SCC - SECCOMP

Cómo evitar las llamadas al sistema

Jose Ángel Gumiel

Contenido

Introducción 3
Entorno de desarrollo 3
Desarrollo 3
Comprobaciones 3
Paso 1: Desarrollo de la aplicación inicial 4
Paso 2: Añadiendo filtros a la aplicación 5
Paso 3: Reportando las llamadas al sistema 7
Paso 4: Depurando o permitiendo llamadas 10
Conclusiones 10
Referencias 10

Introducción

En este documento se va a explicar los pasos que se han dado para poner en marcha y probar el funcionamiento de "seccomp". Ésta es una herramienta que sirve para filtrar las llamadas al sistema, y del mismo modo, depurar un programa ante posibles vulnerabilidades

Entorno de desarrollo

Se ha utilizado una distribución de Linux ligera basada en Debian 9. Las pruebas se están haciendo sobre una máquina virtual, pero del mismo modo se puede hacer en cualquier entorno Linux. Se está utilizando VMWare Workstation 11.0 sobre Windows 10.

Desarrollo

En este apartado se va a explicar el desarrollo que se ha seguido para la elaboración de este laboratorio sobre "seccomp".

Comprobaciones

Se comprueba que la distribución utilizada tenga activado "seccomp". Para ello usamos los siguientes comandos:

Conocer la versión del Kernel de Linux:

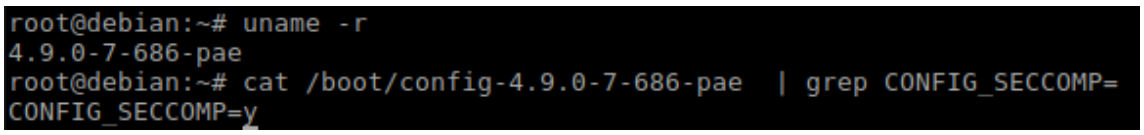
```
uname –r
```

Con "cat", se lee y se muestra por pantalla un fichero, en este caso, el fichero de configuración que aparece en la ruta. El comando grep sirve como filtro.

Mediante un "pipeline" se consigue que, al imprimir ese fichero, le llegue el resultado a grep. Este hará un filtro y sólo mostrará la línea que contenga el texto que se le ha definido, es decir, la línea que contenga "CONFIG_SECCOMP=".

```
cat /boot/config-4.9.0-7-686-pae  | grep CONFIG_SECCOMP=
```

Se ha obtenido una captura de pantalla, y el resultado es el siguiente:



Se va a programar una aplicación en lenguaje C. Se va a utilizar también "autoconf", "autoheader" y "make". Si no está instalado en el sistema, habrá que actualizar la lista de repositorios y añadir estos programas.

```
apt-get update
apt-get install gcc
apt-get install autoconf
apt-get install make
```

Paso 1: Desarrollo de la aplicación inicial

En este apartado se va a desarrollar una aplicación sencilla que haga una llamada al sistema. Se ha pensado en una aplicación que haga un "fork()", es decir, que cree proceso un hijo.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "config.h"

int main() {
    pid_t forkStatus;
    forkStatus = fork();

    /* Hijo... */
    if (forkStatus == 0) {
        printf("Hijo en ejecucion. Espere.\n");
        sleep(2);
        printf("Hijo terminado, saliendo..\n");

    /* Padre... */
    } else if (forkStatus != -1) {
        printf("Proceso padre a la espera...\n");
        wait(NULL);
        printf("Padre terminado...\n");

    } else {
        perror("Error al llamar a la funcion fork()");
    }
    return 0;
}
```

Código: procFork.c

```
CC=gcc
CFLAGS=-Wall

all: procFork

procFork: procFork.o

.PHONY: clean
clean:
    rm -f procFork procFork.o
```

Código: Makefile

```
AC_INIT([procFork], 0.1)
AC_PREREQ([2.59])
AC_CONFIG_HEADERS([config.h])
AC_PROG_CC
AC_OUTPUT
```

Código: configure.ac

El programa se compila del siguiente modo:

```
autoconf
autoheader
./configure
make
```

```
root@debian:~/Escritorio/SCC/paso1# autoconf
root@debian:~/Escritorio/SCC/paso1# autoheader
root@debian:~/Escritorio/SCC/paso1# ./configure
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
configure: creating ./config.status
config.status: creating config.h
config.status: config.h is unchanged
root@debian:~/Escritorio/SCC/paso1# make
gcc -Wall   -c -o procFork.o procFork.c
gcc   procFork.o   -o procFork
```

Compilación: El programa se ha compilado con éxito.

Se ha generado ya el programa procFork. Se ejecuta para ver su resultado:

```
root@debian:~/Escritorio/SCC/paso1# ./procFork
Proceso padre a la espera...
Hijo en ejecucion. Espere.
Hijo terminado, saliendo..
Padre terminado...
root@debian:~/Escritorio/SCC/paso1#
```

Resultado: El programa se ejecuta.

Hay un fork(), que es una llamada al sistema y se ejecuta bien cuando no está seccomp.

Paso 2: Añadiendo filtros a la aplicación

A continuación, hay que añadir la librería “seccomp-bpf.h” y actualizar el fichero “configure.ac” para agrefar también la librería “linux/seccomp.h”.

En el fichero de programación habrá que hacer una arquitectura. Se comprobará si las llamadas al sistema están permitidas. En el caso de no estarlo, se terminará el proceso.

Se va a marcar en color verde las nuevas líneas de código.

```
AC_INIT( [procFork], 0.1)
AC_PREREQ([2.59])
AC_CONFIG_HEADERS([config.h])
AC_PROG_CC
AC_CHECK_HEADERS([linux/seccomp.h])
AC_OUTPUT
```

Código: configure.ac

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/wait.h>
```

```c
#include "config.h"
#include "seccomp-bpf.h"

static int install_syscall_filter(void)
{
    struct sock_filter filter[] = {
        /* Validate architecture. */
        VALIDATE_ARCHITECTURE,
        /* Grab the system call number. */
        EXAMINE_SYSCALL,
        /* List allowed syscalls. */
        ALLOW_SYSCALL(rt_sigreturn),
#ifdef __NR_sigreturn
        ALLOW_SYSCALL(sigreturn),
#endif
        ALLOW_SYSCALL(exit_group),
        ALLOW_SYSCALL(exit),
        ALLOW_SYSCALL(read),
        ALLOW_SYSCALL(write),
        KILL_PROCESS,
    };
    struct sock_fprog prog = {
        .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
        .filter = filter,
    };

    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("prctl(NO_NEW_PRIVS)");
        goto failed;
    }
    if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
        perror("prctl(SECCOMP)");
        goto failed;
    }
    return 0;

failed:
    if (errno == EINVAL)
        fprintf(stderr, "SECCOMP_FILTER is not available. :(\n");
    return 1;
}

int main() {

    if (install_syscall_filter())
        return 1;

    pid_t forkStatus;
    forkStatus = fork();

    /* Hijo... */
    if (forkStatus == 0) {
        printf("Hijo en ejecucion. Espere.\n");
        sleep(2);
        printf("Hijo terminado, saliendo...\n");

    /* Padre... */
    } else if (forkStatus != -1) {
        printf("Proceso padre a la espera...\n");
        wait(NULL);
```

```
        printf("Padre terminado...\n");

    } else {
        perror("Error al llamar a la funcion fork()");
    }
    return 0;
}
```

Código: procFork.c

Siguiendo el mismo método que antes, se compila el programa y se ejecuta:

```
root@debian:~/Escritorio/SCC/paso2# ./procFork
Llamada al sistema errónea
root@debian:~/Escritorio/SCC/paso2#
```

Resultado: Llamada al sistema errónea.

Salta un mensaje de error que no da demasiada información. Lo que se sí que se puede deducir de este resultado es que hace el filtro, ya que hay al menos una llamada al sistema que no está contemplada y que "seccomp" no deja que se haga, terminando el proceso.

Lo que falta en este punto es obtener la llamada al sistema que se está efectuando, de este modo, se podrá depurar.

Paso 3: Reportando las llamadas al sistema

En este paso se va a utilizar una de las características añadidas del filtro de "seccomp". Esto va a permitir capturar las llamadas al sistema fallidas e informar al usuario inmediatamente antes de terminar.

El significado de esta etapa es lanzar una ejecución, ver qué llamadas al sistema se producen y conocerlas. De este modo, el programador puede decidir si permitirlas y añadirlas a la "lista blanca" o si por el contrario, debe de pensar en otra forma de programar la aplicación sin que se haga esa llamada. Esto último en el caso de que pueda ser una vulnerabilidad para el sistema.

Para este escenario son necesarios tres ficheros: "syscall-reporter.c", "syscall-reporter.h" y "syscall-reporter.mk". Este último ha habido que modificarlo, ya que en Debian 9, la ruta de la librería es diferente a la que aparece en el archivo. Aparece en rojo la línea eliminada y el verde la modificación.

```
syscall-names.h: /usr/include/sys/syscall.h syscall-reporter.mk
syscall-names.h: /usr/include/syscall.h syscall-reporter.mk
	echo "static const char *syscall_names[] = {" > $@ ;\
	echo "#include <sys/syscall.h>" | cpp -dM | grep '^#define __NR_'
| \
	    LC_ALL=C sed -r -n -e 's/^\#define[ \t]+__NR_([a-z0-9_]+)[
\t]+([0-9]+)(.*)/ [\2] = "\1",/p' >> $@ ;\
	echo "};" >> $@

syscall-reporter.o: syscall-reporter.c syscall-names.h
```

Código: syscall-reporter.mk

Ahora hay que cambiar el código fuente y el Makefile.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/wait.h>

#include "config.h"
#include "seccomp-bpf.h"
#include "syscall-reporter.h"

static int install_syscall_filter(void)
{
    struct sock_filter filter[] = {
        /* Validate architecture. */
        VALIDATE_ARCHITECTURE,
        /* Grab the system call number. */
        EXAMINE_SYSCALL,
        /* List allowed syscalls. */
        ALLOW_SYSCALL(rt_sigreturn),
#ifdef __NR_sigreturn
        ALLOW_SYSCALL(sigreturn),
#endif
        ALLOW_SYSCALL(exit_group),
        ALLOW_SYSCALL(exit),
        ALLOW_SYSCALL(read),
        ALLOW_SYSCALL(write),
        KILL_PROCESS,
    };
    struct sock_fprog prog = {
        .len = (unsigned short)(sizeof(filter)/sizeof(filter[0])),
        .filter = filter,
    };

    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("prctl(NO_NEW_PRIVS)");
        goto failed;
    }
    if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
        perror("prctl(SECCOMP)");
        goto failed;
    }
    return 0;

failed:
    if (errno == EINVAL)
        fprintf(stderr, "SECCOMP_FILTER is not available. :(\n");
    return 1;
}

int main() {
    if (install_syscall_reporter())
        return 1;
    if (install_syscall_filter())
        return 1;

    pid_t forkStatus;
    forkStatus = fork();

    /* Hijo... */
    if (forkStatus == 0) {
```

```
        printf("Hijo en ejecucion. Espere.\n");
        sleep(2);
        printf("Hijo terminado, saliendo...\n");

    /* Padre... */
    } else if (forkStatus != -1) {
        printf("Proceso padre a la espera...\n");
        wait(NULL);
        printf("Padre terminado...\n");

    } else {
        perror("Error al llamar a la funcion fork()");
    }
    return 0;
}
```

Código: procFork.c

En el código fuente "procFork.c" se ha añadido la librería y se hace la llamada a la función de aviso de llamadas al sistema.

```
CC=gcc
CFLAGS=-Wall

all: procFork

include syscall-reporter.mk

procFork: procFork.o syscall-reporter.o

.PHONY: clean
clean:
    rm -f procFork procFork.o
```

Código: Makefile

En el caso del makefile, se añade el fichero mk, que da instrucciones de cómo compilar, y se modifica para crear código objeto de "syscall-reporter". Hay que combinar los dos en la misma aplicación.

Se compila igual que en pasos anteriores. En la siguiente imagen se ve el resultado.

```
root@debian:~/Escritorio/SCC/paso3# make
gcc -Wall   -c -o procFork.o procFork.c
In file included from procFork.c:10:0:
syscall-reporter.h:21:2: warning: #warning "You've included the syscall reporter
. Do not use in production!" [-Wcpp]
 #warning "You've included the syscall reporter. Do not use in production!"
  ^~~~~~~
gcc   procFork.o syscall-reporter.o   -o procFork
root@debian:~/Escritorio/SCC/paso3# ./procFork
Looks like you also need syscall: clone(120)
root@debian:~/Escritorio/SCC/paso3#
```

Resultado: Se avisa de la llamada al sistema y termina el proceso.

El compilador avisa de que se ha incluido el "syscall-reporter" en el programa. Y dice que no se use en producción. Está bien para depurar fallos, pero en una aplicación final no está bien visto.

Paso 4: Depurando o permitiendo llamadas

En este paso se pretende ver qué pasa cuando se permite la llamada al sistema “clone”, propia del “fork()”. Lo que no se va a permitir es el “wait”.

Para ello se agregan al código fuente procFork.c las siguientes líneas:

```
        ALLOW_SYSCALL(mmap),
        ALLOW_SYSCALL(rt_sigprocmask),
        ALLOW_SYSCALL(rt_sigaction),
        ALLOW_SYSCALL(nanosleep),
        ALLOW_SYSCALL(clone),
        ALLOW_SYSCALL(fstat64),
        ALLOW_SYSCALL(brk),
```

```
root@debian:~/Escritorio/SCC/paso4# ./procFork
Proceso padre a la espera...
Looks like you also need syscall: wait4(114)
root@debian:~/Escritorio/SCC/paso4# Hijo en ejecucion. Espere.
Hijo terminado, saliendo...
```

Resultado: El proceso hijo se ejecuta, pero el padre es terminado.

Se ha permitido el “sleep” y el clone, pero no el “wait”. El padre hace “wait” mientras se ejecuta el hijo, entonces se “mata” a ese proceso. Sin embargo, el hijo usa un “sleep”, que está permitido, y este termina de ejecutarse correctamente.

Conclusiones

La librería “seccomp” es una buena utilidad para conocer lo que se ejecuta por debajo del programa, a nivel de sistema operativo. Es una herramienta que permite al programador ver llamadas al sistema, que se ejecutan, y que tal vez no había contemplado antes.

Por otra parte, también puede servir cuando se usa software de terceros. Se puede dar la situación de tener un código fuente de otro desarrollador. Si no hay tiempo para analizar ese código, se puede ejecutar de forma segura con “seccomp”. Puede que no sea un programa malicioso, pero ante la incertidumbre, se puede ejecutar en un entorno que no permita ciertas llamadas.

La parte más complicada es la puesta en marcha. Cuando ya se conocen las librerías y el funcionamiento, añadir la capa de “seccomp” al código es un proceso bastante sencillo.

Referencias

https://www.gnu.org/software/autoconf/manual/autoconf-2.61/html_node/Writing-configure_002eac.html

https://outflux.net/teach-seccomp/