第3章 指针安全

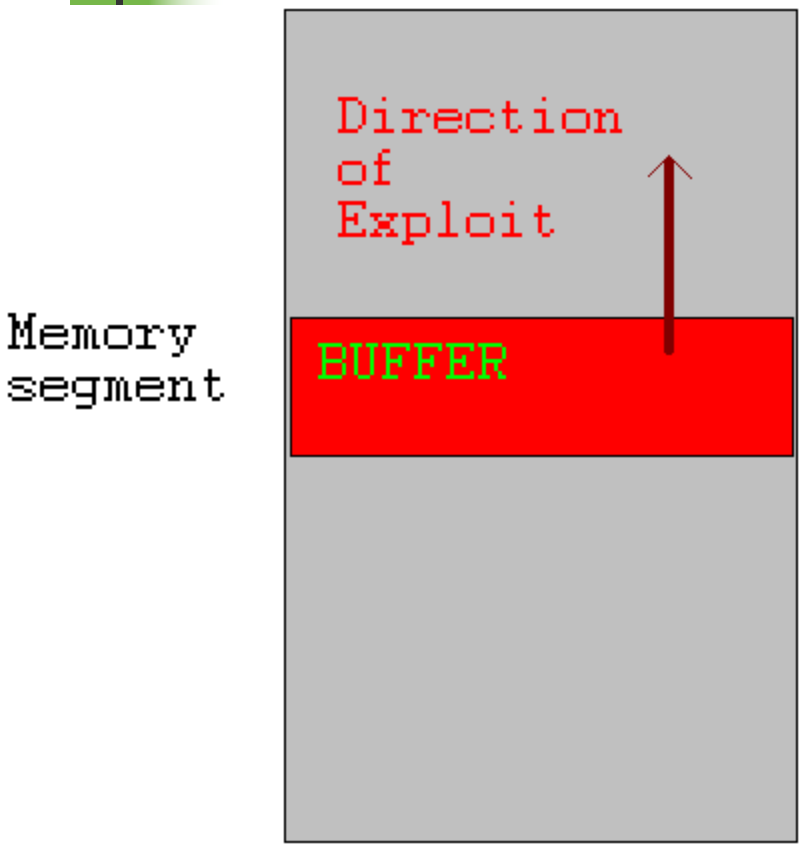
北京邮电大学 徐国胜

guoshengxu@bupt.edu.cn

指针安全

- 指针安全是通过修改指针值来利用程序漏洞的方法的统称
  - 可以通过覆盖函数指针将程序的控制权转移到攻击者提供的外壳代码
  - 对象指针也可以被修改，从而执行任意代码

指针安全



- 缓冲区溢出覆写指针条件：
  - 缓冲区与目标指针必须分配在同一个段内
  - 缓冲区必须位于比目标指针更低的内存地址处
  - 该缓冲区必须是界限不充分的，因此容易被缓冲区溢出利用

指针安全

- UNIX可执行文件包含data段和BSS段
- data段包含了所有已初始化的全局变量和常数
- BSS（ Block Started by Symbols ）段包含了所有未初始化的全局变量
- 已初始化的全局变量和未初始化变量分开是为了让汇编器不将未初始化的变量内容写入目标文件

指针安全

```
1. static int GLOBAL_INIT = 1;      /* data segment, global */
2. static int global_uninit;         /* BSS segment, global */
3.
4. void main(int argc, char **argv) {       /* stack, local */
5.   int local_init = 1;                    /* stack, local */
6.   int local_uninit;                      /* stack, local */
7.   static int local_static_init = 1;  /* data seg, local */
8.   static int local_static_uninit; /* BSS segment, local */
                  /* storage for buff_ptr is stack, local */
                       /* allocated memory is heap, local */
9.   }
```

指针安全

```
void good_function(const char *str) {
        //do something
}

int main(int argc, char **argv) {
        if (argc !=2){
                printf("Usage: prog_name <string1>\n");
                exit(-1);
        }
        static char buff [BUFFSIZE];
        static void (*funcPtr)(const char *str);
        funcPtr = &good_function;
        strncpy(buff, argv[1], strlen(argv[1]));
        (void)(*funcPtr)(argv[2]);
        return 0;
}
```

指针安全

- 程序存在漏洞，可被缓冲区溢出利用
- 缓冲区和函数指针都未初始化，因此存在于BSS段

函数指针举例

```
1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3.   static char buff[BUFFSIZE];
4.   static void (*funcPtr)(const char *str);
5.   funcPtr = &good_function;
6.   strncpy(buff, argv[1], strlen(argv[1]));
7.   (void)(*funcPtr)(argv[2]);
8. }
```

静态字符数组buff

funcPtr都是未初始化的，并且存储于BSS段

函数指针举例

```
1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3.   static char buff[BUFFSIZE];
4.   static void (*funcPtr)(const char *str);
5.   funcPtr = &good_function;
6.   strncpy(buff, argv[1], strlen(argv[1]));
7.   (void)(*funcPtr)(argv[2]);
8. }
```

当argv[1]的长度大于BUFFSIZE的时候，就会发生缓冲区溢出

函数指针举例

```
1. void good_function(const char *str) {...}
2. void main(int argc, char **argv) {
3.   static char buff[BUFFSIZE];
4.   static void (*funcPtr)(const char *str);
5.   funcPtr = &good_function;
6.   strncpy(buff, argv[1], strlen(argv[1]));
7.   (void)(*funcPtr)(argv[2]);
8. }
```

当程序执行到funcPtr标识的函数的时候， shellcode将会取代good_function()得以执行

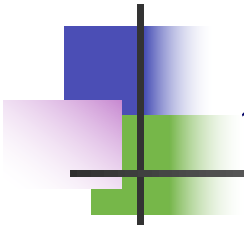
对象指针举例

```
void foo(void * arg, size_t len)        {
        char buff[100];
        long val = …;
        long *ptr = …;
        memcpy(buff, arg, len);
        *ptr = val;
        …
        return;
}
```

缓冲区容易被漏洞利用

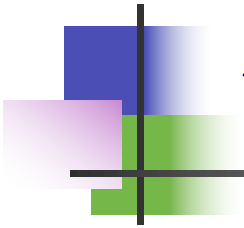
在溢出缓冲区后，攻击者可以覆写ptr和val

会发生任意内存写



对象指针

- “任意内存写”可以改变程序控制流
- 如果指针长度等于重要的数据结构长度，任意内存写会更容易
  - Intel 32 Architectures:
    - sizeof(void*) = sizeof(int) = sizeof(long) = 4B



修改指令指针

- Instruction Counter (IC) (a.k.a Program Counter (PC))
  存储了将要执行的下一条指令地址
  - Intel: EIP 寄存器
- IC不能被直接访问
- IC在顺序执行代码时递增，也可以由控制转移指令间接修改
  - jmp
  - Conditional jumps
  - call
  - ret

修改指令指针

```
int _tmain(int argc, _TCHAR* argv[]) {
        if (argc !=1){
                printf("Usage: prog_name\n");
                exit(-1);
        }
        static void (*funcPtr)(const char *str);
        funcPtr = &good_function;
        (void)(*funcPtr)("hi ");
        good_function("there!\n");
        return 0;
}
```

使用指针调用good function

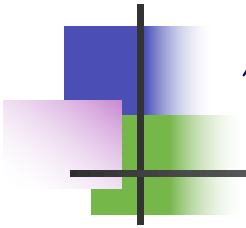
直接调用good function

修改指令指针

```
void)(*funcPtr)("hi ");
00424178 mov esi, esp
0042417A push offset string "hi" (46802Ch)
0042417F call dword ptr [funcPtr (478400h)]
00424185 add  esp, 4
00424188 cmp  esi, esp

good_function("there!\n");
0042418F push offset string "there!\n" (468020h)
00424194 call good_function (422479h)
00424199 add  esp, 4
```

第一个调用good function
机器码
ff 15 00 84 47 00
最后4个字节包含了被调用函数的地址

第二个调用good function
机器码
e8 e0 e2 ff ff
最后4字节被调用函数的相对偏移量



修改指令指针

- 静态调用对于函数地址使用立即数
  - 指令中地址被编码
  - 计算地址，然后放入IC
  - 不改变执行指令，IC不会改变
- 通过函数指针的调用是间接引用
  - IC的下一个值，存储在内存中，其可以被改变

修改指令指针

- 控制IC使得攻击者可以选择要执行的代码
  - 攻击者能够任意写的话，很容易
  - 间接的函数引用与无法在编译期间决定的函数调用可以被利用，从而使程序的控制权转移到任意代码

修改指令指针的目标：
全局偏移表

- Windows和Linux在库函数的链接和控制转移方面使用了类似的机制
- Linux使用的方法是可被利用的，而Windows则不然

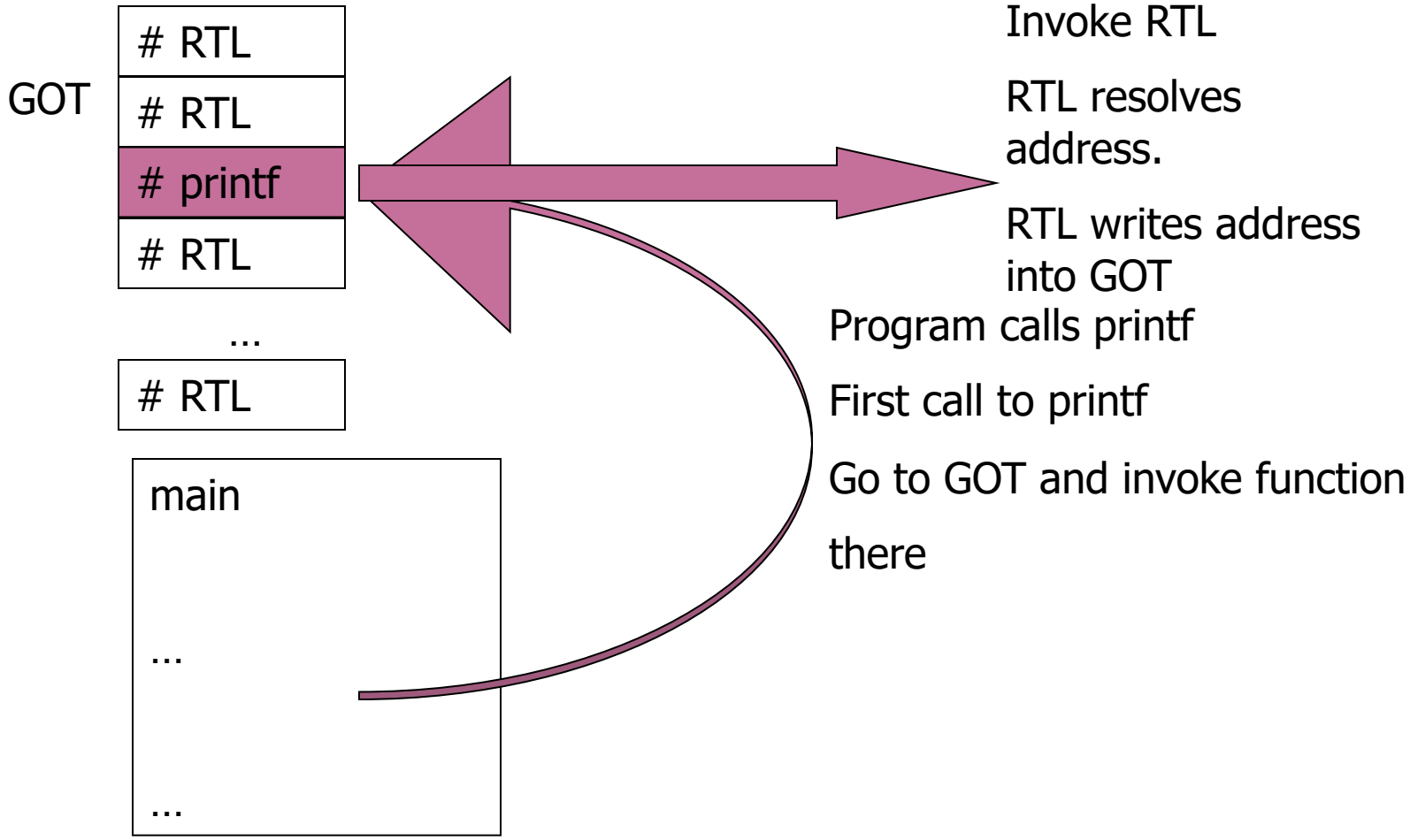
修改指令指针的目标：全局偏移表

- ELF (executable & linking format)
  - 默认二进制格式，对于
    - Linux
    - Solaris 2.x
    - SVR4
    - Adopted by TIS (Tool Interface Standards Committee)
  - Global Offset Table (GOT)
    - 被包含在ELF的二进制文件的进程空间里
    - GOT存放**绝对**地址
    - 要使得动态链接的进程能够工作，其必不可少

修改指令指针的目标：
全局偏移表

- GOT
  - 程序首次使用一个函数前，GOT入口项包含运行时连接器RTL（runtime linker）的地址
  - 如果该函数被程序调用，则程序的控制权被转移到RTL，然后函数的实际地址被确定且被插入到GOT中
  - 接下来就可以通过GOT中的入口项直接调用函数

修改指令指针的目标：
全局偏移表



修改指令指针的目标：全局偏移表

- 在ELF可执行文件中的GOT入口项的地址是固定的
  - 对任何可执行进程映像而言GOT入口项都位于相同的地址
- 可以利用objdump命令查看某一个函数的GOT入口项位置

修改指令指针的目标：全局偏移表

```
% objdump --dynamic-reloc test-prog
format:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET         TYPE                    VALUE
08049bc0 R_386_GLOB_DAT                __gmon_start__
08049ba8 R_386_JUMP_SLOT               __libc_start_main
08049bac R_386_JUMP_SLOT               strcat
08049bb0 R_386_JUMP_SLOT               printf
08049bb4 R_386_JUMP_SLOT               exit
08049bb8 R_386_JUMP_SLOT               sprintf
08049bbc R_386_JUMP_SLOT               strcpy
```

为每一个R_386_JUMP_SLOT重定位记录指定的偏移量，包含了指定函数（或RTL链接函数）的地址

修改指令指针的目标：全局偏移表

- 如何利用GOT？
  - 攻击者需要有自己的shellcode
  - 攻击者需要能够向任意地址写入任意值
  - 攻击：
    - 攻击者用自己的shellcode地址覆写GOT地址（将要被使用的）

修改指令指针的目标：.dtors

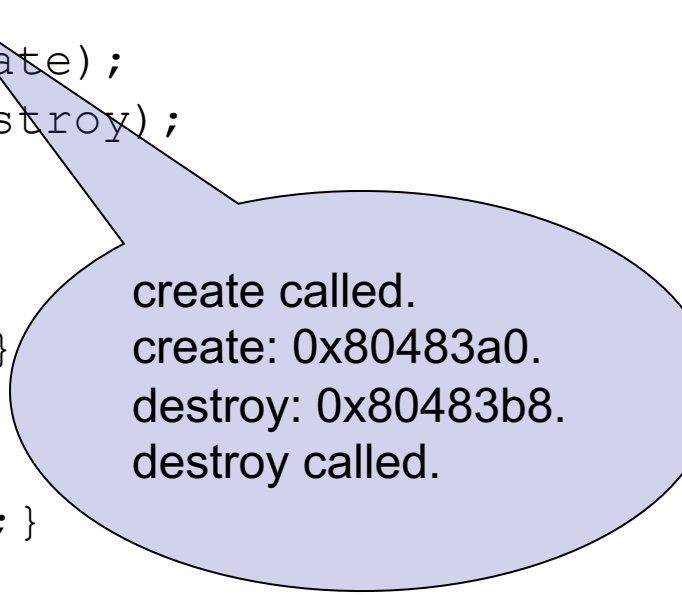
- gcc 允许
  - keyword is __attribute__

```
static void start(void) __attribute__ ((constructor));
static void stop(void) __attribute__ ((destructor));
```

  - constructor 属性:
    - 函数在main()之前被调用
  - destructor 属性:
    - 函数将在main()执行完成后进行调用
  - 构造函数和析构函数分别存储于生成的ELF可执行映像的.ctors和.dtors区中

修改指令指针的目标：.dtors

```
static void create(void)
       __attribute__ ((constructor));
static void destroy (void)
       __attribute__ ((destructor));

int main(int argc, char *argv[]) {
       printf("create: %p.\n", create);
       printf("destroy: %p.\n", destroy);
       exit(EXIT_SUCCESS);}

void create(void) {
       printf("create called.\n");}

void destroy(void) {
       printf("destroy called.\n");}
```



修改指令指针的目标：.dtors

- 这两个区都有如下的布局形式：
  - `0xffffffff {function-address} 0x00000000`
- .ctors和.dtors区映射到进程地址空间后，默认属性为可写
- 漏洞利用程序从未利用过构造函数，因为它们都在main()函数之前执行
- 攻击者的兴趣集中在析构函数和.dtors区上
- 可以使用objdump命令检查可执行映像中.dtors区中的内容

修改指令指针的目标：.dtors

- 攻击者可以通过覆写.dtors区中的函数指针的地址从而将程序控制权转移到任意的代码
- 如果攻击者能够读取到目标二进制文件，那么通过分析ELF映像，很容易就能确定要覆写的确切位置
- 即使没有指定任何析构函数.dtors区仍然存在
- 在这种情况下，.dtors区中只含有头、尾标签而中间没有函数地址
- 仍然可以通过将尾标签0x00000000覆写未攻击者提供的外壳代码的地址，从而将控制转移过去
- 如果外壳代码返回，则进程将会继续调用接下来的函数直到遇到尾标签或发现错误为止

修改指令指针的目标：.dtors

- 对于攻击者而言，覆写.dtors区的好处在于：
  - 该区总是存在并且会映射到内存中
- 然而：
  - .dtors仅存在于用GCC编译和链接的程序中
  - 有时候，很难找到合适的外壳代码注入点，使得在main()退出后外壳代码仍然能够驻留在内存中

修改指令指针的目标：虚函数

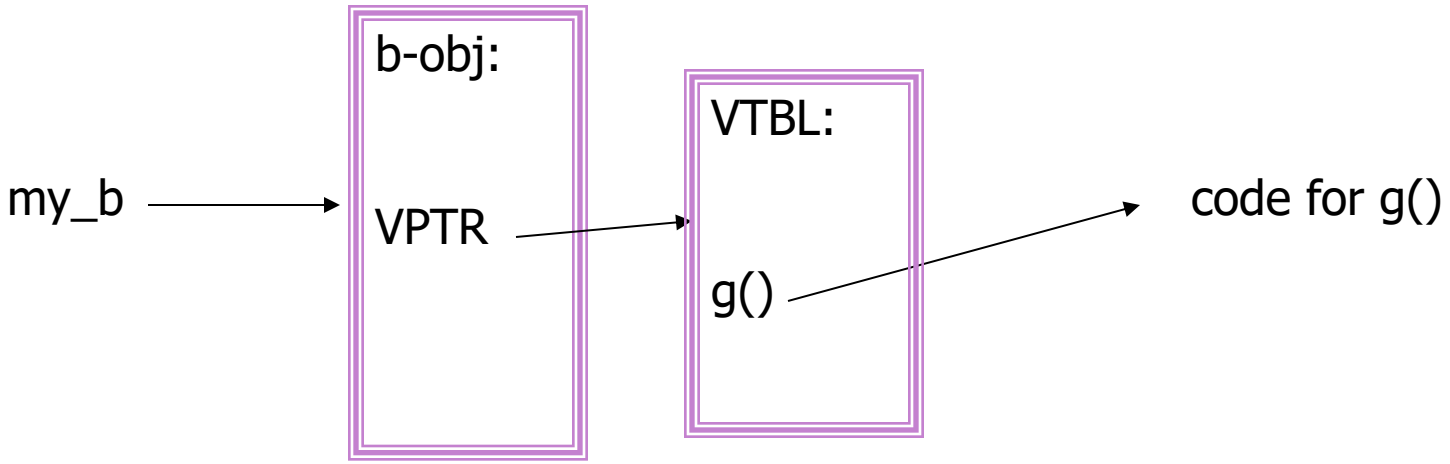
- 面向对象编程的重要特性
  - 允许函数调用的动态绑定（运行时解析）
  - 虚函数 是:
    - 类成员函数
    - 用virtual关键字声明
    - 可由派生类中的同名函数重写
    - 函数调用在运行时解析

修改指令指针的目标：虚函数

```
#include <iostream>
using namespace std;

class a {
public: void f(void) { cout << "base f" << endl;};
        virtual void g(void) { cout << "base g" << endl;};
};

class b: public a {
public: void f(void) { cout << "derived f" << endl;};
        void g(void) { cout << "derived g" << endl;};
};

int main(int argc, char *argv[]) {
        a *my_b = new b();
        my_b->f();
        my_b->g();
        return 0; }
```

修改指令指针的目标：虚函数

- 虚函数实现：
  - 虚函数表Virtual Function Table （VTBL）
  - VTBL是一个函数指针数组，用于在运行时派发虚函数调用
  - 在每一个对象的头部，都包含一个指向VTBL的虚指针VPTR（Virtual Pointer）
  - VTBL含有指向虚函数的每一个实现的指针



修改指令指针的目标：虚函数

- 攻击者可以：
  - 覆写VTBL中的函数指针或者
  - 改写VPTR使其指向其他任意的VTBL

- 攻击者可以通过任意内存写或者利用缓冲区溢出直接写入对象实现这一操作

修改指令指针的目标：atexit() on_exit()

- atexit()
  - C99定义的通用工具函数
  - 可以注册无参函数，并在程序正常结束后调用该函数
  - C99要求实现支持至少32个函数的注册

修改指令指针的目标：atexit() on_exit()

- on_exit()
  - 在SunOS下相似功能
  - 也存在libc4, libc5, glibc

修改指令指针的目标：
atexit() on_exit()

```
#include <stdio.h>
char *glob;
void test(void) {
        printf("%s", glob);
}

int main(void) {
        atexit(test);
        glob = "Exiting.\n";
}
```

修改指令指针的目标：atexit() on_exit()

- atexit()通过向一个退出时将被调用的已有函数数组中添加指定的函数完成工作
- exit()以后进先出（Last-in, First-out, LIFO）的顺序调用函数
- 数组被分配为全局性的符号
  - __atexit in BSD
  - __exit_funcs in Linux

修改指令指针的目标：
atexit() on_exit()

```
(gdb) b main
Breakpoint 1 at 0x80483f6: file atexit.c, line 6.
(gdb) r
Starting program: /home/rcs/book/dtors/atexit

Breakpoint 1, main (argc=1, argv=0xbfffe744) at atexit.c:6
6 atexit(test);
(gdb) next
7 glob = "Exiting.\n";
(gdb) x/12x __exit_funcs
0x42130ee0 <init>:    0x00000000 0x00000003 0x00000004 0x4000c660
0x42130ef0 <init+16>: 0x00000000 0x00000000 0x00000004 0x0804844c
0x42130f00 <init+32>: 0x00000000 0x00000000 0x00000004 0x080483c8
(gdb) x/4x 0x4000c660
0x4000c660 <_dl_fini>: 0x57e58955 0x5ce85356 0x81000054 0x0091c1c3
(gdb) x/3x 0x0804844c
0x804844c <__libc_csu_fini>: 0x53e58955 0x9510b850 x102d0804
(gdb) x/8x 0x080483c8
0x80483c8 <test>: 0x83e58955 0xec8308ec 0x2035ff08 0x68080496
```

修改指令指针的目标：atexit() on_exit()

- 在该调试会话中，在main()中调用atexit()之前设了一个断点，然后运行程序
- 接下来执行atexit()函数，注册test()函数
- 在test()函数注册后，显示了在__exit_funcs 位置处的内存
- 每一个函数都保存在由4个双字构成的结构中
  - 每一个结构的最后一个双字保存着函数的实际地址

修改指令指针的目标：
atexit() on_exit()

```
(gdb) b main
Breakpoint 1 at 0x80483f6: file atexit.c, line 6.
(gdb) r
Starting program: /home/rcs/book/dtors/atexit

Breakpoint 1, main (argc=1, argv=0xbfffe744) at atexit.c:6
6 atexit(test);
(gdb) next
7 glob = "Exiting.\n";
(gdb) x/12x __exit_funcs
0x42130ee0 <init>:    0x00000000 0x00000003 0x00000004 0x4000c660
0x42130ef0 <init+16>: 0x00000000 0x00000000 0x00000004 0x0804844c
0x42130f00 <init+32>: 0x00000000 0x00000000 0x00000004 0x080483c8
(gdb) x/4x 0x4000c660
0x4000c660 <_dl_fini>: 0x57e58955 0x5ce85356 0x81000054 0x0091c1c3
(gdb) x/3x 0x0804844c
0x804844c <__libc_csu_fini>: 0x53e58955 0x9510b850 x102d0804
(gdb) x/8x 0x080483c8
0x80483c8 <test>: 0x83e58955 0xec8308ec 0x2035ff08 0x68080496
```

修改指令指针的目标：atexit() on_exit()

- 示例中：
  - 3个函数已经被注册：
    - _dl_fini()
    - __libc_csu_fini()
    - test()
  - 攻击者可以覆写__exit_funcs 结构

修改指令指针的目标：longjmp()

- C99定义的可选函数调用和返回规则
  - 用于处理程序的低级子程序中遇到的错误和中断
  - setjmp() 宏
    - 保存调用环境
  - longjmp(), siglongjmp()
    - 非局部的跳转到保存的栈环境

修改指令指针的目标：longjmp()

```
#include <setjmp.h>

jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void){ setjmp(buf); g(n);}

void g(int n){ h(n);}

void h(int n) { longjmp(buf, 2);}

int main (void){
        f();
        return 0;
}
```

- longjmp() 示例：
  - longjmp()返回控制权给调用set_jmp()的指针

修改指令指针的目标：longjmp()

```
1. typedef int __jmp_buf[6];

2. #define JB_BX 0
3. #define JB_SI 1
4. #define JB_DI 2
5. #define JB_BP 3
6. #define JB_SP 4
7. #define JB_PC 5
8. #define JB_SIZE 24

9. typedef struct __jmp_buf_tag {
10.     __jmp_buf __jmpbuf;
11.     int __mask_was_saved;
12.     __sigset_t __saved_mask;
13. } jmp_buf[1]
```

- jmp_buf的Linux实现
- 注意JB_PC 域
- 这是攻击目标
- 任意写可以用缓冲区溢出shellcode的地址覆盖这个字段

修改指令指针的目标：longjmp()

Assembly instructions in Linux

```
longjmp(env, i)
movl i, %eax /* return i */
movl env.__jmpbuf[JB_BP], %ebp
movl env.__jmpbuf[JB_SP], %esp
jmp (env.__jmpbuf[JB_PC])
```

第2行movl指令恢复BP

第3行的movl 指令恢复栈指针

第4行则将程序控制权转移到保存的PC

修改指令指针的目标：异常处理

- 异常
  - 函数操作中发生的意外情况
  - Windows提供了三种形式的异常处理程序：
    - 向量化异常处理 Vectored Exception Handling (VEH)
      - Windows XP增加了对这种异常处理程序的支持
      - VEH首先调用以重写SEH
    - 结构化异常处理 Structured Exception Handling (SEH)
      - 被实现为每函数或每线程的异常处理程序
    - 系统默认异常处理 System Default Exception Handling

修改指令指针的目标：异常处理

- SEH
  - 通过try … catch块实现
    - try块中引发的任何异常都将被匹配的catch块处理
    - 如果catch块无法处理该异常，那么它将被传回之前的范围块
    - __finally 是微软对C/C++语言的扩展
      - 被调用来清理由try块说明的任何东西

```
try {
      // Do stuff here
}
catch(…){
      // Handle exception here
}
 __finally {
     // Handle cleanup here
}
```

修改指令指针的目标：
异常处理

- SEH
  - 使用EXECPTION_REGISTRATION结构
  - 栈上分配
  - Pre EXECPTION_REGISTRATION处于栈中较高的地址
  - 如果可执行映像头部列出了 SAFE SEH 处理程序地址，那么处理器地址必须被列为SAFE SEH处理程序。反之，任何结构化异常处理程序都可以被调用

```
EXCEPTION_REGISTRATION struc
    prev                    dd      ?
    handler                 dd      ?
_EXCEPTION_REGISTRATION ends
```

修改指令指针的目标：异常处理

栈帧初始化

- 注意
  - 异常处理程序地址紧跟在局部变量之后
  - 如果栈变量发生缓冲区溢出，那么异常处理程序地址就可以被覆写

```
push        ebp
mov         ebp, esp
and         esp, 0FFFFFFF8h
push        0FFFFFFFFh
push        ptr [Exception_Handler]
mov         eax, dword ptr fs:[00000000h]
push        eax
mov         dword ptr fs:[0], esp
```

修改指令指针的目标：异常处理

- 攻击者可以:
  - 覆写异常处理程序地址 （supra）
  - 替换Thread Environment Block （TEB）中的指针
    - TEB包含已注册的异常处理程序列表
      - 攻击者仿造一个列表入口作为攻击代码的一部分
      - 利用任意内存写技术修改第一个异常处理程序域
    - 似乎仍是可能攻击成功的，尽管最新版本的Windows已经加入了列表入口有效性检验功能

修改指令指针的目标：
异常处理

- Windows为进程提供了一个全局异常过滤器和处理程序，如果之前的异常处理程序都没能处理异常，那么该处理程序就会被调用
- 往往为整个进程实现一个未处理异常，使得程序能够优雅地处理非预期的错误或者只是为了调试方便
- 未处理异常过滤器函数利用 `SetUnhandledExceptionFilter()` 函数进行设置
- 如果攻击者利用任意内存写技术覆盖了某特定内存地址，则未处理异常过滤器可以被重定向去执行任意代码

修改指令指针的目标：异常处理

- 消除“允许内存被不正确地覆写”的漏洞
  - 这些错误出现在：
    - 覆写对象指针（本章中已有讨论）
    - 常见的动态内存管理错误
    - 字符串格式化漏洞
- 减少目标暴露
  - W^X
    - 降低有漏洞的进程的权限
    - 内存区域要么可写要么可执行，但不可同时二者兼备
      - 很难实现
      - 不能防止类似于atexit() 这样的同时需要运行时写入和可执行的目标覆写

修改指令指针的目标：异常处理

- 栈探测仪
  - 能够保护
    - 通过溢出栈缓冲区来覆写栈指针或者其他保护区域的漏洞利用
  - 不能保护
    - 包括栈段在内的任何位置发生缓冲区溢出
    - 修改的漏洞利用
      - 变量
      - 对象指针
      - 函数指针

修改指令指针的目标：总结

- 指针诡计攻击主要是针对抗击栈粉碎措施而发展起来的
  - 栈探测仪, Stack-Guard
- 方法是在目标指针的临近位置进行缓冲区溢出攻击
  - 函数指针覆写
    - 攻击者可以将控制转移到任意的攻击代码
  - 对象指针覆写
    - 能导致内存任意写
    - 对象指针覆写能修改大量的目标，可被攻击者利用

谢谢大家！