COMPASS CTF Tutorial 1: Introduction to Capture the Flag

COMPASS CTF 教程【1】：夺旗赛（Capture the Flag）介绍

30016794 Zhao, Li (Research Assistant)

COMPuter And System Security Lab, Computer Science and Technology Department, College of Engineering (CE), SUSTech University.
南方科技大学 工学院 计算机科学与技术系 计算机与系统安全实验室

2023 年 8 月 1 日

CTF（Capture The Flag）中文一般译作夺旗赛，在网络安全领域中指的是网络安全技术人员之间进行技术竞技的一种比赛形式。CTF 起源于 1996 年 DEFCON 全球黑客大会，以代替之前黑客们通过互相发起真实攻击进行技术比拼的方式。发展至今，已经成为全球范围网络安全圈流行的竞赛形式，2013 年全球举办了超过五十场国际性 CTF 赛事。而 DEFCON 作为 CTF 赛制的发源地，DEFCON CTF 也成为了目前全球最高技术水平和影响力的 CTF 竞赛，类似于 CTF 赛场中的“世界杯”。[1]

CTF 为团队赛，通常以三人为限，要想在比赛中取得胜利，就要求团队中每个人在各种类别的题目中至少精通一类，三人优势互补，取得团队的胜利。同时，准备和参与 CTF 比赛是一种有效将计算机科学的离散面、聚焦于计算机安全领域的方法。

赛事介绍

CTF 是一种流行的信息安全竞赛形式，其英文名可直译为“夺得 Flag”，也可意译为“夺旗赛”。其大致流程是，参赛团队之间通过进行攻防对抗、程序分析等形式，率先从主办方给出的比赛环境中得到一串具有一定格式的字符串或其他内容，并将其提交给主办方，从而夺得分数。为了方便称呼，我们把这样的内容称之为“Flag”。

赛事介绍

CTF 竞赛模式具体分为以下三类：

- 解题模式（Jeopardy）
  在解题模式 CTF 赛制中，参赛队伍可以通过互联网或者现场网络参与，这种模式的 CTF 竞赛与 ACM 编程竞赛、信息学奥赛比较类似，以解决网络安全技术挑战题目的分值和时间来排名，通常用于在线选拔赛。题目主要包含逆向、漏洞挖掘与利用、Web 渗透、密码、取证、隐写、安全编程等类别。
- 攻防模式（Attack-Defense）
  在攻防模式 CTF 赛制中，参赛队伍在网络空间互相进行攻击和防守，挖掘网络服务漏洞并攻击对手服务来得分，修补自身服务漏洞进行防御来避免丢分。攻防模式 CTF 赛制可以实时通过得分反映出比赛情况，最终也以得分直接分出胜负，是一种竞争激烈，具有很强观赏性和高度透明性的网络安全赛制。在这种赛制中，不仅仅是比参赛队员的智力和技术，也比体力（因为比赛一般都会持续 48 小时及以上），同时也比团队之间的分工配合与合作。

赛事介绍

- 混合模式（Mix）
  结合了解题模式与攻防模式的 CTF 赛制，比如参赛队伍通过解题可以获取一些初始分数，然后通过攻防对抗进行得分增减的零和游戏，最终以得分高低分出胜负。采用混合模式 CTF 赛制的典型代表如 iCTF 国际 CTF 竞赛。

题目类别

- Reverse
  - 题目涉及到软件逆向、破解技术等，要求有较强的反汇编、反编译功底。主要考查参赛选手的逆向分析能力。
  - 所需知识：汇编语言、加密与解密、常见反编译工具
- Pwn
  - Pwn 在黑客俚语中代表着攻破，获取权限，在 CTF 比赛中它代表着溢出类的题目，其中常见类型溢出漏洞有整数溢出、栈溢出、堆溢出等。主要考查参赛选手对漏洞的利用能力。
  - 所需知识：C，OD+IDA，数据结构，操作系统

题目类别

- Web
  - Web 是 CTF 的主要题型，题目涉及到许多常见的 Web 漏洞，如 XSS、文件包含、代码执行、上传漏洞、SQL 注入等。也有一些简单的关于网络基础知识的考察，如返回包、TCP/IP、数据包内容和构造。可以说题目环境比较接近真实环境。
  - 所需知识：PHP、Python、TCP/IP、SQL
- Crypto
  - 题目考察各种加解密技术，包括古典加密技术、现代加密技术甚至出题者自创加密技术，以及一些常见编码解码，主要考查参赛选手密码学相关知识点。通常也会和其他题目相结合。
  - 所需知识：矩阵、数论、密码学

题目类别

- Misc
  - Misc 即安全杂项，题目涉及隐写术、流量分析、电子取证、人肉搜索、数据分析、大数据统计等，覆盖面比较广，主要考查参赛选手的各种基础综合知识。
  - 所需知识：常见隐写术工具、Wireshark 等流量审查工具、编码知识
- Mobile
  - 主要分为 Android 和 iOS 两个平台，以 Android 逆向为主，破解 APK 并提交正确答案。
  - 所需知识：Java，Android 开发，常见工具

线下赛 AWD 模式

Attack With Defence，简而言之就是你既是一个 hacker，又是一个 manager。
比赛形式：一般就是一个 ssh 对应一个服务，可能是 web 也可能是 pwn，然后 flag 五分钟一轮，各队一般都有自己的初始分数，flag 被拿会被拿走 flag 的队伍均分，主办方会对每个队伍的服务进行 check，check 不过就扣分，扣除的分值由服务 check 正常的队伍均分。

原始页面见参考资料[2]。

在提问之前

在你准备要透过电子邮件、新闻群组或者聊天室提出技术问题前，请先做到以下事情：

1. 尝试在你准备提问的论坛的旧文章中搜寻答案。
2. 尝试上网搜寻来找到答案。
3. 尝试阅读手册来找到答案。
4. 尝试阅读常见问题文件（FAQ）来找到答案。
5. 尝试自己检查或试验来找到答案
6. 向你身边的强者朋友打听来找到答案。
7. 如果你是程式开发者，请尝试阅读原始码来找到答案

当你提出问题的时候，请先表明你已经做了上述的努力；这将有助于树立你并不是一个不劳而获且浪费别人的时间的提问者。如果你能一并表达在做了上述努力的过程中所学到的东西会更好，因为我们更乐于回答那些表现出能从答案中学习的人的问题。

运用某些策略，比如先用 Google 搜寻你所遇到的各种错误讯息（既搜寻 Google 论坛，也搜寻网页），这样很可能直接就找到了能解决问题的文件或邮件列表线索。即使没有结果，在邮件列表或新闻组寻求帮助时加上一句 我在 Google 中搜过下列句子但没有找到什么有用的东西 也是件好事，即使它只是表明了搜寻引擎不能提供哪些帮助。这么做（加上搜寻过的字串）也让遇到相似问题的其他人能被搜寻引擎引导到你的提问来。

别着急，不要指望几秒钟的 Google 搜寻就能解决一个复杂的问题。在向专家求助之前，再阅读一下常见问题文件（FAQ）、放轻松、坐舒服一些，再花点时间思考一下这个问题。相信我们，他们能从你的提问看出你做了多少阅读与思考，如果你是有备而来，将更有可能得到解答。不要将所有问题一股脑抛出，只因你的第一次搜寻没有找到答案（或者找到太多答案）。
准备好你的问题，再将问题仔细的思考过一遍，因为草率的发问只能得到草率的回答，或者根本得不到任何答案。越是能表现出在寻求帮助前你为解决问题所付出的努力，你越有可能得到实质性的帮助。

小心别问错了问题。如果你的问题基于错误的假设，某个普通骇客（J. Random Hacker）多半会一边在心里想着蠢问题…，一边用无意义的字面解释来答覆你，希望着你会从问题的回答（而非你想得到的答案）中汲取教训。
绝不要自以为够格得到答案，你没有；你并没有。毕竟你没有为这种服务支付任何报酬。你将会是自己去挣到一个答案，靠提出有内涵的、有趣的、有思维激励作用的问题　一个有潜力能贡献社群经验的问题，而不仅仅是被动的从他人处索取知识。另一方面，表明你愿意在找答案的过程中做点什么是一个非常好的开始。谁能给点提示？、我的这个例子里缺了什么？以及我应该检查什么地方比请把我需要的确切的过程贴出来更容易得到答覆。因为你表现出只要有人能指出一个正确方向，你就有完成它的能力和决心。

当你提问时

1. 慎选提问的论坛
2. Stack Overflow
3. 网站和 IRC 论坛
4. 第二步，使用专案邮件列表
5. 使用有意义且描述明确的标题
6. 使问题容易回覆
7. 用清晰、正确、精准并合乎文法的语句
8. 使用易于读取且标准的文件格式发送问题
9. 精确的描述问题并言之有物

当你提问时

1. 话不在多而在精
2. 别动辄声称找到 Bug
3. 别用低声下气取代你真正该做的事
4. 描述问题症状而非猜测
5. 按发生时间先后列出问题症状
6. 描述目标而不是过程
7. 别要求使用私人电邮回覆

当你提问时

1. 清楚明确的表达你的问题以及需求
2. 询问有关程式码的问题时
3. 别把自己家庭作业的问题贴上来
4. 去掉无意义的提问句
5. 即使你很急也不要在标题写紧急
6. 礼多人不怪，而且有时还很有帮助
7. 问题解决后，加个简短的补充说明

如何解读答案

RTFM 和 STFW：如何知道你已完全搞砸了

有一个古老而神圣的传统：如果你收到 RTFM（Read The Fucking Manual）的回应，回答者认为你应该去读那该死的手册。当然，基本上他是对的，你应该去读一读。RTFM 有一个年轻的亲戚。如果你收到 STFW（Search The Fucking Web）的回应，回答者认为你应该到该死的网路上搜寻过了。那人多半也是对的，去搜寻一下吧。（更温和一点的说法是 Google 是你的朋友！）

在论坛，你也可能被要求去爬爬论坛的旧文。事实上，有人甚至可能热心地为你提供以前解决此问题的讨论串。但不要依赖这种关照，提问前应该先搜寻一下旧文。

通常，用这两句之一回答你的人会给你一份包含你需要内容的手册或者一个网址，而且他们打这些字的时候也正在读着。这些答覆意味着回答者认为
你需要的资讯非常容易获得；* 你自己去搜寻这些资讯比灌给你能让你学到更多。
你不应该因此不爽；依照骇客的标准，他已经表示了对你一定程度的关注，而没有对你的要求视而不见。你应该对他祖母般的慈祥表示感谢。

如果还是搞不懂

如果你看不懂回应，别立刻要求对方解释。像你以前试着自己解决问题时那样（利用手册，FAQ，网路，身边的高手），先试着去搞懂他的回应。如果你真的需要对方解释，记得表现出你已经从中学到了点什么。

比方说，如果我回答你：看来似乎是 zentry 卡住了；你应该先清除它。，然后，这是一个很糟的后续问题回应：zentry 是什么？好的问法应该是这样：哦　我看过说明了但是只有 -z 和 -p 两个参数中提到了 zentries，而且还都没有清楚的解释如何清除它。你是指这两个中的哪一个吗？还是我看漏了什么？

处理无礼的回应

很多骇客圈子中看似无礼的行为并不是存心冒犯。相反，它是直接了当，一针见血式的交流风格，这种风格更注重解决问题，而不是使人感觉舒服而却模模糊糊。

如果你觉得被冒犯了，试着平静地反应。如果有人真的做了出格的事，邮件列表、新闻群组或论坛中的前辈多半会招呼他。如果这没有发生而你却发火了，那么你发火对象的言语可能在骇客社群中看起来是正常的，而你将被视为有错的一方，这将伤害到你获取讯息或帮助的机会。

另一方面，你偶而真的会碰到无礼和无聊的言行。与上述相反，对真正的冒犯者狠狠地打击，用犀利的语言将其驳得体无完肤都是可以接受的。然而，在行事之前一定要非常非常的有根据。纠正无礼的言论与开始一场毫无意义的口水战仅一线之隔，骇客们自己莽撞地越线的情况并不鲜见。如果你是新手或外人，避开这种莽撞的机会并不高。如果你想得到的是讯息而不是消磨时光，这时最好不要把手放在键盘上以免冒险。

不该问的问题

以下是几个经典蠢问题，以及骇客没回答时心中所想的：

问题：我能在哪找到 X 程式或 X 资源？
问题：我怎样用 X 做 Y？
问题：如何设定我的 shell 提示？
问题：我可以用 Bass-o-matic 文件转换工具将 AcmeCorp 档案转换为 TeX 格式吗？
问题：我的程式/设定/SQL 语句没有用
问题：我的 Windows 电脑有问题，你能帮我吗？
问题：我的程式不会动了，我认为系统工具 X 有问题
问题：我在安装 Linux（或者 X）时有问题，你能帮我吗？
问题：我怎么才能破解 root 帐号/窃取 OP 特权/读别人的邮件呢？

问题：我能在哪找到 X 程式或 X 资源？
回答：就在我找到它的地方啊，白痴 – 搜寻引擎的那一头。天哪！难道还有人不会用 Google 吗？
问题：我怎样用 X 做 Y？
回答：如果你想解决的是 Y，提问时别给出可能并不恰当的方法。这种问题说明提问者不但对 X 完全无知，也对 Y 要解决的问题糊涂，还被特定形势禁锢了思维。最好忽略这种人，等他们把问题搞清楚了再说。
问题：如何设定我的 shell 提示？？
回答：如果你有足够的智慧提这个问题，你也该有足够的智慧去 RTFM，然后自己去找出来。

问题：我可以用 Bass-o-matic 文件转换工具将 AcmeCorp 档案转换为 TeX 格式吗？
回答：试试看就知道了。如果你试过，你既知道了答案，就不用浪费我的时间了。
问题：我的程式/设定/SQL 语句没有用
回答：这不算是问题吧，我对要我问你二十个问题才找得出你真正问题的问题没兴趣 – 我有更有意思的事要做呢。在看到这类问题的时候，我的反应通常不外如下三种

- 你还有什么要补充的吗？
- 真糟糕，希望你能搞定。
- 这关我屁事？

问题：我的 Windows 电脑有问题，你能帮我吗？
回答：能啊，扔掉微软的垃圾，换个像 Linux 或 BSD 的开放原始码作业系统吧。
注意：如果程式有官方版 Windows 或者与 Windows 有互动（如 Samba），你可以问与 Windows 相关的问题，只是别对问题是由 Windows 作业系统而不是程式本身造成的回覆感到惊讶，因为 Windows 一般来说实在太烂，这种说法通常都是对的。
问题：我的程式不会动了，我认为系统工具 X 有问题
回答：你完全有可能是第一个注意到被成千上万使用者反覆使用的系统呼叫与函式库档案有明显缺陷的人，更有可能的是你完全没有根据。不同凡响的说法需要不同凡响的证据，当你这样声称时，你必须有清楚而详尽的缺陷说明文件作后盾。

问题：我在安装 Linux（或者 X）时有问题，你能帮我吗？
回答：不能，我只有亲自在你的电脑上动手才能找到毛病。还是去找你当地的 Linux 使用群组者寻求实际的指导吧（你能在这儿[3]找到使用者群组的清单）。
注意：如果安装问题与某 Linux 的发行版有关，在它的邮件列表、论坛或本地使用者群组中提问也许是恰当的。此时，应描述问题的准确细节。在此之前，先用 Linux 和所有被怀疑的硬体作关键词仔细搜寻。
问题：我怎么才能破解 root 帐号/窃取 OP 特权/读别人的邮件呢？
回答：想要这样做，说明了你是个卑鄙小人；想找个骇客帮你，说明你是个白痴！

常用基础命令

ls 用来显示目标列表
cd [path] 用来切换工作目录
pwd 以绝对路径的方式显示用户当前工作目录
man [command] 查看 Linux 中的指令帮助、配置文件帮助和编程帮助等信息
apropos [whatever] 在一些特定的包含系统命令的简短描述的数据库文件里查找关键字
echo [string] 打印一行文本，参数“-e”可激活转义字符
cat [file] 连接文件并打印到标准输出设备上
less [file] 允许用户向前或向后浏览文字档案的内容
mv [file1] [file2] 用来对文件或目录重新命名，或者将文件从一个目录移到另一个目录中
cp [file1] [file2] 用来将一个或多个源文件或者目录复制到指定的目的文件或目录
rm [file] 可以删除一个目录中的一个或多个文件或目录，也可以将某个目录及其下属的所有文件及其子目录均删除掉[4]

ps 用于报告当前系统的进程状态
top 实时查看系统的整体运行情况
kill 杀死一个进程
ifconfig 查看或设置网络设备
ping 查看网络上的主机是否工作
netstat 显示网络连接、路由表和网络接口信息
nc(netcat) 建立 TCP 和 UDP 连接并监听
su 切换当前用户身份到其他用户身份
touch [file] 创建新的空文件
mkdir [dir] 创建目录
chmod 变更文件或目录的权限
chown 变更某个文件或目录的所有者和所属组
nano / vim / emacs 字符终端的文本编辑器
exit 退出 shell

管道与变量

使用管道

管道命令符 "|" 将一个命令的标准输出作为另一个命令的标准输入

使用变量

var=value 给变量 var 赋值 value

$var, $var 取变量的值

‘cmd‘, $(cmd) 代换标准输出

'string' 非替换字符串

"string" 可替换字符串

Bash 快捷键

Up(Down) 上（下）一条指令
Ctrl + c 终止当前进程
Ctrl + z 挂起当前进程，使用“fg”可唤醒
Ctrl + d 删除光标处的字符
Ctrl + l 清屏
Ctrl + a 移动到命令行首
Ctrl + e 移动到命令行尾
Ctrl + b 按单词后移（向左）
Ctrl + f 按单词前移（向右）
Ctrl + Shift + c 复制
Ctrl + Shift + v 粘贴

根目录结构

由于不同的发行版会有略微的不同，我们这里使用的是基于 Arch 的发行版 Manjaro，以上就是根目录下的内容，我们介绍几个重要的目录：

- /bin、/sbin：链接到 /usr/bin，存放 Linux 一些核心的二进制文件，其包含的命令可在 shell 上运行。
- /boot：操作系统启动时要用到的程序。
- /dev：包含了所有 Linux 系统中使用的外部设备。需要注意的是这里并不是存放外部设备的驱动程序，而是一个访问这些设备的端口。
- /etc：存放系统管理时要用到的各种配置文件和子目录。
- /etc/rc.d：存放 Linux 启动和关闭时要用到的脚本。
- /home：普通用户的主目录。
- /lib、/lib64：链接到 /usr/lib，存放系统及软件需要的动态链接共享库。

根目录结构

- /mnt：这个目录让用户可以临时挂载其他的文件系统。
- /proc：虚拟的目录，是系统内存的映射。可直接访问这个目录来获取系统信息。
- /root：系统管理员的主目录。
- /srv：存放一些服务启动之后需要提取的数据。
- /sys：该目录下安装了一个文件系统 sysfs。该文件系统是内核设备树的一个直观反映。当一个内核对象被创建时，对应的文件和目录也在内核对象子系统中被创建。
- /tmp：公用的临时文件存放目录。
- /usr：应用程序和文件几乎都在这个目录下。
- /usr/src：内核源代码的存放目录。
- /var：存放了很多服务的日志信息。

进程管理

- top
  - 可以实时动态地查看系统的整体运行情况。
- ps
  - 用于报告当前系统的进程状态。可以搭配 kill 指令随时中断、删除不必要的程序。
  - 查看某进程的状态：$ ps -aux | grep [file]，其中返回内容最左边的数字为进程号 (PID)。
- kill
  - 用来删除执行中的程序或工作。
  - 删除进程某 PID 指定的进程：$ kill [PID]

UID 和 GID

Linux 是一个支持多用户的操作系统，每个用户都有 User ID(UID) 和 Group ID(GID)，UID 是对一个用户的单一身份标识，而 GID 则对应多个 UID。知道某个用户的 UID 和 GID 是非常有用的，一些程序可能就需要 UID/GID 来运行。可以使用 id 命令来查看：

```
$ id root
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),19(log)
$ id firmy
uid=1000(firmy) gid=1000(firmy) groups=1000(firmy),3(sys),7(lp),10(wheel),90(network),91(video),93(optical),95(storage),96(scanner),98(power),56(bumblebee)
```

UID 为 0 的 root 用户类似于系统管理员，它具有系统的完全访问权。我自己新建的用户 firmy，其 UID 为 1000，是一个普通用户。GID 的关系存储在 /etc/group 文件中：

```
$ cat /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,firmy
.....
```

所有用户的信息（除了密码）都保存在 /etc/passwd 文件中，而为了安全起见，加密过的用户密码保存在 /etc/shadow 文件中，此文件只有 root 权限可以访问。

```
$ sudo cat /etc/shadow
root:$6$root$wvK.pRXFEH80GYkpiu1tEWYMOueo4tZtq7mYnldiyJBZDMe.mKwt.WI-
Jnehb4bhZchL/93Oe1ok9UwxYf79yR1:17264::::::
firmy:$6$firmy$dhGT.WP91lnpG5/10GfGdj5L1fFV-
SoYlxwYHQn.llc5eKOvr7J8nqqGdVFKykMUSDNxix5Vh8zbXI-
apt0oPd8.:17264:0:99999:7:::
```

由于普通用户的权限比较低，这里使用 sudo 命令可以让普通用户以 root 用户的身份运行某一命令。使用 su 命令则可以切换到一个不同的用户：

```
$ whoami
firmy
$ su root
# whoami
root
```

whoami 用于打印当前有效的用户名称，shell 中普通用户以 $ 开头，root 用户以 # 开头。在输入密码后，我们已经从 firmy 用户转换到 root 用户了。

权限设置

在 Linux 中，文件或目录权限的控制分别以读取、写入、执行 3 种一般权限来区分，另有 3 种特殊权限可供运用。
使用 ls -l [file] 来查看某文件或目录的信息：

```
$ ls -l /
lrwxrwxrwx 1 root root 7 Jun 21 22:44 bin -> usr/bin
drwxr-xr-x 4 root root 4096 Jul 28 08:48 boot
-rw-r–r– 1 root root 18561 Apr 2 22:48 desktopfs-pkgs.txt
```

第一栏从第二个字母开始就是权限字符串，权限表示三个为一组，依次是所有者权限、组权限、其他人权限。每组的顺序均为 rwx，如果有相应权限，则表示成相应字母，如果不具有相应权限，则用 - 表示。

- r：读取权限，数字代号为“4”
- w：写入权限，数字代号为“2”
- x：执行或切换权限，数字代号为“1”

通过第一栏的第一个字母可知，第一行是一个链接文件 (l)，第二行是个目录（d)，第三行是个普通文件（-)。
用户可以使用 chmod 指令去变更文件与目录的权限。权限范围被指定为所有者 (u)、所属组（g)、其他人（o）和所有人（a)。

- -R：递归处理，将指令目录下的所有文件及子目录一并处理；

1	2	3	4	5	6	7	8	9	10
File Type	User Permission			Group Permission			Other Permission		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
d/l/s/p/-/c/b	r	w	e	r	w	e	r	w	e

图: 权限设置标识符

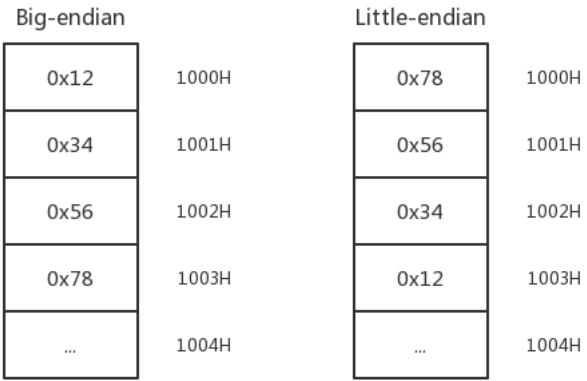
字节序

目前计算机中采用两种字节存储机制：大端（Big-endian）和小端（Little-endian）。

- MSB (Most Significan Bit/Byte)：最重要的位或最重要的字节。
- LSB (Least Significan Bit/Byte)：最不重要的位或最不重要的字节。

Big-endian 规定 MSB 在存储时放在低地址，在传输时放在流的开始；LSB 存储时放在高地址，在传输时放在流的末尾。Little-endian 则相反。常见的 Intel 处理器使用 Little-endian，而 PowerPC 系列处理器则使用 Big-endian，另外 TCP/IP 协议和 Java 虚拟机的字节序也是 Big-endian。

例如十六进制整数 0x12345678 存入以 1000H 开始的内存中：



图：不同字节序的存储示例

我们在内存中实际地看一下，在地址 0xffffd584 处有字符 1234，在地址 0xffffd588 处有字符 5678。

```
gdb-peda$ x/w 0xffffd584
0xffffd584: 0x34333231
gdb-peda$ x/4wb 0xffffd584
0xffffd584: 0x31 0x32 0x33 0x34
gdb-peda$ python print('\x31\x32\x33\x34')
1234
```

```
gdb-peda$ x/w 0xffffd588
0xffffd588: 0x38373635
gdb-peda$ x/4wb 0xffffd588
0xffffd588: 0x35 0x36 0x37 0x38
gdb-peda$ python print('\x35\x36\x37\x38')
5678
```

```
gdb-peda$ x/2w 0xffffd584
0xffffd584: 0x34333231 0x38373635
gdb-peda$ x/8wb 0xffffd584
0xffffd584: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38
gdb-peda$ python print('\x31\x32\x33\x34\x35\x35\x36\x37\x38')
123455678
db-peda$ x/s 0xffffd584
0xffffd584: "12345678"
```

输入输出

- 使用命令的输出作为可执行文件的输入参数
  - $ ./vulnerable ‘your_command_here‘
  - $ ./vulnerable $(your_command_here)
- 使用命令作为输入
  - $ your_command_here | ./vulnerable
- 将命令行输出写入文件
  - $ your_command_here > filename
- 使用文件作为输入
  - $ ./vulnerable < filename

文件描述符

在 Linux 系统中一切皆可以看成是文件，文件又分为：普通文件、目录文件、链接文件和设备文件。文件描述符（file descriptor）是内核管理已被打开的文件所创建的索引，使用一个非负整数来指代被打开的文件。
标准文件描述符如下：

文件描述符	用途	stdio 流
0	标准输入	stdin
1	标准输出	stdout
2	标准错误	stderr

当一个程序使用 fork() 生成一个子进程后，子进程会继承父进程所打开的文件表，此时，父子进程使用同一个文件表，这可能导致一些安全问题。如果使用 vfork()，子进程虽然运行于父进程的空间，但拥有自己的进程表项。

核心转储

当程序运行的过程中异常终止或崩溃，操作系统会将程序当时的内存、寄存器状态、堆栈指针、内存管理信息等记录下来，保存在一个文件中，这种行为就叫做核心转储（Core Dump）。

会产生核心转储的信号

Signal	Action	Comment
SIGQUIT	Core	Quit from keyboard
SIGILL	Core	Illegal Instruction
SIGABRT	Core	Abort signal from abort
SIGSEGV	Core	Invalid memory reference
SIGTRAP	Core	Trace/breakpoint trap

开启核心转储

输入命令 ulimit -c，输出结果为 0，说明默认是关闭的。
输入命令 ulimit -c unlimited 即可在当前终端开启核心转储功能。
如果想让核心转储功能永久开启，可以修改文件 /etc/security/limits.conf，增加一行：

#<domain> <type> <item> <value>
* soft core unlimited

修改转储文件保存路径

通过修改 /proc/sys/kernel/core_uses_pid，可以使生成的核心转储文件名变为 core.[pid] 的模式。

```
# echo 1 > /proc/sys/kernel/core_uses_pid
```

还可以修改 /proc/sys/kernel/core_pattern 来控制生成核心转储文件的保存位置和文件名格式。

```
# echo /tmp/core-%e-%p-%t > /proc/sys/kernel/core_pattern
```

此时生成的文件保存在 /tmp/ 目录下，文件名格式为 core-[filename]-[pid]-[time]。

使用 gdb 调试核心转储文件

gdb [filename] [core file]

例子

```
$ cat core.c
#include <stdio.h>
void main(int argc, char **argv)
char buf[5];
scanf("%s", buf);
$ gcc -m32 -fno-stack-protector core.c
$ ./a.out
AAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

使用 gdb 调试核心转储文件

例子

```
$ file /tmp/core-a.out-12444-1503198911
/tmp/core-a.out-12444-1503198911: ELF 32-bit LSB core file Intel 80386, version 1
(SYSV), SVR4-style, from './a.out', real uid: 1000, effective uid: 1000, real gid: 1000,
effective gid: 1000, execfn: './a.out', platform: 'i686'
$ gdb a.out /tmp/core-a.out-12444-1503198911 -q
Reading symbols from a.out...(no debugging symbols found)...done.
[New LWP 12444]
Core was generated by `./a.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
```

使用 gdb 调试核心转储文件

例子

#0 0x5655559b in main ()
gdb-peda$ info frame
Stack level 0, frame at 0x41414141:
eip = 0x5655559b in main; saved eip = <not saved>
Outermost frame: Cannot access memory at address 0x4141413d
Arglist at 0x41414141, args:
Locals at 0x41414141, Previous frame's sp is 0x41414141
Cannot access memory at address 0x4141413d

调用约定

函数调用约定是对函数调用时如何传递参数的一种约定。关于它的约定有许多种，下面我们分别从内核接口和用户接口介绍 32 位和 64 位 Linux 的调用约定。

内核接口

x86-32 系统调用约定：Linux 系统调用使用寄存器传递参数。eax 为 syscall_number，ebx、ecx、edx、esi、ebp 用于将 6 个参数传递给系统调用。返回值保存在 eax 中。所有其他寄存器（包括 EFLAGS）都保留在 int 0x80 中。

x86-64 系统调用约定：内核接口使用的寄存器有：rdi、rsi、rdx、r10、r8、r9。系统调用通过 syscall 指令完成。除了 rcx、r11 和 rax，其他的寄存器都被保留。系统调用的编号必须在寄存器 rax 中传递。系统调用的参数限制为 6 个，不直接从堆栈上传递任何参数。返回时，rax 中包含了系统调用的结果。而且只有 INTEGER 或者 MEMORY 类型的值才会被传递给内核。

用户接口

x86-32 函数调用约定：参数通过栈进行传递。最后一个参数第一个被放入栈中，直到所有的参数都放置完毕，然后执行 call 指令。这也是 Linux 上 C 语言函数的方式。
x86-64 函数调用约定：x86-64 下通过寄存器传递参数，这样做比通过栈有更高的效率。它避免了内存中参数的存取和额外的指令。根据参数类型的不同，会使用寄存器或传参方式。如果参数的类型是 MEMORY，则在栈上传递参数。如果类型是 INTEGER，则顺序使用 rdi、rsi、rdx、rcx、r8 和 r9。所以如果有多于 6 个的 INTEGER 参数，则后面的参数在栈上传递。

环境变量

环境变量字符串都是 name=value 这样的形式。大多数 name 由大写字母加下划线组成，一般把 name 部分叫做环境变量名，value 部分则是环境变量的值，而且 value 需要以 '\0' 结尾。
环境变量定义了该进程的运行环境。

分类

- 按照生命周期划分
  - 永久环境变量：修改相关配置文件，永久生效。
  - 临时环境变量：使用 export 命令，在当前终端下生效，关闭终端后失效。
- 按照作用域划分
  - 系统环境变量：对该系统中所有用户生效。
  - 用户环境变量：对特定用户生效。

设置方法

在文件 /etc/profile 中添加变量，这种方法对所有用户永久生效。如:

```
# Set our default path
PATH="/usr/local/sbin:/usr/local/bin:/usr/bin"
export PATH
```

添加后执行命令 source /etc/profile 使其生效。

在文件 /.bash_profile 中添加变量，这种方法对当前用户永久生效。其余同上。

直接运行命令 export 定义变量，这种方法只对当前终端临时生效。

常用变量

使用命令 echo 打印变量：

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/lib/jvm/de-
fault/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl
$ echo $HOME
/home/firmy
$ echo $LOGNAME
firmy
$ echo $HOSTNAME
firmy-pc
$ echo $SHELL
/bin/bash
$ echo $LANG
en_US.UTF-8
```

使用命令 env 可以打印出所有环境变量：

```
$ env
COLORFGBG=15;0
COLORTERM=truecolor
...
```

使用命令 set 可以打印出所有本地定义的 shell 变量：

```
$ set
'!'=0
'#'=0
...
```

使用命令 unset 可以清除变量：

```
unset $变量名
```

LD_PRELOAD

该环境变量可以定义在程序运行前优先加载的动态链接库。在 pwn 题目中，我们可能需要一个特定的 libc，这时就可以定义该变量：

LD_PRELOAD=/path/to/libc.so ./binary

一个例子：

```
$ ldd /bin/true
linux-vdso.so.1 => (0x00007fff9a9fe000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1c083d9000)
/lib64/ld-linux-x86-64.so.2 (0x0000557bcce6c000)
$ LD_PRELOAD=/libc.so.6 ldd /bin/true
linux-vdso.so.1 => (0x00007ffee55e9000)
/home/firmy/libc.so.6 (0x00007f4a28cfc000)
/lib64/ld-linux-x86-64.so.2 (0x000055f33bc50000)
```

注意，在加载动态链接库时需要使用 ld.so 进行重定位，通常被符号链接到 /lib64/ld-linux-x86-64.so 中。动态链接库在编译时隐式指定 ld.so 的搜索路径，并写入 ELF Header 的 INTERP 字段中。从其他发行版直接拷贝已编译的 .so 文件可能会引发 ld.so 搜索路径不正确的问题。相似的，在版本依赖高度耦合的发行版中 (如 ArchLinux)，版本相差过大也会引发 ld.so 的运行失败。

本地同版本编译后通常不会出现问题。如果有直接拷贝已编译版本的需要，可以对比 interpreter 确定是否符合要求，但是不保证不会失败。

上面的例子中两个 libc 是这样的：

```
$ file /lib/x86_64-linux-gnu/libc-2.23.so
/lib/x86_64-linux-gnu/libc-2.23.so: ELF 64-bit LSB shared object, x86-64, version 1
(GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=088a6e00a1814622219f346b41e775b8dd46c518, for GNU/Linux 2.6.32,
stripped
$ file  /libc.so.6
/home/firmy/libc.so.6: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=088a6e00a1814622219f346b41e775b8dd46c518, for GNU/Linux 2.6.32,
stripped
```

都是 interpreter /lib64/ld-linux-x86-64.so.2，所以可以替换。

而下面的例子是在 Arch Linux 上使用一个 Ubuntu 的 libc，就会出错：

```
$ ldd /bin/true
linux-vdso.so.1 (0x00007ffc969df000)
libc.so.6 => /usr/lib/libc.so.6 (0x00007f7ddde17000)
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007f7dde3d7000)
$ LD_PRELOAD= /libc.so.6 ldd /bin/true
Illegal instruction (core dumped)
```

```
$ file /usr/lib/libc-2.26.so
/usr/lib/libc-2.26.so: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux),
dynamically linked, interpreter /usr/lib/ld-linux-x86-64.so.2,
BuildID[sha1]=458fd9997a454786f071cfe2beb234542c1e871f, for GNU/Linux 3.2.0,
not stripped
$ file  /libc.so.6
/home/firmy/libc.so.6: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=088a6e00a1814622219f346b41e775b8dd46c518, for GNU/Linux 2.6.32,
stripped
```

一个在 interpreter /usr/lib/ld-linux-x86-64.so.2，**而另一个在** interpreter /lib64/ld-linux-x86-64.so.2。

environ

libc 中定义的全局变量 environ 指向环境变量表。而环境变量表存在于栈上，所以通过 environ 指针的值就可以泄露出栈地址。

```
gdb-peda$ vmmap libc
Start End Perm Name
0x00007ffff7a1c000 0x00007ffff7bcf000 r-xp /usr/lib/libc-2.27.so
0x00007ffff7bcf000 0x00007ffff7dce000 —p /usr/lib/libc-2.27.so
0x00007ffff7dce000 0x00007ffff7dd2000 r–p /usr/lib/libc-2.27.so
0x00007ffff7dd2000 0x00007ffff7dd4000 rw-p /usr/lib/libc-2.27.so
```

environ

```
gdb-peda$ vmmap stack
Start End Perm Name
0x00007ffffffde000 0x00007ffffffff000 rw-p [stack]
gdb-peda$ shell nm -D /usr/lib/libc-2.27.so | grep environ
00000000003b8ee0 V environ
00000000003b8ee0 V _environ
00000000003b8ee0 B __environ
gdb-peda$ x/gx 0x00007ffff7a1c000 + 0x00000000003b8ee0
0x7ffff7dd4ee0 <environ>: 0x00007ffffffde48
```

environ

```
gdb-peda$ x/5gx 0x00007fffffffde48
0x7fffffffde48: 0x00007fffffffe1da 0x00007fffffffe1e9
0x7fffffffde58: 0x00007fffffffe1fd 0x00007fffffffe233
0x7fffffffde68: 0x00007fffffffe25f
gdb-peda$ x/5s 0x00007fffffffe1da
0x7fffffffe1da: "COLORFGBG=15;0"
0x7fffffffe1e9: "COLORTERM=truecolor"
0x7fffffffe1fd: "DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus"
0x7fffffffe233: "DESKTOP_SESSION=/usr/share/xsessions/plasma"
0x7fffffffe25f: "DISPLAY=:0"
```

procfs

procfs 文件系统是 Linux 内核提供的虚拟文件系统，为访问系统内核数据的操作提供接口。之所以说是虚拟文件系统，是因为它不占用存储空间，而只是占用了内存。用户可以通过 procfs 查看有关系统硬件及当前正在运行进程的信息，甚至可以通过修改其中的某些内容来改变内核的运行状态。

/proc/cmdline

在启动时传递给内核的相关参数信息，通常由 lilo 或 grub 等启动管理工具提供：

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.14-x86_64
root=UUID=8e79a67d-af1b-4203-8c1c-3b670f0ec052 rw quiet
resume=UUID=a220ecb1-7fde-4032-87bf-413057e9c06f
```

/proc/cpuinfo

记录 CPU 相关的信息：

```
$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i5-4210H CPU @ 2.90GHz
stepping : 3
...
```

/proc/crypto

已安装的内核所使用的密码算法及算法的详细信息：

```
$ cat /proc/crypto
name : ccm(aes)
driver : ccm_base(ctr(aes-aesni),cbcmac(aes-aesni))
module : ccm
priority : 300
refcnt : 2
selftest : passed
internal : no
...
```

/proc/devices

已加载的所有块设备和字符设备的信息，包含主设备号和设备组（与主设备号对应的设备类型）名:

```
$ cat /proc/devices
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
...
```

/proc/interrupts

X86/X86_64 系统上每个 IRQ 相关的中断号列表，多路处理器平台上每个 CPU 对于每个 I/O 设备均有自己的中断号：

```
$ cat /proc/interrupts
CPU0 CPU1 CPU2 CPU3
0: 15 0 0 0 IR-IO-APIC 2-edge timer
1: 46235 1277 325 156 IR-IO-APIC 1-edge i8042
8: 0 1 0 0 IR-IO-APIC 8-edge rtc0
...
NMI: 0 0 0 0 Non-maskable interrupts
LOC: 7363806 5569019 6138317 5442200 Local timer interrupts
SPU: 0 0 0 0 Spurious interrupts
...
```

/proc/kcore

系统使用的物理内存，以 ELF 核心文件（core file）格式存储：

```
$ sudo file /proc/kcore
/proc/kcore: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from
'BOOT_IMAGE=/boot/vmlinuz-4.14-x86_64
root=UUID=8e79a67d-af1b-4203-8c1c-3b670f0e'
```

/proc/meminfo

系统中关于当前内存的利用状况等的信息:

```
$ cat /proc/meminfo
MemTotal: 12226252 kB
MemFree: 4909444 kB
MemAvailable: 8776048 kB
Buffers: 288236 kB
Cached: 3953616 kB
...
```

/proc/mounts

每个进程自身挂载名称空间中的所有挂载点列表文件的符号链接：

```
$ cat /proc/mounts
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
sys /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0
dev /dev devtmpfs rw,nosuid,relatime,size=6106264k,nr_inodes=1526566,mode=755
0 0
...
```

/proc/modules

当前装入内核的所有模块名称列表，可以由 lsmod 命令使用。其中第一列表示模块名，第二列表示此模块占用内存空间大小，第三列表示此模块有多少实例被装入，第四列表示此模块依赖于其它哪些模块，第五列表示此模块的装载状态：Live（已经装入）、Loading（正在装入）和 Unloading（正在卸载），第六列表示此模块在内核内存（kernel memory）中的偏移量：

```
$ cat /proc/modules
fuse 118784 3 - Live 0xffffffffc0d9b000
ccm 20480 3 - Live 0xffffffffc0d95000
rfcomm 86016 4 - Live 0xffffffffc0d7f000
bnep 24576 2 - Live 0xffffffffc0d78000
...
```

/proc/slabinfo

保存着监视系统中所有活动的 slab 缓存的信息：

```
$ sudo cat /proc/slabinfo
slabinfo - version: 2.1
# name <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> :
tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs>
<num_slabs> <sharedavail>
fuse_request 0 20 400 20 2 : tunables 0 0 0 : slabdata 1 1 0
fuse_inode 1 39 832 39 8 : tunables 0 0 0 : slabdata 1 1 0
drm_i915_gem_request 765 1036 576 28 4 : tunables 0 0 0 : slabdata 37 37 0
...
```

/proc/[pid]

在 /proc 文件系统下，还有一些以数字命名的目录，这些数字是进程的 PID 号，而这些目录是进程目录。目录下的所有文件如下，然后会介绍几个比较重要的：

```
$ cat - &
[1] 1060
$ ls /proc/1060/
attr comm fd maps ns personality smaps syscall
autogroup coredump_filter fdinfo mem numa_maps projid_map smaps_rollup task
auxv cpuset gid_map mountinfo oom_adj root stack timers
cgroup cwd io mounts oom_score sched stat timerslack_ns
clear_refs environ limits mountstats oom_score_adj schedstat statm uid_map
cmdline exe map_files net pagemap setgroups status wchan
```

/proc/[pid]/cmdline

启动当前进程的完整命令:

$ cat /proc/1060/cmdline

cat-

/proc/[pid]/exe

指向启动当前进程的可执行文件的符号链接:

```
$ file /proc/1060/exe
/proc/1060/exe: symbolic link to /usr/bin/cat
```

/proc/[pid]/root

当前进程运行根目录的符号链接:

```
$ file /proc/1060/root
/proc/1060/root: symbolic link to /
```

/proc/[pid]/mem

当前进程所占用的内存空间，由 open、read 和 lseek 等系统调用使用，不能被用户读取。但可通过下面的 /proc/[pid]/maps 查看。

/proc/[pid]/maps

这个文件大概是最常用的，用于显示进程的内存区域映射信息：

```
$ cat /proc/1060/maps
56271b3a5000-56271b3ad000 r-xp 00000000 08:01 24904069 /usr/bin/cat
56271b5ac000-56271b5ad000 r--p 00007000 08:01 24904069 /usr/bin/cat
56271b5ad000-56271b5ae000 rw-p 00008000 08:01 24904069 /usr/bin/cat
56271b864000-56271b885000 rw-p 00000000 00:00 0 [heap]
7fefb66cd000-7fefb6a1e000 r--p 00000000 08:01 24912207
/usr/lib/locale/locale-archive
7fefb6a1e000-7fefb6bd1000 r-xp 00000000 08:01 24905238 /usr/lib/libc-2.27.so
...
```

/proc/[pid]/stack

这个文件表示当前进程的内核调用栈信息，只有在内核编译启用 CONFIG_STACKTRACE 选项，才会生成该文件：

```
$ sudo cat /proc/1060/stack
[<ffffffff8e08fa2e>] do_signal_stop+0xae/0x1f0
[<ffffffff8e090ec1>] get_signal+0x191/0x580
[<ffffffff8e02ae56>] do_signal+0x36/0x610
[<ffffffff8e003669>] exit_to_usermode_loop+0x69/0xa0
[<ffffffff8e0039d1>] do_syscall_64+0xf1/0x100
[<ffffffff8e800081>] entry_SYSCALL_64_after_hwframe+0x3d/0xa2
[<ffffffffffffffff>] 0xffffffffffffffff
```

/proc/[pid]/auxv

该文件包含了传递给进程的解释器信息，即 auxv(AUXiliary Vector)，每一项都是由一个 unsigned long 长度的 ID 加上一个 unsigned long 长度的值构成：

```
$ xxd -E -g8 /proc/1060/auxv
00000000: 0000000000000021 00007ffde574b000 !.........t.....
00000010: 0000000000000010 00000000bfebfbff ................
00000020: 0000000000000006 0000000000001000 ................
00000030: 0000000000000011 0000000000000064 ........d.......
00000040: 0000000000000003 000056271b3a5040 ........@P:.'V..
00000050: 0000000000000004 0000000000000038 ........8.......
```

/proc/[pid]/auxv

每个值具体是做什么的，可以用下面的办法显示出来，对比看一看，更详细的可以查看 /usr/include/elf.h 和 man ld.so：

```
$ LD_SHOW_AUXV=1 cat -
AT_SYSINFO_EHDR: 0x7ffd16be5000
AT_HWCAP: bfebfbff
AT_PAGESZ: 4096
AT_CLKTCK: 100
AT_PHDR: 0x55eb4c59a040
AT_PHENT: 56
AT_PHNUM: 9
```

值得一提的是，AT_SYSINFO_EHDR 所对应的值是一个叫做的 VDSO(Virtual Dynamic Shared Object) 的地址。在 ret2vdso 漏洞利用方法中会用到。

/proc/[pid]/environ

该文件包含了进程的环境变量：

```
$ strings /proc/1060/environ
GS_LIB=/home/firmy/.fonts
KDE_FULL_SESSION=true
VIRTUALENVWRAPPER_WORKON_CD=1
VIRTUALENVWRAPPER_HOOK_DIR=/home/firmy/.virtualenvs
LANG=zh_CN.UTF-8
...
```

/proc/[pid]/fd

该文件包含了进程打开文件的情况:

```
$ ls -al /proc/1060/fd
total 0
dr-x—— 2 firmy firmy 0 6 月 7 23:37 .
dr-xr-xr-x 9 firmy firmy 0 6 月 7 23:37 ..
lrwx—— 1 firmy firmy 64 6 月 7 23:44 0 -> /dev/pts/3
lrwx—— 1 firmy firmy 64 6 月 7 23:44 1 -> /dev/pts/3
lrwx—— 1 firmy firmy 64 6 月 7 23:44 2 -> /dev/pts/3
```

/proc/[pid]/status

该文件包含了进程的状态信息：

```
$ cat /proc/1060/status
Name: cat
Umask: 0022
State: T (stopped)
Tgid: 1060
Ngid: 0
Pid: 1060
...
```

/proc/[pid]/task

一个目录，包含当前进程的每一个线程的相关信息，每个线程的信息分别放在一个由线程号（tid）命名的目录中：

```
$ ls /proc/1060/task/
1060
$ ls /proc/1060/task/1060/
attr clear_refs cwd fdinfo maps net oom_score projid_map setgroups stat uid_map
auxv cmdline environ gid_map mem ns oom_score_adj root smaps statm wchan
cgroup comm exe io mountinfo numa_maps pagemap sched smaps_rollup status
children cpuset fd limits mounts oom_adj personality schedstat stack syscall
```

/proc/[pid]/syscall

该文件包含了进程正在执行的系统调用：

```
$ sudo cat /proc/1060/syscall
0 0x0 0x7fefb6fdd000 0x20000 0x22 0xffffffff 0x0 0x7ffde5677d48 0x7fefb6b07901
```

第一个值是系统调用号，后面跟着是六个参数，最后两个值分别是堆栈指针和指令计数器的值。

参考文献

[1] https://firmianay.gitbook.io/ctf-all-in-one/1_basic/1.1_ctf.

[2] Ryanhanwu. Ryanhanwu/how-to-ask-questions-the-smart-way: 本文原文由知名 hacker Eric S. Raymond 所撰写，教你如何正确的提出技术问题并获得你满意的答案。[EB/OL]. https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way.

[3] https://www.meetup.com/topics/linux/.

[4] https://firmianay.gitbook.io/ctf-all-in-one/1_basic/1.3_linux_basic.