Programación Segura y CHROOT

Jesús Cea Avión

Ingeniero de Telecomunicación

Ingeniero de Sistemas

jcea@argo.es

http://www.argo.es/~jcea/

Argo RST, S.A.

Madrid, 6 de Abril de 2.000

Programación Segura y CHROOT

Recursos en Internet

- Copia de estas transparencias:
  - http://www.argo.es/~artic/seguridad-dsk00.pdf
- CHROOT
  - http://www.argo.es/~jcea/artic/chroot.htm
  - http://www.argo.es/~jcea/artic/chroot2.htm
- Programación segura
  - http://www.argo.es/~jcea/artic/prg_seg.htm
- Seguridad en general
  - http://www.argo.es/~jcea/artic/
  - http://www.argo.es/~jcea/artic/hack-faq.htm
  - http://www.hispasec.com/

Programación Segura y CHROOT

CHROOT

- Virtualiza exclusivamente el sistema de ficheros
  - No impone otras restricciones al acceso a recursos que el hecho de que estos suelen implementarse como ficheros.
    - “casi todo” en Unix es un fichero.
  - No está diseñado como un sistema de seguridad.
    - No hay seguridad absoluta; la cuestión es desplegar niveles complementarios.
    - Se requieren medidas adicionales para hacer seguro el sistema.
  - Supone un nivel muy primitivo de máquina virtual.

Programación Segura y CHROOT

CHROOT

- Ventajas del CHROOT:
  - Aislamiento de procesos en entornos del sistema de ficheros independientes.
  - Permite limitar las posibilidades de manipulación y ocultación tras un ataque.
  - Cada entorno CHROOT puede trabajar con versiones de librerías, dispositivos, etc., diferentes
  - “bunker” para la prueba de programas experimentales o de dudoso comportamiento.

Programación Segura y CHROOT

CHROOT

- Desventajas del CHROOT:
  - No es una verdadera máquina virtual.
    - Consumo de recursos globales, como tiempo de CPU, memoria, cuota de disco, etc.
      - Pueden implementarse con mecanismos adicionales, como cuotas y límites.
  - No todos los servicios pueden lanzarse dentro de un CHROOT sin problemas.
  - Dificultad para mantener actualizado el entorno CHROOT.
    - Librerías, módulos kernel dinámicos.

Programación Segura y CHROOT

CHROOT

- Es necesario ser “root” para hacer un CHROOT.
  - No hay problema de seguridad en que haga CHROOT cualquier usuario, salvo que se le permita ejecutar procesos SETUID/SETGID.
- El cambio afecta al proceso invocador y a todos sus hijos.
- Es posible realizar nuevos CHROOT dentro de un CHROOT.
  - Posible problema de seguridad, con la semántica actual de la llamada.

Programación Segura y CHROOT

CHROOT

- Pasos a dar:
  - Identificación de los ficheros y librerías invocados por el programa en cuestión.
    - Solaris: “ldd”, “pldd”, “pmap”, “pmem”, “pfiles”, “truss”
    - Linux: “ldd”, interfaz “/proc” y comando “find” por “inode”
  - Replicación de la estructura de directorios demandada por el programa
    - Si reside en la misma partición, se pueden usar “hard-links”
    - Cuidado con comprometer la seguridad al replicar dispositivos o interfaces especiales (como “/proc”)

Programación Segura y CHROOT

CHROOT

- Reglas de seguridad dentro del entorno CHROOT:
  - No debe haber procesos SETUID/SETGID a “root” dentro del CHROOT, a menos que estén extraordinariamente auditados.
  - Si se montan particiones remotas, deben hacerse NOSUID y, a poder ser, como “sólo lectura”.
  - Utilizar un Kernel actualizado, sin fallos conocidos.
  - No debe ser posible cargar módulos en el Kernel.

Programación Segura y CHROOT

CHROOT

- Reglas de seguridad dentro del entorno CHROOT (cont):
  - Ojo con interfaces peligrosas: “/dev”, “/proc”.
  - Ojo con “hardlinks” a directorios externos al CHROOT.
  - Revisar los permisos de escritura de los ficheros “hardlinked” a ficheros externos al CHROOT.

Programación Segura y CHROOT

CHROOT

- Reglas de seguridad en el proceso que realiza el CHROOT:
  - Debe abandonar sus privilegios “root”.
    - Idealmente, no debe ser posible alcanzar privilegios “root” desde dentro del CHROOT.
  - No debe importar ficheros abiertos provenientes del “exterior”.
  - El directorio actual de trabajo debe estar dentro del CHROOT.

Programación Segura y CHROOT

CHROOT

- Notas varias:
  - Es preferible que el CHROOT lo realice un proceso externo, a que lo hagan los propios “demonios”.
    - Auditoría más sencilla.
    - No se requieren modificaciones en los demonios.
  - Cuando sea posible, utilizar el “loopback” para montar “de nuevo” particiones locales en el CHROOT.
    - Facilidad de actualización y menor espacio en disco.
    - Montarlas NOSUID y “sólo lectura”.
  - Ser cuidadoso con “/dev”, “/prov” y los “hardlinks”.

Programación Segura y CHROOT

CHROOT

- ¿Es el CHROOT la panacea?:
  - No, pero ayuda.
    - Portable a cualquier entorno UNIX.
    - Sencillo de implementar y razonablemente seguro si se realiza correctamente.
  - Se necesitan otras opciones (máquinas virtuales, “capabilities”):
    - Pesadilla para el administrador de sistemas, si cada proceso tiene una visión diferente de la estructura del disco.
    - Fallos “misteriosos” por ficheros no presentes.
    - No permite ejecutar procesos “root” de forma segura.

Programación Segura y CHROOT

Programación Segura

- Necesidad de la programación segura:
  - Procesos invocados por usuarios maliciosos
    - En local, típicamente procesos SETUID/SETGID.
  - Procesos cuyas entradas están bajo control de usuarios maliciosos
    - En remoto, cualquier demonio que pueda ofrecer acceso a la máquina, o denegación de servicio.
- Resultado final:
  - Acceso a recursos y provilegios no disponibles de forma “normal”.

Programación Segura y CHROOT

Programación Segura

- Algunos lenguajes de programación son más susceptibles a problemas de seguridad que otros:
  - C/C++
    - Punteros
    - Gestión dinámica de memoria
    - Gestión de cadenas
  - Perl
    - Ejecución de comandos Unix de forma inadvertida en cualquier
    - Compilación dinámica de código
    - Código con “efectos colaterales”

Programación Segura y CHROOT

Programación Segura

- Problemas de la programación segura:
  - Teóricamente menos eficiente
    - Más comprobaciones aparentemente redundantes
  - Tediosa
    - Hay que verificar condiciones “obvias”
    - Hay que escribir más código (y ¡¡depurarlo!!)
  - Ciclos de desarrollo largos
    - Diferencia entre “prototipaje” y aplicación completa
  - No basta con saber programar. Hay que saber programar BIEN

Programación Segura y CHROOT

Programación Segura

- Las Reglas de Oro de la programación segura:
  - Paranoia
    - El usuario fiable no lo será
  - No dar nada por supuesto
    - Si algo no puede ir mal, fallará
    - Si algo puede ir mal, no sólo irá mal, sino que lo hará en el momento más inoportuno, y fallará de la manera más espectacular posible.

Programación Segura y CHROOT

Programación Segura

- Nos vamos a centrar en el lenguaje C/C++:
  - Uno de los lenguajes más universales que existe
    - Hay implementaciones de C para cualquier arquitectura imaginable
  - Lenguaje típico de programación de sistemas
    - Diseñado originariamente para implementar Unix
    - Lenguaje empleado en el kernel Linux
  - La mayoría de los programas que nos vamos a encontrar están en C
    - Perl tiene también gran presencia, y Python es un valor en alza

Programación Segura y CHROOT

Programación Segura

- ¿Por qué fallan los programas?
  - Errores de programación
  - Entradas con valores inesperados
    - Hay que contar con lo “imposible”
  - Condiciones poco conocidas o poco documentadas en las librerías o en el sistema operativo
  - Idiosincrasias del entorno/lenguaje seleccionado

```
char *p=xxx;
int contador[256];
while(*p) contador[*p++]++;
```

Programación Segura y CHROOT

Programación Segura

- Tipos de fallos:
  - Condiciones Internas
    - Bugs o presunciones en el propio código
    - Desbordamientos
    - strcpy()
  - Condiciones Externas
    - Circunstancias no sujetas a nuestro control
    - “race conditions”
    - Enlaces simbólicos

Programación Segura y CHROOT

Programación Segura

- Tipos de fallos (cont.):
  - Ataques Remotos
    - A través de una red (por ejemplo, Internet)
    - Sólo relevantes en programas accesibles desde el exterior
    - Por regla general, el atacante no precisa acceso a nuestro sistema
  - Ataques Locales
    - Casi todo ataque remoto puede convertirse en un ataque local
    - Acceso a un gran número de programas y servicios no accesibles de forma remota (procesos SETUID/SETGID)
    - Acceso a ficheros locales

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Sólo los privilegios imprescindibles
    - CHROOT, “capabilities”, máquinas virtuales
    - Por defecto, sin acceso a nada, y es el administrador quien le va asignando recurso a recurso
    - Entre los privilegios se incluyen también memoria, disco y CPU
  - Usar procesos SUID/SGID de forma inteligente
    - Pueden usarse también para restringir privilegios
      - SUID a otro usuario con privilegios distintos
      - SUID a “root”, seguido de “chroot” y “setgid()”, “setuid()”
    - Librarse de cualquier privilegio lo antes posible

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Usar procesos SUID/SGID de forma inteligente (Cont.)
    - Uso como “wrappers”
    - Pequeño proceso “externo” que nos proporciona un servicio claro con un API bien definida
    - Para encontrar los ejecutables SUID en el sistema, ejecutamos "find . -perm -4000 -print”
    - Nunca usar un script SHELL como SUID, debido a “race conditions”
      - Algunos Unix lo tienen solucionado, y otros no pero, en todo caso, es algo no portable

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Afinar el control de accesos
    - "Access Control Lists"
      - Tecnología originaria de "mainframes"
      - Mucho más flexible que los flags "rwx" tradicionales
      - En Solaris, comandos "setfacl", "getfacl", "acl()", "facl()"
      - No es portable y muchas herramientas no saben tratarlas
    - "capabilities" (credenciales)
      - Originarios de entornos distribuidos
      - No sólo restringen acceso a ficheros, sino también a llamadas del sistema, etc.
      - No portable

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Cuidado con los “buffer overflow”
    - Pueden permitir la ejecución de código arbitrario
    - Error de programación muy habitual
    - Problema inexistente en lenguajes con gestión automática de punteros y memoria: perl, java, python, basic
      - En otros lenguajes se detecta el problema en tiempo de ejecución: ADA
    - Programación defensiva
      - Evaluar cuidadosamente el tamaño de las entradas del usuario
      - el cambio de “strcpy()” por “strncpy()” no es la solución

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Cuidado con los “buffer overflow” (cont.)
    - Entradas típicas:
      - Línea de comando
      - Variables de entorno
      - Ficheros de datos
      - Comandos remotos enviados por una red
    - El frecuente que el código de chequeo contenga errores, ya que no suele ejecutarse en condiciones normales
    - “Buffer overflow” en el sistema y en librerías fuera de nuestro control
      - syslog

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Manejo de ficheros
    - Tarea compleja y delicada. Muy poco intuitiva
    - “access()” sufre de “race conditions”
      - if(access("/path/del/fichero",W_OK)==0) open(...);
      - Los sustitutos de “access()” evalúan el acceso en función del usuario efectivo, no del usuario real
      - Hay que jugar con “stat()”, “lstat()”, “fstat()”
      - Pérdida de portabilidad
      - código complicado
    - Todas las verificaciones deben realizarse siempre sobre el usuario real, no el efectivo

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Manejo de ficheros (cont)
    - Apertura de un fichero ya existente, como lectura o como "append"
      - lstat()
        - » Verificación de las características del fichero (enlace simbólico, fichero regular, dispositivo)
      - open()
      - fstat()
        - » Comparar el resultado de “lstat()” y “fstat()”
      - “race condition” entre el “lstat()” y el “open()”
        - » Importante si el fichero es un dispositivo físico

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Manejo de ficheros (cont)
    - Creación de un fichero nuevo
      - open(O_CREAT|O_EXCL)
        - » Si el fichero ya existe, la orden se aborta

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Manejo de ficheros (cont)
    - Truncado de un fichero
      - lstat()
        - » Vemos si el fichero es válido: enlace simbólico, dispositivo, fichero regular...
      - open()
      - fstat()
        - » Comparamos el resultado de “fstat()” con “lstat()”
      - ftruncate()
      - “race condition” entre el “lstat()” y el “open()”
        - » Importante si el fichero es un dispositivo físico

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Manejo de ficheros (cont)
    - Borrado de un fichero
      - setegid(GID real)
      - seteuid(UID real)
      - unlink()
      - seteuid(UID privilegiado)
      - setegid(GID privilegiado)

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Manejo de ficheros (cont)
    - Todas las verificaciones de acceso deben realizarse sobre el usuario real, no el efectivo
      - Uso de “setgid()”, “setuid()”
      - Especialmente importante cuando en el path existen directorios, ya que habría que verificar componente a componente
        - » Posible uso de “fork()” y “pipe()”, junto a “setgid()” y “setuid()”
      - En sistemas no POSIX que no tengan “saved UID/GID”,se puede intercambiar el usuario real y efectivo con
        - » setreuid(geteuid(),getuid());
      - La verificación es en el “open()”, no en “read()” o “write()”

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Gestión de bloqueos
    - uso de “link()” y “unlink()”
      - Funciona aunque los invoque “root”
        - » “creat()” tiene siempre éxito si somos “root”, aunque el fichero tenga permisos “000”
    - open(O_CREAT|O_EXCL)
      - Ocupa disco y debe crearse con permisos que no permitan a otro proceso escribir sobre él
    - “flock()”
      - Permite bloqueos compartidos y exclusivos (RW y W)
        - » Antiguamente no funcionaba bien sobre NFS

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Gestión de bloqueos (Cont.)
    - En general estos bloqueos son sólo recomendaciones
      - Los procesos se los pueden saltar
      - En algunos Unix se puden fijar modos para que “flock()” sea de cumplimiento obligatorio
        - » Esto sí que no es portable sobre NFS
    - Requisitos especiales en procesos “multithread”
      - Protección de recursos compartidos
        - » Monitores
        - » Mútex
        - » Semáforos

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Ficheros Temporales
    - Ficheros de trabajo
      - Intercambio de datos entre procesos, tipo “pipelining”
      - Tipicamente “/tmp”
        - » “Sticky bit”
    - “mktemp()”
      - Race conditions
    - “tmpfile()”
      - No disponible en todos los Unix
      - Debe abrir el fichero como “O_EXCL”
      - Borra el fichero al terminar con él

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Ficheros Temporales (Cont.)
    - “tmpfile()”
      - “tmpnam()”
        - » Elige un nombre único
      - Crea el fichero con “O_EXCL” y devuelve su descriptor
      - “unlink()”
        - » El fichero ya no “existe”, pero sigue siendo accesible a través de su descriptor
      - El programa accede al fichero usando el descriptor
        - » Puede moverse con “rewind()” y “fseek()”
      - Al cerrar el descriptor, el fichero desaparece del todo

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Ficheros Temporales (Cont.)
    - “tmpfile()” (Cont.)
      - El fichero ocupa espacio en disco, aunque no sea visible en el directorio
      - El fichero no puede usarse para intercambiar datos con un proceso independiente, porque no tiene un Path
    - FIFO
      - Muy útiles cuando existen
        - » No portable
      - No ocupan espacio en disco

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Ficheros “core”
    - En algunos Kernel los “core” siguen enlaces simbólicos
      - No se requiere ningún privilegio especial para crear un enlace simbólico a cualquier fichero, aunque no nos pertenezca ni tengamos acceso a él
    - Un core de un proceso SUID puede filtrar información confidencial
      - Los procesos SUID no deberían generar “core”
      - Los kernel modernos no generan “core” si el usuario real de un proceso no coincide con su usuario efectivo
        - » Un proceso puede limpiar sus privilegios sin borrar memoria

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Ficheros “core” (Cont.)
    - Limitar el tamaño de los “core” o prohibirlos
      - “ulimit”, “setrlimit()”
    - Solaris
      - función “gcore”
    - “core” se graba con los permisos “umask()”
      - Delicado si permite escribir a cualquier usuario
        - » Rebosamiento de cuotas
        - » Ataque DoS por saturación de una partición de disco
        - » Permiso para leer

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - “umask()” heredados
    - Permisos por defecto
    - Heredado del proceso padre
    - Debería prohibir el acceso de escritura para el grupo y para el resto de usuarios
    - Los procesos SUID deben ser cuidadosos con “umask()”
      - Generación de ficheros sobre los que pueden escribir otros usuarios
      - “core”

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - El Shell es demasiado listo
    - Gestión complicada de los metacaracteres

```
sprintf(buf,"/usr/bin/ls -la /var/mail/%s",usuario);
system(buf);
```

    - Variables de entorno, especialmente “IFS”
      - Borrar el entorno actual y fijarlas de forma explícita
    - Funciones “system()”, “popen()”, “exec*()”
      - Limpiar metcaracteres y usar PATH completo

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Entradas de usuario
    - El usuario es el enemigo
      - Paranoia
      - Validación
    - Resolución inversa
      - Obtenemos la lista de inversas para esa IP
      - Obtenemos la lista de IPs para esas inversas
      - Debe existir alguna coincidencia

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Entradas de usuario (Cont.)
    - “argc”=0
    - Manipulación de descriptores estándar (0-2)
    - Directorio actual ilegible o inexistente
    - Señales
    - “alarm()” en curso al lanzar el proceso
    - Ejecución paso a paso o “tracing”
      - Los kernel modernos no permiten “tracing” de un SUID
    - El proceso puede tener hijos que no ha creado
      - Pipes

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Propagación entre procesos
    - Un proceso cuidadoso puede invocar procesos que no lo sean, o que sean maliciosos
      - Filtrado de descriptores de fichero
        - » Configuración “ioctl()” y “fcntl()” para que se cierren al hacer un “exec*()”
    - Cifrado y autentificación en los intercambios entre máquinas diferentes

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Librerías dinámicas
    - Variables de entorno “LD_LIBRARY_PATH” y “LD_PRELOAD”
      - Posibilidad de ejecutar código arbitrario en procesos SUID
      - La mayoría de los “dinamic linker” ignoran estas variables si el proceso es SUID. Al menos si no contienen ningún “/”
      - Pero existe el riesgo de que ese proceso invoque otros procesos no SUID
        - » El proceso debe “limpiar” sus variables de entorno
        - » Idealmente esta limpieza debería hacerla el “dinamic linker”
      - Los procesos SUID deberían estar enlazados estáticamente
        - » gcc -static

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Librerías dinámicas (Cont.)
    - Algunos sistemas tienen variables de precarga de librerías dinámicas adicionales
      - “NLSPATH” en Solaris
  - Verificación exhaustiva y programación conservativa
    - Comprobar el resultado de todas las funciones
      - “Malloc()”
      - Descriptores de ficheros
      - Disco lleo o cuota superada
      - Incapacidad para hacer “fork()”

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Verificación exhaustiva y programación conservativa (Cont.)
    - Actuar como un atacante intentando localizar un error
    - El código privilegiado debe ser lo más corto y simple posible
    - Librarse de los privilegios cuanto antes
    - Los paths a comandos externos deben ser absolutos
    - Fijación coherente del directorio de trabajo
    - Evitar “deadlocks”
      - Un proceso externo puede “morir” o funcionar mal

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Verificación exhaustiva y programación conservativa (Cont.)
    - Evitar “race conditions”
      - Múltiples instancias del proceso en paralelo
      - Caso típico de “CGIs”
        - » Limitación de carga
        - » Limitación de concurrencia
    - Compilar con “warnings” al máximo
      - Usar herramientas con “lint”, “purify”, etc.
    - Documentación detallada y precisa

Programación Segura y CHROOT

Programación Segura

- Reglas Básicas:
  - Verificación exhaustiva y programación conservativa (Cont.)
    - Regla de Oro: “Si todo va mal, no lo arregles”
      - Si un proceso no puede recuperarse de forma segura de un error, debe dejar constancia y morir de la forma menos traumática posible
        - » Eliminación de ficheros temporales
        - » Eliminación de bloqueos
      - Ante situaciones límite, la única actuación razonable es delegar la responsabilidad en el administrador de sistemas

Muchas gracias por su atención

Madrid, 6 de Abril de 2.000