POSIX 文件能力：分配根用户的能力

如何安全地将根特权交给更多用户

Serge Hallyn
2007 年 11 月 12 日发布

一些程序需要以非特权用户的身份执行特权操作。例如，passwd 程序经常对 /etc/passwd 和 UNIX® 系统上，这种控制是通过设置限制文件上的 setuid 位实现的。这个位告诉系统，在运行 是谁，都应该把它看作属于拥有这个文件的用户（通常是根用户）。因为用户不能编写 passwd 操作有严格限制，所以这个设置是安全的。更复杂的程序使用保存的 uid 在根用户和非根用

POSIX 能力将根特权划分成更小的特权，所以可以只用根用户特权的一个子集来运行任务。文 样的特权，这大大简化了能力的使用。在 Linux 中已经可以使用 POSIX 能力了。与将用户切换 处：

- 可以将能力从有效集（effective set）中删除，但是保留在允许集（permitted set）中，从 而防止滥用能力。
- 可以从允许集中删除所有不需要的能力，这样就无法恢复这些能力。坦率地说，大多数能力 是危险的，可能被滥用，减少攻击者可以利用的能力有助于保护系统。
- 在对常规的可执行文件执行 exec(3) 之后，所有能力都会丢失。（细节比较复杂，而且这种 情况可能不久就会改变。本文后面会进一步解释这个问题。）

本文讲解程序如何使用 POSIX 能力，如何确定一个程序需要哪些能力，以及如何为程序分配这

进程能力

多年以来，POSIX 能力只能分配给进程，而不能分配给文件。因此，程序必须由根用户启动（setuid 位），然后才能放弃某些根特权，同时保留其他特权。另外，放弃能力的操作次序也非

2. 程序将它的 userid 改为非根用户。
3. 程序构造所需的能力集并设置为活动集。

进程有三个能力集：*允许（permitted，P）*、*可继承（inheritable，I）* 和 *有效（effective，E）*程复制能力集。当一个进程执行一个新程序时，根据公式计算新的能力集（稍后讨论这些公式

有效集 中的能力是进程当前可以使用的。有效集必须是 *允许集* 的子集。只要有效集不超过允许改有效集的内容。*可继承集* 只用于在执行 `exec()` 之后计算新的能力集。

清单 1 给出三个公式，它们表示在文件执行之后根据 POSIX 草案计算出的新能力集（参见 参考的链接）。

清单 1. 在执行 exec() 之后计算新能力集的公式

```
1  pI' = pI
2  pP' = fP | (fI & pI)
3  pE' = pP' & fE
```

以 ' 结尾的值表示新计算出的值。 f头的值表示进程能力。以 f 开头的值表示文件能力。

可继承集按原样从父进程继承，没有任何修改，所以进程一旦从可继承集中删除一个能力，就面对 `SECURE_NOROOT` 的讨论）。 新的允许集是文件的允许集与文件和进程的可继承集的交集合允许集和文件有效集的交集。从技术上说，在 Linux 中，`fE` 不是一个集，而是一个布尔值。如为 `pP'`。如果是 false，`pE'` 就是空的。

如果进程要在执行一个文件之后保留任何能力，那么这些能力必须被包含在文件的允许集或可时期内没有实现文件能力，所以这是一个难以实施的限制。为了解决这个问题，实现了“安全组成：

- 如果没有设置 `SECURE_NOROOT`，那么当进程执行文件时，就按照完全填充的文件能力集计算
  - 如果进程的真实 uid 或有效 uid 是 0（根用户），或者文件是 setuid root，那么文件的
  - 如果进程的有效 uid 是根用户，或者文件是 setuid root，那么文件有效集就是满的。
- 如果没有设置 `SECURE_NO_SETUID_FIXUP`，那么当进程将它的真实或有效 uid 切换到 0 或切力集：

- 如果进程将它的有效 uid 从 0 切换到非 0，那么它的有效能力集被清空。

内容

这套规则让进程可以根据根用户或者通过运行 setuid root 文件拥有能力。但是，SECURE_NO_S
之后保留任何能力。但是，如果没有设置 SECURE_NOROOT，那么一个已经放弃一些能力的根进
复它的能力。所以为了能够使用能力并保证系统安全，根进程必须能够不可逆转地将它的 uid

通过使用 prctl(3)，进程可以请求在下一次调用 setuid(2) 时保留它的能力。这意味着进程可

- 通过根用户身份或者执行 setuid root 二进制文件，作为根进程启动。
- 通过调用 prctl(2) 设置 PR_SET_KEEPCAPS，这请求系统在调用 setuid(2) 时保留它的能
- 调用 setuid(2) 或相关的系统调用来修改 userid。
- 调用 cap_set_proc(3) 来删除能力。

现在，进程可以一直用根特权的一个子集运行。如果攻击者突破了这个程序，他也只能使用有
cap_set_proc(3)，也只能使用
的能力。另外，如果攻击者迫使这个程序执行另一个
作为非特权用户执行这个文件。

清单 2 中的 exec_with_caps()
缩减代码的能力，setuid root 程序可以通过它作为指
数，执行时的能力集由一个字符

清单 2. 用缩减的能力执行代码

```
#include <sys/prctl.h>
#include <sys/capability.h>
#include <sys/types.h>
#include <stdio.h>

int printmycaps(void *d)
{
    cap_t cap = cap_get_proc();
    printf("Running with uid %d\n", getuid());
    printf("Running with capabilities: %s\n", cap_to_text(cap, NULL));
    cap_free(cap);
    return 0;
}

int exec_with_caps(int newuid, char *capstr, int (*f)(void *data), void *data
{
    int ret;
    cap_t newcaps;

    ret = prctl(PR_SET_KEEPCAPS, 1);
    if (ret) {
        perror("prctl");
        return -1;
```

```
    }
    ret = setresuid(newuid, newuid, newuid);
    if (ret) {
```



```
    newcaps = cap_from_text(capstr);
    ret = cap_set_proc(newcaps);
    if (ret) {
        perror("cap_set_proc");
        return -1;
    }
    cap_free(newcaps);
    f(data);
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("Usage: %s <capability_list>\n",
            argv[0]);
        return 1;
    }
    return exec_with_caps(1000, argv[1], printmycaps, NULL);
}
```

为了测试这个函数，将代码复制到一个文件中并保存为 execwithcaps.c，编译并作为根用户运

```
gcc -o execwithcaps execwithcaps.c -lcap
./execwithcaps cap_sys_admin=eip
```

文件能力

文件能力特性当前是在 -mm 内核树中实现的，有望在 2.6.24 版中被包含在主线内核中。可以利
序。例如，ping 程序需要 CAP_NET_RAW。因此，它一直是一个 setuid root 程序。有了文件能力
的特权数量：

```
chmod u-s /bin/ping
setfcaps -c cap_net_admin=p -e /bin/ping
```

这需要从 GoogleCode 获得 libcap 库和相关程序的最新版本（参见 参考资料 中的链接）。以上
setuid 位，然后给它分配所需的 CAP_NET_RAW 特权。现在，任何用户都可以用 CAP_NET_RAW 特
被突破了，攻击者也无法掌握其他特权。

问题在于，如何判断一个非特权用户在运行某个程序时需要的最小能力集。如果只考虑一个程
身、它的动态链接库和内核源代码。但是，需要对所有 setuid root 程序都重复这个过程。当然
行一个应用程序之前，采用这种方法进行检查并不是个坏主意，但是这种方法不切实际。

概览
进程能力
文件能力

如果一个程序提供详细的错误输出而且表现正常，那么不使用任何特权来运行这个程序，然后根据结果修改。我们来对 ping 试试这种方法。


```
3 su - myuser
4 ping google.com
5     ping: icmp open socket: Operation not permitted
```

如果我们了解 icmp 的实现，这种技巧可以帮助我们判断问题，但是它确实没有把问题说清楚。

接下来，我们可以试着在 strace 之下运行这个程序（同样不设置 suid 位）。strace 会报告这回值，所以可以通过查看 strace 输出中的返回值来判断缺少的权限。

```
1 strace -oping.out ping google.com
2 grep EPERM ping.out
3    socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = -1 EPERM (Operation not permitted
```

我们缺少创建套接字类型 SOCK_RAW 的权限。查看 /usr/include/linux/capability.h，会看到：

```
1 /* Allow use of RAW sockets */
2 /* Allow use of PACKET sockets */
3
4 #define CAP_NET_RAW          13
```

显然，为了允许非特权用户使用 ping，需要的能力是 CAP_NET_RAW。但是，有些程序可能会试作，-EPERM 会拒绝这些操作。判断它们真正需要的能力并不这么容易。

另一种更可行的方法是，在内核中检查能力的地方插入一个探测。这个探测输出关于被拒绝的

开发人员可以用 kprobes 编写小的内核模块，从而在函数的开头（jprobe）、函数的结尾（kr（kprobe）运行代码。可以利用这个功能收集信息，了解内核在运行某些程序时需要哪些能力启用了 kprobes 和文件能力。）

清单 3 是一个内核模块，它插入一个 jprobe 来探测 cap_capable() 函数的开头。

清单 3. capable_probe.c

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/kprobes.h>
4 #include <linux/sched.h>
5
6 static const char *probed_func = "cap_capable";
7
8 int cr_capable (struct task_struct *tsk, int cap)
```

```
 9  {
10      printk(KERN_NOTICE "%s: asking for capability %d for %s\n",
11          __FUNCTION__, cap, tsk->comm);
```



```
15
16  static struct jprobe jp = {
17      .entry = JPROBE_ENTRY(cr_capable)
18  };
19
20  static int __init kprobe_init(void)
21  {
22      int ret;
23      jp.kp.symbol_name = (char *)probed_func;
24
25      if ((ret = register_jprobe(&jp)) < 0) {
26          printk("%s: register_jprobe failed, returned %d\n",
27              __FUNCTION__, ret);
28          return -1;
29      }
30      return 0;
31  }
32
33  static void __exit kprobe_exit(void)
34  {
35      unregister_jprobe(&jp);
36      printk("capable kprobes unregistered\n");
37  }
38
39  module_init(kprobe_init);
40  module_exit(kprobe_exit);
41
42  MODULE_LICENSE("GPL");
```

当插入这个内核模块时，对 cap_capable() 的任何调用都被替换为对 cr_capable() 函数的调
的名称和被核查的能力。然后，通过调用 jprobe_return() 继续执行实际的 cap_capable() i

使用清单 4 中的 makefile 编译这个模块：

清单 4. capable_probe 的 makefile

```
1  obj-m := capable_probe.o
2  KDIR := /lib/modules/$(shell uname -r)/build
3  PWD := $(shell pwd)
4  default:
5      $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
6  clean:
7      rm -f *.mod.c *.ko *.o
```

然后作为根用户执行它：

```
1  /sbin/insmod capable_probe.ko
```

现在在一个窗口中，用以下命令查看系统日志：

在另一个窗口中，作为非根用户执行没有设置 setuid 位的 ping 二进制程序：

内容

```
1 /bin/ping google.com
```

系统日志现在包含关于 ping 的几条记录。这些记录指出这个程序试图使用的能力。这些能力并是 21、13 和 7，可以检查 /usr/include/linux/capability.h，将整数转换为能力名称：

- 21 是 CAP_SYS_ADMIN。不要把这个能力授予*任何*程序。
- 7 是 CAP_SETUID。ping 应该不需要这个能力。
- 13 是 CAP_NET_RAW。ping 应该需要这个能力。

我们将这个能力授予 ping，看看它是否能够成功执行。

```
1 setfcaps -c cap_net_raw=p -e /bin/ping
2 (become non root user)
3 ping google.com
```

不出所料，ping 成功了。

复杂情况

现有的软件常常编写得尽可能可靠，在许多 UNIX 变体上很少有改动。发行版有时候会在此之. 些情况下不可能用文件能力替代 setuid 位。

这种情况的一个例子是 Fedora 上的 at。at 程序允许用户将作业安排在以后某个时间执行。例话：

```
1 echo "xterm -display :0.0 -e \
2 \"echo Call customer 555-5555; echo ^V^G; sleep 10m\" " | \
3 at 14:00
```

所有 UNIX 系统上都有 at 程序，任何用户都可以使用它。用户共享 /var/spool 下面的一个公用极其重要，但是它是跨许多系统工作，所以不能使用系统特有的安全机制（比如能力）。无论减少特权。在此基础上，Fedora 通过应用补丁使用 PAM 模块。

要想查明非根用户是否可以运行不带 setuid 位的 at，最快的方法是删除 setuid 位，然后授予


```
4 /usr/bin/at
```

内容

通过指定 -c all=p，我们请求在 /usr/bin/at 上设置包含所有能力的允许能力集。所以，运行根特权。但是在 Fedora 7 上，运行 /usr/bin/at 会产生以下结果：

```
1 You do not have permission to run at.
```

如果下载并研究源代码，就可以找到原因，但是这些细节对本文没有帮助。肯定可以修改源代是在 Fedora 上简单地分配文件能力并不能取代 setuid 位。

文件能力细节

在前面，我们使用一种专用的格式为可执行程序分配能力。我们对 ping 使用了以下命令：

```
1 setfcaps -c cap_net_raw=p -e /bin/ping
```

setfcaps 程序通过设置一个名为 *security.capability* 的扩展属性，设置目标文件的能力。-c 标列表：

```
1 capability_list=capability_set(s)
```

capability_set 可以包含 i 和 p，capability_list 可以包含任何有效能力。能力类型分别个集指定单独的能力列表。-e 或 -d 标志分别表示允许集中的能力在启动时是否在程序的有效中，那么程序必须能够感知能力，必须自己启用有效集中的位，才能使用能力。

到目前为止，我们已经在允许集中设置了所需的能力，但是还没有在可继承集中设置。实际上大的效果。下面回忆一下清单 1：

重复清单 1. 在执行 exec() 之后计算新能力集的公式

```
1 pI' = pI
2 pP' = fP | (fI & pI)
3 pE' = pP' & fE
```

文件可继承集决定进程的哪些可继承能力可以放在新的进程允许集中。如果文件可继承集中只将这个能力继承到新的进程允许集中。

最后，文件有效位表示任务的新允许集中的位是否应该在新的有效集中设置；也就是说，程序需要用 cap_set_proc(3) 显式地请求它们。

如果没有设置 SECURE_NOROOT，系统会对根用户做一些修改。就是说，系统假设在执行文件时和有效集（fE）包含所有能力。所以二进制文件上的 fI 集只对具有非空能力集的非根进程有作力的程序，将应用上面的公式，而不会使用上面的假设。SECURE_NOROOT 以后可能会成为每个使用本身的能力，还是使用 root-user-is-privileged 模型。但是到编写本文时，在任何实际系统置，它的默认设置让根用户总是拥有所有能力。

为了演示这些集的相互作用，假设管理员用以下命令在 /bin/some_program 上设置了文件能力

```
setfcaps -c cap_sys_admin=i,cap_dac_read_search=p -e \
/bin/some_program
```

如果一个非根用户在拥有所有能力 下运行这个程序，首先计算它的可继承集（pI）和 fI cap_sys_admin。接下来，计算 f 个集的并集，所以结果是 cap_sys_admin+cap_dac_re 允许集。

最后，因为设置了有效位，新的任 集将包含新允许集中的两个能力。

另一方面，如果一个完全没有特权的用户运行同一个程序，他的可继承集是空的，这个集与 fI 空集与 fP 求并集，产生 cap_dac_read_search。这个集成为新的任务允许集。最后，因为设置 允许集，同样只包含 cap_dac_read_search。

在这两种情况下，如果没有设置有效位，那么任务需要使用 cap_set_proc(3) 将它所需的位从

总结和练习

下面总结一下：

- 文件有效位表示程序在默认情况下是否能够使用它的允许能力。
- 文件允许集中的能力总会在产生的进程上启用。
- 文件可继承集中的能力可以从父进程的可继承集继承到新的允许集。

为了演示前面讨论的内容，我们编写了清单 5 和清单 6 中的程序。在清单 5 中，print_caps 中，尝试作为根用户执行 exec_as_nonroot_priv。它请求在下一次调用 setuid(2) 时保留它

清单 5. print_caps.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/capability.h>

int main(int argc, char *argv[])
{
    cap_t cap = cap_get_proc();

    if (!cap) {
        perror("cap_get_proc");
        exit(1);
    }
    printf("%s: running with caps %s\n", argv[0], cap_to_text(cap, NULL));
    cap_free(cap);
    return 0;
}
```

清单 6. exec_as_nonroot_priv.c

```
#include <sys/prctl.h>
#include <sys/capability.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void printmycaps(void)
{
    cap_t cap = cap_get_proc();

    if (!cap) {
        perror("cap_get_proc");
        return;
    }
    printf("%s\n",  cap_to_text(cap, NULL));
    cap_free(cap);
}

int main(int argc, char *argv[])
{
    cap_t cur;
    int ret;
    int newuid;

    if (argc<4) {
        printf("Usage: %s <uid> <capset>"
            "<program_to_run>\n", argv[0]);
        exit(1);
    }
    ret = prctl(PR_SET_KEEPCAPS, 1);
    if (ret) {
        perror("prctl");
        return 1;
    }
```

```
    newuid = atoi(argv[1]);
    printf("Capabilities before setuid: ");
    printmycaps();
```



```
        return 1;
    }
    printf("Capabilities after setuid, before capset: ");
    printmycaps();
    cur = cap_from_text(argv[2]);
    ret = cap_set_proc(cur);
    if (ret) {
        perror("cap_set_proc");
        return 1;
    }
    printf("Capabilities after capset: ");
    cap_free(cur);
    printmycaps();
    ret = execl(argv[3], argv[3], NULL);
    if (ret)
        perror("exec");
}
```

我们用这些程序检验一下可继承集和允许集的效果。在 print_caps 上设置文件能力，然后用初始进程能力集并执行 print_caps。首先，只在 print_caps 的允许集中设置一些能力：

```
gcc -o print_caps print_caps.c -lcap
setfcaps -c cap_dac_override=p -d print_caps
```

现在，作为非根用户执行 print_

```
su - (username)
./print_caps
```

接下来，作为根用户通过 exec_as_nonroot_priv 执行 print_caps：

```
./exec_as_nonroot_priv 1000 cap_dac_override=eip ./print_caps
```

在这两种情况下，print_caps 运行时的能力集都是 cap_dac_override=p。注意，有效位是空调用 cap_set_proc(3)，然后才能使用 cap_dac_override 能力。要想改变这种情况，可以在有效位。

```
setfcaps -c cap_dac_override=p -e print_caps
```

print_caps 的 fI 是空的，所以进程的 pI 中的能力都不能继承到 pP' 中。pP' 只包含来自文件

另一个有意思的测试检验可继承文件能力的效果，同样作为非根用户和通过 exec_as_nonroot print_caps：


```
3      ./print_caps
4      exit
5  ./exec_as_nonroot_priv 1000 cap_dac_override=eip ./print_caps
```

这一次，非根用户的能力集是空的，作为根用户启动的进程的允许集和有效集中包含 cap_dac

再次运行 print_caps，这一次直接作为根用户运行，而不通过 exec_as_nonroot_priv。注意何设置，根用户在执行程序之后总是获得完整的能力集。exec_as_nonroot_priv 并不作为根用根用户的特权为非根进程设置一些可继承能力。

结束语

现在，您已经了解了如何判断一个程序所需的能力，如何设置能力，以及如何用文件能力做其

在处理能力时一定要小心：它们 特权中比较危险的部分。另一方面，sendmail capab 接）表明，提供的能力太少 也可 金。无论如何，对系统二进制代码谨慎地应用文件能力 式，可以更好地保护系统。

相关主题

- 您可以参阅本文在 developerWorks 全球站点上的 英文原文。
- 在 developerWorks 上的 *Secure programmer* 系列中，有几篇文章讨论了 setuid()：
  - “安全编程: 警惕输入”（developerWorks，2003 年 12 月）：讨论将数据输入程序的方
  - “安全编程: 最小化特权”（2004 年 5 月）：讨论如何提供满足系统用户需求的最少特权
  - “安全编程: 安全地调用组件”（2004 年 12 月）：解释如何防止攻击者通过组件调用攻
- “让 Linux 更安全，第 3 部分: 加固系统”（developerWorks，2005 年 4 月）：提供了帮助引导进程和本地文件系统，保护服务和守护进程，实施限额和限制，启用强制性的访问控制性时可能引入的安全弱点。
- “对话 UNIX，第 8 部分: UNIX 进程”（developerWorks，2007 年 4 月）：讲解如何控制进
- “Linux 内核剖析”（developerWorks，2007 年 6 月）：介绍 Linux 的各个部分如何相互配
- 查阅 sendmail capabilities bug。

- Linux 手册页上指出：exec() 函数系列用一个新的进程映像替换当前的进程映像。
- 在 Linux 手册页上指出：在进程上调用 [illegible] 时，第一个参数描述要做什么（使用 [illegible]

- 在 Linux 手册页上指出：setuid() 设置当前进程的有效用户 ID。如果调用者的有效 UID 是保存的 set-user-ID。在 Linux 中，它的实现方法与带 _POSIX_SAVED_IDS 特性的 POSIX 版根）程序能够放弃它的所有用户特权，执行一些非特权操作，然后恢复原来的有效用户 ID。
- 在 Linux 手册页上指出：cap_set_proc() 用 cap_p 表示的能力状态为所有能力设置能力标程的新能力状态完全由 cap_p 的内容决定。如果对调用进程中不允许的能力设置 cap_p 的败，进程的能力状态保持不变。
- POSIX Threads Programming 是一个出色的教程，它介绍了基本概念并涉及许多主题，比
- POSIX（也称为 IEEE Std 1003.1-2001）定义了一个标准的操作系统界面和环境，包括命序，促进源代码级的应用程序可移植性。应用程序开发人员和系统实现者都可以使用它。
- 阅读 “在 Linux 中使用 ReiserFS 文件系统”（developerWorks，2006 年 4 月），了解这种
- 在 “Differentiating UNIX and Linux”（developerWorks，2006 年 3 月）中，讨论了 Linux —— 寻找 “Filesystem support” 部分。
- “系统管理员工具包: 迁移和移 X 文件系统”（developerWorks，2006 年 7 月）提供正在运行的系统上如何迁移整 系统，包括如何创建、复制和重新启用。
- 从 GoogleCode 获得 libcap 库 [illegible]程序。
- Linux PAM 是一种灵活的用户 证机制，它使开发人员可以建立独立于身份验证方案的必重新编码所有身份验证支持 。
- 在 developerWorks Linux 专区 中可以找到为 Linux 开发人员准备的更多参考资料，还可以
- 查阅 developerWorks 上的所有 Linux 技巧 和 Linux 教程。
- 使用 IBM 试用软件 构建您的下一个 Linux 开发项目，这些软件可以从 developerWorks 直

评论

添加或订阅评论，请先登录或注册。

有新评论时提醒我