5. Codificación Segura de Aplicaciones II

5.1. Introducción y objetivos

La posibilidad de desarrollar software no confiable es atribuible a todas las partes involucradas en el ciclo de vida de desarrollo de software y, en concreto, a los desarrolladores de software que escriben el código, quienes desempeñan un papel vital en el desarrollo de software seguro libre de vulnerabilidades.

La formación de los programadores en esta materia es muy importante, porque si se conoce qué tipo de defectos de seguridad se pueden cometer en un lenguaje de programación, se lograrán evitar muchos de ellos desde el principio. Además, se hacen una serie de recomendaciones para tener en cuenta a la hora de la implementación, tanto de buenas como de malas prácticas.

El encontrar *bugs* de manera temprana conlleva una reducción del coste económico, en varios órdenes, con respecto a su descubrimiento en fases posteriores. Este hecho tiene que incitar a las organizaciones a poner especial interés en esta actividad y encuadrarla en los procesos y procedimientos de desarrollo.

De todas formas, inevitablemente, siempre se dejarán defectos en el código una vez terminada la fase de desarrollo; es por ello por lo que todavía se puede hacer más por eliminar la mayor parte de defectos posible al realizar un análisis de código fuente durante la codificación y al finalizarla.

Los principales objetivos de este tema son los siguientes:

* Estudiar una serie de recomendaciones de buenas y malas prácticas de implementación.
* Conocer los principales errores de programación que derivan en vulnerabilidades explotables, relativos a *integers overflows*, errores y excepciones, privacidad y confidencialidad y programas privilegiados.
* Prepararse para poder analizar el código en base al conocimiento de los defectos de programación que se pueden cometer.

Si se entienden bien los defectos que se pueden cometer, se estará en disposición de poder analizar el código para encontrarlos y, posteriormente, corregirlos.

5.2. Integers overflows

Todos los tipos de representación de números enteros tienen una limitada capacidad debido a que se representan con un limitado número de bits. Esto es un hecho que normalmente se les olvida a los programadores, quienes pueden ocasionar errores de este tipo al intentar almacenar un valor demasiado grande en la variable asociada y generar que sobrepase esos límites inferiores y superiores, o un valor grande positivo se convierta en un valor grande negativo y viceversa, lo que provoca un resultado inesperado (valores negativos, valores inferiores, etc.).

En el documento de INCIBE (2012) se indica que:

«Este tipo de error puede tener consecuencias graves cuando el valor que genera el *integer overflow* es resultado de alguna entrada de usuario (es decir, que puede ser controlado por el mismo) y cuando, de este valor, se toman decisiones de seguridad, se toma como base para hacer asignaciones de memoria, índice de un array, concatenar datos, hacer bucles. etc. Este error ha sido el caso de vulnerabilidades como CVE-2001-014459 en SSH1 y que permitiría a un atacante ejecutar código arbitrario con los privilegios del servicio ssh; o algunos más recientes como CVE-2011-2371 en Mozilla Firefox».

El código de la siguiente figura muestra un ejemplo de *integer overflow* que se da con una función que reserva memoria para un *string* con un tamaño que se lee de un fichero. Antes de reservar el *buffer,* varias líneas de código comprueban que el tamaño no es mayor a 1024 y decrementa su valor en uno para no copiar el carácter de la nueva línea del fichero.

Sin embargo, dependiendo del valor de readamt, decrementar su valor podría causar resultados erróneos; si se declara sin signo y un atacante ocasiona que getstringsize() devuelva 0, malloc() será invocada con 4 294 267 295, que es el valor más grande en una representación de 32 bit y la operación fallará por insuficiencia de memoria.

A close-up of a card

Description automatically generated

Figura 1. *Integer overflow.* Fuente: Chess y West, 2007.

Otro error de tipo aritmético es el siguiente:

A screenshot of a computer code

Description automatically generated

Figura 2. Error aritmético. Fuente: presentación de Microsoft «Basic of Secure Development Test».

Errores de truncado

Otro tipo de problemas que se pueden presentar en operaciones con números es que se pueden truncarcuando un tipo de dato entero, con gran cantidad de bits, se convierte a otro con menor cantidad de bits. El lenguaje C, por ejemplo, desecha los bits de mayor orden; si un entero con signo se convierte desde otro más pequeño en número de bits, los bits extra se rellenan de tal forma que el número nuevo conserve el mismo signo.

Esto significa que, si un número negativo es convertido a un tipo de datos más grande, su signo será el mismo, pero su valor aumentará considerablemente, porque sus bits más significativos se modificarán. Si un programador no entiende cuándo y cómo las conversiones tienen lugar, pueden surgir vulnerabilidades.

Conversiones entre enteros con signo y sin signo

Suponiendo que se tienen 4 bits para representar enteros con y sin signo, se tendría:

A screenshot of a computer

Description automatically generated

Figura 3. Error de truncado que ocurre cuando un entro de 8 bits es truncado a uno de 4 bits y una de extensión de signo cuando un entero de 4 bits es truncado a un entero con signo de 8 bits. Fuente: Chess y West, 2007.

No pueden representar el mismo número de valores y solo algunos valores pueden ser convertidos desde un dato de tipo sin signo a otro con signo, y viceversa, sin cambiar el significado, mientras otros no. En el caso de valores positivos, el problema es que el 50 % de los valores sin signo requiere poner a uno el bit más significativo.



Figura 4. Error que ocurre cuando un entero de 4 bit es truncado con signo a sin signo y viceversa. Fuente: Chess y West, 2007.

En el código de la siguiente figura se procesa la entrada de usuario de una serie de *strings* de longitud variable almacenadas en una posición . Los primeros dos octetos de entrada dictan el tamaño de la estructura a ser procesada. El programador ha puesto una cota superior en el tamaño del *string:* la entrada será procesada solo si es menor o igual a 512.

El problema es que está declarado como tipo *short* (con signo); entonces, la comprobación contra la longitud de *string* máxima es una comparación con signo. Más tarde, , implícitamente, se convierte a un número entero sin signo en la llamada a . Si el valor de es negativo, aparecerá que el *string* tiene un tamaño apropiado (se toma la alternativa *if*), pero la cantidad de memoria copiada por será muy grande (mucho más grande que 512) y el atacante será capaz de desbordar *buff* con los datos en

A computer code on a white background

Description automatically generated

Figura 5. *Integer overflow* causado por conversión signo-sin signo. Fuente: Chess y West, 2007.

5.3. Errores y excepciones

Los problemas de seguridad, muchas veces, comienzan cuando un atacante busca la forma de violar las expectativas del programador, quienes es usual que den menos importancia a las condiciones de error y manejo de excepciones. Por el contrario, es uno de los aspectos que tienen en cuenta los atacantes:

* C usa códigos de error que generan las funciones.
* C++ usa una combinación de códigos de error y excepciones sin comprobar.
* Java usa excepciones comprobadas.

Manejar errores mediante códigos de retorno

El lenguaje C usa el valor de retorno de una función para comunicar el éxito o el fracaso. Esta aproximación tiene varios efectos secundarios no deseables:

* Hace fácil no tratar los errores: simplemente, no haciendo caso del valor de retorno de una función.
* Conectar la información del error con el código para manejarlo hace los programas más difíciles de leer. La lógica de manejo del error se mezcla con la lógica para manejar casos esperados. Esto aumenta la tentación de ignorar errores.
* No hay ninguna convención universal para comunicar la información de error; por lo tanto, los programadores deben investigar el mecanismo que se maneja en cada función.

Vale la pena notar que estos son algunos motivos para que los diseñadores de C++ y Java incluyeran excepciones como un elemento del lenguaje. El efecto que tiene ignorar un código de error, como en el ejemplo de la siguiente figura, es que el programador espera que el *buff* termine en nulo; pero, si no es así, debido, por ejemplo, a un error I/O, como no se notifica el error en la siguiente llamada a , puede resultar en *buffer* *overflow* (*buff* no tiene terminador nulo).



Figura 6. No se comprueba el código de retorno. Fuente: Chess y West, 2007.

El código Java de la siguiente figura recorre un conjunto de usuarios al leer un fichero de datos privado de cada usuario. El programador asume que los archivos tienen siempre exactamente 1 kb de tamaño y, por lo tanto, no hace caso del valor de retorno de . Si un atacante puede crear un archivo más pequeño, el programa reutilizará el resto de los datos del archivo anterior, haciendo que el código trate los datos de otro usuario como si los datos pertenecieran al atacante.

A computer code with black text

Description automatically generated

Figura 7. No se comprueba el código de retorno del método read(). Fuente: Chess y West, 2007.

Manejo de errores mediante excepciones

Las excepciones solucionan muchos problemas de manejo de errores. Aunque sea fácil ignorar el valor de retorno de una función simplemente omitiendo la comprobación del código, no hacer caso de una excepción comprobada requiere justo lo contrario: un programador tiene que escribir el códigoexpresamente para ignorarlo. Las excepciones también permiten la separación entre el código que sigue un camino esperado y el código que maneja circunstancias anormales.

Las excepciones tienen dos versiones, la diferencia es relativa a si el compilador usa análisis estático para asegurar que la excepción es manejada:

* *Cheked.* Si un método declara que lanza una excepción *checked,* todos los objetos que lo utilizan deben manejar la excepción o declarar que también lo lanzan. Esto fuerza al programador a pensar en excepciones *checked*en cualquier parte donde pudieran ocurrir. Los compiladores de Java hacen cumplir las reglas en cuanto a excepciones *checked,* y la biblioteca de clases de Java hace un empleo liberal de excepciones *checked.* La clase Java java.lang.Exception es una excepción *checked.*
* *Unchecked.* Las excepciones *unchecked*no tienen que ser declaradas o manejadas. Todas las excepciones en C++ son *unchecked,* que quiere decir que un programador podría completamente ignorar las excepciones y el compilador no se quejaría. Java ofrece excepciones *unchecked,* también. Sin embargo, el peligro con las excepciones *unchecked* consiste en que los programadores podrían ser inconscientes sobre que una excepción puede ocurrir en un contexto dado y omitir el manejo del error apropiado. Por ejemplo, la función , de Windows, asigna memoria en el *stack.* Si una petición de asignación es demasiado grande para el espacio de *stack*disponible, lanza una excepción de desbordamiento de *stack unchecked.* Si la excepción no es capturada, el programa fallará, lo que potencialmente posibilite un ataque de denegación de servicio. Siempre que sea posible, se deberá:
  + Capturar las excepciones en el nivel superior, las excepciones se pueden capturar o volver a lanzar.
  + No incluir sentencias *return* en el bloque *finally* de manejo de la excepción, el programa se comporta como si ninguna excepción hubiera ocurrido.
  + Capturar solamente lo que se esté preparando para manejar. La captura de todas las excepciones en el nivel superior es una buena idea, pero la captura de las excepciones dentro de un programa después de una gran cantidad de código puede causar problemas.
  + No dejar excepciones *checked* con el bloque *catch* vacío. Si no se sabe qué hacer para tratar una excepción capturada, se puede lanzar una excepción que, por lo menos, notifique que algo raro ha ocurrido:



Figura 8. Excepción con el bloque*catch* relleno.

Desasignar los recursos que no se van a utilizar en adelante

No liberar recursos como memoria, ficheros, conexiones de red, etc., puede causar problemas de rendimiento. Este concepto se conoce como *resource leaks,* que supone un riesgo de seguridad, pues favorecen claramente ataques de denegación de servicio. En cualquier lenguaje que se utilice se tiene que procurar desasignar cualquier recurso que no se vaya a utilizar en adelante en el resto de la aplicación. En el código de la siguiente figura se muestra un ejemplo de *memory leak,* que es, también, un subconjunto de esta categoría.

A white background with black text

Description automatically generated

Figura 9. No se libera la memoria de *buff.* Fuente: Chess y West, 2007.

5.4. Privacidad y confidencialidad

En este apartado se va a analizar cómo hay que manejar las contraseñas, tanto las que utilizan los usuarios para autenticarse *(inbound passwords),*como las que utiliza un programa que actúa como cliente para autenticarse frente a un servicio final *(outbound passwords),* como una base de datos o un servidor LDAP. *Intbound passwords* pueden ser *hashed;* sin embargo, las *outbound passwords* deben ser accesibles al programa, en texto claro.

Para profundizar sobre *intbound passwords,* se recomienda la lectura de este libro: <http://www.cl.cam.ac.uk/~rja14/book.html>

Mantener las *passwords* fuera del código fuente

Se deberán cifrar las contraseñas con algoritmo seguro y almacenarlas fuera del código. La siguiente figura muestra un trozo de código que muestra un mal uso de*password* codificada dentro de código Java.

A screen shot of a computer code

Description automatically generated

Figura 10. *Password hardcoded.* Fuente: Chess y West, 2007.

Una buena estrategia consiste en:

* Cifrar las *passwords* con una implementación pública de algoritmos criptográficos robustos y almacenarlas cifradas en un fichero de configuración: almacenar la clave necesaria para descifrar la *password* en un fichero separado donde la aplicación pueda acceder, pero no la mayoría de los administradores de sistemas.
* Usar *passwords* robustas: usar generadores de números aleatorios seguros para las *passwords* de conexiones *outbounds.* Hay un compromiso entre los conceptos de robustez y de facilidad a la hora de elegir una *password.* Las *outbound passwords* no requieren ser recordadas, deben ser largas y aleatoriamente generadas usando un método seguro de generación de números aleatorios.
* Números aleatorios: si se consigue una buena fuente de generación de números aleatorios, se tiene una buena fuente de secreto. La generación de nuevos secretos es una parte crítica en el establecimiento de muchas clases de relaciones de confianza. Los números aleatorios son importantes en la criptografía.

Los computadores son máquinas intrínsecamente deterministas y no son una fuente de aleatoriedad. Si un programa requiere un volumen grande de valores realmente arbitrarios, hay que pensar en comprar un generador de números aleatorios hardware*.* Los generadores de números pseudo aleatorios caen en dos categorías:

* PRNGS estadísticos: producen valores que son uniformemente distribuidos a través de un rango especificado, pero son calculados usando algoritmos que crean una corriente de valores fáciles de reproducir y triviales de predecir.
* PRNGS criptográficos: usan algoritmos criptográficos para ampliar la entropía de una semilla de valor aleatorio en una corriente de valores imprevisibles que, con alta probabilidad, no pueden ser distinguidos de un valor realmente arbitrario. Los PRNGS criptográficos son convenientes cuando la imprevisibilidad es importante, incluyendo los usos siguientes:
  + Criptografía.
  + Generación de contraseñas.
  + Aleatoria de puertos para seguridad.
  + Identificadores externos únicos (identificadores de sesión).

Los errores más comunes y obvios relacionados con números aleatorios ocurren cuando se usa PRNG estadístico en una situación que exige que los valores sean sumamente imprevisibles y, por lo tanto, producidos por PRNG criptográfico. Otro error ocurre cuando un PRNG no se crea con entropía insuficiente, causando que la corriente de números sea predecible.

* Números aleatorios en Java. Java tiene métodos de generación de números aleatorios estadísticos y criptográficos:
  + Método estadístico:

A screen shot of a computer code

Description automatically generated

Figura 11. Función aleatoria no segura. Fuente: Chess y West, 2007.

* Clase con métodos criptográficos: Java.security.SecureRandom. El algoritmo, por defecto, y las fuentes de entropía usadas por SecureRandom, son opciones buenas para la mayor parte de aplicaciones de software y los usuarios raras veces deberían tener que proporcionar su propia semilla.
* Números aleatorios en C y C++. C/C++ tienen métodos de generación de números aleatorios estadísticos y aleatorios:
  + Funciones estadísticas: evitar utilizar
  + Funciones criptográficas: Sobre plataformas de Microsoft, (del API de seguridad de Microsoft CRT) y (de Microsoft CryptoAPI) proporcionan el acceso a valores criptográficamente fuertes arbitrarios producidos por el subyacente

### **5.5. Programas privilegiados**

La mayor parte de los programas se ejecutan con un **conjunto de privilegios** heredados del usuario con el que se ejecutan. Por ejemplo, un editor de textos puede mostrar solo los archivos que su usuario tiene el permiso de lectura.

Los programas privilegiados tienen privilegios adicionales que les permiten realizar operaciones que sus usuarios, de otra forma, no realizarían.

* Cuando está **escrito correctamente,** un programa privilegiado concede a usuariosregulares unacantidad limitadade accesos a algún recurso compartido, como la memoria física, un dispositivo de hardware, o archivos especiales, como el archivo de contraseñas o la base de datos de correo.
* Cuando está **escrito incorrectamente,** un programa vulnerable privilegiado deja, a los atacantes, campo para actuar. En el peor caso, esto puede permitir un ataque de escalada de privilegios vertical, en el que el agente malicioso gana acceso incontrolado a los privilegios elevados del programa.

Otro tipo común de ataque, conocido como **escalada de privilegios horizontal,** ocurre cuando un atacante engaña a los mecanismos de control de acceso de una aplicación para tener acceso a recursos que pertenecen a otro usuario. La siguiente figura ilustra la diferencia entre la escalada de privilegio vertical y horizontal.

A diagram of a diagram

Description automatically generated

Figura 12. Ataques de elevación de privilegios vertical y horizontal. Fuente: Chess y West, 2007.

### **Principio de mínimo privilegio**

Evidentemente, es uno de los primordiales principios de diseño que, a la hora de implementar el código, hay que ser lo más consecuente posible con lo que se especificó durante la fase de análisis de requisitos. La motivación detrás de este principio está clara: **el privilegio es peligroso.** Cuantos más privilegios tiene un programa, mayor daño potencial puede causar.

Un conjunto reducido de acciones posibles disminuye el riesgo potencial de un programa. La mejor gestión de privilegios es ninguna gestión de privilegio. El modo más fácil de prevenir vulnerabilidades relacionadas con el privilegio es diseñar los sistemas que no requieren componentes privilegiados.

Cuando se aplica al código, el **principio de mínimos privilegios** implica dos cosas:

* Los programas no deberían requerir que sus usuarios tengan **privilegios extraordinarios** para realizar tareas ordinarias.
* Los programas privilegiados deberían **reducir** al mínimo la cantidad de daño que pueden causar cuando algo va mal.

Dependiendo de la **funcionalidad del programa,** y de sus necesidades de privilegio, se pueden elevar y bajar sus privilegios en puntos diferentes de su ejecución. Los programas requieren privilegios para una variedad de motivos:

* Comunicarse directamente al hardware.
* Modificar el comportamiento del OS.
* Enviar a señales a ciertos procesos.
* Trabajar con recursos compartidos.
* Apertura de puertos de red con baja numeración.
* Cambio de la configuración global (el registro y/o archivos).
* Protecciones de sistema de archivos principales.
* Instalación de nuevos archivos en directorios de sistema.
* Puesta al día de archivos protegidos.
* Tener acceso a archivos que pertenecen a otros usuarios.

Las transiciones entre estados privilegiados y no privilegiados definen el **perfil de privilegio** de un programa. El perfil de privilegio de un programa puede ser colocado en una de las siguientes **cuatro clases:**

* **Programas normales** que corren con los mismos privilegios que sus usuarios. Ejemplo: Emacs.
* **Programas de sistema** que corren con privilegios de root para la duración de su ejecución. Ejemplo: Init.
* Programas que **necesitan privilegios de** root para usar un conjunto fijo de recursos del sistema cuando se ejecutan al principio. Ejemplo: Apache httpd, que necesita el acceso de root para usar puertos bajos numerados.
* Programas que **requieren privilegios de** root intermitentemente a lo largo de su ejecución. Ejemplo: un demonio de FTP que usa puertos de baja numeración intermitentemente en todas las partes de su ejecución. En la Figura 13, se representa un esquema de los ejemplos mencionados y cuándo necesitarían privilegios de root.

A line drawing of a person

Description automatically generated with medium confidence

Figura 13. Ejemplo de utilización de privilegios de rooten la ejecución de varios programas. Fuente: Chess y West, 2007.

El concepto de seguridad que la mayoría de la gente debería repetir es **«no confiar en nada».** Aunque esto sea un ideal que es imposible de realizar, no hay nada que debería ser tan perseguido enérgicamente como un **programa privilegiado.**

Los atacantes usarán, para montar, ataques de escalada de privilegios:

* Los argumentos al programa.
* El entorno de ejecución del programa.
* Los recursos de que el programa depende (sistema de ficheros).

### **Manejo de los privilegios**

Los autores advierten que la mayoría de los programas privilegiados realizan **tres operaciones claves** sobre sus privilegios:

* Eliminar privilegios con la intención de recuperarlos.
* Eliminar privilegios permanentemente.
* Recuperar privilegios previamente almacenados.

Otra aproximación de gestión de privilegios, de acuerdo con el principio de **mínimo privilegio,** es dividir un programa en componentes privilegiados y no privilegiados. Si las operaciones que requieren privilegios pueden ser categorizadas en un proceso separado, esta solución puede reducir bastante el trabajo requerido para asegurar que los privilegios se administran adecuadamente.

Además de la gestión explícita de privilegios y el control de acceso basado en el ID del usuario, se puede restringir los privilegios de un programa limitando su **visión del sistema de archivos** al usar, por ejemplo, la función de sistema en sistemas Linux. Después de una correcta invocación a , un proceso no puede tener acceso a ningún archivo fuera del árbol de directorio especificado para esa llamada. Se denomina a tal entorno **jaula** chroot y se usa comúnmente para prevenir la posibilidad de que se pueda corromper un proceso y usarlo para tener **acceso a archivos protegidos.** Por ejemplo, muchos servidores de FTP se ejecutan en jaulas chroot para prevenir a un atacante que descubre una nueva vulnerabilidad en el servidor. Esta tentativa de crear una jaula chroot tiene**cuatro problemas:**

* Llamar a no afecta a ningún descriptor de archivo que está actualmente abierto; entonces, se puede apuntar a archivos fuera de la jaula chroot. Para estar seguro, se deberá cerrar cada archivo abierto antes de la llamada .
* La llamada a la función no cambia el directorio actual de trabajo del proceso; entonces, rutas relativas como todavía pueden referirse a recursos del sistema de archivos fuera de la jaula chroot, después de que se ha llamado a la función . Siempre, antes de una llamada a , se incluye una llamada a . Verificar que cualquier error de manejo de código entre la llamada a y  no abre ningún archivo y no devuelve el control a ninguna otra parte del programa que no sean rutinas de parada. Las funciones y deberían estar tan cerca como sea posible, para limitar la exposición a esta vulnerabilidad.
* Es también posible, y recomendable, crear una jaula chroot apropiada que llame a primero. La llamada a , entonces, puede tomar un argumento constante , que es más fácil de verificar durante una revisión de código.
* Una llamada a puede fallar, por lo que se deberá comprobar el valor de retorno de para asegurarse del resultado de la llamada.
* Para llamar , el programa debe ejecutarse con privilegios de root. En cuanto la operación privilegiada se ha completado, el programa debería eliminar los privilegios de root y retornar a los privilegios del usuario de invocación. El código en el ejemplo de la Figura 14, sigue ejecutándose como root.

A computer code with text

Description automatically generated with medium confidence

Figura 14. Código de un simple FTP server. Fuente: Chess y West, 2007.

### **Manejo de excepciones en programas privilegiados**

Hay que tener en cuenta una serie de directrices generales para manejar **eventos inesperados y excepciones** que toman una importancia aún mayor en programas privilegiados, donde los ataques son más probables y el coste de un exploit es mayor.

* Comprobar cada condición de error.
* Seguridad sobre robustez: terminación ante errores. No intentar reponerse de errores inesperados o mal entendidos.
* Inhabilitar señales antes de la elevación de privilegios. De esta forma, se evita tener señales que controlan código que se ejecuta con privilegios. Rehabilitar las señales y después volver a los privilegios estándar que tenía el usuario.

### **Ataques de escalada de privilegios**

A veces, un ataque contra un programa privilegiado se realiza por un **usuario legítimo** del sistema. Normalmente, un atacante usa un proceso de dos pasos para obtener privilegios sobre una máquina:

* Intenta **encontrar una debilidad** en un servicio de red o en una cuenta de usuario mal protegida y obtiene acceso a una cuenta con bajos privilegios sobre la máquina.
* Usa este acceso de bajos privilegios para **explotar una vulnerabilidad** en un programa privilegiado y tomar el control de la máquina.

Los ataques de escalada de privilegios pueden tener como objetivo cualquier variedad de vulnerabilidades de software. En este apartado, se tratan las clases de vulnerabilidades que son principalmente un riesgo en programas privilegiados:

* Condiciones de carrera de acceso a archivos.
* Permisos de archivo débiles.
* Archivos temporales inseguros.
* Inyección de comandos.
* Mal uso de descriptores de archivo estándar.

### **Ataque de condiciones de carrera en el acceso a ficheros**

Se encuentran en este grupo las vulnerabilidades conocidas como time-of-check, time-of-use**(TOCTOU).** MITRE clasifica este tipo de vulnerabilidades con el **CWE 367.** Los errores generados como consecuencia de una condición de carrera se producen por el cambio que experimenta el **estado de un recurso** (ficheros, memoria, registros, etc.) cuando un programa intenta verificar una propiedad y, más tarde, toma una decisión, suponiendo que la propiedad no ha cambiado. Los ataques **TOCTOU** sobre vulnerabilidades del sistema de archivos, generalmente, siguen esta secuencia:

A screenshot of a computer

Description automatically generated

Figura 15. Secuencia de ataque TOCTOU. Fuente: elaboración propia.

La siguiente figura muestra el modo que las operaciones en **lpr** podrían intercalarse con la ejecución del código de un agente malicioso de un **ataque con éxito.** Un atacante invoca **lpr** con el argumento y redirecciona el archivo para que apunte al fichero de durante el tiempo que pasa entre que es comprobado y utilizado. En la figura siguiente, se muestra el ejemplo corregido.

A computer code with green text

Description automatically generated

Figura 16. Ejemplo de código con una vulnerabilidad TOCTOU en el acceso a un fichero. Fuente: Chess y West (2007).

En la siguiente tabla se muestra un **esquema de tiempos** con las operaciones que pueden darse intercaladas de un **usuario legítimo y un atacante.** El aprovechamiento de las vulnerabilidades de condiciones de carrera de acceso a un archivo depende del hecho de que los sistemas operativos modernos permiten a muchos programas ejecutarse inmediatamente y no se garantiza que la ejecución de cualquier programa dado no se intercale con otro. Aunque dos operaciones pudieran parecer secuenciales en el **código original** de un programa, cualquier número de instrucciones puede ser ejecutado entre ellas.

A blue and white table with text

Description automatically generated

Tabla 1. Esquema de tiempos de una vulnerabilidad TOCTOU en el acceso a un fichero. Fuente: Chess y West, 2007.

La ventana de vulnerabilidad para tal ataque es el período de tiempo entre cuando una propiedad es **comprobada** y cuando se **usa el archivo.** Los atacantes tienen una variedad de técnicas para ampliar la longitud de esta ventana para hacer los exploitsmás fáciles, como el enviar a una señal al proceso víctima, que hace que ceda la CPU a otro proceso. Incluso, con una pequeña ventana, una tentativa de exploitpuede repetirse hasta que se consiga llevar a cabo.

Hay que tener en cuenta que un ataque de condición de carrera todavía puede darse después de abrir el archivo, si alguna operación posterior depende de una propiedad comprobada anterior a la apertura del archivo.

Por ejemplo, si la estructura pasada a se obtiene antes de que un archivo sea abierto, y una decisión posterior sobre si hay que operar sobre el archivo está basada en un valor leído de la estructura de stat, entonces una condición de carrera TOCTOU puede darse entre la llamada a y la llamada .

### **Ficheros temporales inseguros**

Algunas funciones intentan crear un único nombre de archivo simbólico que el programador puede usar para crear un archivo temporal. Otras funciones, además, van un poco más lejos e intentan abrir un **descriptor de archivo.**

Ambos tipos de funciones son susceptibles a una variedad de vulnerabilidades que podrían permitir a un atacante **inyectar contenido malévolo** en un programa, hacer que el programa realice operaciones malévolas en nombre del atacante, dar acceso al atacante a información sensible que el programa almacena sobre el sistema de archivos, o permitir un ataque de **denegación** de servicio contra el programa.

Existen dos opciones para crear archivos temporales de **forma segura:**

* **Almacenar** los archivos temporales bajo un directorio que no es públicamente accesible; de esta forma, se elimina toda la discusión con respecto a ataques.
* **Generar** los nombres de archivo temporales que sean difíciles de adivinar usando un generador de números criptográficamente seguros pseudoaleatorios (PRNG) para crear un elemento aleatorio en cada nombre del archivo temporal.

### **Inyección de comandos**

Los datos introducidos por un usuario pueden **alterar el propósito** de un comando, consulta, etc., a ejecutar en el entorno de la aplicación, base de datos, etc. La secuencia de actuación de un atacante sería:

* Un atacante **modifica** el entorno de un programa.
* El programa ejecuta un comando que usa el **entorno modificado** sin especificar una ruta absoluta.
* Ejecutando el comando, el programa da **privilegios** a un atacante o la capacidad que, de otra manera, no tendría.

Para entender mejor el **impacto potencial** del entorno sobre la ejecución de comandos en un programa, hay que considerar la vulnerabilidad en la utilidad **ChangePassword,** basada en CGI-Web, que permite a los usuarios **cambiar sus contraseñas** en el sistema (Berkman, 2004). El proceso de actualización de contraseña incluye ejecutar make en el directorio /var/yp. Es importante notar que debido a que el programa pone al día registros de contraseña, debe ser instalado con setuid root. El programa invoca make como sigue:

Como el programa no especifica una ruta absoluta para sus variables de entorno antes de la invocación del comando, un atacante puede aprovecharse del programa ejecutarlo localmente para modificar la variable de entorno con el fin de apuntar su código malicioso llamado make y luego ejecutar un script CGI desde un intérprete de comandos shell. El atacante puede ejecutar código arbitrario con privilegios de root.

En programas seguros privilegiados, debería contener, únicamente, directorios de root, ya que es probable que el programa tenga que ejecutar utilidades de sistema con acceso a directorios controlados por su propietario. No incluir en el directorio actual (.) rutas relativas que un atacante podría ser capaz de manipular. En la mayor parte de los entornos, debería ser puesto a su valor por defecto,

### **Descriptores de ficheros**

Los **descriptores de archivo estándar** en sistemas Linux stdin (FD 0), stdout (FD 1), y stderr (FD 2) son, por lo general, abiertos por el terminal y usados tanto explícita como implícitamente por funciones como . Algunos programas remiten uno o varios de estos descriptores a streams de datos diferentes, como un archivo de log, para reutilizar su comportamiento implícito, de forma que sea más adecuado al diseño del programa, pero la mayor parte de estos nunca cambia sus valores.

Como con umask y con variables de entorno, un **proceso hijo** hereda descriptores de archivo estándar de su padre. Los atacantes pueden aprovechar este hecho para hacer que un programa vulnerable lea o escriba en archivos del sistema, cuando en realidad se espera interactuar con el terminal.

A white rectangular object with black text

Description automatically generated

Figura 17. Código vulnerable a ataques a su descriptor de ficheros estándar. Fuente: Chess y West, 2007.

El ejemplo muestra un programa que, como parte de su actividad normal, abre el archivo en modo lectura-escritura y después, condicionalmente, pasa su primer argumento a usado por Se podría creer que estas dos operaciones no están relacionadas y no tienen ningún impacto la uno sobre la otra, pero el código realmente contiene un bug.

Un ejemplo de un **ataque sobre la vulnerabilidad del ejemplo anterior** se puede ver en la figura siguiente, el código del primer ejemplo cierra (descriptor de archivo 2) y luego ejecuta el programa vulnerable. Cuando el programa abre devolverá el primer descriptor de archivo disponible que, debido a que el programa del atacante cerró , será el descriptor de archivo 2.

Ahora, en vez de inofensivamente pasar su primer parámetro al terminal, el programa escribe la información a . Aunque esta vulnerabilidad fuera perjudicial, independientemente de la información escrita, este ejemplo es, en particular, fatal, porque el atacante puede escribir una entrada válida a . En los sistemas que almacenan entradas de contraseña en se daría acceso con privilegios de rootal atacante sobre el sistema.

A computer code with red text

Description automatically generated

Figura 18. Ejemplo de ataque a vulnerabilidad del ejemplo anterior. Fuente: Chess y West, 2007.

Los programas privilegiados deben asegurar que los tres primerosdescriptores de archivo se abran a **archivos seguros,** conocidos antes del comienzo de su ejecución. Un programa puede: abrir los tres primeros descriptores de archivo al terminal deseado, abrir archivos streams (si el programa tiene la intención de usar funciones que implícitamente dependen de estos valores), o abrir tres veces para asegurar que los tres primeros descriptores de archivo se usan. El ejemplo de la figura siguiente ilustra la segunda opción.

A screenshot of a computer code

Description automatically generated

Figura 19. Código que soluciona vulnerabilidades de descriptor de ficheros asegurando el uso de los tres primeros descriptores de ficheros. Fuente: Chess y West, 2007.

**DevOps** unifica los dos mundos de las tecnologías de la información, desarrollo y operaciones. Propone ciclos completos y automatizados desde el desarrollo hasta el despliegue del producto. De ahí que lenguajes de implementación de infraestructura como Puppet o Ansible hayan encajado tan bien en este paradigma. La idea práctica es introducir la **capa de seguridad** en DevOps para llegar a evolucionar al término DevSecOps. En este vídeo (DevSecOps) se realiza una introducción a DevSecOps.

El **IEEE Center for Secure Design (CSD)** definió una lista de los diez defectos de diseño más importantes del software y las técnicas de diseño para evitarlos. En este vídeo (Principales 10 debilidades de diseño) se presentan y estudian los diez citados defectos de diseño.