Day 21

我们之前第20天的时候已经提前解决了今天开头的这个小问题，所以我们先跳过了。

之前我们所谓的应用程序都是用汇编语言写的，不过用汇编语言写程序真是太累了，我们得想办法。

我们的思路是这样的：用汇编语言写API，c语言写程序，然后都编译成符号文件，然后进行链接。最后用bim2hrb转换一下。

我们写了一个输出一个字符的程序，装入映像中。make run。然而系统却没了反应。

咋回事呢？根据作者提供的方法，我们修改了程序的前6个字节为`E8 16 00 00 00 CB`，就能顺利运行了。

这六个字节，对应的nask命令如下

```
call 0x1b
retf
```

相当于调用0x1b位置的函数，然后进行far return

0x1b位置是什么呢？他正好是我们的写的main函数所处的位置。

这样我们就搞明白整个的工作原理了。

C语言（或者是bim2hrb，不确定）编译（链接）出来的程序的main函数并不存在于逻辑地址的0x00位置，而是再0x1b的位置上，而我们的操作系统则是从文件的开始位置执行的。于是我们在开始位置替换两条命令，使得应用程序能够调用main，并且运行完之后返回系统。

不过每次替换程序的开头的6个字节实在是太麻烦，所以我们把他交给操作系统来做吧！

在进行farcall之前，做如下操作

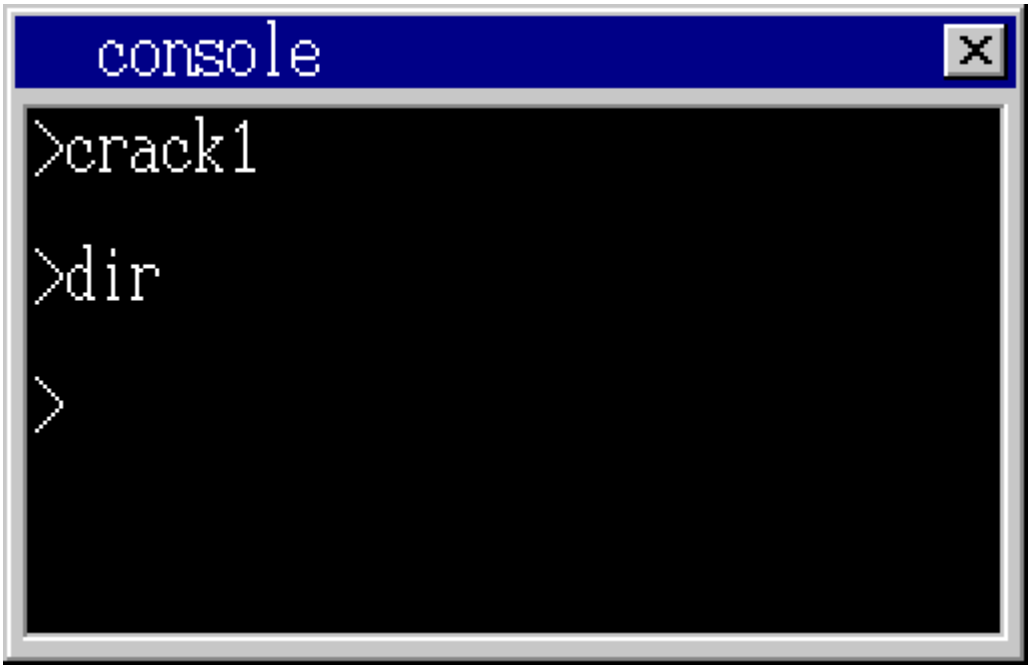
```
if (finfo->size >= 8 && strncmp(p + 4, "Hari", 4) == 0) {
    p[0] = 0xe8;
    p[1] = 0x16;
    p[2] = 0x00;
    p[3] = 0x00;
    p[4] = 0x00;
    p[5] = 0xcb;
}
```

为什么size要至少为8呢，因为我们要比较4到7byte，看这个是不是c语言的程序（bim2hrb生成的文件这4个字节一定是"Hari"）（要是汇编语言写的也做替换的话会出问题）。

接下来我们搞搞系统保护。由于应用程序的开发不受操作系统的控制，我们不能确保每个开发者不具有恶意并且水平足够高，所以我们要进行操作系统保护以防止操作系统遭到意外破坏无法正常运行。

我们进行一个小小的破坏系统的测试，讲0x00102600位置修改为0

```
void HariMain(void)
{
    *((char *) 0x00102600) = 0;
    return;
}
```



可见dir命令已经坏掉了。

如何保护系统呢？我们应当把应用程序对内存的操作和访问限制在一定的范围内。我们创建应用程序专用的数据段

操作系统用代码段	2 * 8
操作系统用数据段	1 * 8
应用程序用代码段	1003 * 8
应用程序用数据段	1004 * 8
TSS段	3 * 8 ~ 1002 * 8

```
if (finfo != 0) {
    p = (char *) memman_alloc_4k(memman, finfo->size);
    q = (char *) memman_alloc_4k(memman, 64 * 1024); // 为数据段分配空间
    *((int *) 0xfe8) = (int) p;
    file_loadfile(finfo->clustno, finfo->size, p, fat, (char *) (ADR_DISKIMG + 0x003e00));
    set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER);
    set_segmdesc(gdt + 1004, 64 * 1024 - 1,   (int) q, AR_DATA32_RW); // 设定gdt表，把刚刚分配的空间注册了
    if (finfo->size >= 8 && strncmp(p + 4, "Hari", 4) == 0) {
        p[0] = 0xe8;
        p[1] = 0x16;
        p[2] = 0x00;
```

```
        p[3] = 0x00;
        p[4] = 0x00;
        p[5] = 0xcb;
    }
    start_app(0, 1003 * 8, 64 * 1024, 1004 * 8); // 指定cs,  ds
    memman_free_4k(memman, (int) p, finfo->size);
    memman_free_4k(memman, (int) q, 64 * 1024); // 释放空间
    cons_newline(cons);
    return 1;
}
```

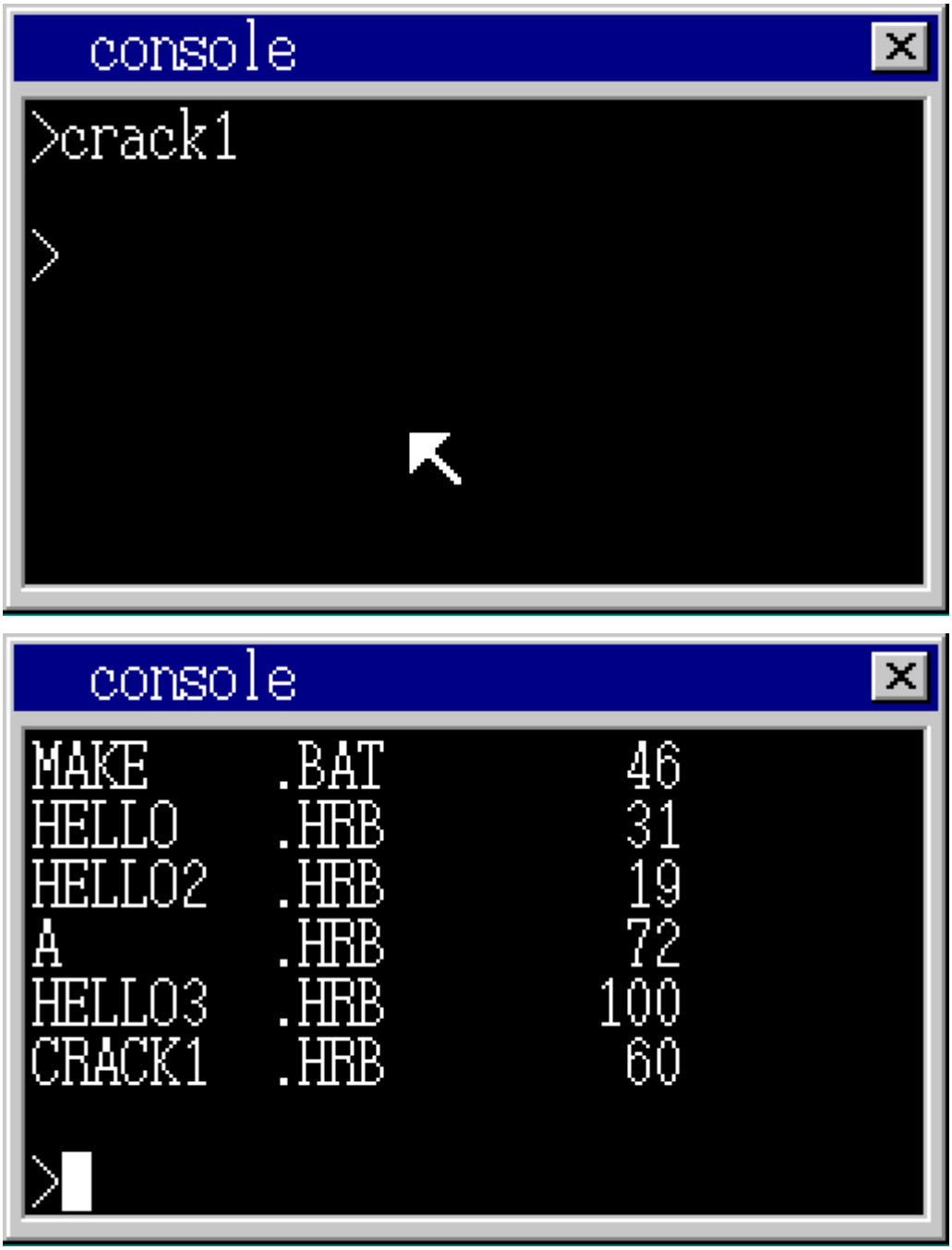
```
_start_app:     ; void start_app(int eip, int cs, int esp, int ds);
        PUSHAD      ; 保存现场
        MOV     EAX,[ESP+36]    ; 保存参数eip
        MOV     ECX,[ESP+40]    ; 保存参数cs
        MOV     EDX,[ESP+44]    ; 保存参数esp
        MOV     EBX,[ESP+48]    ; 保存参数ds
        MOV     [0xfe4],ESP     ; 保存操作系统的esp到内存中
        CLI         ; 切换时关中断
        MOV     ES,BX
        MOV     SS,BX
        MOV     DS,BX
        MOV     FS,BX
        MOV     GS,BX
        MOV     ESP,EDX ; 导入应用程序esp
        STI         ; 切换完成恢复中断
        PUSH    ECX             ; cs
        PUSH    EAX             ; eip
        CALL    FAR [ESP]       ; 调用应用程序

        MOV     EAX,1*8         ; 恢复操作系统的ds
        CLI         ; 切换，关中断
        MOV     ES,AX
        MOV     SS,AX
        MOV     DS,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     ESP,[0xfe4]
        STI         ; 恢复中断
        POPAD   ; 恢复现场
        RET
```

别忘了hrb_api，它应当工作在内核态下，我们应该将段切换回系统，否则无法正常工作

对了，还有中断处理的也需要改！因为我们的程序在运行过程中中断可不会停止产生。

总之，任何可能返回内核态的位置都需要进行修改。真是太麻烦了！

我们就不在这里写代码了。方法和start_app大同小异。



操作系统没有被破坏，好极了！

我们日常使用的操作系统在程序产生错误的时候都可以自动结束并报错，我们也引入这样的功能。

> 当应用程序试图破坏操作系统，或者试图违背操作系统的设置时，就会自动产生0x0d中断

也就是说，我们只要注册0x0d中断，就可以对应用程序的错误进行响应。

由于是中断，我们还是要进行切换的设置（微笑）

```
_asm_inthandler0d:
        STI
        PUSH ES
        PUSH DS
        PUSHAD
        MOV AX,SS
```

```
        CMP AX,1*8
        JNE .from_app ; 判断是操作系统还是应用程序产生的0x0d中断
; 当操作系统活动时产生中断的情况和之前差不多
        MOV EAX,ESP
        PUSH SS ; 保存中断时的SS
        PUSH EAX ; 保存中断时的ESP
        MOV AX,SS
        MOV DS,AX
        MOV ES,AX
        CALL _inthandler0d
        ADD ESP,8
        POPAD
        POP DS
        POP ES
        ADD ESP,4 ; 在INT 0x0d中需要这句
        IRETD
        .from_app:
; 当应用程序活动时产生中断
        CLI
        MOV EAX,1*8
        MOV DS,AX ; 先仅将DS设定为操作系统用
        MOV ECX,[0xfe4] ; 操作系统的ESP
        ADD ECX,-8
        MOV [ECX+4],SS ; 保存产生中断时的SS
        MOV [ECX ],ESP ; 保存产生中断时的ESP
        MOV SS,AX
        MOV ES,AX
        MOV ESP,ECX
        STI
        CALL _inthandler0d
        CLI
        CMP EAX,0
        JNE .kill
        POP ECX
        POP EAX
        MOV SS,AX ; 将SS恢复为应用程序用
        MOV ESP,ECX ; 将ESP恢复为应用程序用
        POPAD
        POP DS
        POP ES
        ADD ESP,4 ; INT 0x0d需要这句
        IRETD
.kill:
; 将应用程序强制结束
        MOV EAX,1*8 ; 操作系统用的DS/SS
        MOV ES,AX
        MOV SS,AX
        MOV DS,AX
        MOV FS,AX
        MOV GS,AX
        MOV ESP,[0xfe4] ; 强制返回到start_app时的ESP
        STI ; 切换完成后恢复中断请求
        POPAD ; 恢复事先保存的寄存器值
```

```
        RET
```

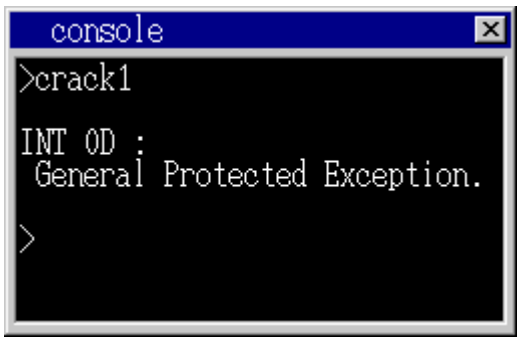
这个函数与20号中断处理函数的区别在于用STI/CLI禁止、允许终端，根据handler的结果决定是否结束程序。

> 在强制结束时，尽管中断处理完成了但却没有使用IRETD指令，而且还把栈强制恢复到start_app时的状态，使程序返回到cmd_app。可能大家会问，这种奇怪的做法真的没问题吗？是的，完全没问题。

然后我们写中断服务程序并将其注册到IDT中

```
int inthandler0d(int *esp)
{
    struct CONSOLE *cons = (struct CONSOLE *) *((int *) 0x0fec);
    cons_putstr0(cons, "\nINT 0D :\n General Protected Exception.\n");
    return 1;
}
```

make run(由于提前知悉了qemu的问题，所以放在virtual box上面跑了)



保护成功!

我们现在可以防御这种简单的bug/攻击了，如果恶意程序直接修改DS寄存器的内容的话，我们的操作系统还是会中招。具体的防御办法是为段加上访问权限

> 在段定义的地方，如果将访问权限加上0x60的话，就可以将段设置为应用程序用。当CS中 的段地址为应用程序用段地址时，CPU会认为“当前正在运行应用程序”，这时如果存入操作系 统用的段地址就会产生异常。

这样CPU就可以自动帮我们做切换了!

```
int cmd_app(struct CONSOLE *cons, int *fat, char *cmdline)
{
    （中略）
    char name[13], *p, *q;
    struct TASK *task = task_now();      /*这里！*/

    （中略）

    if (finfo != 0) {
        /*找到文件的情况*/
        （中略）
        set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER + 0x60);     /*从此开始*/
        set_segmdesc(gdt + 1004, 64 * 1024 - 1,   (int) q, AR_DATA32_RW + 0x60);
        （中略）
        start_app(0, 1003 * 8, 64 * 1024, 1004 * 8, &(task->tss.esp0));              /*到此结束*/
        （中略）
    }
    /*没有找到文件的情况*/
    return 0;
}
```

还有一个问题，使用这种方法的话，我们就没法进行farcall和farjmp了，怎么办呢？我们使用了一个小trick。我们将跳转地址压入栈中，然后retf。这样就可以规避x86的限制了。

```
_start_app: ; void start_app(int eip, int cs, int esp, int ds, int *tss_esp0);
        PUSHAD ; 将32位寄存器的值全部保存下来
        MOV EAX,[ESP+36] ; 应用程序用EIP
        MOV ECX,[ESP+40] ; 应用程序用CS
        MOV EDX,[ESP+44] ; 应用程序用ESP
        MOV EBX,[ESP+48] ; 应用程序用DS/SS
        MOV EBP,[ESP+52] ; tss.esp0的地址
        MOV [EBP ],ESP ; 保存操作系统用ESP
        MOV [EBP+4],SS ; 保存操作系统用SS
        MOV ES,BX
        MOV DS,BX
        MOV FS,BX
        MOV GS,BX
        ; 下面调整栈，以免用RETF跳转到应用程序
        OR ECX,3 ; 将应用程序用段号和3进行OR运算
        OR EBX,3 ; 将应用程序用段号和3进行OR运算
        PUSH EBX ; 应用程序的SS
        PUSH EDX ; 应用程序的ESP
        PUSH ECX ; 应用程序的CS
        PUSH EAX ; 应用程序的EIP
        RETF
        ; 应用程序结束后不会回到这里
```

由于我们并非使用farcall跳转到应用程序，所以我们从应用程序返回系统就不能使用retf了，需要使用别的办法。

我们搞一个结束应用程序的API，给他功能号4

```
} else if (edx == 4) {          /*这里！*/
    return &(task->tss.esp0);   /*这里！*/
}
```

然后我们把中断处理程序改回去。CPU替我们做过切换了，不改回去会出问题的。

然后我们修改一下IDT，给中断加上权限，只允许应用程序调用hrb_api中断(0x40)

```
set_gatedesc(idt + 0x0d, (int) asm_inthandler0d, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x20, (int) asm_inthandler20, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x21, (int) asm_inthandler21, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x27, (int) asm_inthandler27, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x2c, (int) asm_inthandler2c, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x40, (int) asm_hrb_api, 2 * 8, AR_INTGATE32 + 0x60);
```

应用程序也得修改了，因为我们不能用retf结束程序了，要用api。