Linux给普通用户尽可能低的权限，而把全部的系统权限赋予一个单一的帐户--root。root帐户用来管理系统、安装软件、管理帐户、运行某些服务、安装/卸载文件系统、管理用户、安装软件等。这种依赖单一帐户执行特权操作的方式加大了系统的面临风险，而需要root权限的程序可能只是为了一个单一的操作。例如，一个普通用户需要使用ping命令。这是一个SUID命令，会以root的权限运行。而实际上这个程序只是需要RAW套接字建立必要ICMP数据包，除此之外的其它root权限对这个程序都是没有必要的。如果程序编写不好，就可能被攻击者利用，获得系统的控制权。

Linux针对上述问题提出capability，将内核系统资源访问划分为许多的权利属性。使用能力(capability)可以减小这种风险。系统管理员为了系统的安全可以剥夺root权限程序的某些capability，这样即使该进程也将无法进行这些capability的操作。而这个过程又是不可逆的，也就是说如果一种能力被删除，无法重新添加被删除的能力。一个root进程在没有设置capability的时候，所有权利属性默认开启。当一个root进程通过运行execv类型的函数加载新的进程后，进程重新获取所有capability。

Linux所特有的，这些能力在/usr/src/linux/include/linux/capability.h文件中定义。其细节如下：

CAP\_CHOWN 0 允许改变文件的所有权

CAP\_DAC\_OVERRIDE 1 忽略对文件的所有DAC访问限制

CAP\_DAC\_READ\_SEARCH 2 忽略所有对读、搜索操作的限制

。。。。。。。

我们通过lcap库来实现root程序的权限分离：

Static int set\_capabilities(cap\_value\_tcap\_list[], intncap)

{

cap\_t caps;

int ret;

/\* caps should be initialized with all flags cleared... \*/

caps = cap\_init();

if (!caps)

{

perror("cap\_init");

return -1;

}

/\* ... but we better rely on cap\_clear \*/

if (cap\_clear(caps))

{

perror("cap\_clear");

return -1;

}

if ((cap\_list) &&ncap)

{

if (cap\_set\_flag(caps, CAP\_PERMITTED, ncap, cap\_list, CAP\_SET) != 0)

{

perror("cap\_set\_flag CAP\_PERMITTED");

cap\_free(caps);

return -1;

}

if (cap\_set\_flag(caps, CAP\_EFFECTIVE, ncap, cap\_list, CAP\_SET) != 0)

{

perror("cap\_set\_flag CAP\_EFFECTIVE");

cap\_free(caps);

return -1;

}

if (cap\_set\_flag(caps, CAP\_INHERITABLE, ncap, cap\_list, CAP\_SET) != 0)

{

perror("cap\_set\_flag CAP\_INHERITABLE");

cap\_free(caps);

return -1;

}

}

ret = cap\_set\_proc(caps);

if (ret)

{

perror("cap\_set\_proc");

cap\_free(caps);

return -1;

}

if (cap\_free(caps))

{

perror("cap\_free");

return -1;

}

return 0;

}

Static intdrop\_capabilities(cap\_value\_tcap\_list[], intncap)

{

cap\_t caps;

int ret;

/\* caps should be initialized with all flags cleared... \*/

caps = cap\_get\_proc();

if (!caps)

{

perror("cap\_init");

return -1;

}

if ((cap\_list) &&ncap)

{

if (cap\_set\_flag(caps, CAP\_EFFECTIVE, ncap, cap\_list, CAP\_CLEAR) != 0)

{

perror("cap\_set\_flag CAP\_PERMITTED");

cap\_free(caps);

return -1;

}

if (cap\_set\_flag(caps, CAP\_PERMITTED, ncap, cap\_list, CAP\_CLEAR) != 0)

{

perror("cap\_set\_flag CAP\_EFFECTIVE");

cap\_free(caps);

return -1;

}

if (cap\_set\_flag(caps, CAP\_INHERITABLE, ncap, cap\_list, CAP\_CLEAR) != 0)

{

perror("cap\_set\_flag CAP\_INHERITABLE");

cap\_free(caps);

return -1;

}

}

ret = cap\_set\_proc(caps);

if (ret)

{

perror("cap\_set\_proc");

cap\_free(caps);

return -1;

}

return 0;

}

int main()

{

intch, ret = -1;

cap\_value\_tcap\_list[2];

cap\_t caps;

cap\_value\_tdrop\_cap[2];

/\* capabilities we need \*/

cap\_list[0] = CAP\_SETUID;

cap\_list[1] = CAP\_NET\_RAW;

//set the cap we need

if (set\_capabilities(cap\_list, sizeof(cap\_list)/sizeof(cap\_list[0])))

{

fprintf(stderr, "Could not adjust capabilities, aborting\n");

return EXIT\_FAILURE;

}

ret = socket(PF\_INET, SOCK\_RAW, IPPROTO\_ICMP);

assert(ret != -1);

printf("ok, run raw socket sucessful\n");

drop\_cap[0] = CAP\_SETUID;

drop\_cap[1] = CAP\_NET\_RAW;

// drop the caps above

if (drop\_capabilities(drop\_cap, sizeof(drop\_cap)/sizeof(drop\_cap[0])))

{

fprintf(stderr, "Could not adjust capabilities, aborting\n");

return EXIT\_FAILURE;

}

// must be can not raw socket ecause of the drop CAP\_NET\_RAW

ret = socket(PF\_INET, SOCK\_RAW, IPPROTO\_ICMP);

assert(ret == -1);

printf("ok, run raw socket failed\n");

return 0;

}

gcc–o cap cap.c -lcap

使用root用户运行cap程序，发现在设置了CAP\_NET\_RAW时候程序能够进行raw socket操作，当drop掉它的时候系统不允许。