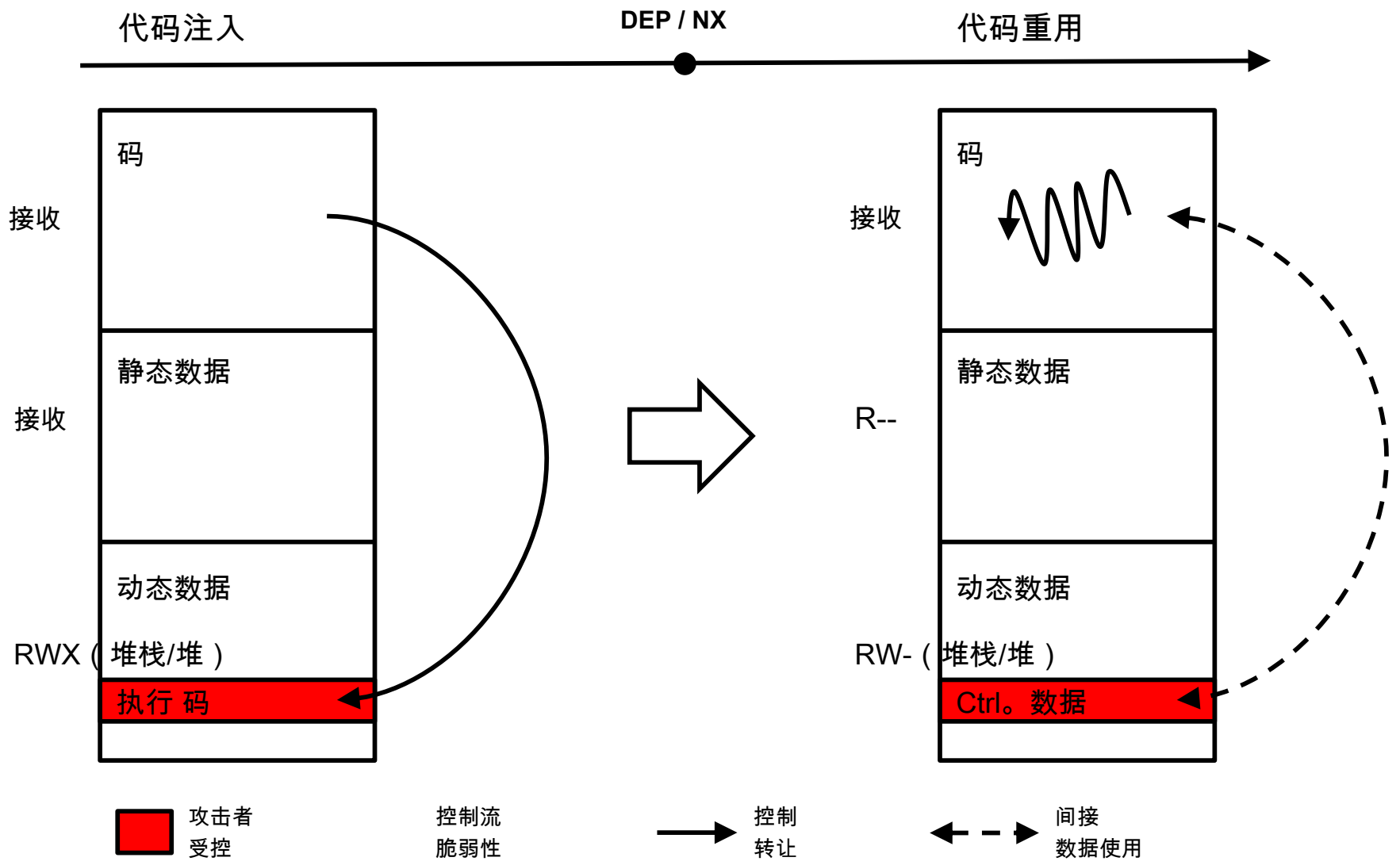
机器代码级别的演变

攻击



返回libc

- NX（W XOR X）使得无法注入代码并执行它。
  - 没有可写和可执行的内存区域
- 想法：重用现有代码
  - “幸运的” libc加载到一个恒定地址
  - 将被利用程序的控制流转移到libc代码中
  - 堆栈上的“加载”参数
  - 无需代码注入：跳转到已知地址
    - exec（），system（），printf（）
- 例如：
  - 执行（“/ bin / sh”）

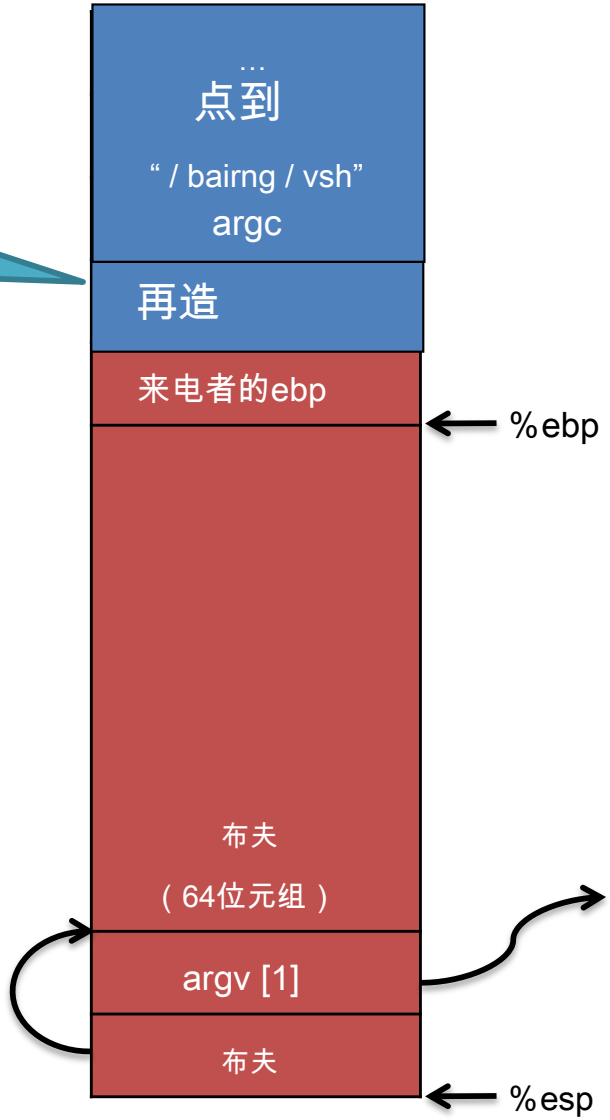
返回libc攻击

ret将控制权移交给 系统，
在堆栈上查找参数

用libc函数的地址覆盖返回地址

- 设置伪造的返回地址和参数
- 退回 将“调用” libc函数

没有注入代码！



面向返回的编程（ROP）

- 返回libc似乎很有限并且很容易被击败
  - 攻击者无法执行任意代码
  - 攻击者依赖libc的内容

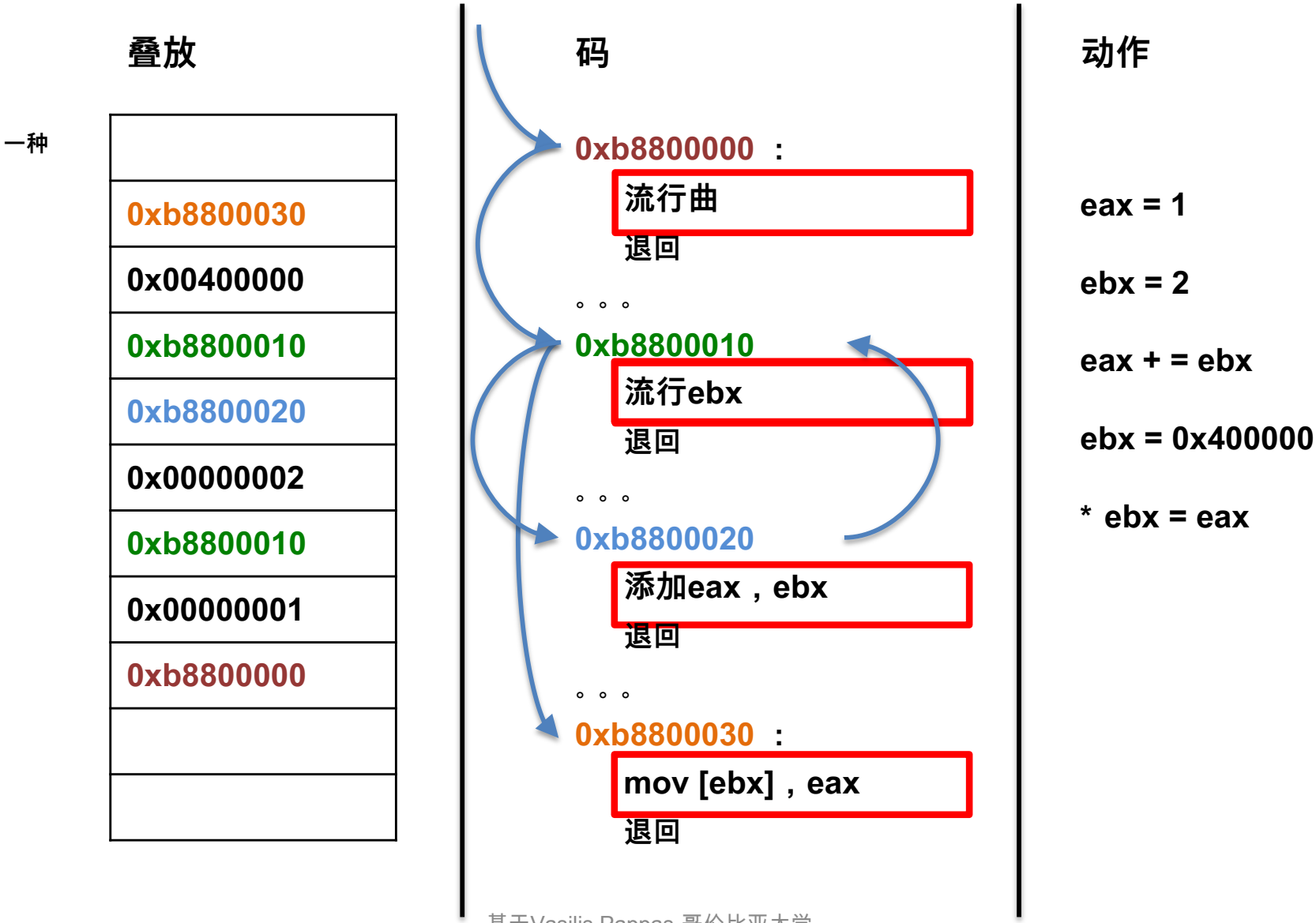
- 这种看法是错误的：面向返回的编程和面向跳转的编程

  - 返回libc的特殊情况
  - 任意攻击者的计算和行为（考虑到可以使用的足够大的代码库）

ROP：方法

- 最直接的灵感来自 *借用的代码块[* 克拉默（2005）
  - 查找允许执行某些给定操作的简短指令序列
  - 小工具
  - 使用“ ret”将它们“连锁”在一起
- JOP攻击=使用 *跳* 代替 *退回*

面向收益的程序设计


叠放
码
动作
一种
0xb8800030
0x00400000
0xb8800010
0xb8800020
0x00000002
0xb8800010
0x00000001
0xb8800000
0xb8800000 :
流行曲
退回
。。。
0xb8800010
流行ebx
退回
。。。
0xb8800020
添加eax，ebx
退回
。。。
0xb8800030 :
mov [ebx]，eax
退回
eax = 1
ebx = 2
eax + = ebx
ebx = 0x400000
* ebx = eax

ROP：方法

- 图灵完整的小工具集可以执行任意计算
  - 漏洞利用不受直线限制
  - 已证明可在大多数架构上使用
  - 等同于拥有虚拟机/解释器
- 根本不调用任何功能
  - 不能通过删除诸如system（ ）之类的功能来击败
  - 必须知道内存映射（无ASLR）
  - 需要找到有趣的小工具并按照给定的顺序将它们链接起来
- 特定的编译器（例如ROPC）
  - 自动化技术来查找那些代码序列
  - 满意度模理论（SMT）求解器

ROP：后果与保护

- 恶意代码检测不能仅限于可执行内存区域
  - 面向回报的rootkit /恶意代码…
  - 甚至非可执行内存也需要验证
- ROP被ASLR击败
  - 链接退货需要提前知道地址
- 盲ROP
  - 可以了解小工具，暴力破解和监视副作用的位置
  - 堆栈学习一次覆盖一个字节并对其进行暴力破解。

堆缓冲区溢出

- 堆是运行时用于动态分配的内存池
  - malloc（）获取堆上的内存
  - 自由（）释放堆上的内存
- 数据块存储在双链表typedef结构__HeapHdr__ {

  结构__HeapHdr__ * next;

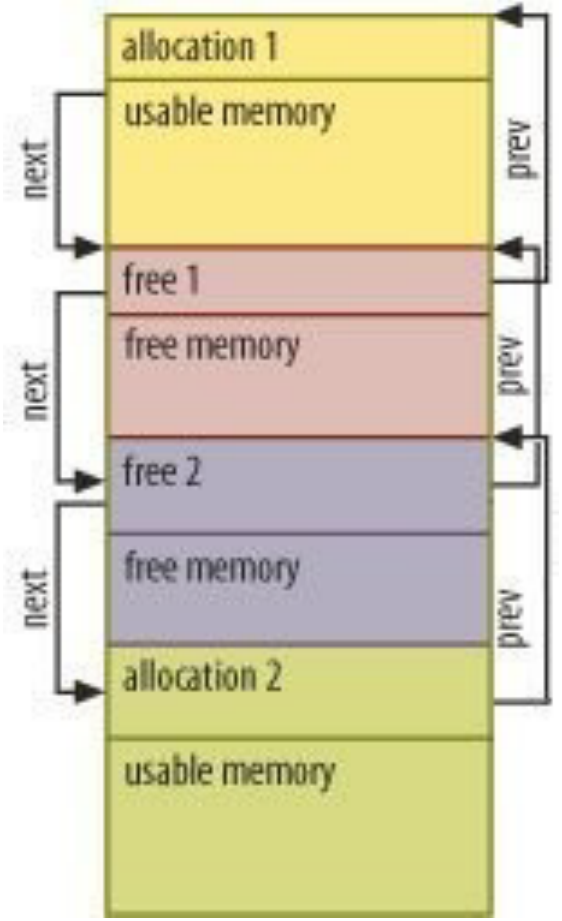
  结构__HeapHdr__ *上一页;

  无符号整数大小；

  使用unsigned int；

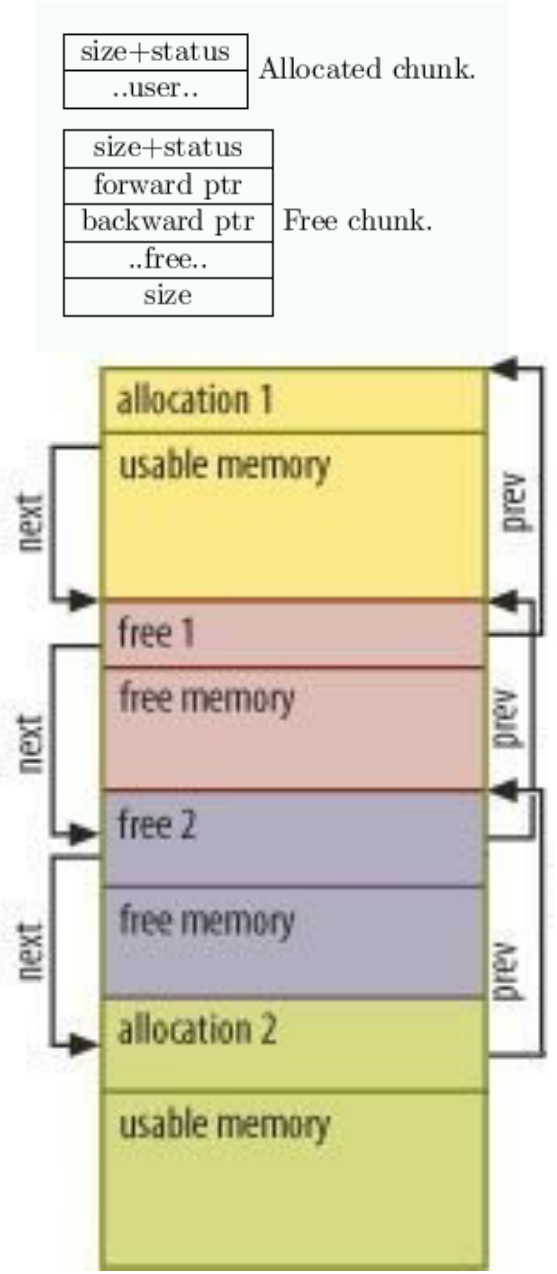
  //可用的数据区域从此处开始} HeapH

dr_t;



堆缓冲区溢出

- 下一个/上一个指针存储在数据之后
  - 溢出：覆盖上一个/下一个指针（标题）
- 释放块=更新双链表
  - 这允许在任意地址（红色=攻击者控制）上写入任意值，例如函数指针
  - FD = hdr-> 下一个
  - BK = hdr-> 上一个
  - FD ->上一个= BK
  - BK ->下一个= FD
  - cf. https://www.win.tue.nl/~aeb/linux/hh/hh-11.html
- 检测很简单：
  - 测试是否（hdr-> prev-> next == hdr）否则正在进行攻击！
  - 金丝雀



堆溢出开发

- 直接攻击：修改函数指针
  - 简单溢出到指针位置
- 通常是对堆栈返回地址的间接攻击
  - 用堆栈中返回地址的地址填充标头
  - 下一个malloc / free操作将随意修改返回地址

- 堆喷涂：
  - 利用连续的块放置（例如，浏览器，PDF，Flash）
  - 用NOP底座+有效负载填满整个块并将其重复喷入堆中

- 可能很复杂
  - 需要预测堆布局，控制程序状态
  - 否则，导致程序处于可利用状态

软件开发：更大透视

- 软件故障注入
  - 为某一目的而构建的软件，但攻击者滥用该软件出于另一目的
  - 特别是通过特制的输入
  - 可以利用任何图灵机
- 硬件故障注入
  - 不要忘记软件在硬件中运行
  - 在代码执行期间干扰执行环境（激光，电源故障，时钟故障）
  - 宇宙/伽玛射线会导致随机错误（位翻转）
  - 特定的内存访问模式会导致DRAM中的位错误

比赛条件

- 并行执行任务
  - 多进程或多线程环境
  - 多用户
  - 任务可以彼此交互
- 存在竞争条件必须具备三个属性：
  - **并发：** 必须至少有两个控制流同时执行。
  - **共享对象：** 两个并发流都必须访问共享的竞争对象。
  - **状态变更：** 至少其中一个控制流必须更改比赛对象的状态
- 任务的结果取决于事件的相对时间
  - 非确定性行为

比赛条件：基础

- 程序员将一组操作视为原子操作
  - 实际上，原子性不是强制性的
  - 调度程序可以随时中断进程
  - 如果存在阻塞的系统调用，则更有可能
- 攻击者可以利用这种差异
- 竞争情况漏洞通常在以下情况下出现：
  - 检查给定的特权，以及
  - 行使特权
- 通过相互冲突的操作消除竞争条件

TOC（T）TOU：检查时间（至）使用时间

- 检查一下 – 建立一些前提条件（不变性），例如访问权限
- 采用 – 假定不变式仍然有效，则在对象上进行操作

- 可以在任何并发系统中发生：
  - 共享内存（或地址空间）
  - 文件系统
  - 信号

共享内存

- 任务之间共享内存可能导致比赛
  - 线程共享整个内存空间
  - 进程可以共享内存映射区域
- 使用同步原语：
  - 锁定，信号量
  - Java：
    - 同步的类和方法（监视器模型）
    - 原子类型（java.util.concurrent.atomic.AtomicInteger等）
- 避免共享内存：
  - 使用消息传递模型
  - 仍然需要正确的同步！

共享内存竞赛：示例

```
public class Counter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest in,
                      HttpServletResponse out)
    {
        out.setContentType("text/plain");
        Printwriter p = out.getWriter();
        count++;
        p.println(c
    }
}
```

看起来是原子的（1行代码！）

- 不是！

简单比赛：

- 2个线程读取计数
- 都写计数+1
- 错过了1个增量

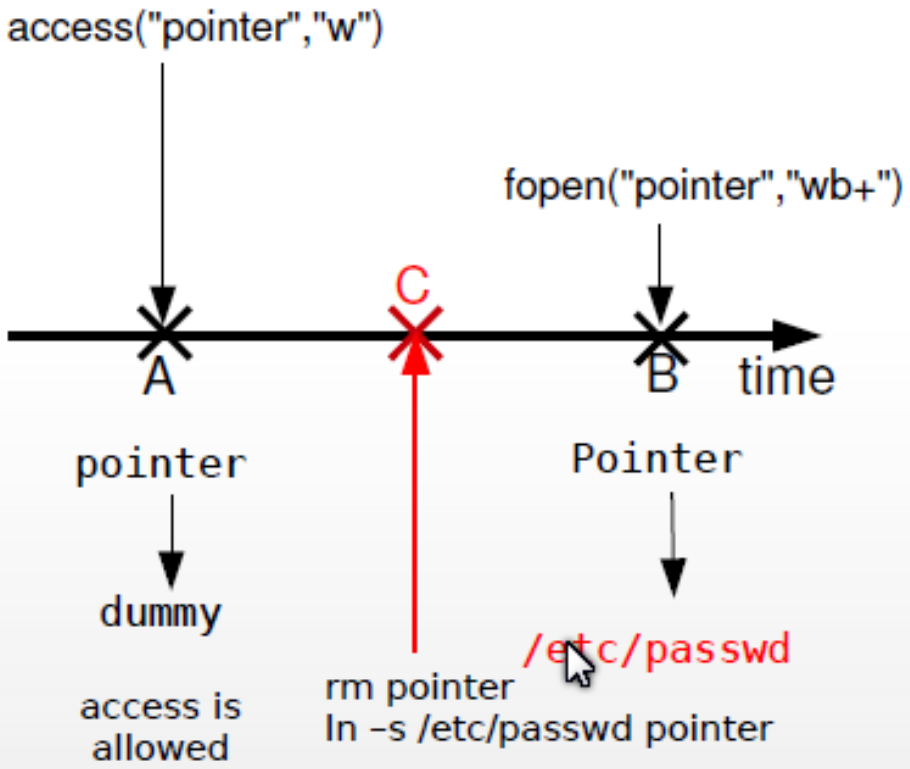
UNIX文件系统安全性

- 访问控制：仅在用户有权访问文件的情况下，他才能访问文件
- 但是，如果用户以setuid-root身份运行怎么办？
  - 例如，为了访问打印机设备，打印程序通常是setuid-root
    - 以“好像”用户具有root特权的方式运行
  - 但是root用户可以访问任何文件！
  - 打印程序如何知道用户有权读取（和打印）任何给定的文件？
- UNIX有一个特殊的 访问（ ） 系统调用

Unix文件系统：访问/公开竞赛

```
/* access returns 0 on success */
if(!access(file, W_OK)) {
    f = fopen(file, "wb+");
    write_to_file(f);
} else {
   fprintf ( stderr , “权限被拒绝 ,
                    无法打开%s。\n“ , 文件 ) ;
}
```

```
$ touch dummy; ln -s dummy pointer
$ rm pointer; ln -s /etc/passwd pointer
```



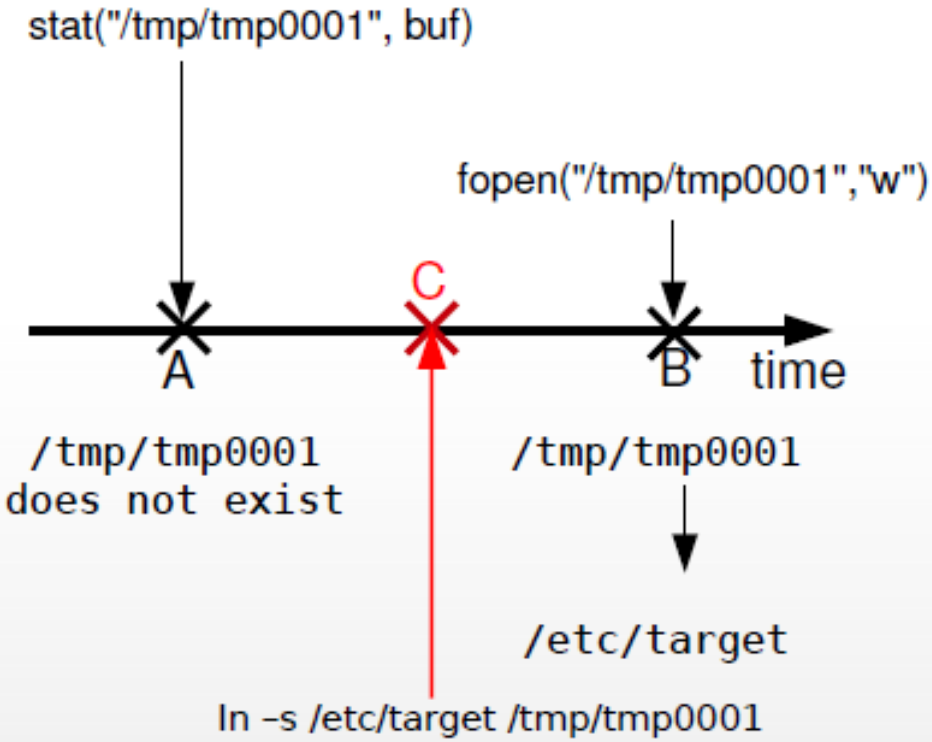
临时文件竞赛

- 与常规文件类似的问题
  - 通常在/ tmp或/ var / tmp中打开
  - 在/ tmp中创建文件不需要特殊权限
  - 通常容易猜到的文件名

- 可能的攻击方式：
  - 猜tmp文件名：“ / tmp / tmp0001”
  - ln -s / etc / target / tmp / tmp0001
  - 当它试图创建临时文件时，受害者程序将为您创建文件/ et c / target！
  - 如果第一个猜想不起作用，请尝试一百万次

临时文件竞赛

- A: program checks if file "/tmp/tmp0001" already exists
- B: program creates file "/tmp/tmp0001"
  - /etc/target is created!



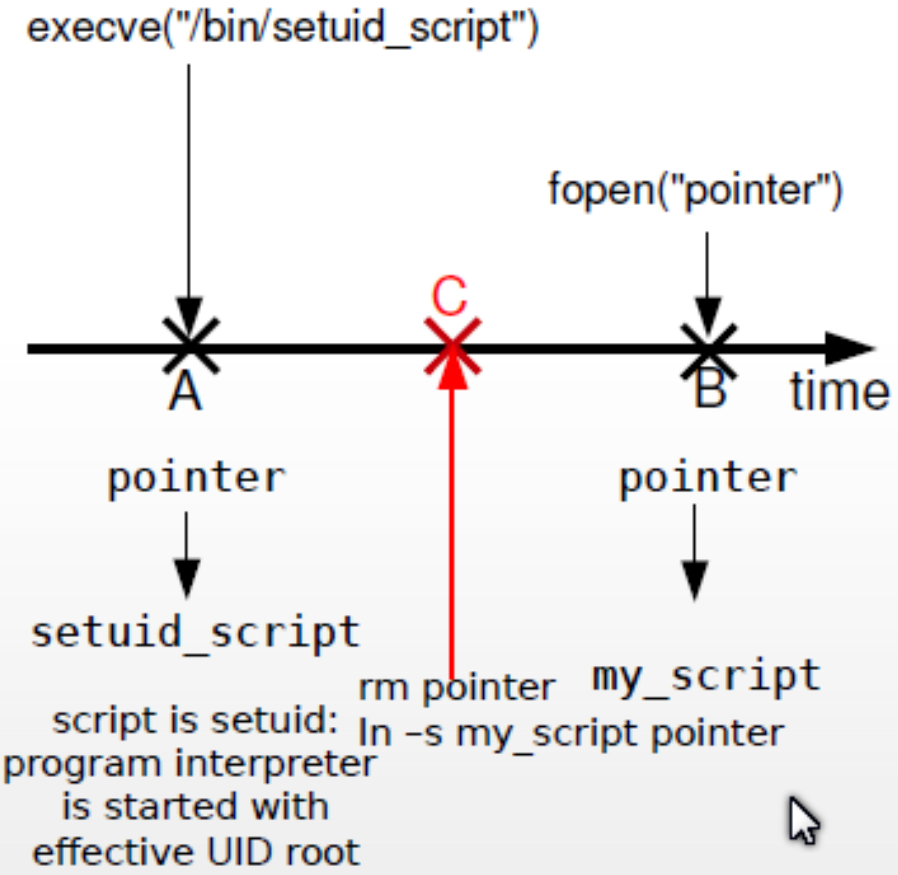
- Attack:

```
$ ln -s /etc/target /tmp/tmp0001
```

Unix文件系统：脚本执行竞赛

- 文件名重定向
  - 再次软链接
- Setuid脚本
  - execve（）系统调用在执行程序之前调用seteuid（）调用
  - 答：程序是一个脚本，因此首先加载命令解释器
  - B：以脚本名称调用程序解释器（具有root特权）
- 攻击者可以在步骤A和B之间替换脚本内容
- 在大多数平台上的脚本上都不允许使用Setuid！
  - 一些解决方法

- A: program interpreter is started (with root privilege)
  - e.g: /bin/sh, /usr/bin/python,
- B: program interpreter opens script pointed to by "pointer"
- Interpreter runs the script



- Attack:

```
$ ln -s /bin/setuid_script pointer
$ rm pointer; ln -s my_script pointer
```

线程程序：售后使用

- Thread 1

```
extern int * a;
a = malloc(10);
// Launch Thread 2

if(some_error)
   free(a);
```

- Thread 2

```
extern int *a;

/* is password checked
   ?*/
if(a[0])
   /* do passwd
   protected stuff */
```

- Thread 3

```
/* same memory
   block allocated
   */
X=malloc(10);
X[0]=1;
```

漏洞窗口

- 漏洞窗口可能很短
  - 测试很难发现种族状况问题
  - 难以复制和调试
- 关于比赛条件的神话
  - ”*种族很难利用*”
  - ”*种族不能被可靠地利用*”
  - ”*在10000中只有1次机会可以起作用！*”
- 攻击者通常可以找到克服困难的方法！
  - 反复尝试
  - 攻击者可以尝试减慢受害机器/进程的速度，以提高可能性（高负载，计算复杂性）
  - 攻击者可以并行进行多次攻击，以增加处理器在正确的时间安排攻击过程的可能性

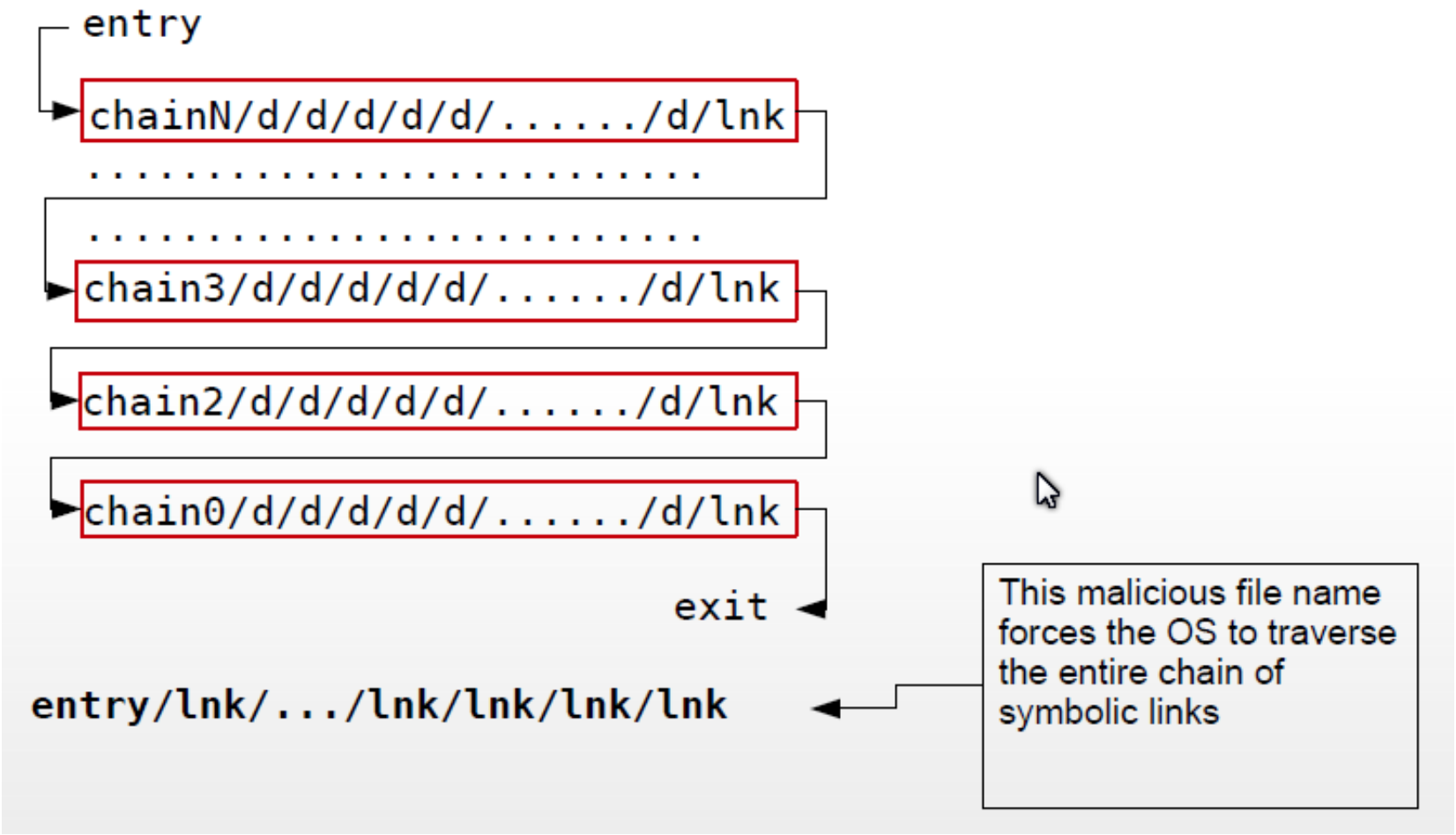
文件查找速度慢

- 深度嵌套的目录结构：
  - d / d / d / d / d / d / d /...../ d / file.txt
- 要解析此文件名，操作系统必须：
  - 在当前工作目录中查找名为d的目录
  - 在该目录中查找名为d的目录
  - 。。。
  - 在最终目录中查找名为file.text的文件
- 限制文件名的长度：
  - MAXPATHLENGTH（在Linux上为4096）
  - 最大深度约2000

使其变慢：File SystemMaze

- 将深层嵌套的目录结构与符号链接链相结合
  - MAXPATHLENGTH将文件参数的长度限制为单个系统调用（例如，打开，访问）
  - 但是文件名的一部分本身可以是链接
  - 链接链的长度受内核参数限制
    - Linux机器上的40
- 文件系统查询总数：
  - 跟随40个连锁店...
  - 。。。每个都有2000个嵌套目录
  - 80000个查询！

文件系统迷宫


entry
chainN/d/d/d/d/d/...../d/lnk
.....
.....
chain3/d/d/d/d/d/...../d/lnk
chain2/d/d/d/d/d/...../d/lnk
chain0/d/d/d/d/d/...../d/lnk
exit
entry/lnk/.../lnk/lnk/lnk/lnk
This malicious file name forces the OS to traverse the entire chain of symbolic links

预防与检测

- 预防：许多解决方案取决于实际比赛
  - 特定于操作系统的解决方案：与ID或文件名相关
  - 分叉：将操作委托给具有EUID（有效UID）的单独流程
  - 锁定：抑制种族，但减慢进程
  - 硬度增加：降低攻击者的成功概率（k种族，伪原子交易）
- 检测：
  - 模式匹配的静态分析
  - 通过模型检查（MOPS，RacerX，rccjava）进行静态分析
  - 动态分析（橡皮擦）