



赛客  
SAIKE

# 内存PWN基础

BASIC OF MEMORY PWN

# 目录

操作系统基础

01

02

进程管理

进程虚拟空间与  
链接

03

04

\*nix软件调试环境

二进制程序漏洞挖掘

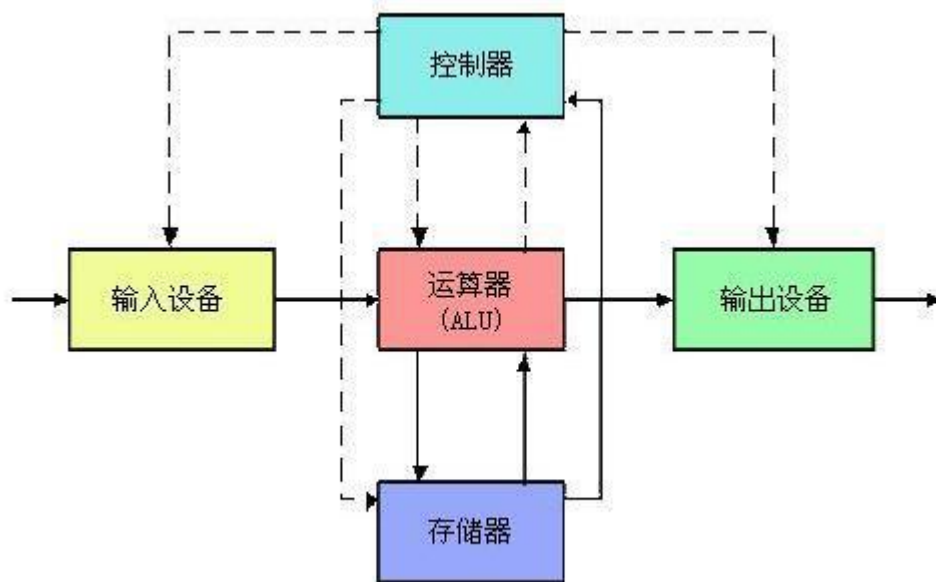
05

# 第一章

## 操作系统 基础

# 第一章 操作系统基础

## 01 计算机基础---- 冯·诺依曼计算机工作模型

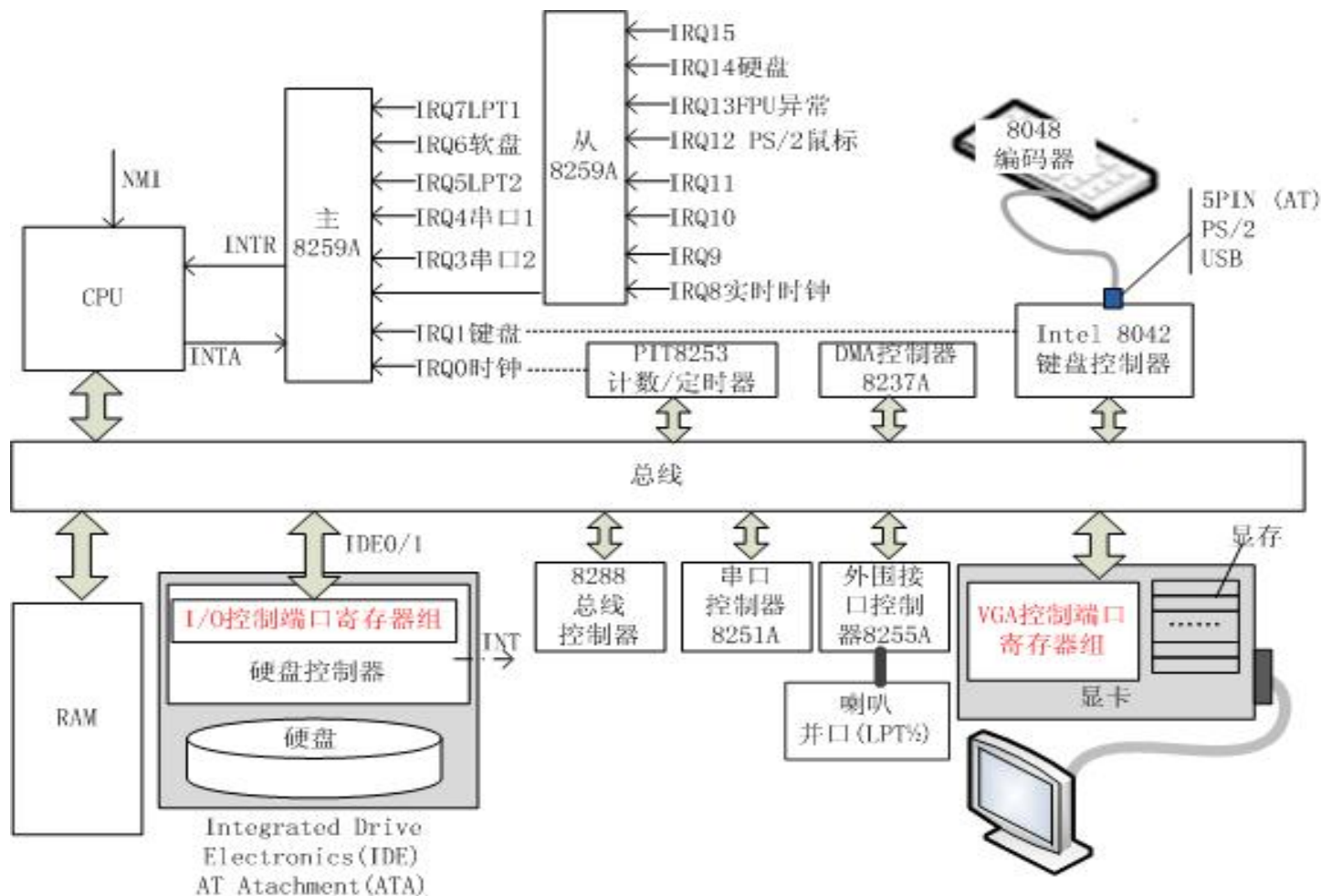


存储器用来容纳程序和数据

程序由指令组成，并和**数据一起存储**在计算机内存中。

# 第一章 操作系统基础

## 01 计算机基础---- 一种简明的计算机结构



# 第一章 操作系统基础

## 02：操作系统目标

是计算机系统的一个**系统软件**，是一些具有如下功能的程序模块的集合：

- ✓ 能有效地组织和管理计算机硬件和软件资源
- ✓ 能合理组织计算机的工作流程，控制程序的执行
- ✓ 能透明地向用户提供各种服务功能，使用户能够灵活、方便地使用计算机，使整个计算机系统能高效地运行

# 第一章 操作系统基础

## 02：操作系统目标

### ● 操作系统的目标（质量模型）

- ✓ 方便性（方便，易学、易用）
- ✓ 有效性（有效管理各类系统核心资源，提高系统的利用率和吞吐率）
- ✓ 可扩充性（可修改性，可扩展性好）
- ✓ 开放性（移植性，互操作性好）

### ● 操作系统的作用

- ✓ 作为计算机系统资源的管理者；
- ✓ 作为用户与计算机硬件系统之间的接口；

### ● 操作系统结构模型

- ✓ 一般采用基于特权级保护的层次化结构模型



# 第一章 操作系统基础

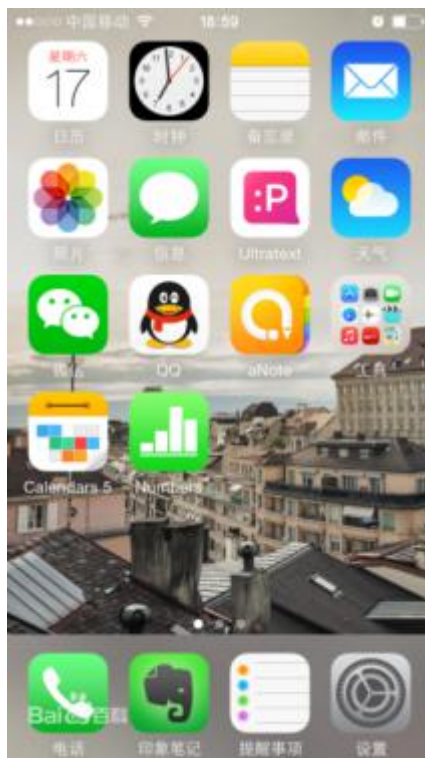
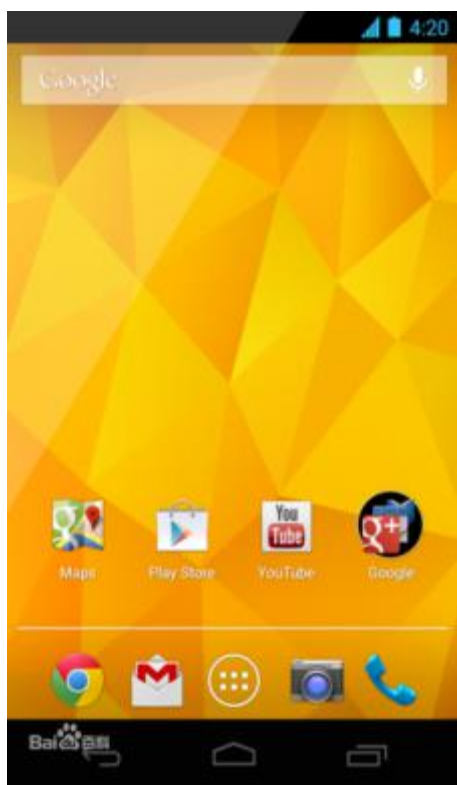
## 03：典型操作系统





# 第一章 操作系统基础

## 03：典型操作系统



# 第一章 操作系统基础

## 04: Unix V.S. Windows

- UNIX:

- ✓ 内核和外层有机结合。内核小、简洁，常驻内存，保证系统高效运行；外层包含非常的实用程序和丰富的支持软件——各种工具容易串接组合。
- ✓ 可移植性、可扩展性好和安全性好；
- ✓ 是一个支持多任务多用户系统
- ✓ 开放源代码

- WINDOWS

- ✓ 多任务操作环境
- ✓ 图形化工作环境和用户界面，界面友好（→傻瓜机）
- ✓ 属于准微内核体系，兼顾性能和效率



V.S.



# 第一章 操作系统基础

## 05：操作系统主要功能概述

### 一、进程与线程管理

- 主要任务是对CPU的分配和运行实施有效管理
- 具体功能包括
  - 进程控制：负责进程的创建、撤销和状态转换
  - 进程同步：对并发执行的多进程进行协调
  - 进程通信：负责完成进程间的信息交换
  - 进程调度：按一定的算法进行CPU分配

# 第一章 操作系统基础

## 05：操作系统主要功能概述

### 二、存储管理

- 主要任务是对内存进行分配、保护和扩充
- 具体功能包括
  - 内存分配：按一定的策略为每道程序分配内存
  - 内存保护：保证各程序在自己的内存区域内运行不受其它并发执行程序影响。
  - 内存扩充：为允许大型作业或多作业并发运行，必须借助虚拟存储技术来获得更大“虚拟”内存

# 第一章 操作系统基础

## 05：操作系统主要功能概述

### 三、设备管理

- 是OS中最庞杂、最琐碎部分
- 具体功能包括
  - 设备分配：按一定原则对设备进行分配。为使设备能与主机并行工作，需大量采用缓冲技术和虚拟技术
  - 设备传输控制：实现物理设备的I/O操作，包括启动、中断处理和结束处理等操作。

# 第一章 操作系统基础

## 05：操作系统主要功能概述

### 四、文件管理

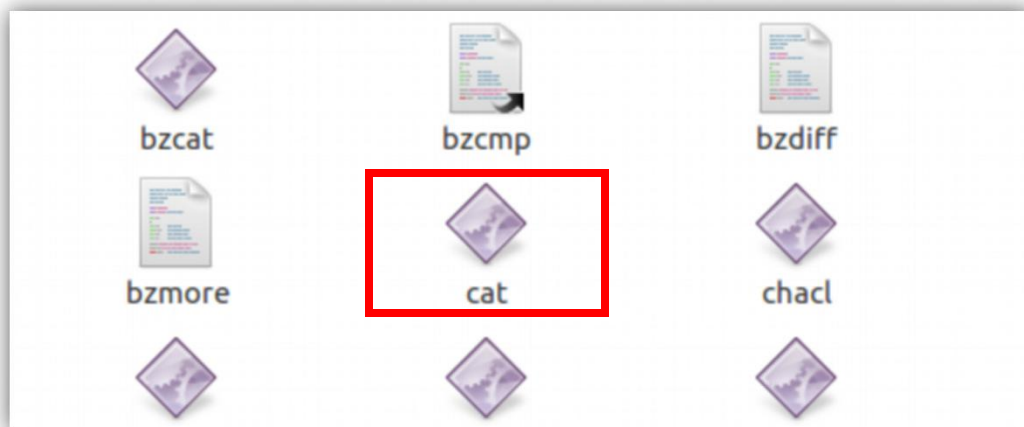
- OS中负责信息管理部分称为文件系统
- 具体功能包括
  - 文件的存储空间管理（分配、回收）
  - 目录管理：目录是为方便文件管理而采用的基本数据结构，它能提供“按名存取”功能。
  - 文件操作管理：实现文件的基本操作，包括打开、关闭、读、写等。
  - 文件保护：提供文件安全保护的有关功能和设施。



## 第二章 进程管理

### 进程(Task)的概念

- 进程是执行程序的一个实例
- 进程和程序的区别
  - 几个进程可以并发的执行一个程序
  - 一个进程可以顺序的执行几个程序



A screenshot of a terminal window displaying a list of running processes. The processes are listed with their PID, PPID, and name. The processes 'cat' at PID 8424 and 8425 are highlighted with a red rectangular box.

PID	PPID	Name
8386	?	gvfsd-http
8418	?	chrome
8424	?	cat
8425	?	cat
8428	?	chrome
8429	?	nacl_helper
8432	?	chrome
8472	?	chrome
8537	?	chrome
8724	?	chrome
8743	?	chrome
8834	?	chrome
8857	?	chrome
14099	?	cupsd
14101	?	cups-browsed
14117	?	dbus
14600	?	dhclient
15306	?	kworker/3:1
15432	?	kworker/0:0
16212	?	vmware
16253	?	vmware-tray
16769	?	jbd2/sdb7-8
16770	?	ext4-rsv-conver

## 第二章 进程管理

### 01: 进程描述符

为了管理进程，内核必须对每个进程进行清晰的描述。  
进程描述符提供了内核所需了解的进程信息

include/linux/sched.h

struct [task\\_struct](#)

数据结构很庞大

- 基本信息
- 管理信息
- 控制信息

```
1391 struct task_struct {
1392     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
1393     void *stack;
1394     atomic_t usage;
1395     unsigned int flags; /* per process flags, defined below */
1396     unsigned int ptrace;
1397
1398     #ifdef CONFIG_SMP
1399     struct llist_node wake_entry;
1400     int on_cpu;
1401     unsigned int wakee_flips;
1402     unsigned long wakee_flip_decay_ts;
1403     struct task_struct *last_wakee;
1404
1405     int wake_cpu;
1406     #endif
1407     int on_rq;
1408
1409     int prio, static_prio, normal_prio;
1410     unsigned int rt_priority;
1411     const struct sched_class *sched_class;
1412     struct sched_entity *se;
1413 }
```

```
1391 struct task_struct {
1392     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
1393     void *stack;
1394     atomic_t usage;
1395     unsigned int flags; /* per process flags, defined below */
1396     unsigned int ptrace;
1397 }
```

```
1427
1428     unsigned int policy;
1429     int nr_cpus_allowed;
1430     cpumask_t cpus_allowed;
1431
1432     #ifdef CONFIG_PREEMPT_RCU
1433     int rcu_read_lock_nesting;
1434     union rcu_special rcu_read_unlock_special;
1435     struct list_head rcu_node_entry;
1436     struct rcu_node *rcu_blocked_node;
1437     #endif /* #ifdef CONFIG_PREEMPT_RCU */
1438 }
```

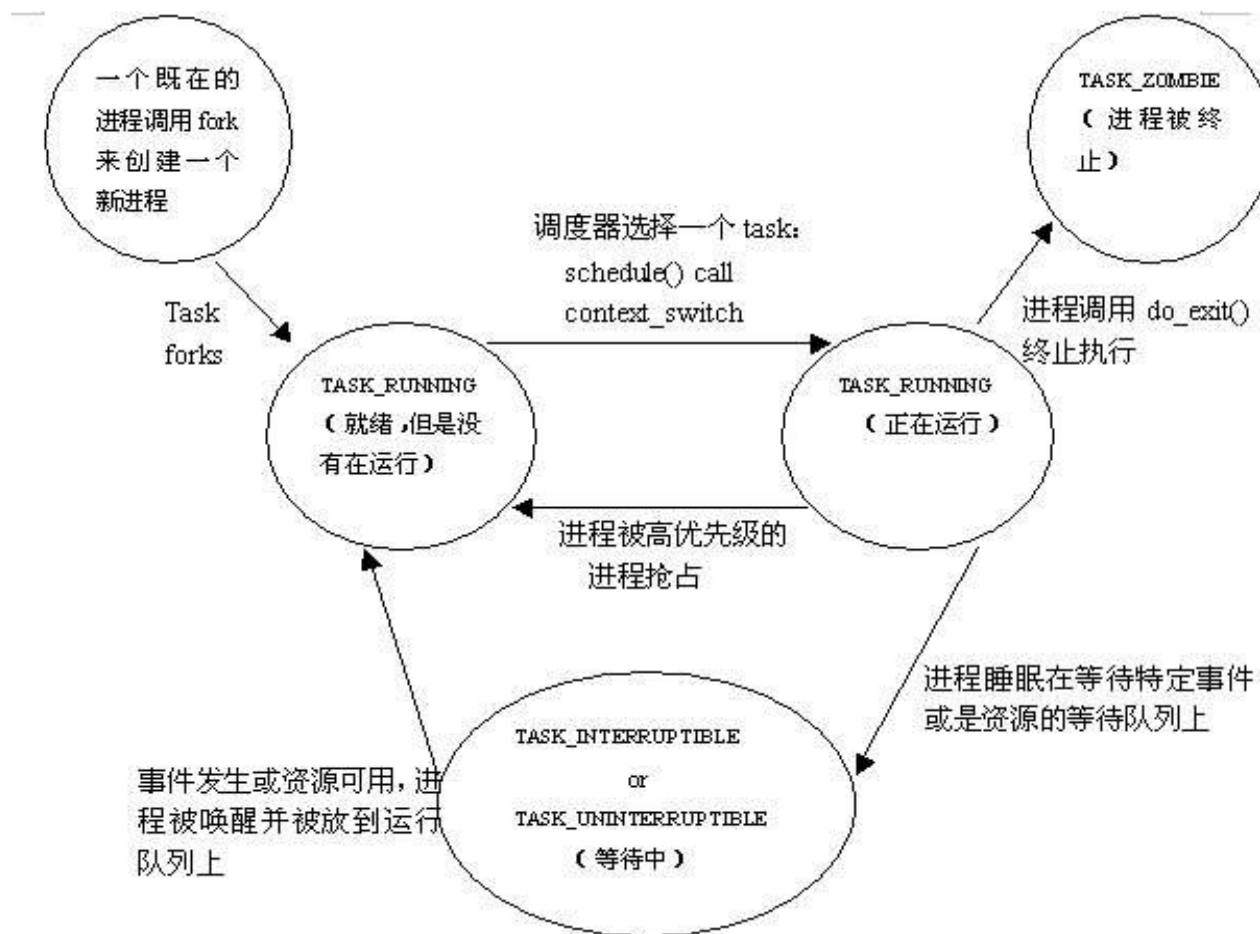
# 第二章 进程管理

## 02：进程状态

- 可运行状态(TASK\_RUNNING)
- 可中断的等待状态(TASK\_INTERRUPTIBLE)
- 不可中断的等待状态(TASK\_UNINTERRUPTIBLE)
- 暂停状态(TASK\_STOPPED)
- 僵死状态(TASK\_ZOMBIE)

## 第二章 进程管理

### 03: 进程状态转换图



## 第二章 进程管理

### 03：进程标识

- 使用进程描述符地址

进程和进程描述符之间有非常严格的一一对应关系，使得用32位进程描述符地址标识进程非常方便

- 使用PID (Process ID , PID)

- 每个进程的PID都存放在进程描述符的pid域中
- 新pid的产生： `get_pid`
  - +1
  - 循环

## 第二章 进程管理

### 04：进程创建

- 许多进程可以并发的运行同一程序，这些进程共享内存中程序正文的单一副本，但每个进程有自己的单独的数据和堆栈区
- 一个进程可以在任何时刻可以执行新的程序，并且在它的生命周期中可以运行几个程序
- 又如，只要用户输入一条命令，shell进程就创建一个新进程



## 第二章 进程管理

### 04：进程创建

传统的UNIX操作系统采用统一的方式来创建进程，即子进程复制父进程所拥有的所有资源。

缺点：

- 创建过程慢、效率低
- 事实上，子进程复制的很多资源是不会使用到的

现代UNIX内核通过引入写时复制技术来解决这个问题

## 第二章 进程管理

### 04：进程创建

#### 写时复制技术，Copy-On-Writing，COW

COW技术允许父子进程能读相同的物理页。

内核只为新生成的子进程创建虚拟空间结构，但是不为这些段分配物理内存，它们共享父进程的物理空间，当父子进程中有更改相应段的行为发生（即发生写操作时）时，再为子进程相应的段分配物理空间。

## 第二章 进程管理

### 05：进程相关API

Linux提供了几个系统调用来创建和终止进程，以及执行新程序。

- fork，vfork和clone系统调用创建新进程，其中，clone创建轻量级进程，必须指定要共享的资源。
- exec系统调用执行一个新程序。
- exit系统调用终止进程（进程也可以因收到信号而终止）。

## 第二章 进程管理

### 05: 进程相关API

#### fork系统调用创建一个新进程

- 调用fork的进程称为父进程
- 新进程是子进程
- 子进程几乎就是父进程的完全复制。它的地址空间是父进程的复制，一开始也是运行同一程序。

fork为父子进程返回不同的值

```
FORK(3am)                                GNU Awk Extension Modules                                FORK(3am)

NAME
    fork, wait, waitpid - basic process management

SYNOPSIS
    @load "fork"

    pid = fork()

    ret = waitpid(pid)

    ret = wait();

DESCRIPTION
    The fork extension adds three functions, as follows.

    fork() This function creates a new process. The return value is the
    zero in the child and the process-id number of the child in the
    parent, or -1 upon error. In the latter case, ERRNO indicates
    the problem. In the child, PROCINFO["pid"] and PROCINFO["ppid"]
    are updated to reflect the correct values.
```

## 第二章 进程管理

### 05：进程相关API

#### exec系统调用执行一个新程序

很多情况下，子进程从fork返回后很多会调用exec来开始执行新的程序。

这种情况下，子进程根本不需要读或者修改父进程拥有的所有资源。所以fork中地址空间的复制依赖于Copy On Write技术，降低fork的开销。

## 第二章 进程管理

### 05: 进程相关API

```
if (result = fork() == 0) {  
    /* 子进程代码 */  
    ...  
    if (execve("new_program",...) < 0) {  
        perror("execve failed");  
        exit(1);  
    }  
} else if (result < 0) {  
    perror("fork failed")  
} else {  
    /* result == 子进程的pid, 父进程将会从这里继续执行 */  
    ...  
}
```



## 第二章 进程管理

### 05：进程相关API

分开这两个系统调用是有好处的：

- 比如服务器可以fork许多进程执行同一个程序
- 有时程序只是简单的exec，执行一个新程序
- 在fork和exec之间，子进程可以有选择的执行一系列操作以确保程序以所希望的状态运行。比如重定向输入输出、关闭不需要的打开文件、重置信号处理程序等。

若单一的系统调用试图完成所有这些功能将是笨重而低效的

## 第二章 进程管理

### 05：进程相关API

#### 进程终止的一般方式是exit()系统调用

- 这个系统调用可能由编程者明确的在代码中插入
- 另外，在控制流到达主过程[C中的main()函数]的最后一条语句时，执行exit()系统调用

#### 内核可以强迫进程终止

- 当进程接收到一个不能处理或忽视的信号时，比如SIGSEGV崩溃信号
- 当在内核态产生一个不可恢复的CPU异常而内核此时正代表该进程在运行

# 第三章 进程虚拟空间及链接

## 01：进程的虚拟空间

为了管理进程，内核必须对每个进程进行清晰的描述，进程描述符提供了内核所需了解的进程信息。

include/linux/sched.h  
struct [task\\_struct](#)

数据结构非常庞大

- 基本信息
- 管理信息
- 控制信息

```
1391 struct task_struct {
1392     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
1393     void *stack;
1394     atomic_t usage;
1395     unsigned int flags; /* per process flags, defined below */
1396     unsigned int ptrace;
1397
1398     #ifdef CONFIG_SMP
1399     struct llist_node wake_entry;
1400     int on_cpu;
1401     unsigned int wakee_flips;
1402     unsigned long wakee_flip_decay_ts;
1403     struct task_struct *last_wakee;
1404
1405     int wake_cpu;
1406     #endif
1407     int on_rq;
1408
1409     int prio, static_prio, normal_prio;
1410     unsigned int rt_priority;
```

```
1391 struct task_struct {
1392     volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
1393     void *stack;
1394     atomic_t usage;
1395     unsigned int flags; /* per process flags, defined below */
1396     unsigned int ptrace;
```

```
1425     unsigned int btrace_seq;
1426     #endif
1427
1428     unsigned int policy;
1429     int nr_cpus_allowed;
1430     cpumask_t cpus_allowed;
1431
1432     #ifdef CONFIG_PREEMPT_RCU
1433     int rcu_read_lock_nesting;
1434     union rcu_special rcu_read_unlock_special;
1435     struct list_head rcu_node_entry;
1436     struct rcu_node *rcu_blocked_node;
1437     #endif /* #ifdef CONFIG_PREEMPT_RCU */
```

## 第三章 进程虚拟空间及链接

### 01: 进程的虚拟空间

- 进程最多能访问**4GB**的线性地址空间 (32位操作系统)
- 但进程在访问某个线性空间之前，必须获得该线性空间的许可
- 因此，一个进程的地址空间是由**允许该进程访问的全部线性地址组成**
- 内核使用线性区资源来表示线性地址空间
- 每个线性区由起始线性地址、长度和一些存取权限描述

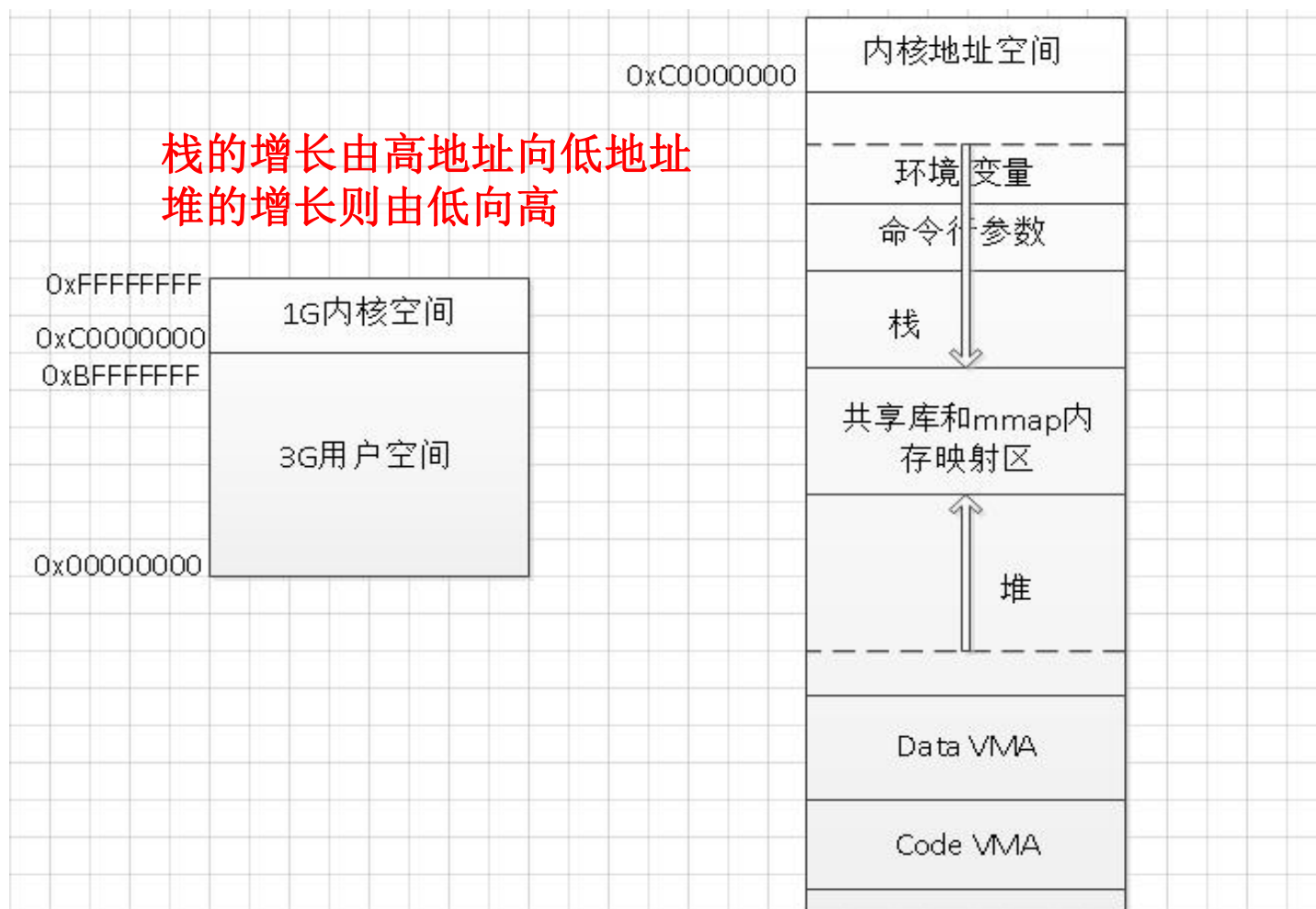
## 第三章 进程虚拟空间及链接

### 01：进程的虚拟空间

- 线性区的开始和结束都必须**4KB对齐**
- 进程获得新线性区的一些典型情况：
  - 刚刚创建的新进程
  - 使用exec系统调用装载一个新的程序运行
  - 将一个文件（或部分）映射到进程地址空间中
  - 当用户堆栈不够用的时候，扩展堆栈对应的线性区
  - .....

# 第三章 进程虚拟空间及链接

## 01：进程的虚拟空间





## 第三章 进程虚拟空间及链接

### 01：进程的虚拟空间

#### 线性区（Memory Area）

- 比如0x08048000——0x0804C000这段线性地址空间被分配给了一个进程，进程就可以访问这段地址空间
- 进程只能访问某个有效的memory area。
- 如果进程试图访问一个有效的area之外的地址或者用不正确的方式访问一个有效的area，内核将通过段异常(segmentation fault)杀死这个进程，即**程序崩溃**。

## 第三章 进程虚拟空间及链接

### 01：进程的虚拟空间

```
~/DefCC/alldatasonserver/virt_root_dir$ ./cbs/pwn47-dffffba924671101521d932e5a0007a40 < cras  
hes/dffffba924671101521d932e5a0007a40-0ffa9d9562bcc9e30cfe44e0c828bcfe 1 >/dev/null  
Segmentation fault (core dumped)
```

gdb调试信息：

Program received signal SIGSEGV, Segmentation fault.

decode\_4 (in=in@entry=0x2c615760 <Address 0x2c615760 out of  
bounds>, inlen=inlen@entry=4096, outp=outp@entry=0xbfffd4cc,  
outleft=outleft@entry=0xbfffd49c) at lib/base64.c:417

分析结果：

程序尝试访问非法内存地址0x2c615760

## 第三章 进程虚拟空间及链接

### 01：进程的虚拟空间

#### 线性区中可以包含各种内容

- 可执行文件代码段的内存映射，就是.text section
- 数据段的内存映射，.data section
- zero page的内存映射用来包含未初始化的全局变量，.bss section
- 为库函数和链接器附加的代码、数据、bss段
- 文件的内存映射
- 共享内存的映射
- 匿名内存区域的映射，比如通过malloc()函数申请的内存区域

## 第三章 进程虚拟空间及链接

### 01: 进程的虚拟空间

```
root@kali:~# cat /proc/self/maps
08048000-08054000 r-xp 00000000 08:01 391694      /bin/cat
08054000-08055000 r--p 0000b000 08:01 391694      /bin/cat
08055000-08056000 rw-p 0000c000 08:01 391694      /bin/cat
09523000-09544000 rw-p 00000000 00:00 0          [heap]
b734b000-b736d000 rw-p 00000000 00:00 0
b736d000-b73ac000 r--p 00000000 08:01 262305      /usr/lib/locale/zh_TW.utf8/LC_CTYPE
b73ac000-b7510000 r--p 00000000 08:01 262192      /usr/lib/locale/zh_CN.utf8/LC_COLLATE
b7510000-b7511000 rw-p 00000000 00:00 0
b7511000-b76c1000 r-xp 00000000 08:01 915083      /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
b76c1000-b76c4000 r--p 001af000 08:01 915083      /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
b76c4000-b76c6000 rw-p 001b2000 08:01 915083      /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
b76c6000-b76c8000 rw-p 00000000 00:00 0
b76d4000-b76d5000 r--p 00000000 08:01 262312      /usr/lib/locale/zh_TW.utf8/LC_NUMERIC
b76d5000-b76d6000 r--p 00000000 08:01 262203      /usr/lib/locale/zh_CN.utf8/LC_TIME
b76d6000-b76d7000 r--p 00000000 08:01 262198      /usr/lib/locale/zh_CN.utf8/LC_MONETARY
b76d7000-b76d8000 r--p 00000000 08:01 262309      /usr/lib/locale/zh_TW.utf8/LC_MESSAGES/SYS_LC_MESSAGES
b76d8000-b76d9000 r--p 00000000 08:01 262341      /usr/lib/locale/zu_ZA.utf8/LC_PAPER
b76d9000-b76da000 r--p 00000000 08:01 262311      /usr/lib/locale/zh_TW.utf8/LC_NAME
b76da000-b76db000 r--p 00000000 08:01 262191      /usr/lib/locale/zh_CN.utf8/LC_ADDRESS
b76db000-b76dc000 r--p 00000000 08:01 262202      /usr/lib/locale/zh_CN.utf8/LC_TELEPHONE
b76dc000-b76dd000 r--p 00000000 08:01 262335      /usr/lib/locale/zu_ZA.utf8/LC_MEASUREMENT
b76dd000-b76e4000 r--s 00000000 08:01 1941        /usr/lib/i386-linux-gnu/gconv/gconv-modules.cache
b76e4000-b76e5000 r--p 00000000 08:01 262194      /usr/lib/locale/zh_CN.utf8/LC_IDENTIFICATION
b76e5000-b76e7000 rw-p 00000000 00:00 0
b76e7000-b76e9000 r--p 00000000 00:00 0          [vvar]
b76e9000-b76ea000 r-xp 00000000 00:00 0          [vdso]
b76ea000-b770b000 r-xp 00000000 08:01 915119      /lib/i386-linux-gnu/ld-2.21.so
b770b000-b770c000 r--p 00020000 08:01 915119      /lib/i386-linux-gnu/ld-2.21.so
b770c000-b770d000 rw-p 00021000 08:01 915119      /lib/i386-linux-gnu/ld-2.21.so
bf919000-bf93a000 rw-p 00000000 00:00 0          [stack]
```



## 第三章 进程虚拟空间及链接

### 01: 进程的虚拟空间

- 进程地址空间中所有有效的线性地址都确定的存在于一个area中
  - memory areas **不重叠**
- 进程中每个单独的area对应一个不同内存区：
  - 堆栈、二进制代码、全局变量、文件映射等等

## 第三章 进程虚拟空间及链接

### 01: 进程的虚拟空间

```
struct task_struct {
    /*
     * offsets of these are hardcoded elsewhere - touch w
     */
    volatile long state;    /* -1 unrunnable, 0 runnable,
    unsigned long flags;    /* per process flags, defined
    int sigpending;
    mm_segment_t addr_limit; /* thread address space:
                             0-0xBFFFFFFF for user-thread
                             0-0xFFFFFFFF for kernel-thread
                             */
    struct exec_domain *exec_domain;
    volatile long need_resched;
    unsigned long ptrace;

    int lock_depth;        /* Lock depth */

    /*
     * offset 32 begins here on 32-bit platforms. We keep
     * all fields in a single c
     * the goodness() loop in s
     */
    long counter;
    long nice;
    unsigned long policy;
    struct mm_struct *mm;
```

内核使用**内存描述符**来描述进程的整个地址空间（即进程的全部线性区）

```

struct mm_struct {
    struct vm_area_struct * mmap;           /* list of VMAs */
    rb_root_t mm_rb;
    struct vm_area_struct * mmap_cache;     /* last find vma result */
    pgd_t * pgd;
    atomic_t mm_users;
    atomic_t mm_count;
    int map_count;
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;            /* Protects task page tables and mm->

struct list_head mmlist;
    /* together with mmlist of other processes
    * by mmlist
    */

    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_address;

    unsigned dumpable:1;

    /* Architecture-specific MM context */
    mm_context_t context;
} ? end mm_struct ? ;

```

**mmap**和**mm\_rb**是两个不同的数据结构，但是包含了相同的東西：進程地址空間中所有的memory areas  
 前者使用鏈表存儲areas  
 後者用紅黑樹存儲areas

所有的**mm\_struct**結構通過**mmlist**域鏈接在一個双向鏈表上。這個鏈表的第一個元素是idle進程的**mm\_struct**結構

## 内存描述符

- 内存描述符包含了跟进程地址空间相关的所有信息



## 第三章 进程虚拟空间及链接

### 01：进程的虚拟空间

vm\_flags域描述有关这个线性区全部页的信息。例如，进程访问每个页的权限是什么。还有一些标志描述线性区自身，例如它应该如何增长

- VM\_READ, VM\_WRITE, VM\_EXEC
- VM\_SHARED
- VM\_RESERVED
- VM\_GROWSUP

在漏洞利用过程中经常会采取通过篡改内存页面读写执行权限来达到利用目的。

# 第三章 进程虚拟空间及链接

## 01：进程的虚拟空间

### 缺页异常

如前所述，内核只是通过mmap()等调用分配了一些线性地址空间给进程，并没有真正的把实际的物理页框分配给进程，当进程试图访问这些分配给它的地址空间时，比如一段线性地址空间映射的是二进制代码，则进程被调度执行的时候会跳转到这个地址上去执行。但此时，并没有物理页框对应于这些线性地址，从而会引发一个缺页异常。

## 第三章 进程虚拟空间及链接

### 01：进程的虚拟空间

Linux内核中通过缺页异常处理程序do\_page\_fault处理缺页异常。

它可以判断出这不是不是一个合法的缺页异常，如果是，则负责给这段线性地址分配一些物理页框并把磁盘中对应的文件写入这些物理页框，这样进程得以正常运行。

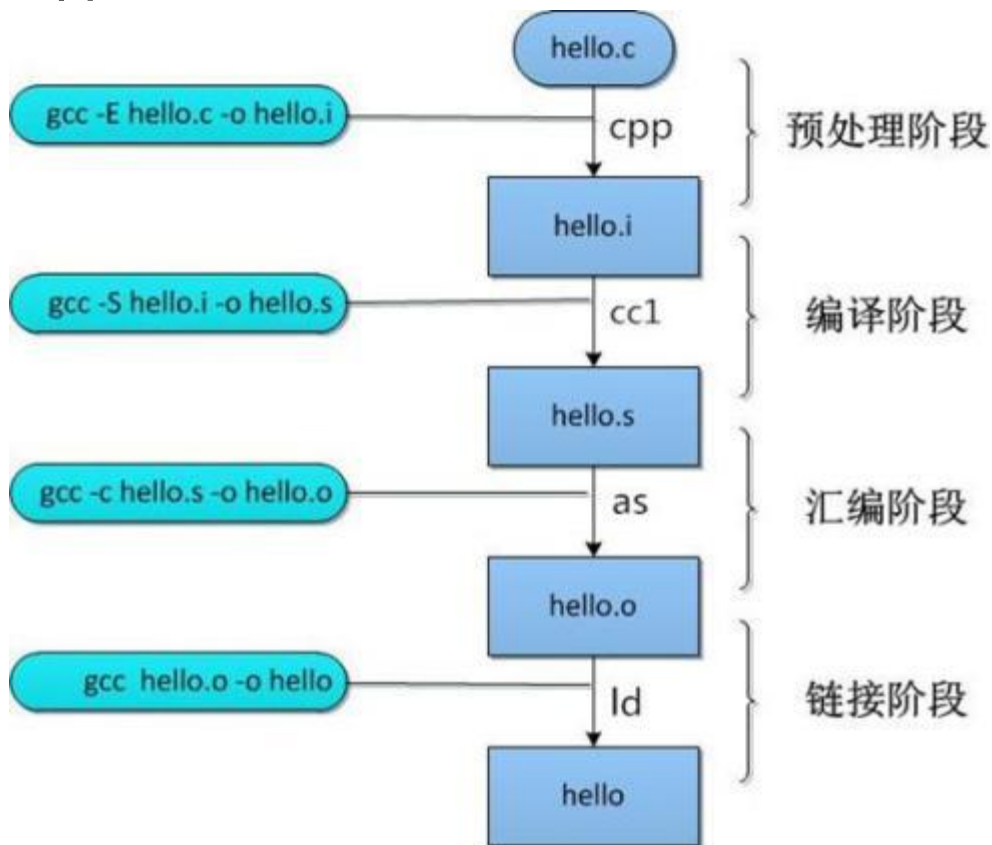
对于非法的缺页异常，内核将通过段异常(segmentation fault)杀死该进程。

# 第三章 进程虚拟空间及链接

## 02: 链接

从源码到可执行文件发生了什么？

在Linux下使用GCC将源码编译成可执行文件的过程可以分解为4个步骤，分别是预处理、编译、汇编和链接。



# 第三章 进程虚拟空间及链接

## 02：链接

### 什么是链接？

链接过程的本质就是要把多个不同目标文件粘合成一个整体，目标文件之间相互拼合实际上是目标文件之间对地址的引用，即对函数和变量的地址的引用。在链接中，我们将函数和变量统称为符号（Symbol），函数名和变量名就是符号名

（Symbol Name），我们可以将符号看做是链接中的粘合剂，整个链接过程正是基于符号才能够正确完成。每个目标文件都会有一个符号表（Symbol Table），即 .symtab 段，这个表里记录了目标文件所用到的所有符号。每个定义的符号有一个对应的值，叫做符号值（Symbol Value），对于变量和函数来说，符号值就是它们的地址。我们可以通过 readelf 工具来查看符号表中所有的符号信息：



# 第三章 进程虚拟空间及链接

## 02: 链接

什么是链接？

`readelf -s `which cat``

```
Symbol table '.dynsym' contains 78 entries:
Num:  Value                               Size Type Bind  Vis      Ndx Name
 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
 1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __uflow@GLIBC_2.2.5 (2)
 2: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getenv@GLIBC_2.2.5 (2)
 3: 0000000000000000 0 FUNC GLOBAL DEFAULT UND free@GLIBC_2.2.5 (2)
 4: 0000000000000000 0 FUNC GLOBAL DEFAULT UND abort@GLIBC_2.2.5 (2)
 5: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __errno_location@GLIBC_2.2.5 (2)
 6: 0000000000000000 0 FUNC GLOBAL DEFAULT UND strncmp@GLIBC_2.2.5 (2)
 7: 0000000000000000 0 FUNC GLOBAL DEFAULT UND _exit@GLIBC_2.2.5 (2)
 8: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __fpending@GLIBC_2.2.5 (2)
 9: 0000000000000000 0 FUNC GLOBAL DEFAULT UND iconv@GLIBC_2.2.5 (2)
10: 0000000000000000 0 FUNC GLOBAL DEFAULT UND iswcntrl@GLIBC_2.2.5 (2)
11: 0000000000000000 0 FUNC GLOBAL DEFAULT UND write@GLIBC_2.2.5 (2)
12: 0000000000000000 0 FUNC GLOBAL DEFAULT UND textdomain@GLIBC_2.2.5 (2)
13: 0000000000000000 0 FUNC GLOBAL DEFAULT UND fclose@GLIBC_2.2.5 (2)
14: 0000000000000000 0 FUNC GLOBAL DEFAULT UND bindtextdomain@GLIBC_2.2.5 (2)
15: 0000000000000000 0 FUNC GLOBAL DEFAULT UND stpcpy@GLIBC_2.2.5 (2)
16: 0000000000000000 0 FUNC GLOBAL DEFAULT UND dcgettext@GLIBC_2.2.5 (2)
17: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __ctype_get_mb_cur_max@GLIBC_2.2.5 (2)
18: 0000000000000000 0 FUNC GLOBAL DEFAULT UND strlen@GLIBC_2.2.5 (2)
19: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __stack_chk_fail@GLIBC_2.4 (3)
20: 0000000000000000 0 FUNC GLOBAL DEFAULT UND getopt_long@GLIBC_2.2.5 (2)
21: 0000000000000000 0 FUNC GLOBAL DEFAULT UND mbrtowc@GLIBC_2.2.5 (2)
22: 0000000000000000 0 FUNC GLOBAL DEFAULT UND strchr@GLIBC_2.2.5 (2)
23: 0000000000000000 0 FUNC GLOBAL DEFAULT UND strrchr@GLIBC_2.2.5 (2)
24: 0000000000000000 0 FUNC GLOBAL DEFAULT UND lseek@GLIBC_2.2.5 (2)
25: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __assert_fail@GLIBC_2.2.5 (2)
26: 0000000000000000 0 FUNC GLOBAL DEFAULT UND memset@GLIBC_2.2.5 (2)
27: 0000000000000000 0 FUNC GLOBAL DEFAULT UND fscanf@GLIBC_2.2.5 (2)
28: 0000000000000000 0 FUNC GLOBAL DEFAULT UND ioctl@GLIBC_2.2.5 (2)
29: 0000000000000000 0 FUNC GLOBAL DEFAULT UND strlen@GLIBC_2.2.5 (2)
30: 0000000000000000 0 FUNC GLOBAL DEFAULT UND close@GLIBC_2.2.5 (2)
31: 0000000000000000 0 FUNC GLOBAL DEFAULT UND posix_fadvise@GLIBC_2.2.5 (2)
32: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __strdup@GLIBC_2.2.5 (2)
33: 0000000000000000 0 FUNC GLOBAL DEFAULT UND memchr@GLIBC_2.2.5 (2)
34: 0000000000000000 0 FUNC GLOBAL DEFAULT UND read@GLIBC_2.2.5 (2)
35: 0000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
36: 0000000000000000 0 FUNC GLOBAL DEFAULT UND memcmp@GLIBC_2.2.5 (2)
37: 0000000000000000 0 FUNC GLOBAL DEFAULT UND fputs_unlocked@GLIBC_2.2.5 (2)
38: 0000000000000000 0 FUNC GLOBAL DEFAULT UND calloc@GLIBC_2.2.5 (2)
39: 0000000000000000 0 FUNC GLOBAL DEFAULT UND strcmp@GLIBC_2.2.5 (2)
40: 0000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
41: 0000000000000000 0 FUNC GLOBAL DEFAULT UND memcpy@GLIBC_2.14 (4)
```

# 第三章 进程虚拟空间及链接

## 02: 链接

### 程序的链接方式

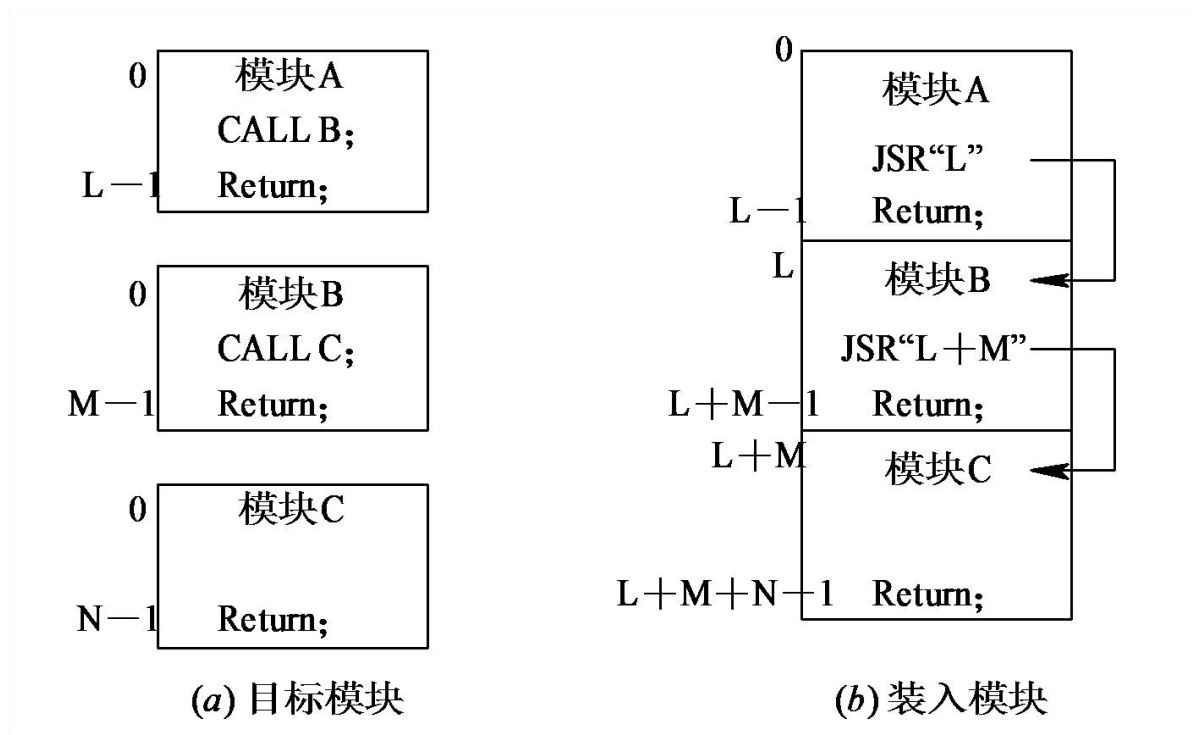
1. 静态链接方式(Static Linking)
2. 装入时动态链接(Load time Dynamic Linking)
3. 运行时动态链接(Run-time Dynamic Linking)



# 第三章 进程虚拟空间及链接

## 02: 链接

### 1. 静态链接方式(Static Linking)



# 第三章 进程虚拟空间及链接

## 02: 链接

### 2. 装入时动态链接(Loadtime Dynamic Linking)

优点:

((1))便于修改和更新。

(2) 便于实现对目标模块的共享。

缺点:

符号解析过程繁琐

# 第三章 进程虚拟空间及链接

## 02: 链接

### 3. 运行时动态链接(Runtime Dynamic Linking)

对模块的链接推迟到执行时才执行，即在执行过程中，当发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将之装入内存，把它链接到调用者模块上。

凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。

## 第三章 进程虚拟空间及链接

### 03：深入理解运行时动态链接

如何解决模块间的数据访问以及调用、跳转？

objdump -R `which cat`

Linux ELF的做法是通过在数据段中建立一个指向变量、目标函数地址的指针数组，当代码需要引用该变量或者调用该函数时，可以通过该数组中对应的项进行间接引用，该数据成为**全局偏移表** (Global Offset Table, GOT)。

```
/bin/cat:      file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET          TYPE          VALUE
000000000000bfff R_X86_64_GLOB_DAT  __gmon_start__
000000000000c300 R_X86_64_COPY      __progrname@@GLIBC_2.2.5
000000000000c308 R_X86_64_COPY      stdout@@GLIBC_2.2.5
000000000000c310 R_X86_64_COPY      optind@@GLIBC_2.2.5
000000000000c318 R_X86_64_COPY      __progrname_full@@GLIBC_2.2.5
000000000000c320 R_X86_64_COPY      stderr@@GLIBC_2.2.5
000000000000c018 R_X86_64_JUMP_SLOT __uflow@GLIBC_2.2.5
000000000000c020 R_X86_64_JUMP_SLOT getenv@GLIBC_2.2.5
000000000000c028 R_X86_64_JUMP_SLOT free@GLIBC_2.2.5
000000000000c030 R_X86_64_JUMP_SLOT abort@GLIBC_2.2.5
000000000000c038 R_X86_64_JUMP_SLOT __errno_location@GLIBC_2.2.5
000000000000c040 R_X86_64_JUMP_SLOT strcmp@GLIBC_2.2.5
000000000000c048 R_X86_64_JUMP_SLOT _exit@GLIBC_2.2.5
000000000000c050 R_X86_64_JUMP_SLOT _fpending@GLIBC_2.2.5
000000000000c058 R_X86_64_JUMP_SLOT lconv@GLIBC_2.2.5
000000000000c060 R_X86_64_JUMP_SLOT lswcntrl@GLIBC_2.2.5
000000000000c068 R_X86_64_JUMP_SLOT write@GLIBC_2.2.5
000000000000c070 R_X86_64_JUMP_SLOT textdomain@GLIBC_2.2.5
000000000000c078 R_X86_64_JUMP_SLOT fclose@GLIBC_2.2.5
000000000000c080 R_X86_64_JUMP_SLOT bindtextdomain@GLIBC_2.2.5
000000000000c088 R_X86_64_JUMP_SLOT stpcpy@GLIBC_2.2.5
000000000000c090 R_X86_64_JUMP_SLOT dcgettext@GLIBC_2.2.5
000000000000c098 R_X86_64_JUMP_SLOT __ctype_get_mb_cur_max@GLIBC_2.2.5
000000000000c0a0 R_X86_64_JUMP_SLOT strlen@GLIBC_2.2.5
000000000000c0a8 R_X86_64_JUMP_SLOT __stack_chk_fail@GLIBC_2.4
000000000000c0b0 R_X86_64_JUMP_SLOT getopt_long@GLIBC_2.2.5
000000000000c0b8 R_X86_64_JUMP_SLOT mbrtowc@GLIBC_2.2.5
000000000000c0c0 R_X86_64_JUMP_SLOT strchr@GLIBC_2.2.5
000000000000c0c8 R_X86_64_JUMP_SLOT strrchr@GLIBC_2.2.5
000000000000c0d0 R_X86_64_JUMP_SLOT lseek@GLIBC_2.2.5
000000000000c0d8 R_X86_64_JUMP_SLOT __assert_fail@GLIBC_2.2.5
000000000000c0e0 R_X86_64_JUMP_SLOT memset@GLIBC_2.2.5
000000000000c0e8 R_X86_64_JUMP_SLOT fscanf@GLIBC_2.2.5
000000000000c0f0 R_X86_64_JUMP_SLOT ioctl@GLIBC_2.2.5
```

## 第三章 进程虚拟空间及链接

### 03：深入理解运行时动态链接

模块间函数的链接推迟到执行时才执行，我们也称之为“**延迟绑定 (Lazy Binding)**”。

程序开始执行时，模块间的函数调用都没有进行绑定，而是在需要用的时候才由动态链接器来负责绑定。这样的做法**可以大大加快程序的启动速度**，特别有利于一些有大量函数引用和大量模块的程序。

## 第三章 进程虚拟空间及链接

### 03：深入理解运行时动态链接

**ELF采用PLT（procedure linkage table）机制来实现延迟绑定，这种方法使用了很多精巧的指令序列来完成。**

在Linux中，动态链接在扫描可执行文件时，一旦需要进行模块间外部函数的调用，则会启用绑定函数`_dl_runtime_resolve(mod, func)`来确定模块`mod`中的`func`函数真实地址，并完成对对应GOT表项的填充工作。

## 第三章 进程虚拟空间及链接

### 03：深入理解运行时动态链接

**PLT (Procedure Linkage Table)** 在GOT表基础又做了一次间接跳转。即模块内关于外部函数的地址引用，并不直接通过GOT跳转，而是通过一个叫做PLT项的结构来进行，每个外部函数引用都对应PLT表中的一个表项，比如read()函数在PLT表中的表项称为read@plt，实现如下：

```
read@plt:  
    jmp  *(read@GOT)  
    push  n  
    push  link_map  
    jump  _dl_runtime_resolve
```



## 第三章 进程虚拟空间及链接

### 03: 深入理解运行时动态链接

```
[-----code-----]
0x8048e30 <open64@plt>:      jmp     DWORD PTR ds:0x8055010
0x8048e36 <open64@plt+6>:   push    0x8
0x8048e3b <open64@plt+11>:  jmp     0x8048e10
=> 0x8048e40 <read@plt>:    jmp     DWORD PTR ds:0x8055014
| 0x8048e46 <read@plt+6>:   push    0x10
| 0x8048e4b <read@plt+11>:  jmp     0x8048e10
| 0x8048e50 <fflush@plt>:   jmp     DWORD PTR ds:0x8055018
| 0x8048e56 <fflush@plt+6>: push    0x18
|-> 0x8048e46 <read@plt+6>: push    0x10
    0x8048e4b <read@plt+11>: jmp     0x8048e10
    0x8048e50 <fflush@plt>: jmp     DWORD PTR ds:0x8055018
    0x8048e56 <fflush@plt+6>: push    0x18

[-----stack-----]
0000| 0xbffff32c --> 0x804c663 (add     esp,0x10)
0004| 0xbffff330 --> 0x0
0008| 0xbffff334 --> 0xb7c3f000 --> 0x0
0012| 0xbffff338 --> 0x20000
0016| 0xbffff33c --> 0x804d22e (add     esp,0x10)
0020| 0xbffff340 --> 0x20fff
0024| 0xbffff344 --> 0x0
0028| 0xbffff348 --> 0xbffff458 --> 0x0

Legend: code, data, rodata, value

Breakpoint 1, 0x08048e40 in read@plt ()
gdb-peda$ x/x 0x8055014
0x8055014 <read@got.plt>:      0x08048e46
```

JUMP is taken

首先跳转到GOT表，但此时GOT表中填充的为plt+0x6的地址，因此，相当于顺序执行。

## 第三章 进程虚拟空间及链接

### 03: 深入理解运行时动态链接

```
[-----code-----]
0x8048e36 <open64@plt+6>:  push    0x8
0x8048e3b <open64@plt+11>:  jmp     0x8048e10
0x8048e40 <read@plt>:      jmp     DWORD PTR ds:0x8055014
=> 0x8048e46 <read@plt+6>:  push    0x10
0x8048e4b <read@plt+11>:  jmp     0x8048e10
0x8048e50 <fflush@plt>:      jmp     DWORD PTR ds:0x8055018
0x8048e56 <fflush@plt+6>:  push    0x18
0x8048e5b <fflush@plt+11>:  jmp     0x8048e10
[-----stack-----]
0000| 0xbffff32c --> 0x804c663 (add esp,0x10)
0004| 0xbffff330 --> 0x0
0008| 0xbffff334 --> 0xb7c3f000 --> 0x0
0012| 0xbffff338 --> 0x20000
0016| 0xbffff33c --> 0x804d22e (add esp,0x10)
0020| 0xbffff340 --> 0x20fff
0024| 0xbffff344 --> 0x0
0028| 0xbffff348 --> 0xbffff458 --> 0x0
[-----]
Legend: code, data, rodata, value
0x08048e46 in read@plt ()
gdb-peda$ x/x 0x8048bac+0x10
0x8048bbc: 0x08055014
gdb-peda$ x/x 0x08055014
0x08055014 <read@got.plt>: 0x08048e46
gdb-peda$
```

这个0x10是read函数在重定位表".rel.plt"中的下标。



## 第三章 进程虚拟空间及链接

```
root@kali:~# readelf -S /bin/cat
共有 27 个节头，从偏移量 0xc2b0 开始：
```

节头：

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	4
[ 3]	.note.gnu.build-id	NOTE	08048188	000188	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	080481ac	0001ac	000044	04	A	5	0	4
[ 5]	.dynsym	DYNSYM	080481f0	0001f0	000500	10	A	6	1	4
[ 6]	.dynstr	STRTAB	080486f0	0006f0	00035a	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	08048a4a	000a4a	0000a0	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	08048aec	000aec	000090	00	A	6	1	4
[ 9]	.rel.dyn	REI	08048b7c	000b7c	000030	08	A	5	0	4
[10]	.rel.plt	REL	08048bac	000bac	000238	08	AI	5	12	4
[11]	.init	PROGBITS	08048de4	000de4	000023	00	AX	0	0	4
[12]	.plt	PROGBITS	08048e10	000e10	000480	04	AX	0	0	16
[13]	.text	PROGBITS	08049290	001290	006dec	00	AX	0	0	16
[14]	.fini	PROGBITS	0805007c	00807c	000014	00	AX	0	0	4
[15]	.rodata	PROGBITS	080500c0	0080c0	000f46	00	A	0	0	64
[16]	.eh_frame_hdr	PROGBITS	08051008	009008	000304	00	A	0	0	4
[17]	.eh_frame	PROGBITS	0805130c	00930c	00218c	00	A	0	0	4
[18]	.init_array	INIT_ARRAY	08054f08	00bf08	000004	00	WA	0	0	4
[19]	.fini_array	FINI_ARRAY	08054f0c	00bf0c	000004	00	WA	0	0	4
[20]	.jcr	PROGBITS	08054f10	00bf10	000004	00	WA	0	0	4
[21]	.dynamic	DYNAMIC	08054f14	00bf14	0000e8	08	WA	6	0	4
[22]	.got	PROGBITS	08054ffc	00bffc	000004	04	WA	0	0	4
[23]	.got.plt	PROGBITS	08055000	00c000	000128	04	WA	0	0	4
[24]	.data	PROGBITS	08055140	00c140	000080	00	WA	0	0	32
[25]	.bss	NOBITS	080551c0	00c1c0	0005c4	00	WA	0	0	64
[26]	.shstrtab	STRTAB	00000000	00c1c0	0000ed	00		0	0	1

## 第三章 进程虚拟空间及链接

### 03: 深入理解运行时动态链接

```
[-----code-----]
0x8048e36 <open64@plt+6>:  push    0x8
0x8048e3b <open64@plt+11>: jmp     0x8048e10
0x8048e40 <read@plt>:      jmp     DWORD PTR ds:0x8055014
=> 0x8048e46 <read@plt+6>:  push    0x10
0x8048e4b <read@plt+11>:  jmp     0x8048e10
0x8048e50 <fflush@plt>:     jmp     DWORD PTR ds:0x8055018
0x8048e56 <fflush@plt+6>:  push    0x18
0x8048e5b <fflush@plt+11>: jmp     0x8048e10
[-----stack-----]
0000| 0xbffff32c --> 0x804c663 (add    esp,0x10)
0004| 0xbffff330 --> 0x0
0008| 0xbffff334 --> 0xb7c3f000 --> 0x0
0012| 0xbffff338 --> 0x20000
0016| 0xbffff33c --> 0x804d22e (add    esp,0x10)
0020| 0xbffff340 --> 0x20fff
0024| 0xbffff344 --> 0x0
0028| 0xbffff348 --> 0xbffff458 --> 0x0
[-----]
Legend: code, data, rodata, value
0x08048e46 in read@plt ()
gdb-peda$ x/x 0x8048bac+0x10
0x8048bbc: 0x08055014
gdb-peda$ x/x 0x08055014
0x08055014 <read@got.plt>: 0x08048e46
gdb-peda$
```

这个0x10是read函数在重定位表“rel.plt”中的下标。

重定位表偏移0x10处的地址恰好为read函数在got表中的位置。



## 第三章 进程虚拟空间及链接

### 03: 深入理解运行时动态链接

```
[-----code-----]
0x8048e3b <open64@plt+11>:  jmp     0x8048e10
0x8048e40 <read@plt>:      jmp     DWORD PTR ds:0x8055014
0x8048e46 <read@plt+6>:   push    0x10
=> 0x8048e4b <read@plt+11>: jmp     0x8048e10
| 0x8048e50 <fflush@plt>:  jmp     DWORD PTR ds:0x8055018
| 0x8048e56 <fflush@plt+6>: push    0x18
| 0x8048e5b <fflush@plt+11>: jmp     0x8048e10
| 0x8048e60 <_exit@plt>:  jmp     DWORD PTR ds:0x805501c
|-> 0x8048e10:  push    DWORD PTR ds:0x8055004
      0x8048e16:  jmp     DWORD PTR ds:0x8055008
      0x8048e1c:  add     BYTE PTR [eax],al
      0x8048e1e:  add     BYTE PTR [eax],al

[-----stack-----]
0000| 0xbffff328 --> 0x10
0004| 0xbffff32c --> 0x804c663 (add     esp,0x10)
0008| 0xbffff330 --> 0x0
0012| 0xbffff334 --> 0xb7c3f000 --> 0x0
0016| 0xbffff338 --> 0x20000
0020| 0xbffff33c --> 0x804d22e (add     esp,0x10)
0024| 0xbffff340 --> 0x20fff
0028| 0xbffff344 --> 0x0

Legend: code, data, rodata, value
0x08048e4b in read@plt ()
jdb-peda$ x/x 0x8055008
0x8055008: 0xb7ff1150
jdb-peda$ p _dl_runtime_resolve
$3 = {<text variable, no debug info>} 0xb7ff1150 <dl runtime resolve>
```

Link\_map=\*(GOT+4)  
该结构包含符号表syntab、  
字符串表strtab等。

在参数入栈之后，进入  
dl\_runtime\_resolve。

## 第三章 进程虚拟空间及链接

### 03: 深入理解运行时动态链接

```
[-----code-----]
0xb7fdcc34 <__kernel_vsyscall+12>:  nop
0xb7fdcc35 <__kernel_vsyscall+13>:  nop
0xb7fdcc36 <__kernel_vsyscall+14>:  int      0x80
=> 0xb7fdcc38 <__kernel_vsyscall+16>:  pop      ebp
0xb7fdcc39 <__kernel_vsyscall+17>:  pop      edx
0xb7fdcc3a <__kernel_vsyscall+18>:  pop      ecx
0xb7fdcc3b <__kernel_vsyscall+19>:  ret
0xb7fdcc3c:  ret
```

```
[-----stack-----]
0000| 0xbffff318 --> 0xbffff458 --> 0x0
0004| 0xbffff31c --> 0x20000
0008| 0xbffff320 --> 0xb7c3f000 --> 0x0
0012| 0xbffff324 --> 0xb7edbe33 (<__read_nocancel+25>:  pop      ebx)
0016| 0xbffff328 --> 0x20000
0020| 0xbffff32c --> 0x804c663 (add      esp,0x10)
0024| 0xbffff330 --> 0x0
0028| 0xbffff334 --> 0xb7c3f000 --> 0x0
```

Legend: code, data, rodata, value

Stopped reason: SIGINT

0xb7fdcc38 in \_\_kernel\_vsyscall ()

```
gdb-peda$ x/x 0x08055014
```

```
0x8055014 <read@got.plt>:      0xb7edbe10
```














```
gdb-peda$ p read
```

```
$2 = {<text variable, no debug info>} 0xb7edbe10 <read>
```

dl\_runtime\_resolve函数返回后，  
read@got的内存已经被替换为真实的  
read函数地址。这样，在后续调用read  
函数时，可以直接跳转到真实地址。

## 第三章 进程虚拟空间及链接

### 03：深入理解运行时动态链接

Name	Start	End	R	W	X	D	L	Align	Base
 .init	08048DE4	08048E07	R	.	X	.	L	dword	0001
 .plt	08048E10	08049290	R	.	X	.	L	para	0002
 .text	08049290	0805007C	R	.	X	.	L	para	0003
 .fini	0805007C	08050090	R	.	X	.	L	dword	0004
 .rodata	080500C0	08051006	R	.	.	.	L	64byte	0005
 .eh_frame_hdr	08051008	0805130C	R	.	.	.	L	dword	0006
 .eh_frame	0805130C	08053498	R	.	.	.	L	dword	0007
 .init_array	08054F08	08054F0C	R	W	.	.	L	dword	0008
 .fini_array	08054F0C	08054F10	R	W	.	.	L	dword	0009
 .jcr	08054F10	08054F14	R	W	.	.	L	dword	000A
 .got	08054FFC	08055000	R	W	.	.	L	dword	000B
 .got.plt	08055000	08055128	R	W	.	.	L	dword	000C
 .data	08055140	080551C0	R	W	.	.	L	32byte	000D

既然GOT表的权限为R/W，那是不是可以搞点事情呢？



## 第三章 进程虚拟空间及链接

### 03： 深入理解运行时动态链接

演示手动劫持GOT表

# 第四章 \*nix程序基本调试环境

## 01: 信息收集

熟练掌握使用\*nix系统的coreutils&binutils工具进行初步的信息收集

```
root@kali:~$ file `which cat`  
/bin/cat: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,  
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=  
2267d831560007f67fa4388d830192fd89861061, stripped  
root@kali:~$
```

- ELF文件格式
- 64bit
- X86-64架构
- 动态链接
- stripped

其它工具:

readelf/objdump/hexdump/size等

# 第四章 \*nix程序基本调试环境

## 01: 信息收集

```
~$ readelf -S 'which cat'
There are 29 section headers, starting at offset 0xc430:
```

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[ 0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0	0
[ 1]	.interp	PROGBITS	0000000000400238	00000238
	000000000000001c	0000000000000000	A 0 0	1
[ 2]	.note.ABI-tag	NOTE	0000000000400254	00000254
	0000000000000020	0000000000000000	A 0 0	4
[ 3]	.note.gnu.build-id	NOTE	0000000000400274	00000274
	0000000000000024	0000000000000000	A 0 0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000400298	00000298
	0000000000000040	0000000000000000	A 5 0	8
[ 5]	.dynsym	DYNSYM	00000000004002d8	000002d8
	00000000000000750	0000000000000018	A 6 1	8
[ 6]	.dynstr	STRTAB	0000000000400a28	00000a28
	00000000000000316	0000000000000000	A 0 0	1
[ 7]	.gnu.version	VERSYM	0000000000400d3e	00000d3e
	000000000000009c	0000000000000002	A 5 0	2
[ 8]	.gnu.version_r	VERNEED	0000000000400de0	00000de0
	0000000000000060	0000000000000000	A 6 1	8
[ 9]	.rela.dyn	RELA	0000000000400e40	00000e40
	0000000000000090	0000000000000018	A 5 0	8
[10]	.rela.plt	RELA	0000000000400ed0	00000ed0
	00000000000000678	0000000000000018	AI 5 24	8
[11]	.init	PROGBITS	0000000000401548	00001548
	000000000000001a	0000000000000000	AX 0 0	4
[12]	.plt	PROGBITS	0000000000401570	00001570
	00000000000000460	0000000000000010	AX 0 0	16
[13]	.plt.got	PROGBITS	00000000004019d0	000019d0
	0000000000000008	0000000000000000	AX 0 0	8

```
~$ objdump -R 'which cat'
```

/bin/cat: file format elf64-x86-64

DYNAMIC RELOCATION RECORDS

OFFSET	TYPE	VALUE
000000000060bffb	R_X86_64_GLOB_DAT	__gmon_start__
000000000060c300	R_X86_64_COPY	__progname@@GLIBC_2.2.5
000000000060c308	R_X86_64_COPY	stdout@@GLIBC_2.2.5
000000000060c310	R_X86_64_COPY	optind@@GLIBC_2.2.5
000000000060c318	R_X86_64_COPY	__progname_full@@GLIBC_2.2.5
000000000060c320	R_X86_64_COPY	stderr@@GLIBC_2.2.5
000000000060c018	R_X86_64_JUMP_SLOT	__uflow@GLIBC_2.2.5
000000000060c020	R_X86_64_JUMP_SLOT	getenv@GLIBC_2.2.5
000000000060c028	R_X86_64_JUMP_SLOT	free@GLIBC_2.2.5
000000000060c030	R_X86_64_JUMP_SLOT	abort@GLIBC_2.2.5
000000000060c038	R_X86_64_JUMP_SLOT	__errno_location@GLIBC_2.2.5
000000000060c040	R_X86_64_JUMP_SLOT	strncmp@GLIBC_2.2.5
000000000060c048	R_X86_64_JUMP_SLOT	__exit@GLIBC_2.2.5
000000000060c050	R_X86_64_JUMP_SLOT	__fpending@GLIBC_2.2.5
000000000060c058	R_X86_64_JUMP_SLOT	lconv@GLIBC_2.2.5
000000000060c060	R_X86_64_JUMP_SLOT	iswcntrl@GLIBC_2.2.5
000000000060c068	R_X86_64_JUMP_SLOT	write@GLIBC_2.2.5
000000000060c070	R_X86_64_JUMP_SLOT	textdomain@GLIBC_2.2.5
000000000060c078	R_X86_64_JUMP_SLOT	fclose@GLIBC_2.2.5
000000000060c080	R_X86_64_JUMP_SLOT	bindtextdomain@GLIBC_2.2.5
000000000060c088	R_X86_64_JUMP_SLOT	stpncpy@GLIBC_2.2.5
000000000060c090	R_X86_64_JUMP_SLOT	dcgettext@GLIBC_2.2.5
000000000060c098	R_X86_64_JUMP_SLOT	__ctype_get_mb_cur_max@GLIBC_2.2.5
000000000060c0a0	R_X86_64_JUMP_SLOT	strlen@GLIBC_2.2.5
000000000060c0a8	R_X86_64_JUMP_SLOT	__stack_chk_fail@GLIBC_2.4
000000000060c0b0	R_X86_64_JUMP_SLOT	getopt_long@GLIBC_2.2.5
000000000060c0b8	R_X86_64_JUMP_SLOT	nbrtowc@GLIBC_2.2.5
000000000060c0c0	R_X86_64_JUMP_SLOT	strchr@GLIBC_2.2.5
000000000060c0c8	R_X86_64_JUMP_SLOT	strrchr@GLIBC_2.2.5
000000000060c0d0	R_X86_64_JUMP_SLOT	lseek@GLIBC_2.2.5
000000000060c0d8	R_X86_64_JUMP_SLOT	__assert_fail@GLIBC_2.2.5

# 第四章 \*nix程序基本调试环境

## 02: 静态分析



[https://nominis.cef.fr/contenus/saints\\_966.html](https://nominis.cef.fr/contenus/saints_966.html)



# 第四章 \*nix程序基本调试环境

IDA - /home/epelus/Desktop/cat

File Edit Jump Search View Debugger Options Windows Help

Library function Data Regular function Unexplored Instruction External symbol

Functions window

Function name	Segment	Start	Length
__errno_location	.plt	08049140	00000006
__memchr	.plt	08049150	00000006
__fileno	.plt	08049160	00000006
__printf_chk	.plt	08049170	00000006
__strlen	.plt	08049180	00000006
__uflow	.plt	08049190	00000006
__nl_langinfo	.plt	080491A0	00000006
__setlocale	.plt	080491B0	00000006
__strchr	.plt	080491C0	00000006
__isseek64	.plt	080491D0	00000006
__fprintf_chk	.plt	080491E0	00000006
__bindtextdomain	.plt	080491F0	00000006
__posix_fadvise64	.plt	08049200	00000006
__strncpy	.plt	08049210	00000006
__abort	.plt	08049220	00000006
__iconv_open	.plt	08049230	00000006
__close	.plt	08049240	00000006
__assert_fail	.plt	08049250	00000006
__ctype_b_loc	.plt	08049260	00000006
__calloc	.plt	08049270	00000006
__sprintf_chk	.plt	08049280	00000006
main	.text	08049290	000008ED
start	.text	08049E7D	00000022
sub_8049EA0	.text	08049EA0	00000004
sub_8049EB0	.text	08049EB0	00000028
sub_8049F20	.text	08049F20	0000001E
sub_8049F40	.text	08049F40	0000002B
sub_8049F70	.text	08049F70	00000069
sub_8049FE0	.text	08049FE0	0000031A
__posix_fadvise64	.text	0804A3D0	00000005
sub_804A410	.text	0804A410	00000065
sub_804A480	.text	0804A480	0000009A
sub_804A520	.text	0804A520	0000008C
sub_804AE80	.text	0804AE80	000002FA
sub_804B1A0	.text	0804B1A0	00000026

函数枚举窗体

函数控制流图窗体

```
08049F40
08049F40
08049F40
08049F40 sub_8049F40 proc near
08049F40 ; FUNCTION CHUNK AT 08049EE0 SIZE 00000033 BYTES
08049F40
08049F40 mov     eax, offset unk_8054F10
08049F45 mov     edx, [eax]
08049F47 test    edx, edx
08049F49 jnz     short loc_8049F50

08049F50
08049F50 loc_8049F50:
08049F50 mov     edx, 0
08049F55 test    edx, edx
08049F57 jz      short loc_8049F4B

08049F4B jmp     short loc_8049EE0

08049F33 push    ebp
08049F3A mov     ebp, esp
08049F3C sub     esp, 14h
08049F3F push    eax
08049F40 call    edx
08049F42 add     esp, 10h
08049F45 leave
08049F46 jmp     loc_8049EE0
08049F48 sub     sub_8049F40 endp
08049F49

08049EE0 ; START OF FUNCTION CHUNK FOR sub_8049F40
08049EE0
08049EE0 loc_8049EE0:
08049EE0 mov     eax, offset program_invocation_short_name
08049EE3 sub     eax, offset program_invocation_short_name
08049EE5 sar     eax, 2
08049EE8 mov     edx, eax
08049EEF shr     edx, 1Fh
08049EF2 add     eax, edx
08049EF4 sar     eax, 1
08049EF6 jz      short locret_8049F13

08049EF8 mov     edx, 0
08049EFD test    edx, edx
```

Python交互窗体

The initial autoanalysis has been finished  
Command "JumpAsk" failed

Python

# 第四章 \*nix程序基本调试环境

## 02：静态分析

### IDA pro自动化分析

IDA pro的具有强大的扩展能力，其提供了IDC(IDA 的脚本语言)和SDK(让开发者扩展方便 IDA 插件)以便用户可以根据自己需要定制插件。

2004 年 Gergely 和 Ero Carrera 开发了 IDAPython 插件，将强大的 Python 和 IDA 结合起来，使得自动化分析变得异常简单。而如今IDAPython被广泛的使用于各种商业产品（比如BinNavi）和开源工程（比如PaiMei）中。





# 第四章 \*nix程序基本调试环境

## 02：静态分析

### IDA python常见接口

- **SegByName( string SegmentName )**  
通过段名字返回段基址，比如，如果调用.text 作为参数，就会返回程序中代码段的开始位置
- **SegName( long Address )**  
通过段内的某个地址，获得段名
- **Functions( long StartAddress, long EndAddress )**  
返回一个列表，包含了从 StartAddress 到 EndAddress 之间的所有函数
- **LocByName( string FunctionName )**  
通过函数名返回函数的地址
- **CodeRefsTo( long Address, bool Flow )**  
返回一个列表，告诉我们 Address 处代码被什么地方引用了，Flow 告诉 IDAPython 是否要跟踪这些代码
- **DataRefsTo( long Address )**  
返回一个列表，告诉我们 Address 处数据被什么地方引用了。  
常用于跟踪全局变量。

# 第四章 \*nix程序基本调试环境

## 02：静态分析

### 演示使用IDA python进行自动化分析

寻找软件漏洞 bug 的时候，首先会找一些常用的而且容易被错误使用的函数。比如危险的字符串拷贝函数 (strcpy, sprintf)，内存拷贝函数 (memcpy)等。

那么如何通过IDA python自动识别这些函数在哪里被调用的，  
即危险函数调用自动化识别？

# 第四章 \*nix程序基本调试环境

## 02：动态调试

命令行调试工具GDB



图形化调试工具EDB，类似于windows的Ollydbg

# 第四章 \*nix程序基本调试环境

## 02：动态调试

### 运行

run (简写r) : 运行程序, 当遇到断点后, 程序会在断点处停止运行, 等待用户输入下一步的命令。  
continue (简写c) : 继续执行, 到下一个断点处 (或运行结束)  
next (简写n) : 单步跟踪程序, 当遇到函数调用时, 直接调用, 不进入此函数体;  
step (简写s) : 单步调试如果有函数调用, 则进入函数; 与命令n不同, n是不进入调用的函数的  
until: 运行程序直到退出循环体; / until+行号: 运行至某行  
finish : 运行程序, 直到当前函数完成返回, 并打印函数返回时的堆栈地址和返回值及参数值等信息。  
call 函数(参数): 调用“函数”, 并传递“参数”, 如: call gdb\_test(55)  
quit: 简记为 q , 退出gdb

### 设置断点

break n (简写b n) : 在第n行处设置断点 ; 可以带上代码路径和代码名称: b OAGUPDATE.cpp:578)  
break func: 在函数func()的入口处设置断点, 如: break cb\_button  
delete 断点号n: 删除第n个断点  
disable 断点号n: 暂停第n个断点  
enable 断点号n: 开启第n个断点  
clear 行号n: 清除第n行的断点  
info breakpoints (简写info b) : 显示当前程序的断点设置情况

# 第四章 \*nix程序基本调试环境

## 02：动态调试

### 打印表达式

print 表达式：简记为 p，其中“表达式”可以是任何当前正在被测试程序的有效表达式，比如当前正在调试C语言的程序，那么“表达式”可以是任何C语言的有效表达式，包括数字，变量甚至是函数调用。

print a：将显示整数 a 的值

print ++a：将把 a 中的值加1,并显示出来

print name：将显示字符串 name 的值

print gdb\_test(22)：将以整数22作为参数调用 gdb\_test() 函数

print gdb\_test(a)：将以变量 a 作为参数调用 gdb\_test() 函数

display 表达式：在单步运行时将非常有用，使用display命令设置一个表达式后，它将在每次单步进行指令后，紧接着输出被设置的表达式及值。如： display a

watch 表达式：设置一个监视点，一旦被监视的“表达式”的值改变，gdb将强行终止正在被调试的程序。如： watch a

### 查询运行信息

where/bt：当前运行的堆栈列表

set args 参数:指定运行时的参数

show args：查看设置好的参数

info program：来查看程序的是否在运行，进程号，被暂停的原因

### 分割窗口

layout：用于分割窗口，可以一边查看代码，一边测试：

layout src：显示源代码窗口

layout asm：显示反汇编窗口

layout regs：显示源代码/反汇编和CPU寄存器窗口

layout split：显示源代码和反汇编窗口

Ctrl + L：刷新窗口

# 第四章 \*nix程序基本调试环境

## 02：动态调试

演示使用GDB调试程序



# 第五章 二进制程序漏洞挖掘

## 01：漏洞的定义

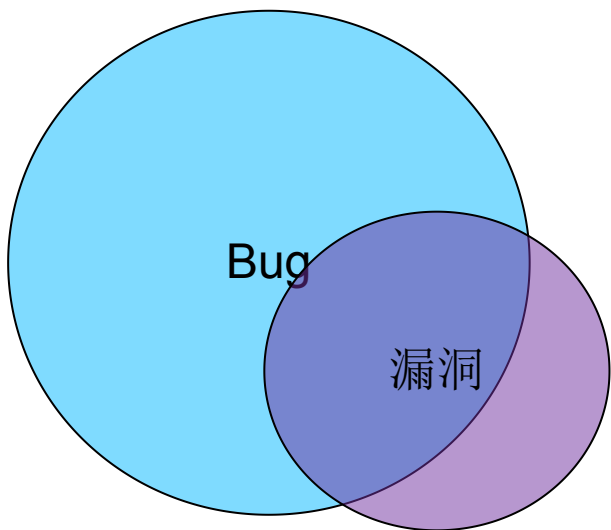
### 什么是漏洞(Vulnerability)?

漏洞是指系统中存在的一些功能性或安全性的缺陷，是信息系统在硬件、软件、协议的设计与实现过程中或系统安全策略上存在的缺陷和不足。

# 第五章 二进制程序漏洞挖掘

## 02：漏洞与BUG的关系

- 大部分Bug只影响功能，不涉及安全性（影响机密性、完整性、可用性），不构成漏洞；
- 大部分漏洞来源于Bug，二者之间有一个很大的交集。



# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞

软件是网络空间的灵魂，几乎网络空间中所有功能都以软件的形式运行体现。

- 纯软的形式 — 操作系统、应用软件
- 固件的形式 — 嵌入式系统（硬件）
- 网络的形式 — 网络软件（协议）

**网络空间中每200行有效代码就产生1个软件漏洞**

# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞

### 内存破坏漏洞

由于某种形式的非预期的内存越界访问（读、写或执行），可导致拒绝服务、信息泄露或执行攻击者指定的任意指令。

- 缓冲区溢出
- 格式化字符串问题
- 越界内存访问
- 二次释放
- 释放后重用

# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞

### 内存破坏漏洞

【例】1988年著名的Morris蠕虫就是利用了finger 服务的一个栈缓冲区溢出漏洞。2008年之前的几乎所有影响面巨大的网络蠕虫也基本利用此类漏洞。

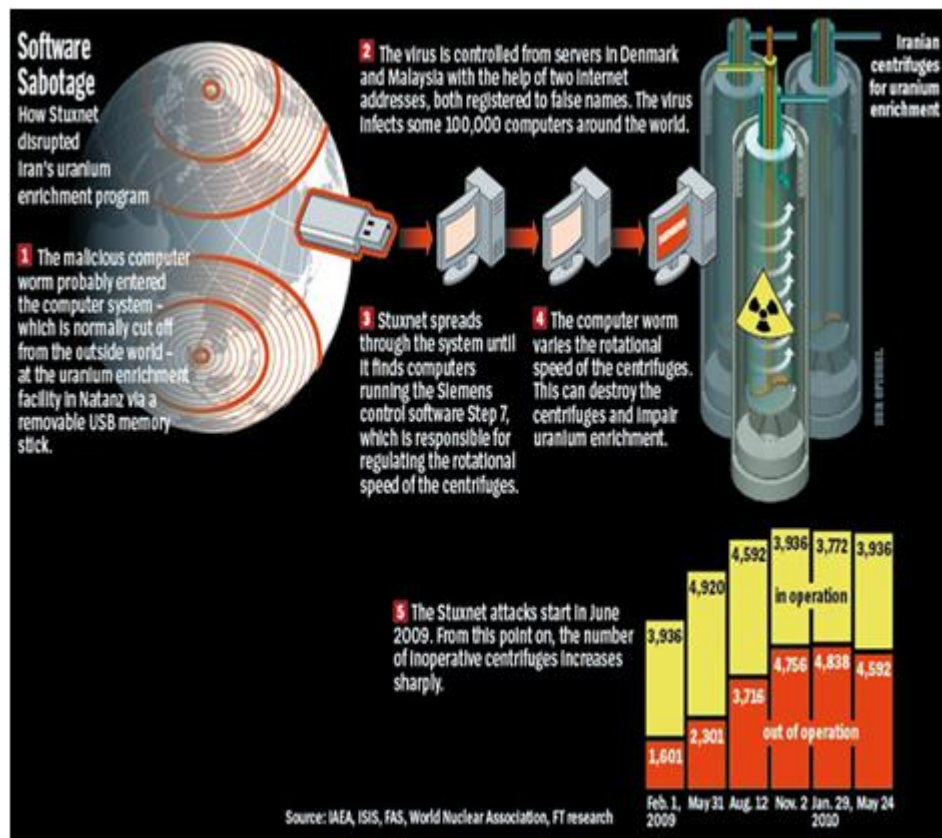
蠕虫	中文名号	MS 公告号	CVE ID	漏洞名
Slammer	蠕虫王	MS02-056	CVE-2002-1123	Microsoft SQL Server 预验证过程远程缓冲区溢出漏洞
MSBlast	冲击波	MS03-026	CVE-2003-0352	Microsoft Windows DCOM RPC 接口长主机名远程缓冲区溢出漏洞
Sasser	震荡波	MS04-011	CVE-2003-0533	Microsoft Windows LSASS 远程缓冲区溢出漏洞
Conficker	飞客蠕虫	MS08-067	CVE-2008-4250	Microsoft Windows Server 服务 RPC 请求缓冲区溢出漏洞

# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞

### 内存破坏漏洞

【例】2010年，名震天下的“震网”病毒通过利用4个未公开漏洞，成功渗透进入重重防护的伊朗核电站核心控制网络，造成20%的浓缩铀离心机被摧毁。





# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞挖掘

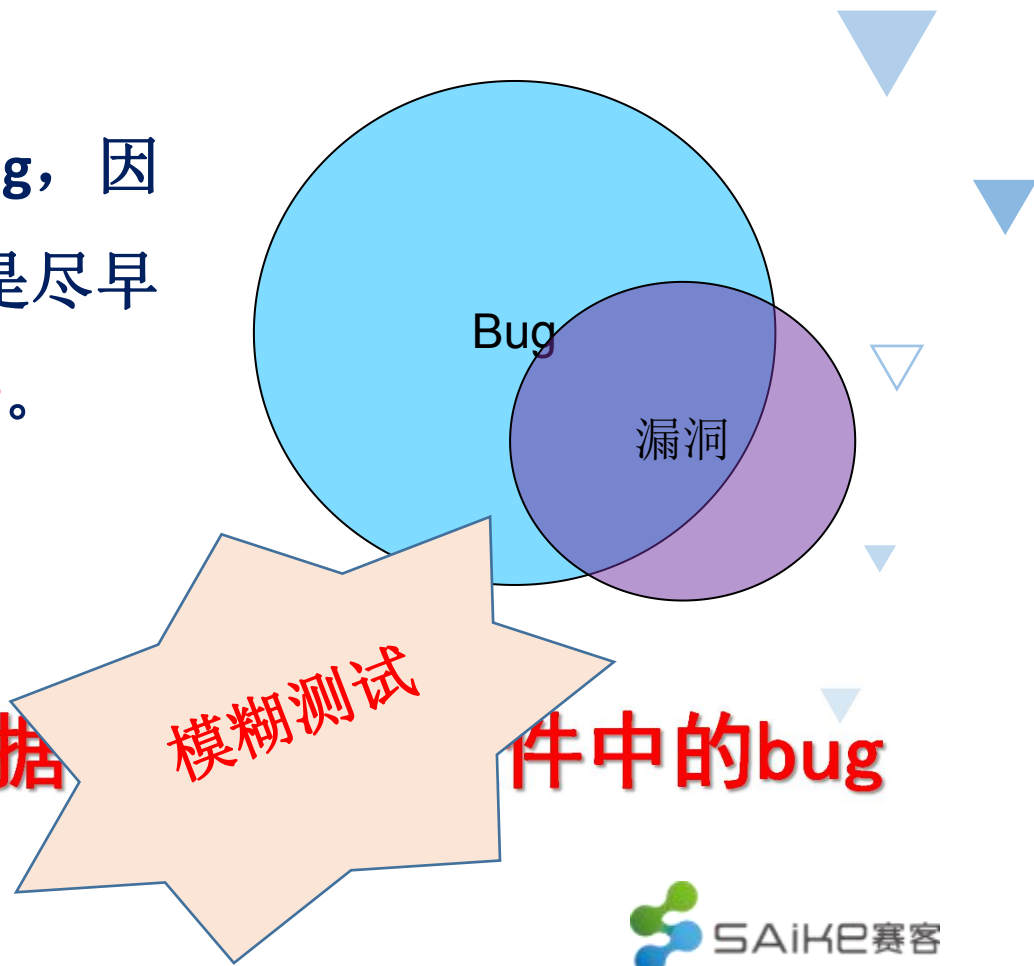
大部分漏洞来源于bug，因此挖掘软件漏洞的前提是尽早尽多地发现软件中的bug。

我们的方法：

向程序发送随机数据

模糊测试

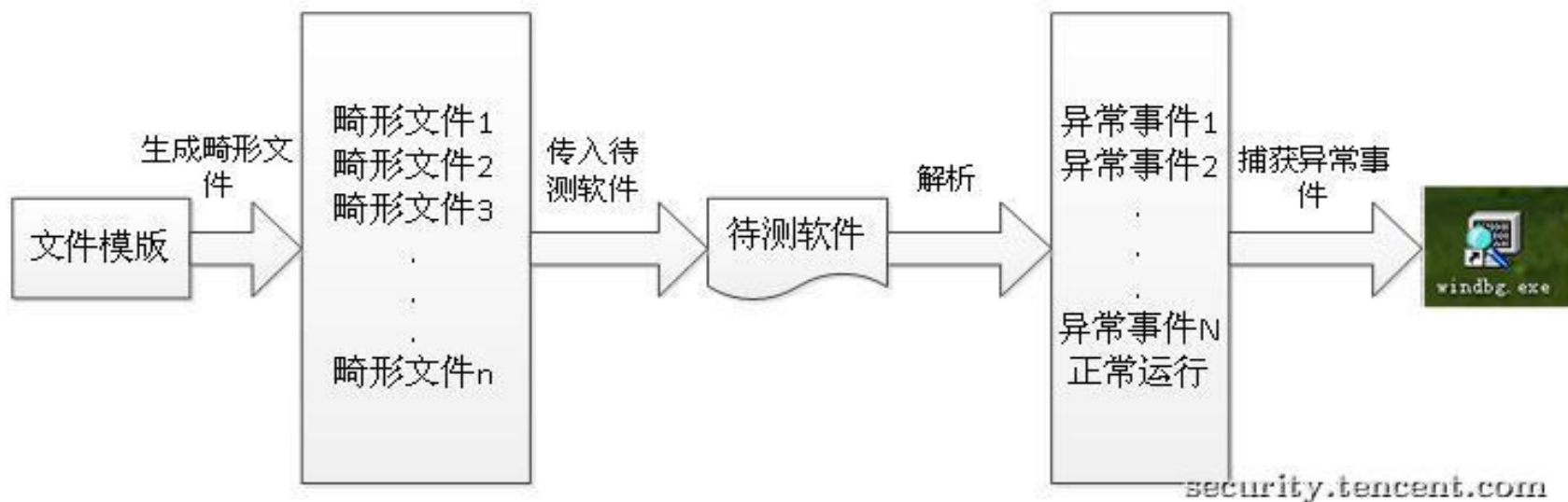
件中的bug



# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞挖掘

模糊测试是一种黑盒测试技术，它将大量的畸形数据输入到目标程序中，通过监测程序的异常来发现被测程序中可能存在的安全漏洞。



# 第五章 二进制程序漏洞挖掘

## 03: 软件漏洞挖掘

american fuzzy lop 0.47b (readpng)

### process timing

run time : 0 days, 0 hrs, 4 min, 43 sec  
last new path : 0 days, 0 hrs, 0 min, 26 sec  
last uniq crash : none seen yet  
last uniq hang : 0 days, 0 hrs, 1 min, 51 sec

### cycle progress

now processing : 38 (19.49%)  
paths timed out : 0 (0.00%)

### stage progress

now trying : interest 32/8  
stage execs : 0/9990 (0.00%)  
total execs : 654k  
exec speed : 2306/sec

### fuzzing strategy yields

bit flips : 88/14.4k, 6/14.4k, 6/14.4k  
byte flips : 0/1804, 0/1786, 1/1750  
arithmetics : 31/126k, 3/45.6k, 1/17.8k  
known ints : 1/15.8k, 4/65.8k, 6/78.2k  
havoc : 34/254k, 0/0  
trim : 2876 B/931 (61.45% gain)

### overall results

cycles done : 0  
total paths : 195  
uniq crashes : 0  
uniq hangs : 1

### map coverage

map density : 1217 (7.43%)  
count coverage : 2.55 bits/tuple

### findings in depth

favorable paths : 128 (65.64%)  
new edges on : 85 (43.59%)  
total crashes : 0 (0 unique)  
total hangs : 1 (1 unique)

### path geometry

levels : 3  
pending : 178  
pend fav : 114  
imported : 0  
variable : 0  
latent : 0

# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞挖掘

**AFL**模糊测试引擎较好地解决了模糊测试的两个核心问题，极大程度上提高了漏洞发现的速度和效率。



# 第三

03:

IJG jpeg <sup>1</sup>	libjpeg-turbo <sup>1 2</sup>	libpng <sup>1</sup>
libtiff <sup>1 2 3 4 5</sup>	mozjpeg <sup>1</sup>	PHP <sup>1 2 3 4 5</sup>
Mozilla Firefox <sup>1 2 3 4</sup>	Internet Explorer <sup>1 2 3 4</sup>	Apple Safari <sup>1</sup>
Adobe Flash / PCRE <sup>1 2 3 4</sup>	sqlite <sup>1 2 3 4 ...</sup>	OpenSSL <sup>1 2 3 4 5 6 7</sup>
LibreOffice <sup>1 2 3 4</sup>	poppler <sup>1</sup>	freetype <sup>1 2</sup>
GnuTLS <sup>1</sup>	GnuPG <sup>1 2 3 4</sup>	OpenSSH <sup>1 2 3</sup>
PuTTY <sup>1 2</sup>	ntpd <sup>1</sup>	nginx <sup>1 2 3</sup>
bash (post-Shellshock) <sup>1 2</sup>	tcpdump <sup>1 2 3 4 5 6 7 8 9</sup>	JavaScriptCore <sup>1 2 3 4</sup>
pdfium <sup>1 2</sup>	ffmpeg <sup>1 2 3 4 5</sup>	libmatroska <sup>1</sup>
libarchive <sup>1 2 3 4 5 6 ...</sup>	wireshark <sup>1 2 3</sup>	ImageMagick <sup>1 2 3 4 5 6 7 8 9 ...</sup>
BIND <sup>1 2 3 ...</sup>	QEMU <sup>1 2</sup>	lcms <sup>1</sup>
Oracle BerkeleyDB <sup>1 2</sup>	Android / libstagefright <sup>1 2</sup>	iOS / ImageIO <sup>1</sup>
FLAC audio library <sup>1 2</sup>	libsndfile <sup>1 2 3 4</sup>	less / lesspipe <sup>1 2 3</sup>
strings (+ related tools) <sup>1 2 3 4 5 6 7</sup>	file <sup>1 2 3 4</sup>	dpkg <sup>1 2</sup>
rcs <sup>1</sup>	systemd-resolved <sup>1 2</sup>	libyaml <sup>1</sup>
Info-Zip unzip <sup>1 2</sup>	libtasn1 <sup>1 2 ...</sup>	OpenBSD pfctl <sup>1</sup>
NetBSD bpf <sup>1</sup>	man & mandoc <sup>1 2 3 4 5 ...</sup>	IDA Pro [reported by authors]
clamav <sup>1 2 3 4 5</sup>	libxml2 <sup>1 2 3 4 5 6 7 8 9 ...</sup>	glibc <sup>1</sup>
clang / llvm <sup>1 2 3 4 5 6 7 8 ...</sup>	nasm <sup>1 2</sup>	ctags <sup>1</sup>
mutt <sup>1</sup>	procmail <sup>1</sup>	fontconfig <sup>1</sup>
ndksh <sup>1 2</sup>	Qt <sup>1 2 ...</sup>	waypack <sup>1</sup>



# 第五章 二进制程序漏洞挖掘

## 03：软件漏洞挖掘

**AFL**模糊测试引擎支持对源代码软件的漏洞挖掘，同时也支持对二进制程序的漏洞挖掘。

使得使用**AFL**对**CTF PWN**题目进行**快速漏洞发现**成为可能！



在 RHG 上，机器人需要完成这些步骤：

第一，利用 `fuzz` 模块，输入错误的东西让程序崩溃；

第二，利用漏洞挖掘引擎，找到可以挖掘的漏洞；

第三，根据漏洞点，生成可利用的 `EXP`；

第四，验证该漏洞利用程序的破坏性。



# 谢谢观赏 THANKS

河南赛客信息技术有限公司  
[www. secseeds. com](http://www.secseeds.com)