Day 22

今天我们来测试一下我们的操作系统到底能防御多少攻击

重新设定定时器

```
MOV AL,0x34
OUT 0x43,AL
MOV AL,0xff
OUT 0x40,AL
MOV AL,0xff
OUT 0x40,AL
MOV EDX,4
INT 0x40
```

防御成功，在应用程序模式下是不允许执行in、out指令的。

关中断

```
CLI
fin:
HTL
JMP fin
```

防御成功，CLI是不允许的

---

farcall

```
CALL 2*8:0xac1
MOV EDX,4
INT 0x40
```

防御成功，far call是不允许的

操作系统有后门，调用特定API

```
} else if (edx == 123456789) {
    *((char *) 0x00102600) = 0;
}
```

```
MOV EDX,123456789
INT 0x40
MOV EDX,4
INT 0x40
```

防御失败，放恶人进来，CPU也救不了你。毕竟你给了应用程序调用0x40中断的权限

bug处理

程序开发人员可能会出各种各样的疏忽，导致应用程序有bug，无意的破坏操作系统。

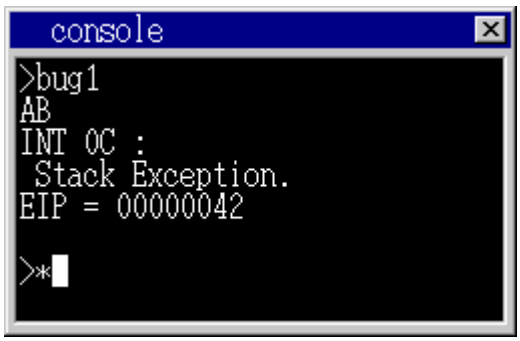
一个很常见的事情就是数组下标访问越界。

```
void api_putchar(int c);
void api_end(void);
void HariMain(void)
{
    char a[100];
    a[10] = 'A';
    api_putchar(a[10]);
    a[102] = 'B'; /*越界*/
    api_putchar(a[102]);
    a[123] = 'C'; /*越界*/
    api_putchar(a[123]);
    api_end();
}
```

virtual box运行，虚拟机重启了

这是因为这个bug触发了新的中断，0x0c，我们没有对他进行处理。

处理函数的代码和0x0d的几乎一模一样，只有输出不太一样。不要忘记注册它到IDT中



我们还可以加上相应的调试信息。

为什么显示了AB之后才产生exception呢?

> 可能有人会问，为什么“C”会被判定为异常而“B”就可以被放过去呢？下面我们就 来简单讲一讲。 a[102]虽然超出了数组的边界，但却没有超出为应用程序分配的数据段的边界，因此虽 然这是个bug，CPU也不会产生异常。另一方面，a[123]所在的地址已经超出了数据段的边界， 因此CPU马上就发现并产生了异常。 其实，CPU产生异常的目的并不是去发现bug，而是为了保护操作系统，它的思路是： “这个程序试图访问自身所在数据段以外的内存地址，一定是想擅自改写操作系统或者其他 应用程序所管理的内存空间，这种行为岂能放任不管？”因此，即便CPU不能帮我们发现所 有的bug，也不可以责怪它哦。

死循环处理

```
void HariMain(void) {
    for (;;) { }
}
```

遇到这样的空循环怎么办？操作系统和用户都无可奈何。我们可以仿照我们日常使用的操作系统中的设计，使用特定的按键来终结正在运行的程序。

命令行窗口在运行的时候并不会检查他的FIFO队列，所以我们不能在console当中进行处理。因为console并不会知道我们按下了结束程序的快捷键。我们把对强制结束的处理放在bootpack。

我们把强制结束键设置成`Shift`+`F1`吧

```
if (i == 256 + 0x3b && key_shift != 0 && task_cons->tss.ss0 != 0) { /* Shift+F1 */
    cons = (struct CONSOLE *) *((int *) 0x0fec);
    cons_putstr0(cons, "\nBreak(key):\n");
    io_cli();
    task_cons->tss.eax = (int) &(task_cons->tss.esp0);
    task_cons->tss.eip = (int) asm_end_app;
    io_sti();
}
```

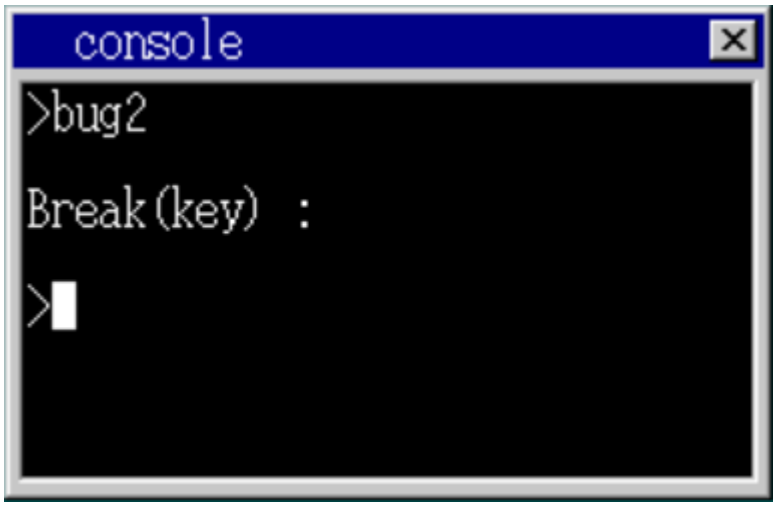
这段代码的原理是，当按下`Shift`+`F1`时，修改命令行窗口的寄存器值，然后goto到asm_end_app。光这样做还不够，我们在应用程序没有运行的时候按下`Shift`+`F1`键会出问题，也会进行跳转。

我们可以用tss中的ss0来进行标记

```
struct TASK *task_alloc(void)
{
    int i;
    struct TASK *task;
    for (i = 0; i < MAX_TASKS; i++) {
        if (taskctl->tasks0[i].flags == 0) {
        task = &taskctl->tasks0[i];
        task->flags = 1;
        task->tss.eflags = 0x00000202; /* IF = 1; */
        task->tss.eax = 0;
        // ........
        task->tss.iomap = 0x40000000;
        task->tss.ss0 = 0; /*这里! */
        return task;
```

```
        }
    }
    return 0; /*已经全部正在使用*/
}
```

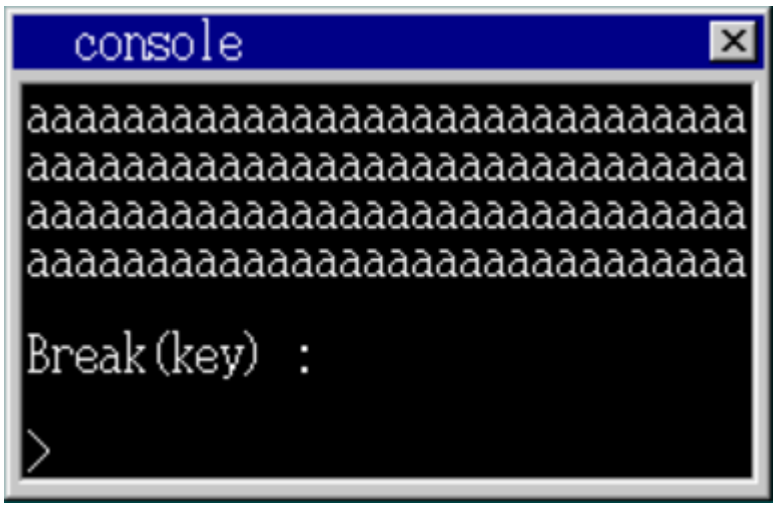
make run



按下 Shift + F1 。成功结束程序

```
void api_putchar(int c);
void api_end(void);
void HariMain(void)
{
    for (;;) {
        api_putchar('a');
    }
}
```

测试一下这个



我们来实现写入字符串的API

由于现在使用API来结束程序，所以我们修改开头6字节的操作也不必要了。我们直接在start_app时设定起始运行地址就ok了

遇到了一些麻烦.....

```
void api_putstr0(char *s);
void api_end(void);
void HariMain(void)
{
    api_putstr0("hello, world\n");
    api_end();
}
```

这段程序并没有正确的打印helloworld，我们的API看起来也似乎没什么问题。

通过阅读我们了解到，程序出现问题的主要原因是无法定位数据部分

hrb前32个字节的内容和定义

0x0000 (DWORD)	请求操作系统为应用程序准备的数据段的大小
0x0004 (DWORD)	“Hari ”（.hrb 文件的标记）
0x0008 (DWORD)	数据段内预备空间的大小
0x000c (DWORD)	ESP 初始值& 数据部分传送目的地址
0x0010 (DWORD)	hrb 文件内数据部分的大小
0x0014 (DWORD)	hrb 文件内数据部分从哪里开始
0x0018 (DWORD)	0xe9000000
0x001c (DWORD)	应用程序运行入口地址 - 0x20
0x0020 (DWORD)	malloc

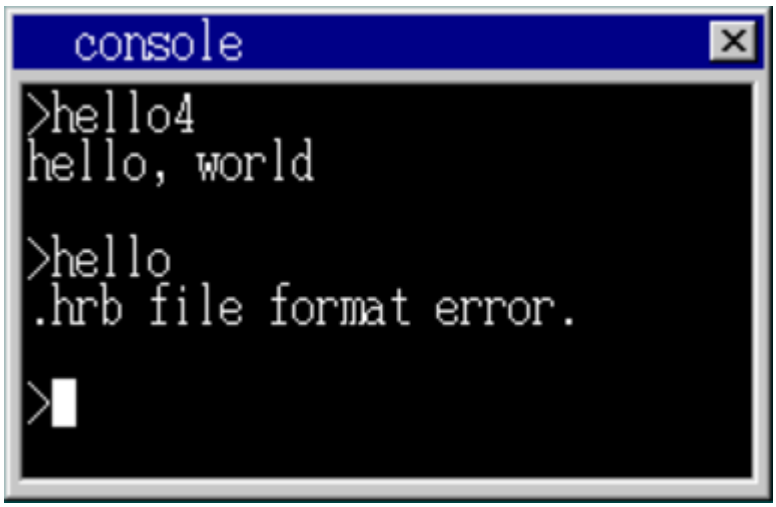
我们要完成以下几个任务

- 文件中找不到“Hari”标志则报错。
- 数据段的大小根据.hrb文件中指定的值进行分配。
- 将.hrb文件中的数据部分先复制到数据段后再启动程序。

```
if (finfo->size >= 36 && strncmp(p + 4, "Hari", 4) == 0 && *p == 0x00) {
    segsiz = *((int *) (p + 0x0000));
    esp = *((int *) (p + 0x000c));
    datsiz = *((int *) (p + 0x0010));
    dathrb = *((int *) (p + 0x0014));  //  根据上面的约定获取各种信息
    q = (char *) memman_alloc_4k(memman, segsiz);
```

```
    *((int *) 0xfe8) = (int) q;
    set_segmdesc(gdt + 1003, finfo->size - 1, (int) p, AR_CODE32_ER + 0x60);
    set_segmdesc(gdt + 1004, segsiz - 1, (int) q, AR_DATA32_RW + 0x60);
    for (i = 0; i < datsiz; i++) {  // 拷贝数据区数据过去
        q[esp + i] = p[dathrb + i];
    }
    start_app(0x1b, 1003 * 8, esp, 1004 * 8, &(task->tss.esp0));
    memman_free_4k(memman, (int) q, segsiz);
}
```

测试



再写一个汇编语言的程序

```
[FORMAT "WCOFF"]
[INSTRSET "i486p"]
[BITS 32]
[FILE "hello5.nas"]
        GLOBAL  _HariMain
[SECTION .text]
_HariMain:
        MOV     EDX,2
        MOV     EBX,msg
        INT     0x40
        MOV     EDX,4
        INT     0x40
[SECTION .data]
msg:
        DB  "hello, world", 0x0a, 0
```

利用bim2hrb我们生成上文所提到的那些信息，我们就可以提前分配好数据区了。

显示窗口。之前我们一直在跟字符界面打交道，我们现在来给我们的应用程序添加UI功能。为系统添加创建窗口的API

EDX	5
EBX	窗口缓冲区
ESI	窗口在x轴方向上的大小（即窗口宽度）
EDI	窗口在y轴方向上的大小（即窗口高度）
EAX	透明色
ECX	窗口名称

调用后的返回值：

EAX	用于操作窗口的句柄（用于刷新窗口等操作）

之前我们写的API都不具有返回值，这次我们要为API添加返回值，那么如何返回寄存器的值呢？

我们在asm_hrb_api中进行了两次push_ad。第一次是为了保存现场，第二次是为了传参。只要我们把参数修改掉，就可以通过popad获得修改后的值，从而实现返回值的功能。

```
struct SHTCTL *shtctl = (struct SHTCTL *) *((int *) 0x0fe4); /*从此开始*/
struct SHEET *sht;
int *reg = &eax + 1; // 保存的现场
/* reg[0] : EDI, reg[1] : ESI, reg[2] : EBP, reg[3] : ESP */
/* reg[4] : EBX, reg[5] : EDX, reg[6] : ECX, reg[7] : EAX */
```

```
else if (edx == 5) {
    sht = sheet_alloc(shtctl);
    sheet_setbuf(sht, (char *) ebx + ds_base, esi, edi, eax);
    make_window8((char *) ebx + ds_base, esi, edi, (char *) ecx + ds_base, 0);
    sheet_slide(sht, 100, 50);
    sheet_updown(sht, 3); /*背景层高度3位于task_a之上*/
    reg[7] = (int) sht;
}
```

```
_api_openwin: ; int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
PUSH EDI
PUSH ESI
PUSH EBX
MOV EDX,5
MOV EBX,[ESP+16] ; buf
MOV ESI,[ESP+20] ; xsiz
MOV EDI,[ESP+24] ; ysiz
MOV EAX,[ESP+28] ; col_inv
MOV ECX,[ESP+32] ; title
INT 0x40
POP EBX
POP ESI
POP EDI
RET
```

以上是我们的API设计

我们编写个测试程序：

```
int api_openwin(char *buf, int xsiz, int ysiz, int col_inv, char *title);
void api_end(void);
char buf[150 * 50];
void HariMain(void)
{
    int win;
    win = api_openwin(buf, 150, 50, -1, "hello");
    api_end();
}
```

make run

成功显示！

再接再厉，添加一些在窗口中绘制其他元素的API

显示字符API

EDX	6
EBX	窗口句柄
ESI	显示位置的x坐标
EDI	显示位置的y坐标
EAX	色号
ECX	字符串长度
EBP	字符串

绘制方块API

EDX	7
EBX	窗口句柄
EAX	x0
ECX	y0
ESI	x1
EDI	y1
EBP	色号

hrb_api部分

```
else if (edx == 6) {
    sht = (struct SHEET *) ebx;
    putfonts8_asc(sht->buf, sht->bxsize, esi, edi, eax, (char *) ebp + ds_base);
    sheet_refresh(sht, esi, edi, esi + ecx * 8, edi + 16);
} else if (edx == 7) {
    sht = (struct SHEET *) ebx;
    boxfill8(sht->buf, sht->bxsize, ebp, eax, ecx, esi, edi);
    sheet_refresh(sht, eax, ecx, esi + 1, edi + 1);
}
```

然后我们修改应用程序

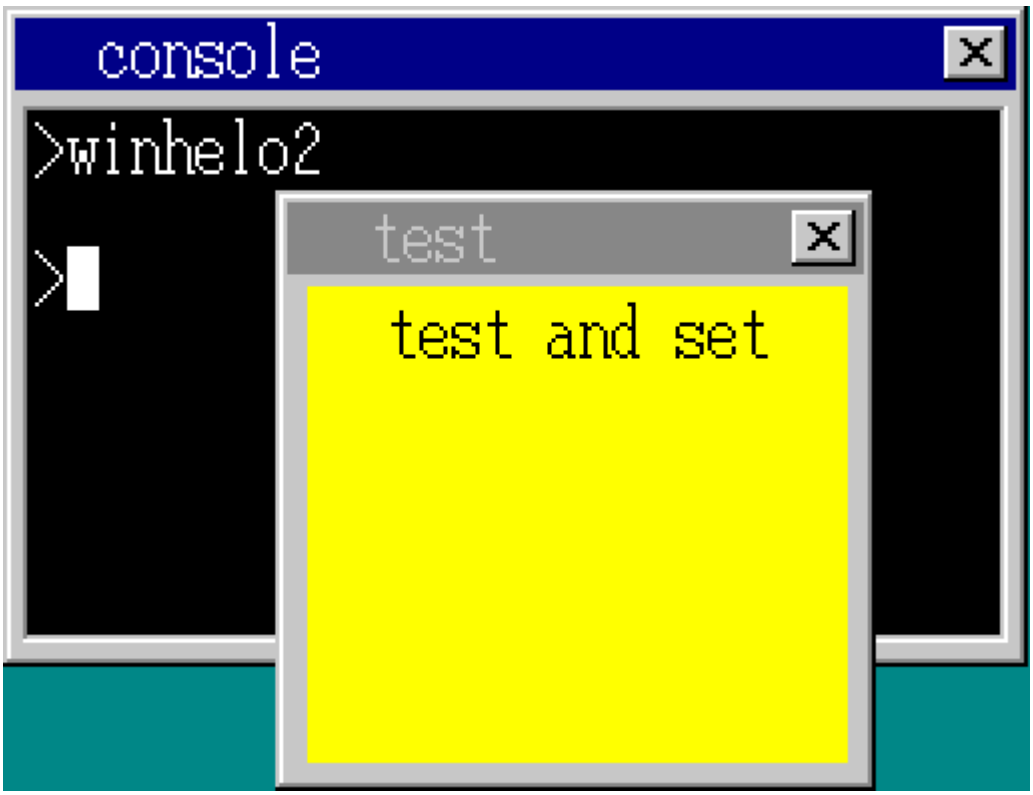
修改a_nask.nas

```
_api_putstrwin: ; void api_putstrwin(int win, int x, int y, int col, int len, char *str);
        PUSH EDI
        PUSH ESI
        PUSH EBP
        PUSH EBX
```

```
        MOV EDX,6
        MOV EBX,[ESP+20] ; win
        MOV ESI,[ESP+24] ; x
        MOV EDI,[ESP+28] ; y
        MOV EAX,[ESP+32] ; col
        MOV ECX,[ESP+36] ; len
        MOV EBP,[ESP+40] ; str
        INT 0x40
        POP EBX
        POP EBP
        POP ESI
        POP EDI
        RET
_api_boxfilwin: ; void api_boxfilwin(int win, int x0, int y0, int x1, int y1, int col);
        PUSH EDI
        PUSH ESI
        PUSH EBP
        PUSH EBX
        MOV EDX,7
        MOV EBX,[ESP+20] ; win
        MOV EAX,[ESP+24] ; x0
        MOV ECX,[ESP+28] ; y0
        MOV ESI,[ESP+32] ; x1
        MOV EDI,[ESP+36] ; y1
        MOV EBP,[ESP+40] ; col
        INT 0x40
        POP EBX
        POP EBP
        POP ESI
        POP EDI
        RET
```

我们测试如下程序

```
void HariMain(void)
{
    int win;
    win = api_openwin(buf, 150, 150, -1, "test");
    api_boxfilwin(win,  8, 24, 142, 142, 3);
    api_putstrwin(win, 28, 28, 0, 12, "test and set");
    api_end();
}
```


console
>winhelo2
>
test
test and set