf文件**（是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface, ABI）而开发和发布的，也是Linux的主要可执行文件格式。https://blog.csdn.net/kang__xi/article/details/79571137）
 - 再找到指向**elf文件**的**link_map**链表的指针，并通过遍历**link_map**中的符号表，找到需要修补的（比如：**printf**）函数及用于将补丁文件加载到进程内存地址空间的**__libc_dlopen_mode**函数的地址。根据**elf文件**的结构信息，首先从**elf文件**的开头开始读取信息，找到头部表（**program header table**），再根据头部表找到**elf文件**的全局偏移量表（**global offset table, GOT表**）。**GOT表**中的每一项都是一个**32bit的Elf32_Addr**地址，其中前两项是两个特殊的数据结构的地址：
- 格式代码如下：
- **GOT[0]**为**PT_DYNAMIC**的起始地址
 - **GOT[1]**为**link_map**结构体的地址
 - 由此可以得到指向**link_map**链表的指针，代码如下：

```
1  #define EI_NIDENT 16
2  typedef struct{
3      unsigned char e_ident[EI_NIDENT];
4      Elf32_Half e_type;
5      Elf32_Half e_machine;
6      Elf32_Word e_version;
7      Elf32_Addr e_entry;
8      Elf32_Off e_phoff;
9      Elf32_Off e_shoff;
10     Elf32_Word e_flags;
11     Elf32_Half e_ehsize;
12     Elf32_Half e_phentsize;
13     Elf32_Half e_phnum;
14     Elf32_Half e_shentsize;
15     Elf32_Half e_shnum;
16     Elf32_Half e_shstrndx;
17 } Elf32_Ehdr;
```

第二步：找到被补的可执行成的函数

```
/*
 * 得到指向link_map链表首项的指针
 */
struct link_map *get_linkmap(int pid)
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *) malloc(sizeof(Elf32_Ehdr));
    Elf32_Phdr *phdr = (Elf32_Phdr *) malloc(sizeof(Elf32_Phdr));
    Elf32_Dyn *dyn = (Elf32_Dyn *) malloc(sizeof(Elf32_Dyn));
    Elf32_Word got;
    struct link_map *map = (struct link_map *) malloc(sizeof(struct link_map));
    int i = 1;
    unsigned long tmpaddr;

    ptrace_read(pid, IMAGE_ADDR, ehdr, sizeof(Elf32_Ehdr));
    phdr_addr = IMAGE_ADDR + ehdr->e_phoff;
    printf("phdr_addr\t %p\n", phdr_addr);

    ptrace_read(pid, phdr_addr, phdr, sizeof(Elf32_Phdr));
    while(phdr->p_type != PT_DYNAMIC)
        ptrace_read(pid, phdr_addr += sizeof(Elf32_Phdr), phdr, sizeof(Elf32_Phdr));
    dyn_addr = phdr->p_vaddr;
    printf("dyn_addr\t %p\n", dyn_addr);

    ptrace_read(pid, dyn_addr, dyn, sizeof(Elf32_Dyn));
    while(dyn->d_tag != DT_PLTGOT) {
        tmpaddr = dyn_addr + i * sizeof(Elf32_Dyn);
        //printf("get_linkmap tmpaddr = %x\n", tmpaddr);
        ptrace_read(pid, tmpaddr, dyn, sizeof(Elf32_Dyn));
        i++;
    }

    got = (Elf32_Word) dyn->d_un.d_ptr;
    got += 4;
    //printf("GOT\t\t %p\n", got);

    ptrace_read(pid, got, &map_addr, 4);
    printf("map_addr\t %p\n", map_addr);
    map = map_addr;
    //ptrace_read(pid, map_addr, map, sizeof(struct link_map));

    free(ehdr);
    free(phdr);
    free(dyn);

    return map;
}
```

遍历link_map链表，
依次对每一个link_map调
用find_symbol_in_linkmap
函数：

find_symbol_in_linkmap函数
负责在指定的link_map中查
找所需要的函数的地址：

```
/*
 * 解析指定符号
 */
unsigned long find_symbol(int pid, struct link_map *map, char *sym_name)
{
    struct link_map *lm = map;
    unsigned long sym_addr;
    char *str;
    unsigned long tmp;

    sym_addr = find_symbol_in_linkmap(pid, lm, sym_name);
    while(!sym_addr) {
        ptrace_read(pid, (char *)lm+12, &tmp, 4); //获取下一个库的link_map地址
        if(tmp == 0)
            return 0;
        lm = tmp;

        if ((sym_addr = find_symbol_in_linkmap(pid, lm, sym_name)))
            break;
    }

    return sym_addr;
}
```

第三步—加载补丁程序

调用__libc_dlopen_mode函数，将补丁文件加载到需要修补的进程的内存空间中，并再一次遍历link_map中的符号表，找到新加载的补丁文件中的新函数newprintf的地址，代码如下：

```
/*
在指定的link_map所指向的符号表中查找符号
*/
unsigned long find_symbol_in_linkmap(int pid, struct link_map *lm, char *sym_name)
{
    Elf32_Sym *sym = (Elf32_Sym *) malloc(sizeof(Elf32_Sym));
    int i = 0;
    char *str;
    unsigned long ret;
    int flags = 0;

    get_sym_info(pid, lm);

    do{
        if(ptrace_read(pid, symtab + i * sizeof(Elf32_Sym), sym, sizeof(Elf32_Sym)))
            return 0;
        i++;
        if (!sym->st_name && !sym->st_size && !sym->st_value) //全为0是符号表的第一项
            continue;
        str = (char *) ptrace_readstr(pid, strtab + sym->st_name);
        if (strcmp(str, sym_name) == 0) {
            printf("\nfind_symbol_in_linkmap str = %s\n",str);
            printf("\nfind_symbol_in_linkmap sym->st_value = %x\n",sym->st_value);
            free(str);
            if(sym->st_value == 0) //值为0代表这个符号本身就是重定向的内容
                continue;
            flags = 1;
            break;
        }
        free(str);
    }while(1);

    if (flags != 1)
        ret = 0;
    else
        ret = link_addr + sym->st_value;

    free(sym);

    return ret;
}
```


第四步—将被补函数的地址改为补丁函数的地址

```
/* 发现__libc_dlopen_mode, 并调用它 */
sym_addr = find_symbol(pid, map, "__libc_dlopen_mode"); /* call _dl_open */
printf("found __libc_dlopen_mode at addr %p\n", sym_addr);
if(sym_addr == 0)
    goto detach;
call__libc_dlopen_mode(pid, sym_addr, libpath); /* 注意装载的库地址 */
waitpid(pid, &status, 0);
/* 找到新函数的地址 */
strcpy(sym_name, newfunname); /* intercept */
sym_addr = find_symbol(pid, map, sym_name);
printf("%s addr\t %p\n", sym_name, sym_addr);
if(sym_addr == 0)
    goto detach;
```

找到要修补的函数printf的重定向地址，
在该地址填入补丁文件中的新函数newprintf
的地址代码如下

```
/* 找到旧函数在重定向表的地址 */
strcpy(sym_name, oldfunname);
rel_addr = find_sym_in_rel(pid, sym_name);
printf("%s rel addr\t %p\n", sym_name, rel_addr);
if(rel_addr == 0)
    goto detach;

/* 函数重定向 */
puts("intercept..."); /* intercept */
if(modifyflag == 2)
    sym_addr = sym_addr - rel_addr - 4;
printf("main modify sym addr = %x\n", sym_addr);
```


第五步-收尾

- 完成patch，恢复现场，脱离需要修补的进程代码如下。

```
ptrace_write(pid, rel_addr, &sym_addr, sizeof(sym_addr));  
puts("patch ok");  
detach:  
printf("prepare to detach\n");  
ptrace_detach(pid);  
  
return 0;
```

初始程序的执行效果:

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./original  
original: 1622280172  
original: 1622280177
```

通过加载程序打上热补丁后的执行效果:

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./original  
original: 1622281125  
original: 1622281130  
original: 1622281135  
original: 1622281140  
patch succeeded  
My student number is xxx.  
My student number is xxx.  
My student number is xxx.  
My student number is xxx.
```

应用热补丁

热补丁加载程序运行情况：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./hotfix 11331 ./patch.so printf newprintf
main pid = 11331
main libpath : ./patch.so
main oldfunname : printf
main newfunname : newprintf
phdr_addr      0x8048034
dyn_addr       0x8049f14
map_addr       0xf7f82940

find_symbol_in_linkmap str = printf
find_symbol_in_linkmap sym->st_value = 50c60
found printf at addr 0xf7dbcc60
get_sym_info exit

find_symbol_in_linkmap str = __libc_dlopen_mode
find_symbol_in_linkmap sym->st_value = 131a90
found __libc_dlopen_mode at addr 0xf7e9da90
get_sym_info exit

find_symbol_in_linkmap str = newprintf
find_symbol_in_linkmap sym->st_value = 4bd
newprintf addr  0xf7f514bd
get_sym_info exit

printf rel addr  0x804a00c
intercept...
main modify sym_addr = f7f514bd
patch ok
prepare to detach
```

内核级热补丁

- 与应用程序热补丁相比，系统内核热补丁需要考虑的因素更多，对稳定性和可靠性的要求也更高。因此，各大操作系统厂商都推出了自己的系统内核热补丁方案。下面介绍几种常见的**Linux**内核热补丁方案。

内核级热补丁

- **Ksplice**是MIT的Jeff Arnold在2008年发明的，是第一个成熟的Linux内核热补丁方案，并于2011年被Oracle收购。它的核心原理是构建新、老两个系统内核，并在此基础上比较二进制代码的差异。对于存在差异的函数，Ksplice会调用stop_machine_run函数来短暂地停止其所在的进程及其线程，并迅速地在原始函数的开头处插入一段跳转代码，将其指向修补后的新函数。之后，Linux内核将执行修补后的新函数。
- **Kgraft**由SUSE在2014年提出。它的原理与Ksplice相近，都是通过差异比较来确定内核中需要修补的函数。但是，与Ksplice不同，Kgraft不会调用stop_machine_run函数来暂时停止进程或线程，而是使用了处理器间非可屏蔽中断机制来完成新、老内核的切换。同时，Kgraft能够为用户进程、内核线程和中断处理程序提供始终一致的视图，这样，老的内核模块将始终调用另一个老的内核模块，而修补后的新内核模块将始终调用另一个新内核模块，从而避免系统在热补丁修补过程中出现混乱。
- **Kpatch**是Redhat在2014年提出的系统内核热补丁方案，它基于ftrace来hook原始函数的mcount调用指令，从而将对原始函数的调用重定向到新的函数。使用Kpatch的系统内核在编译时会在每个函数的入口处保留若干字节，这样，在进行热补丁修补时只需要将待修补的函数入口处保留的字节替换为跳转指令，跳转到Kpatch的相关流程中，即可实现函数级别的执行流程在线替换，最终进入修补后的新函数的执行流程。

补丁的方法

- 硬盘上PE文件的Characteristics和FirstThunk指向2个IMAGE_THUNK_DATA结构, 当加载到内存后, PE装载器通过OriginalFirstThunk找到IMAGE_THUNK_DATA再找到IMAGE_IMPORT_BY_NAME,
- 根据shell32.dll.shellaboutA信息, 计算出函数的真实地址, 把真实地址填写到FirstThunk->IMAGE_THUNK_DATA中。PS: 函数的真实地址 = dll加载基地址 + 函数RAV. 这个地址是PE加载器算出来的, PE文件不保存函数的RAV.
- EXE程序加载后, FirstThunk指向的为函数的真实地址。
- APIHOOK替换地址就是 *FirstThunk, CALL DWORD ptr[FirstThunk]
- IMAGE_IMPORT_DESCRIPTOR STRUCT 20个字节
- union Characteristics DWORD ? //固定的IMAGE_THUNK_DATA结构的地址
- OriginalFirstThunk DWORD ?
- ends TimeDateStamp DWORD ?
- ForwarderChain DWORD ? Name DWORD ?
- FirstThunk DWORD ? //会被PE加载器重写的IMAGE_THUNK_DATA地址, PE装载器加载exe后, 把这个IMAGE_THUNK_DATA改写dll里 函数的真实地址IMAGE_IMPORT_DESCRIPTOR ENDS
- IMAGE_THUNK_DATA STRUC 4个字节 union u1 ForwarderString DWORD ? ; 指向一个转向者字符串的RVA Function DWORD ? ; 被输入的函数的内存地址 Ordinal DWORD ? ; 被输入的API的序数值 AddressOfData DWORD ? ; 指向IMAGE_IMPORT_BY_NAME ends IMAGE_THUNK_DATA ENDS