

## **摘要**

### **一、背景介绍**

- 1.1 通用概念
  - 1.1.1 系统调用
  - 1.1.2 ptrace

- 1.2 赛题描述
  - 1.2.1 基础功能
  - 1.2.2 扩展功能

### **二、比赛题目分析和相关资料调研**

- 2.1 比赛题目分析
- 2.2 相关资料调研
  - 2.2.1 论文参考
  - 2.2.2 博客参考

### **三、系统框架设计**

#### **3.1 目前已实现的远程系统调用**

- 3.2 基于Ptrace的框架设计
  - 3.2.1 系统调用的拦截
  - 3.2.2 远程系统调用的执行
- 3.3 框架设计详细解释
  - 3.3.1 指针参数分类
    - 3.3.1.1 按照级数分类
    - 3.3.1.2 按指针指向内容
    - 3.3.1.3 按远程系统调用的行为
  - 3.3.2 通过Ptrace实现系统调用的截取
  - 3.3.3 远程系统调用请求编组
  - 3.3.4 远程系统调用请求解组
  - 3.3.5 远程系统调用请求执行结果编组
  - 3.3.6 远程系统调用请求执行结果解组

### **四、开发计划**

### **五、比赛过程中的主要进展**

### **六、系统测试情况**

### **七、遇到的主要问题和解决方式**

- 7.1 ptrace截获系统调用的弊端
  - 7.1.1 ptrace无法跳过系统调用的执行
  - 7.1.2 使用PTTRACE\_SYSEMU
  - 7.1.3 重定向系统调用号
- 7.2 对于指针类型的参数如何进行传参
  - 7.2.1 系统调用参数的传递和分类
  - 7.2.2 指针类型参数传递
  - 7.2.3 重定向指针并使用额外的缓冲区
- 7.3 有选择性远程执行系统调用

### **八、分工和协作**

- 8.1 参赛队伍
- 8.2 协同和分工

### **九、提交项目文件简介**

- 9.1 项目目录结构
- 9.2 项目文件简介
  - 9.2.1 include
  - 9.2.2 client
  - 9.2.2 server
- 9.3 项目运行
  - 9.3.1 server端
  - 9.3.2 client端
  - 9.3.3 演示结果
  - 9.3.4 演示视频录制

# 摘要

远程系统调用这一概念最初由Caceres M 在 *Syscall proxying-simulating remote execution* 中提出，不过他给其命名为 *Syscall proxying*。提出目的是为了实现一种旨在简化权限提升阶段的技术：通过向目标操作系统提供直接接口，它允许攻击代码和工具自动控制远程资源。系统调用代理透明地将进程的系统调用“代理”到远程服务器，有效地模拟远程执行。

当然，这一技术不仅仅可以被用于漏洞渗透，Casek 在 *22nd Chaos Communication Congress* 的PPT中提到，该技术也可被用于：

- Remote pathcing for minor upgrades
- Remote debugging
- Transparent remote IPC

除此之外，通过进行系统调用级的client-server间协同，也可被用于资源卸载、任务卸载等研究领域，具体的技术细节还需要根据实现目的来决定。

## 一、背景介绍

### 1.1 通用概念

#### 1.1.1 系统调用

典型的软件进程在其执行的某个时刻需要与某些资源交互：磁盘中的文件、屏幕、网卡、打印机等。进程可以通过系统调用访问这些资源。这些系统调用是操作系统服务，通常标识为用户模式进程和操作系统内核之间的最低通信层，在linux中，系统调用是用户态访问内核的唯一合法方式，除异常和陷入外。如下图所示：

系统调用由操作系统核心提供，运行于内核态，用户进程通过软中断（INT 0x80）来触发系统调用。

类似中断机制，Linux内核为每一个系统调用实现了一个子程序来实现系统调用的功能，并设置了一张系统调用表 *syscall\_table*，通过系统调用号来寻址相应的系统调用子程序。而系统调用所需的参数则依次通过以下寄存器来传递：RDI, RSI, RDX, R10, R8, R9，而系统调用号则通过RAX来传递。

当然，一般情况下我们并不需关系这些，所有的调用细节全由库函数如 *glibc* 实现了，我们只需要调用库函数提供给我们的API就行了。

#### 1.1.2 ptrace

*ptrace()* 是Linux下的一个系统调用，通过该系统调用可以使得一个进程（tracer）去观察和控制另一个进程（tracee）的执行，并检查和更改tracee的内存和寄存器。主要实现于断点调试和系统调用跟踪，是反调试技术的基础手段，我们熟悉的 *gdb* 和 *strace* 就是利用 *ptrace()* 进行编写的。

*ptrace()* 的使用方法如下所示：

```
1 #include <sys/ptrace.h>
2 long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void
3             *data);
4 # return
5 #   成功返回0, 错误返回-1, errno被设置
6 # error
7 #   EPERM    - 特殊进程, 不可以被跟踪或进程已经被跟踪
```

```

7 # ESRCH - 指定进程不存在
8 # EIO - 请求非法
9 # EBUSY - (i386 only) 分配或释放调试寄存器时出错
10 # EFAULT - 试图读取或写入跟踪器或被跟踪者内存中的无效区域，可能是因为该区域未映射或不可访问。不幸的是，在 Linux 下，此故障的不同变体将或多或少任意返回 EIO 或 EFAULT。
11 # EPERM - 无法追踪指定的进程。这可能是因为跟踪器没有足够的权限（所需的能力是 CAP_SYS_PTRACE）；出于显而易见的原因，非特权进程无法跟踪它们无法向其发送信号的进程或
12 # 运行 set-user-ID/set-group-ID 程序的进程。或者，该进程可能已经被跟踪，或者（在
13 # 2.6.26 之前的内核上）是 init(1) (PID 1)。
14 #
15 #

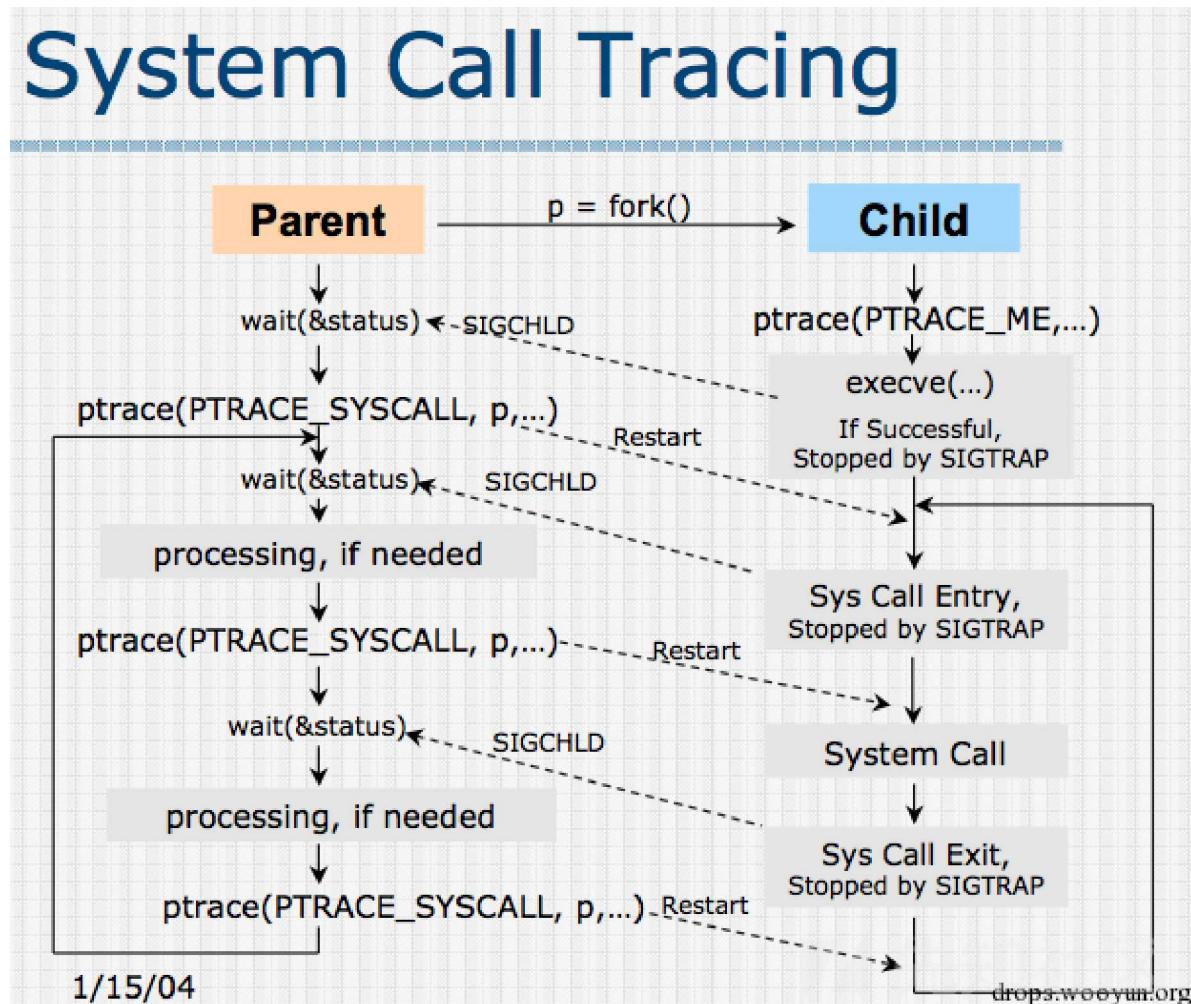
```

*ptrace()* 通过不同的标志位来实现不同的功能。一般而言，使用 *ptrace()* 跟踪进程有两种方式：

1. 父子进程间：使用**PTRACE\_TRACEME**标志，子进程作为被跟踪者（tracee），父进程作为跟踪者（tracer）。
2. 不相关进程间：使用**PTRACE\_ATTACH**标志，跟踪进程（tracer）使用指定进程PID跟踪指定进程（tracee）。

*ptrace*的工作原理：**tracee**会在接收到一个信号时进入暂停态，哪怕这个信号被忽略掉（除了**SIGKILL**信号，它会正常工作）。紧接着**tracer**会通过[waitpid\(2\)](#)（或其他wait类的系统调用）得知**tracee**进入了暂停态，该调用（[waitpid\(2\)](#)）会返回一个status值来解释**tracee**暂停的原因。而当**tracee**暂停后，**tracer**可以使用不同的 *ptrace request* 来检查和更改**tracee**的内存和寄存器。当**tracer**完成自己的逻辑之后，**tracer**会通过 *ptrace request* (**PTRACE\_SYSCALL**, **PTRACE\_SYSEMU**或其他标志) 使**tracee**继续运行，可以选择忽略掉传递来的信号（或者发送不同的信号作为代替）。

以下是通过 *ptrace()* 跟踪进程的流程（图为引用）：



可以向ptrace()传入不同的标志参数来实现不同的操作，ptrace()常用的标志参数有以下几种：

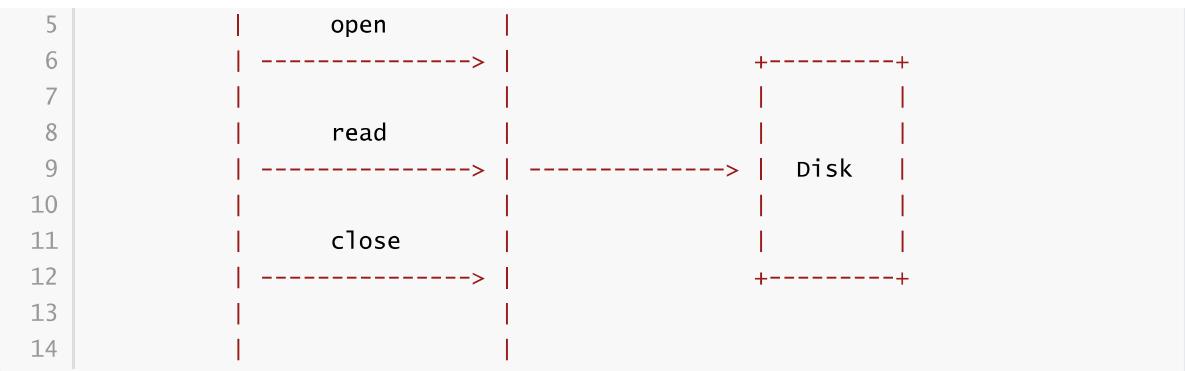
1. **PTRACE\_TRACEME**: 在子进程中使用，指明孩子进程将要被其父进程所跟踪，除了该标志位在子进程（即tracee）中被使用，其余ptrace request都要在父进程（tracer）中使用。除了PTRACE\_ATTACH, PTRACE\_SEIZE, PTRACE\_INTERRUPT和PTRACE\_KILL之外的其他ptrace request，都需要在tracee处于暂停态才能使用。在子进程中使用该请求时，pid、addr、data参数被忽略，可为0。
2. **PTRACE\_PEEKTEXT, PTRACE\_PEEKDATA**: 从addr指向的tracee的虚拟内存地址空间读取一个字的内容，读取内容作为ptrace的结果返回。Linux没有独立的代码和数据地址空间，所以两个标志等同。（data参数被忽略）通过查看源码可知，**PTRACE\_PEEKTEXT, PTRACE\_PEEKDATA**操作的字节大小为*unsigned long* (x86, Linux-5.10)，而函数的操作是以words为单位的，即4个字，而不是4个字节。
3. **PTRACE\_PEEKUSER**: 从addr指向的tracee的USER区域读取一个字的内容，该区域包含进程寄存器和一些关于进程的信息（看<sys/user.h>）。读取的内容作为ptrace的调用结果返回。通常来说，addr指向的地址偏移必须是字对齐的，尽管在不同体系下这一点可能有些细微的不同。（data参数被忽略）
4. **PTRACE\_POKETEXT, PTRACE\_POKEDATA**: 从data复制一份数据到addr指向的tracee的内存区域。两个标志目前没啥区别。
5. **PTRACE\_POKEUSER**: 从data复制一份数据到addr指向的tracee的USER区域。和PTRACE\_PEEKUSER相同，addr的地址必须是字对齐的。为了维护内核的完整性，不允许对USER区域进行一些修改。
6. **PTRACE\_GETREGS, PTRACE\_GETFPREGS**: 复制tracee的通用寄存器（PTRACE\_GETREGS）和浮点寄存器（PTRACE\_GETFPREGS）内容到data指向的tracer的内存中，data的数据格式参见<sys/user.h> (x86体系的通常在/usr/include/x86\_gnu\_linux/sys/user.h, addr参数被忽略)。注意，在SPARC系统中，data和addr的作用相反，data参数会被忽略且寄存器内容复制到addr中。注意，PTRACE\_GETREGS, PTRACE\_GETFPREGS并非在所有架构上都存在。
7. **PTRACE\_GETREGSET** (Linux 2.6.34加入) : 读取tracee的寄存器。addr以与体系结构相关的方式指定要读取的寄存器类型。**NT\_PRSTATUS** (数值为1) 通常会导致读取通用寄存器。例如，如果CPU具有浮点和/或向量寄存器，则可以通过将addr设置为相应的**NT\_foo**常量来检索它们。data指向一个struct iovec，它描述了目标缓冲区的位置和长度。返回时，内核修改iov.len以指示返回的实际字节数。
8. **PTRACE\_SETREGS, PTRACE\_SETFPREGS**: 修改tracee的通用寄存器（PTRACE\_SETREGS）和浮点寄存器内容（PTRACE\_SETFPREGS），修改内容来自tracer的data。和PTRACE\_POKEUSER标志相同，对于某些通用寄存器的修改将不被允许（addr参数会被忽略）。注意，与PTRACE\_GETREGS, PTRACE\_GETFPREGS相同，在SPARC系统上，addr和data参数的意义和作用相反。
9. **PTRACE\_SETREGSET** (Linux 2.6.34加入) : 修改tracee的寄存器，data和addr的作用与PTRACE\_GETREGSET中的类似。

有关ptrace()更详细的说明请自行查阅[Linux手册](#)

## 1.2 赛题描述

系统调用(Syscall)作为操作系统对用户态提供的编程接口，通常是用户态进程和内核之间最底层的通信接口。这里我们只考虑为Unix Like的系统平台做相关设计，在Unix Like环境下OS为每个特定的系统调用提供了系统调用编号，通过栈和寄存器来传递系统调用的参数，通常系统调用服务的个数是有限的，并且libc为我们封装好了更加友好的系统调用接口。这里以进程从文件中读取内容为例：

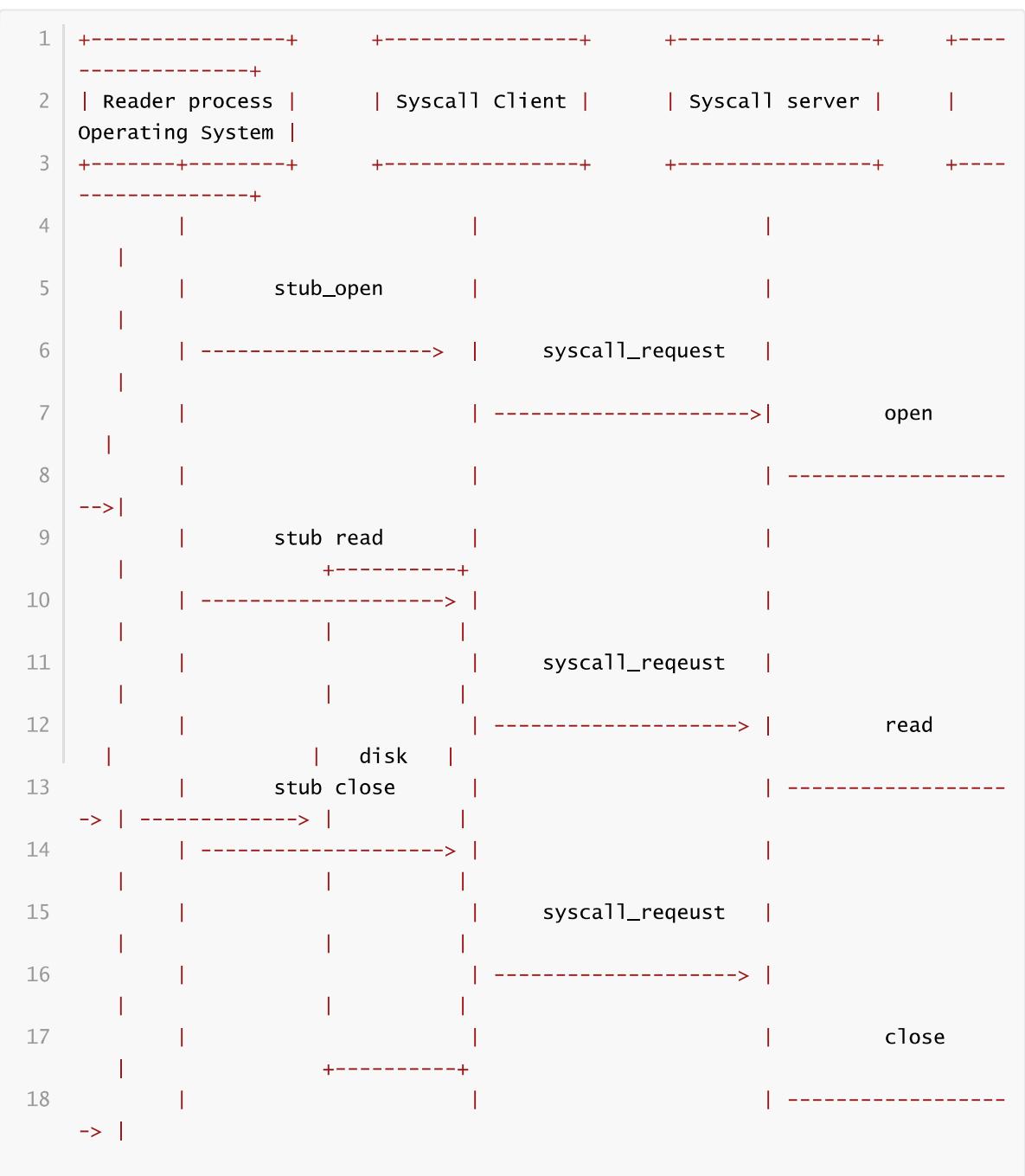
1	+-----+	+-----+
2	Reader	Linux
3	+-----+	+-----+
4		



本项目的目标是实现一个Remote Syscall执行框架，通过这个执行框架用户可以将客户端的一个指定进程的系统调用进行截获，并转发到服务端上下文进行执行，然后将系统调用结果返回给客户端。项目的本质是为了实现一个远程系统调用模拟环境。

### 1.2.1 基础功能

- 1、支持远程系统的调用的基本功能，能够将**进程内特定的系统调用**拉远到服务端执行并返回对应的结果。



- 2、支持将某个特定的本地进程的所有系统调用拉远到服务端执行，使得本地进程能够执行在远程服务端上下文并返回正确的执行结果。
- 3、支持通过tcp或者virtio-vsock**两种不同通信通道来传递系统调用**，通信通道做到独立解耦。

## 1.2.2 扩展功能

- 1、支持对应用程序代码系统调用修改的方式，将应用程序的系统调用拉远到服务端上下文执行。
- 2、支持通过**LD\_PRELOAD**，或者**ptrace**和**ebpf**规则的方式将进程内特定的系统调用，做到非侵入式系统调用拉远。
- 3、尽量简化服务端执行上下文的架构。

# 二、比赛题目分析和相关资料调研

## 2.1 比赛题目分析

赛题要求是实现进程的系统调用拉到云端去执行，获取云端执行结果之后返回到本地进程，使得本地进程正常运行，但是实际上与之交互的操作系统是云端的服务器操作系统。

远程系统调用这一概念最初由Caceres M 在 *Syscall proxying-simulating remote execution* 中提出，不过他给其命名为 *Syscall proxying*。提出目的是为了实现一种旨在简化权限提升阶段的技术：通过向目标操作系统提供直接接口，它允许攻击代码和工具自动控制远程资源。系统调用代理透明地将进程的系统调用“代理”到远程服务器，有效地模拟远程执行。

当然，这一技术不仅仅可以被使用于漏洞渗透，Casek 在 22nd Chaos Communication Congress 的PPT 中提到，该技术也可被用于：

- Remote pathcing for minor upgrades
- Remote debugging
- Transparent remote IPC

除此之外，通过进行系统调用级的client-server间协同，也可被用于资源卸载、任务卸载等研究领域，具体的技术细节还需要根据实现目的来决定。

在与赛题提出方进行沟通交流之后，暂时明确了实现目的为资源卸载，将性能较弱的客户机上对于性能需求等系统调用需求较强的放在远程执行，只要求实现常用的2, 30个系统调用，较强的需求是尽量简化其执行框架。

较为理想的实现手段是通过动态库来拦截相应的函数发起的系统调用，类似 *glibc* 在应用程序与内核之间发挥的作用。但是由于与赛题方沟通较晚，所以我们首先实现目标是实现全部的系统调用，故采用了基于 *ptrace* 的远程执行框架，这就导致了相关框架较为臃肿，因为要考虑到大部分的系统调用。

而基于动态库的框架还没有完成。

## 2.2 相关资料调研

### 2.2.1 论文参考

1. [Caceres M. Syscall proxying-simulating remote execution\[J\]. Core Security Technologies, 2002](#)
2. [Kerber F, Teubert D, Meyer U. Covert remote syscall communication at kernel level: A SPOOKY backdoor\[C\]//2015 10th International Conference on Malicious and Unwanted Software \(MALWARE\). IEEE, 2015: 74-81.](#)

3. Casek. Syscall proxying fun and applications. 22nd Chaos Communication Congress, <http://events.ccc.de/congress/2005/fahrplan/events/553.en.html>, 2005
4. F. Balestra and R. Branco. Syscall Proxying 2Pivoting Systems. Hackers to Hackers Conference III, <http://www.kernelhacking.com/rodrigo/docs/H2HCIII.pdf>, 2007.

## 2.2.2 博客参考

ptrace:

1. [使用 Ptrace 拦截和模拟 Linux 系统调用](#)

google gvisor:

1. [gVisor: 谷歌发布的一个用于提供安全隔离的轻量级容器运行时沙箱](#)
2. [Kubernetes的Kata Containers 与 gVisor - ghostwritten](#)
3. [gVisor是什么? 可以解决什么问题](#)

LD\_PRELOAD:

1. [LD\\_PRELOAD的偷梁换柱之能](#)

ebpf:

1. [从Falco看如何利用eBPF检测系统调用](#)
2. [BPF 学习路径总结](#)
3. [https://github.com/DavadDi/bpf\\_study](https://github.com/DavadDi/bpf_study)

## 三、系统框架设计

### 3.1 目前已实现的远程系统调用

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
257	sys_openat	int dfd	const char *filename	int flags	int mode		
1	sys_write	unsigned int fd	const char *buf	size_t count			
0	sys_read	unsigned int fd	char *buf	size_t count			

### 3.2 基于Ptrace的框架设计

整体框架主要分为两部分：

1. 系统调用的拦截

2. 系统调用的远程执行

### 3.2.1 系统调用的拦截

#### 3.2.1.1 基于ptrace

*ptrace()* 是Linux下的一个系统调用，用来实现对程序的系统调用的跟踪、控制。

*ptrace()* 通过不同的标志位来实现不同的功能。一般而言，使用 *ptrace()* 跟踪进程有两种方式：

1. 父子进程间：使用**PTRACE\_TRACEME**标志，子进程作为被跟踪者（tracee），父进程作为跟踪者（tracer）。
2. 不相关进程间：使用**PTRACE\_ATTACH**标志，跟踪进程（tracer）使用指定进程PID跟踪指定进程（tracee）。

通过 *ptrace()* 对于被跟踪进程进行监控，获取其每一次系统调用的参数（通过**PTRACE\_GETREGS**和**PTRACE\_SETREGS**），以及其对应的内存数据（通过**PTRACE\_PEEKDATA**和**PTRACE\_POKEDATA**），然后通过框架的第二部分将其编组为远程端可识别的消息格式，发送到远程执行。

#### 3.2.1.2 远程系统调用执行的代码区

然而，这两种方式都有一个共性：tracee被全程跟踪。即程序初始化时、程序正式执行时、程序销毁时 tracee 的所有系统调用都会被 tracer 所跟踪。这必然会干扰到我们的实现目标（我们并非是盲目的远程发送所有系统调用，如果那样还不如 *rpc* 或 *ssh*）。那么接下来我们就有了一个新的实现目标：所以我们需要跳过程序初始化时以及程序销毁时 tracee 执行的系统调用。

所以需要对被跟踪进程进行一些特殊处理，目前采取的思路是使用进程间同步手段来实现对某一特定代码区进行远程系统调用执行。

### 3.2.2 远程系统调用的执行

#### 3.2.2.1 系统调用的规律观察

在这一部分需要对拦截到的系统调用参数进行编组，并考虑到不同的系统调用之间的区别分别对其进行相应的处理。

虽然系统中存在300多种系统调用，且这些系统调用设计到系统运行的各个方面，但是在对所有系统调用进行仔细观察之后，可以发现其大致规律（基于X86）：

1. 系统调用参数传递：都是通过RDI, RSI, RDX, R10, R8, R9, RAX这七个寄存器来传递参数和系统调用号。
2. 系统调用的执行执行：都是根据查找系统调用表来找到相应的系统调用子程序的位置来执行系统调用。
3. 系统调用的分类：
  1. 按指针指向内容：
    1. 指向整型的指针
    2. 指向字符串的指针
    3. 指向缓冲区的指针
    4. 指向结构体的指针
  2. 按远程系统调用的行为：
    1. 输入指针
    2. 输出指针

我们可以根据以上的特性对系统调用制定一个统一的框架。

### 3.2.2.2 系统调用的编组和解组

之后需要将系统调用的类别分别进行编组，将其编组为远程服务器可识别的消息格式，之后再对远程执行结果进行解组，将相应执行结果返回到被跟踪进程中去，如此便完成了一次远程系统调用的执行。

为了设置本地端-服务端间数据同步以及设置一个统一的格式，我们定义了以下的数据结构：

```
1  /*
2   * 远程系统调用请求格式:
3   -----
4   | struct rsc_header |
5   -----
6   |                   |
7   |       buffer      |
8   |                   |
9   -----
10 */
11 // 系统调用请求头
12 struct rsc_header {
13     unsigned long long int syscall;           // 系统调用号
14     unsigned int p_flag;                      // 系统调用分类
15     unsigned int size;                        // 远程系统调用请求长度，字节为单位
16     unsigned int error;                      // 出错信息
17
18     // 系统调用传参寄存器
19     unsigned long long int rax;
20     unsigned long long int rdi;
21     unsigned long long int rsi;
22     unsigned long long int rdx;
23     unsigned long long int r10;
24     unsigned long long int r8;
25     unsigned long long int r9;
26
27     // 系统调用指针参数描述信息
28     unsigned int p_location_in;              // 输入指针所在位置
29     unsigned int p_location_out;             // 输出指针所在位置
30     unsigned int p_count_in;                // 输入指针需要操作的字节数
31     unsigned int p_count_out;               // 输出指针需要操作的字节数
32
33     char * p_addr_in;                     // 输入指针指向的内存，用于服务端进行指针重定向
34     char * p_addr_out;                    // 输出指针指向的内存，用于服务端进行指针重定向
35 };
```

一些系统调用中可能会存在指针参数指向的是本地的内存地址，我们称之为**带输入指针参数的系统调用**，或者是远程系统调用会将执行结果存放到一片缓冲区中，该缓冲区的地址由指针参数表示，我们称之为**带输出指针参数的系统调用**。以上两种系统调用会使本地端-服务端存在数据同步的问题，故此需要对这两种系统调用进行特殊处理，常见的方法是在本地端发送给远程端的系统调用请求后附加一个缓冲区 *buffer* 来存放需要同步的数据，这也是我们的做法。

## 3.3 框架设计详细解释

### 3.3.1 指针参数分类

#### 3.3.1.1 按照级数分类

- 一级指针
- 二级（可能二级以上）指针

#### 3.3.1.2 按指针指向内容

- 指向整型的指针
- 指向字符串的指针
- 指向缓冲区的指针
- 指向结构体的指针

#### 3.3.1.3 按远程系统调用的行为

- 输入指针
- 输出指针

### 3.3.2 通过Ptrace实现系统调用的截取

*ptrace()* 是Linux下的一个系统调用，用来实现对程序的系统调用的跟踪、控制。

*ptrace()* 通过不同的标志位来实现不同的功能。一般而言，使用 *ptrace()* 跟踪进程有两种方式：

1. 父子进程间：使用**PTRACE\_TRACEME**标志，子进程作为被跟踪者（tracee），父进程作为跟踪者（tracer）。
2. 不相关进程间：使用**PTRACE\_ATTACH**标志，跟踪进程（tracer）使用指定进程PID跟踪指定进程（tracee）。

然而，这两种方式都有一个共性：tracee被全程跟踪。即程序初始化时、程序正式执行时、程序销毁时 tracee 的所有系统调用都会被 tracer 所跟踪。这必然会干扰到我们的实现目标（我们并非是盲目的远程发送所有系统调用，如果那样还不如 *rpc* 或 *ssh*）。那么接下来我们就有了一个新的实现目标：所以我们需要跳过程序初始化时以及程序销毁时 tracee 执行的系统调用。

#### 3.3.2.1 通过编译器手段

Finding the address of the actual *main()* function, a non-exported and compiler handled symbol, was therefore one challenge that had to be solved to make our solution generally applicable

by *Covert Remote Syscall Communication at Kernel Level: A SPOOKY Backdoor*

从一篇引用论文中可以找到一些提示：通过寻找 tracee 的真正开始执行地址 *main()* 来跳过程序初始化操作，程序销毁时同理。

该种方法当前还未考虑和实现。

#### 3.3.2.2 通过进程间同步手段

更进一步，我们可以考虑对特定一部分代码实现系统调用的拉远执行。对于这部分特定代码，我们称之为**目标代码**。

实现思路：使用进程间通信手段，当要执行目标代码（目标进程中指定的要监控部分代码）时，使得目标进程陷入沉睡或暂停，等待框架进程使用**PTRACE\_ATTACH**进行系统调用追踪；当目标代码区执行结束后，目标进程陷入沉睡或暂停，框架进程使用**PTRACE\_DETACH**解除系统调用追踪。

实现手段：

**使用System V信号量**

伪代码如下：

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <sys/ptrace.h>
6
7
8 attach = 0;
9 detach = 0;
10 target = 0;
11 target_exit = 0;
12
13 /* tracee */
14 post(attach);           // 通知tracer准备开始追踪
15 wait(target);          // 阻塞等待追踪进程开始追踪
16
17 target code...          // 要追踪的目标代码区
18
19 sem_post(target_exit);  // 通知tracer准备结束追踪
20 sem_wait(detach);       // 阻塞等待追踪进程结束追踪
21
22
23
24 /* tracer */
25 for(;;){                // 可能存在多段目标代码区
26     wait(attach);        // 阻塞等待tracee请求跟踪
27     PTRACE_ATTACH;        // 开始追踪
28     post(target);         // 通知tracee已经开始追踪
29
30     while(TRUE){          // 一段目标代码区开始追踪
31         if(trywait(target_exit)){ // tracer准备结束追踪
32             PTRACE_DETACH;      // 解除跟踪
33             post(detach);       // 通知tracee已经结束跟踪了
34             break;
35         }
36         trace...              // tracer的追踪逻辑
37     }
38 }
39 }
```

## 共享内存区

### 管道通信

需要使用到 `sys_read` 和 `sys_write` 系统调用，首先排除。

### ptrace 标志

似乎 `PTRACE_SIZE` 和 `PTRACE_INTERRUPT` 可行？我还没试过，以后再研究。或者 `EVENT_STOP`？

**wait a warm heart people...**

### 3.3.3 远程系统调用请求编组

我们使用 `struct rsc_header` 存放该请求的一些描述性信息，使用 `syscall` 来标识系统调用号；同时，根据上面我们可以知道系统调用可以根据其目的的不同以及指针参数的不同分为五类：

1. 不带指针参数的系统调用
2. 带输入指针参数的系统调用
3. 带输出指针参数的系统调用
4. 带输入/输出指针参数的系统调用
5. 带连续输入指针参数的系统调用（输入参数是结构体数组指针）

我们使用 `p_flag` 参数来标识不同的系统调用类别。当然，使用这种分类是比较粗糙的，可能会存在一些有着特殊参数意义的系统调用，比如 `sys_poll` 系统调用，传入的是一个结构体数组，分别使用两个参数进行描述，对于这些特殊的系统调用，我们可以再开一个分类来进行特殊处理。

`struct rsc_header` 的最后一个参数是 `size`，用来描述整个**远程系统调用请求**的大小，以方便 `server` 接收，防止出现“粘包问题”。

之后我们使用 `struct rsc_regs` 保存X86系统中用来传递系统调用参数的几个寄存器的值。

最后一个问题是对指针参数的处理，我们在**远程系统调用请求**后面开辟了一个 `buffer` 用来存放指针参数所指向的本地内存数据（之后我们简称该内存数据为**本地指针解引用**）。也即是对于带输入指针参数的系统调用，我们事先将**本地指针解引用**取出，将其放在 `buffer` 中。

对于每个本地指针解引用长度的计算，可以根据指针指向内容的不同，将其分为四类：

1. 指向整型的指针
2. 指向字符串的指针
3. 指向缓冲区的指针
4. 指向结构体的指针

除此之外，由于进程间的内存地址不能相互访问，所以需要通过**PTRACE\_PEEKDATA**、**PTRACE\_POKEDATA**来获取被跟踪进程的内存数据，因为以上两个标志一次性只能取 `unsigned long`（8字节）的数据，故此对于输入指针参数是字符串而言并不友好。

### 3.3.4 远程系统调用请求解组

我们使用 `socket` 中的 `TCP` 进行通信，该通信方式是面向字节流的，数据的接收和发送是没有边界的，因此，会出现粘包问题。所以为了避免此类问题，我们可以借鉴通信协议格式，使用一个约定好的固定格式的消息头来存放**远程系统调用请求**的一些信息，其中包括了请求的长度，如下所示：

```
1  /*
2   * 远程系统调用请求格式：
3   * -----
4   * | struct rsc_header |
5   * -----
6   * |          buffer      |
7   * |                      |
8   * |                      |
9   * -----
10  */
```

这样一来，在服务端检测到有消息发来之后，会先读取请求头，在获取到整个请求的长度之后，再读取整个**远程系统调用请求**。

`server` 在读取到整个**远程系统调用请求之后**，会根据请求头中的 `p_flag` 和 `syscall` 变量判断该系统调用请求的类别，进行分别的处理。

### 3.3.5 远程系统调用请求执行结果编组

远程系统调用执行结果放入 struct rsc\_header 的 rax 中，如果执行出错，errno会被赋予相应的出错码。然后消息格式同3.3.3 节。

### 3.3.6 远程系统调用请求执行结果解组

同 3.3.4 节，需要读取两部分信息：struct rsc\_header 和 buffer。执行结果放入 rsc\_header.rax 中，通过 PTRACE\_SETREGS 填到被跟踪进程的 RAX 寄存器中。对于带输出指针参数的系统调用，还需要进行内存拷贝，将 buffer 中的数据拷贝到原来的系统调用指定的内存区域。

## 四、开发计划

较为理想的实现方式是使用动态加载库，对每一个系统调用进行特殊处理，这样可以尽可能的简化执行框架，但是由于我们最初选的是基于ptrace来实现，故先完成了这部分框架的代码，基于动态加载库的另一套框架还没搭建完成。

而赛题中给出的扩展功能中有三种实现方式，分别是LD\_PRELOAD、ptrace和ebpf，正好组内有三人，可以分给不同的队员去实现不同的框架。

因为以学习为主，我们会尽可能的去挖掘这个技术可能作用的领域，所以包括但不限于这三种方式。

#### 6月-7月：

1. 目前基于ptrace的框架只能远程执行几个系统调用，我们会继续完善下去尽可能多的包括所有系统调用
2. 完成基于动态链接库的框架的搭建
3. 学习ebpf并利用ebpf实现赛题

#### 7月-8月：

完善以上三个框架、编写文档并尽可能挖掘还存在的有趣的点。

## 五、比赛过程中的主要进展

几个主要进展情况：

1. 5月1号-5月7号：调研赛题，分析赛题目的，认为赛题目的是实现安全防护工具；
2. 5月7号-5月15号：调研赛题，分析赛题目的，联系赛题导师，认为赛题目的是实现远程的资源控制，为漏洞渗透工具；
3. 5月15号：联系上了赛题导师，确定了赛题目的为资源卸载，但已经选择了ptrace作为基础；
4. 5月15日-5月30日：完成了系统调用拦截和远程系统调用的执行两部分，开始debug和编写文档；
5. 6月5日：文档编写完成，系统勉强能够运行并完成基础功能一和三；

## 六、系统测试情况

# 七、遇到的主要问题和解决方式

## 7.1 ptrace截获系统调用的弊端

### 7.1.1 ptrace无法跳过系统调用的执行

ptrace()是Linux系统中的一个系统调用，是反调试技术中的基础入门手段。ptrace的常用场景有：

1. 编写动态分析工具，如gdb, strace等
2. 反追踪（一个进程只能被一个进程追踪）
3. 代码注入
4. 不退出进程，进行在线升级

在本使用场景中，我们使用 *ptrace()* 中的**PTRACE\_SYSCALL**来暂停系统调用，并使用**PTRACE\_PEEKUSER**, **PTRACE\_GETSEGS**, **PTRACE\_SETREGS**等标志来获取、设置进程寄存器的值。但是，通过 *ptrace()* 来暂停系统调用会有一个弊端：它无法跳过系统调用的执行！也即是，即使我们获取到了本地系统调用的参数并发到云端，在获取到云端的远程调用结果这样过程中，本地的系统调用依旧会正常执行。

### 7.1.2 使用PTRACE\_SYSEMU

自Linux manual page中ptrace部分可以看到关于该参数的使用：

```
1  /*
2   * PTRACE_SYSEMU, PTRACE_SYSEMU_SINGLESTEP (since Linux 2.6.14)
3   * For PTRACE_SYSEMU, continue and stop on entry to the next system call,
4   * which will not be executed.
5   * See the documentation on syscall-stops below. For
6   * PTRACE_SYSEMU_SINGLESTEP, do the same but also
7   * singlestep if not a system call. This call is used by programs like User
8   * Mode Linux that want to
9   * emulate all the tracee's system calls. The data argument is treated as
10  * for PTRACE_CONT. The addr
11  * argument is ignored. These requests are currently supported only on x86.
12  */
```

文档中隐约提到，使用PTRACE\_SYSEMU可以跳过被暂停的系统调用的执行，但是该参数会导致追踪进程不会进入syscall-exit-stop点，也即是该参数不会在本地系统调用结束后暂停（毕竟它都跳过系统调用的执行了）。所以使用该参数也会导致我们难以将云端运行结果返回到本地进程。

### 7.1.3 重定向系统调用号

既然我们需要系统调用的syscall-enter-stop（执行前暂停）与syscall-exit-stop（执行后暂停），并且本地的系统调用无法暂停，那么将要拦截的系统调用重定向到另外一个系统调用似乎是一个不错的选择，我们可以通过在syscall-enter-stop时获取到系统调用号，并做出修改，之后将修改应用到本地进程，使其继续执行。之后在syscall-exit-stop时将远程的系统调用结果返回。

采用这种做法的话又有两种方式：

1. 向内核插入一个新的系统调用DoNothing，该系统调用什么也不做，仅仅满足正常系统调用的执行流程。

2. 将系统调用号重定向到一个错误的系统调用号上，比如10000，保证该系统调用号不在系统调用表syscall\_table中，这样依旧会有返回使得被追踪者会停留在syscall-exit-stop上以满足我们的需求。

由于第一种方法可能会涉及到内核的修改和内核重新编译，因此在本场景中我们使用了第二种方法。

## 7.2 对于指针类型的参数如何进行传参

### 7.2.1 系统调用参数的传递和分类

在x86\_64位操作系统上，系统调用寄存器传递参数：

1. 使用RAX传递系统调用号
2. 当系统调用参数小于等于6个时，参数必须按照顺序放在**RDI, RSI, RDX, R10, R8, R9**中。
3. 当系统调用参数大于六个时，全部参数应该依次放在一块连续的内存区域中，同时在寄存器EBX中保存指向该区域的指针。

对于要传递的系统调用参数，可以粗略的将其总结为四类（参见[LINUX SYSTEM CALL TABLE FOR X86\\_64](#)和[Syscall proxying-simulating remote execution\[1\]](#)）：

1. 整数
2. 指向整数的指针
3. 指向缓冲区的指针
4. 指向结构体的指针

### 7.2.2 指针类型参数传递

如果要传递的系统调用参数仅仅是整数，那么要做的事情就非常简单了。但是要传递的参数涉及到指针时，问题就很棘手了，因为本地进程传递给系统调用的指针参数指向是本地的内存。如果直接将指针参数传递到远程服务器，那么当远程服务器使用该指针参数去寻址时，会发生地址访问错误，如果运气坏一点，正好指向了一个标准但包含有害信息的数据，那么就会导致不可预估的事情发生。

### 7.2.3 重定向指针并使用额外的缓冲区

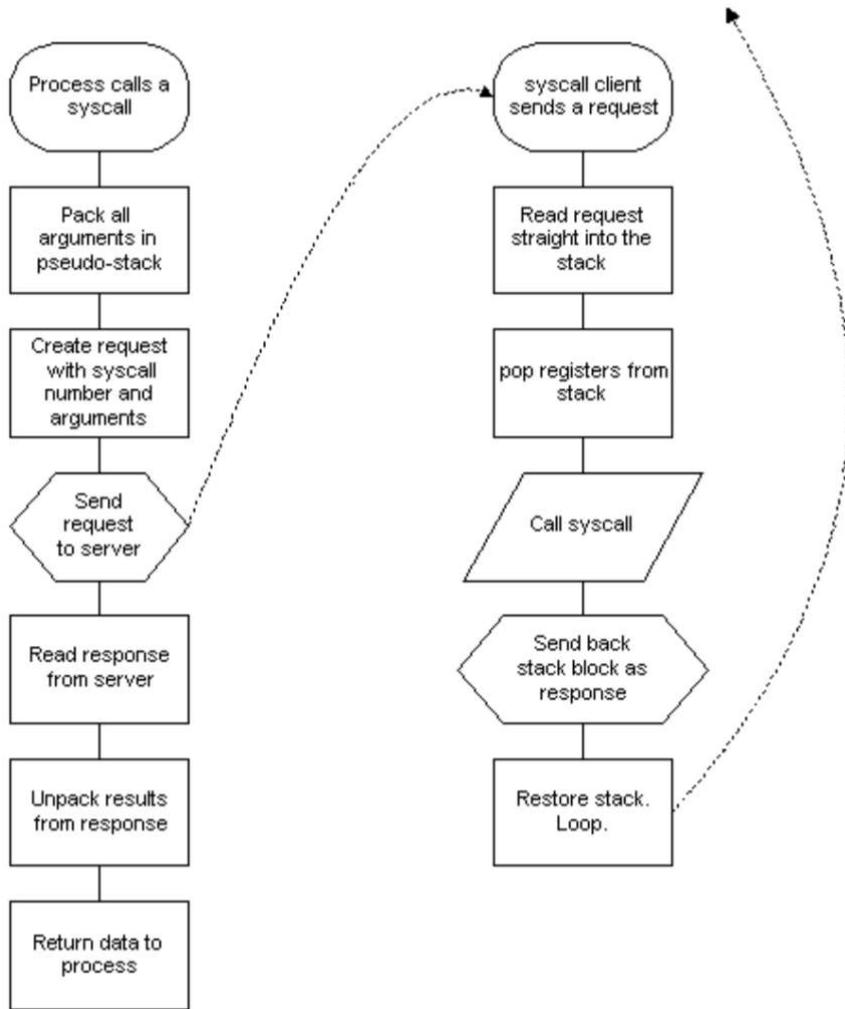
我们由前面可知道在x86\_64中系统调用的参数传递是按照既定的规矩来的，即：使用寄存器传参，参数按照顺序依次压到栈中。

而指针参数的传递会出现问题是因为指针指向的内容存储在本地，那么第一时间我们可以想到：是否可以将指针参数指向的数据也传递给远程服务器。这样做的话，远程服务器就可以获取到本地的数据了。我们可以在客户端请求中增加一个缓冲区来实现该想法。

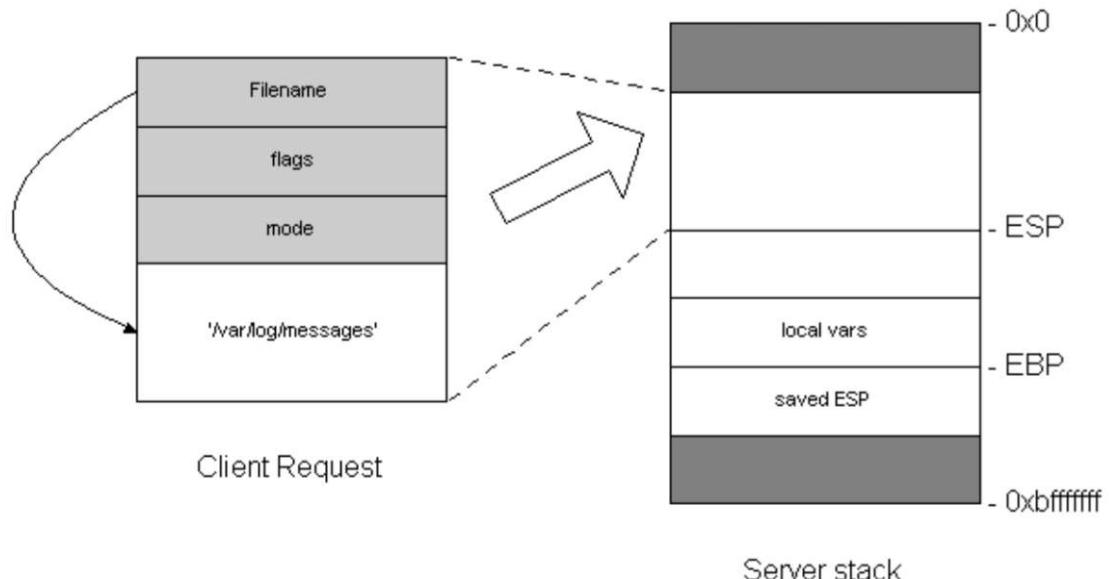
但是该做法还是存在一些问题，即按照指针参数去取值的话，并不能找到该数据。所以还需要做出一些改进：我们可以对指针参数的值进行修改，使其指向传递的具体的数据。但是我们怎么知道接收数据的远程服务器会将数据存储到什么地方以便我们对指针参数进行修改呢？可以借助远程服务器的堆栈和ESP。在客户端将要发送系统调用请求时，远程服务端可以先发送自己的ESP，使客户端根据服务端的ESP对指针参数进行重定向，当服务端接收到客户端请求之后，直接将请求copy到自己的堆栈中，这样服务端就可以对指针参数进行正常寻址了。

客户端和服务端的流程：

## Client host                          Server host



客户端请求结构（以open系统调用举例）：



## 7.3 有选择性远程执行系统调用

一个程序的运行需要许多的系统调用，然而我们并不需要远程执行所有的系统调用，比如程序初始化、销毁时所调用的系统调用，更甚者我们只想实现一部分代码执行远程系统调用。

所以我们考虑之后，决定使用进程间的同步机制来实现这一目的，如System V信号量，来有选择性的选择哪一部分的代码需要在远程执行，而哪一部分的代码不需要。

## 八、分工和协作

### 8.1 参赛队伍

队名：三棵小树苗

指导老师：贾刚勇

校外导师：方应

队长：王国琨

队员：任庆、俞铭辉

### 8.2 协同和分工

王国琨：负责编写文档、debug、调研

任庆：负责基于ptrace的系统调用拦截

俞铭辉：负责远程系统调用请求编组和解组

## 九、提交项目文件简介

### 9.1 项目目录结构

```
1  /*
2  .
3  └── client
4  |   ├── client.c
5  |   ├── example.c
6  |   ├── log.txt
7  |   ├── Makefile
8  |   ├── rsc_client.c
9  |   ├── rsc_semaphore.c
10 |   ├── system_v_yaoshi.txt
11 |   └── test
12 |       └── test.c
13 └── include
14     ├── rsc_client.h
15     ├── rsc.h
16     ├── rsc_semaphore.h
17     └── rsc_server.h
18 └── server
19     ├── ce_w.txt
20     ├── cs_r.txt
21     └── log_e.txt
```

```
22 |   |   Makefile
23 |   |   rsc_server.c
24 |   |   server.c
25 |   |   syscall_table.txt
26 |
27 |
28 4 directories, 20 files
29
30
31 */
```

## 9.2 项目文件简介

### 9.2.1 include

该文件夹包含项目运行所需要的头文件，主要是：

1. rsc\_client.h: client端框架服务运行所需要的一些函数声明。
2. rsc.h: 基础的结构体定义和常量定义
3. rsc\_server.h: server端框架服务运行所需要的一些函数声明
4. rsc\_semaphore.h: 用于client端的进程同步操作所使用的System V信号量操作抽象

### 9.2.2 client

1. client.c: client端框架服务
2. example.c: 测试用例代码
3. Makefile: Makefile
4. rsc\_client.c: rsc\_client.h中声明的函数的实现
5. rsc\_semaphore.c: rsc\_semaphore.h中声明的函数的实现
6. system\_v\_yaoshi.txt: 用于System V信号量机制，被ftok()使用作为信号量Key返回
7. test.c: 如果程序突然退出，那么信号量不会主动销毁，需要调用./test销毁信号量
8. log.txt: 用于debug导出错误信息

### 9.2.2 server

1. server.c: server端框架服务
2. Makefile: Makefile
3. rsc\_server.c: rsc\_server.h中声明的函数的实现
4. ce\_r.txt: 测试用例文件，供本地端读取字符串
5. log\_e.txt: 用于debug导出错误信息

## 9.3 项目运行

### 9.3.1 server端

进入server文件夹，运行makefile，编译server可执行文件

```
1 cd server
2 make server
```

运行server端服务，IP地址和端口作为参数传递进去

```
1 | ./server 127.0.0.1 9999 2>log_e.txt
```

### 9.3.2 client端

进入client文件夹，运行make client 和make example，编译client可执行文件和测试用例

```
1 | cd server  
2 | make client  
3 | make example
```

运行client端服务，IP地址、端口、测试用例作为参数传递进去

```
1 | ./client 127.0.0.1 9999 ./example 2>log.txt
```

### 9.3.3 演示结果

1. 编译并运行server端的框架服务，指定IP为 127.0.0.1 端口为9999

```
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ ls  
cs_r.txt log_e.txt Makefile rsc_server.c rsc_server.o server server.o  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ make clean  
rm server rsc_server.o server.o  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ ls  
cs_r.txt log_e.txt Makefile rsc_server.c server.c  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ make server  
gcc -c rsc_server.c -o rsc_server.o  
gcc -c server.c -o server.o  
gcc rsc_server.o server.o -o server  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ ./server 127.0.0.1 9999
```

2. 编译client端的框架服务和测试代码

```
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ ls  
client.c example.c log.txt Makefile rsc_client.c rsc_semaphore.c system_v_yaoshi.txt test test.c  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ make example  
gcc -c rsc_semaphore.o -o rsc_semaphore.o  
gcc example.c -o example.o  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ make client  
gcc -c rsc_client.c -o rsc_client.o  
gcc -c client.c -o client.o  
gcc rsc_client.o rsc_semaphore.o -o client  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ ls  
client client.o example example.o log.txt Makefile rsc_client.c rsc_semaphore.c rsc_semaphore.o system_v_yaoshi.txt test test.c  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ ls  
client client.o example example.o log.txt Makefile rsc_client.c rsc_client.o rsc_semaphore.c rsc_semaphore.o system_v_yaoshi.txt test test.c  
huomax@huomax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$
```

3. 测试代码如下：

```
1 | #include <stdio.h>  
2 | #include <string.h>  
3 | #include "../include/rsc_semaphore.h"  
4 |  
5 | int main() {  
6 |     int sem_id;  
7 |     sem_id = SemaphoreGet();  
8 |     SemaphorePost(sem_id, ATTACH, 0);           // 通知 tracer 准备开始追  
9 |     踪  
10 |    SemaphoreWait(sem_id, TARGET, 0);          // 阻塞等待 tracer 开始追  
11 |    踪
```

```

10
11 // // 测试 sys_openat, sys_write, sys_close能否远程执行
12 // FILE *fp = fopen("ce_w.txt", "w"); // sys_open,
13 // 服务端打开 ce_w.txt文件
14 // char * buffer = "huamax is a big shuaige!";
15 // fwrite(buffer, sizeof(char), strlen(buffer), fp); // sys_read,
16 // 服务端将字符串写入到 ce_w.txt文件
17 // fclose(fp); // sys_close, 服务端关闭 ce_w.txt文件
18
19 // 测试 sys_openat, sys_read, sys_write, sys_close能否远程执行
20 FILE *fp_r = fopen("ce_r.txt", "r"); // sys_open, 服务端打开
21 // ce_r.txt文件
22 char buffer_r[100];
23 fread(buffer_r, sizeof(char), 100, fp_r); // sys_read, 服务端从
24 // ce_r.txt文件中读取100个字节数据
25 // fclose(fp_r); // sys_close, 服务端关闭打
26 // 开的ce_r.txt文件
27 printf("read data: %s\n", buffer_r); // sys_write, 服务端会显示
28 // 该字符串
29
30 SemaphorePost(sem_id, TARGET_EXIT, 0); // 通知tracer准备结束跟踪
31 SemaphoreWait(sem_id, DETACH, 0); // 阻塞等待追踪进程结束追踪
32 printf("远程系统调用代码区结束了, 我的使命也结束了!\n");
33 return 0;
34 }
```

#### 4. 运行client端服务，指定IP为 127.0.0.1 端口为9999

```

huamax@huamax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ ./test
huamax@huamax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/client$ ./client 127.0.0.1 9999 ./example 2>log.txt
flags=1
[client][ptrace_syscall]: target code area exit
远程系统调用代码区结束了, 我的使命也结束了!
```

5. 运行结果，本地端打开了服务端的 ce\_r.txt文件，并读取100个字节到本地之后再调用 printf()，printf()函数调用 sys\_write系统调用，在服务端显示读取的字符（因为从测试用例进程内存读取数据没注意字节序，故读出的字符顺序会有些混乱）。

```

huamax@huamax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ make server
make: "server"已是最新。
huamax@huamax-virtual-machine:~/helloworld/systemcall-remote/remote-syscall/demo4/server$ ./server 127.0.0.1 9999 2>log_e.txt
read dat data: ha: huomeuomery iry i.hu .hu veruveruai.aai.ax vsx vshuaohuamaxmaxy shh shhuomeuomery iry i.hu .h
```

### 9.3.4 演示视频录制

视频演示了两个测试用例，实现了 sys\_openat, sys\_close, sys\_read, sys\_write四个文件读写操作系统调用的远程执行。

[百度云链接](#)

链接: [https://pan.baidu.com/s/1Sznk-TYYKSoVELocl\\_FQWA](https://pan.baidu.com/s/1Sznk-TYYKSoVELocl_FQWA)

提取码: ud6s

第一个测试用例:

```

1 // 测试 sys_openat, sys_read, sys_write, sys_close能否远程执行
2 FILE *fp_r = fopen("ce_r.txt", "r");           // sys_open, 服务端打开ce_r.txt
文件
3 char buffer_r[100];
4 fread(buffer_r, sizeof(char), 100, fp_r);    // sys_read, 服务端从ce_r.txt文
件中读取100个字节数据
5 fclose(fp_r);                                // sys_close, 服务端关闭打开的
ce_r.txt文件
6 printf("read data: %s\n", buffer_r);        // sys_write, 服务端会显示该字符
串

```

1. 该用例会在服务端打开 *cs\_r.txt* 文件,
2. 然后本地端调用 *sys\_read* 从服务端读取100个字节数据返回本地端,
3. 之后本地端调用 *sys\_close* 请求关闭 *cs\_r.txt* 文件,
4. 最后为了显示读取的字符串, 本地端调用 *printf()* 打印读取的字符串, *printf()* 的底层实现是 *sys\_write*, 又被发送给服务端执行, 最后在服务端界面显示刚刚读取的字符串。

第二个测试用例:

```

1 // // 测试 sys_openat, sys_write, sys_close能否远程执行
2 FILE *fp = fopen("ce_w.txt", "w");           // sys_open, 服务端打开
ce_w.txt文件
3 char * buffer = "huamax is a big shuaige!";
4 fwrite(buffer, sizeof(char), strlen(buffer), fp); // sys_read, 服务端将字
符串写入到 ce_w.txt文件
5 fclose(fp);

```

1. 该用例会在服务端打开 *ce\_w.txt* 文件, 由于服务端并没有这个文件, 故服务端会创建这个文件
2. 然后本地端调用 *sys\_write* 将 *buffer* 指向的数据 "huamax is a big shuaige!" 写到服务端的 *ce\_w.txt* 文件中
3. 之后本地端调用 *sys\_close* 请求关闭 *ce\_w.txt* 文件,
4. 可以在服务端的目录中查看, 会发现多出了 *ce\_w.txt* 文件, 其中存放在本地端请求写入的数据 (字节序问题还未解决, 故顺序有些错乱)

## 十、比赛收获

王国琨:

这次比赛让我收获了许多, 特别是前段时间调研的时候, 几乎每个星期都会修改新的赛题实现目的(当然没联系上校外导师明确赛题方向是我们的问题), 一种技术手段可以用于多种研究领域使我亲身感受到, 特别是与师兄讨论其中对于任务卸载的实现, 让我产生了: "这玩意还可以这样?"的想法, 属实有趣。而在代码编写过程中更是充分锻炼了我们的动手能力以及实践能力。总之, 这个课题很有趣, 我们相信前面还有更有趣的东西在等着我们。

任庆:

这次的比赛收获了很多, 以前写的很多程序都是用面向对象的语言来编写的, 基本上不会设计到底层的一些内容, 这次要实现一个远程系统调用, 学习了一些计算机网络关于TCP传输的内容, 以及实际上linux是如何用程序实现套接字, 还重新认识了Linux的系统调用过程, 明白了如何用ptrace去拦截系统调用的大致原理, 相信这次的比赛会对以后的工作都有一定的影响。

俞铭辉：

这次的比赛令我收获颇丰，主要对于动态链接库的了解更为深入了，还有关于glibc的概念和实现有了较为深刻的理解，并且学会了ptrace工具并且利用它拦截系统调用以及它的大致原理，还加深了linux系统调用过程的理解。在通信协议方面，既了解了序列化和反序列化工具，也了解了如何对消息进行封装等不同的实现方法，总的来说，拓宽了知识面的同时还增长了知识。相信对未来大有裨益。