4. Codificación Segura de Aplicaciones I

4.1. Introducción y objetivos

Las principales características que diferencian a un desarrollador que programa teniendo en cuenta las reglas de codificación segura, de uno que no, son el conocimiento, intención y precaución.

El profesional con conocimiento de seguridad de software es consciente de que una de las maneras de evitar problemas de seguridad en las aplicaciones es, aparte de requisitos adecuados y uso de principios de diseño seguro, el seguimiento por parte de los programadores de prácticas de codificación seguras, que eviten la codificación de errores y debilidades explotables que deriven en vulnerabilidades explotables por los diferentes tipos de agentes maliciosos.

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

Los principales objetivos de este tema son los siguientes:

* Estudiar una serie de recomendaciones de buenas y malas prácticas de implementación.
* Conocer los defectos más comunes que se pueden cometer al codificar en lenguajes como C y Java.
* Prepararse para poder analizar el código en base al conocimiento de los defectos de programación que se pueden cometer.

4.2. Prácticas de codificación segura

Para iniciarse en la implementación segura de código es conveniente empezar con una serie de recomendaciones generales para tener en cuenta, como la línea a seguir en la profundización y conocimiento de esta disciplina. A continuación, se presentan una serie de recomendaciones que caen en las siguientes categorías:

A screenshot of a computer program

Description automatically generated

Figura 1. Buenas y malas prácticas de codificación. Fuente: elaboración propia.

Estos pueden ser solo algunos de los aspectos más generales para tener en cuenta a la hora de desarrollar y cómo se verá más adelante. A la hora de hablar de defectos de implementación de código hay que tener en cuenta los defectos particulares que se pueden cometer con el lenguaje de programación que se está utilizando, el compilador que se utiliza y otrosoftware de terceros que seguramente no se tendrá posibilidad de verificar, pero que formará parte del sistema y, por lo tanto, podría introducir errores en el sistema que se está implementando.

Los defectos de implementación que se pueden cometer caen dentro de las siguientes categorías:

A screenshot of a phone

Description automatically generated

Figura 2. Defectos de implementación. Fuente: elaboración propia.

4.3. Manejo de la entrada de datos

Una de las medidas de defensa más importantes que un programador puede llevar a cabo es validar todas las entradas que el sistema recibe, ya que son la fuente de algunas de las peores vulnerabilidades de las aplicaciones, como el desbordamiento de *buffer,* inyección de SQL y otras.

Los atacantes pueden manipular los diferentes tipos de datos de entrada con el fin de entregar cargas maliciosas a las aplicaciones. Los datos de entrada siempre deben ser validados por una rutina de validación de entradas, para evitar este tipo de ataques.

Validación de la entrada: conjunto de actividades que aceptan la entrada sin confiar en ella y comprueban su validez, limitándola solo a los valores que se sabe que son aceptables.

Validación de toda la entrada

Estas palabras serán repetidas durante todo el apartado: validar todas las entradas. Hay que definir la entrada detalladamente y pensar más allá de los datos que un usuario puede enviar. Si una aplicación consiste en más de un proceso, se deberá validar la entrada de cada proceso, incluso si aquella entrada llega de otra parte de la aplicación. Hay que validar la entrada incluso si llega por una conexión segura, de una fuente confiable o está protegida según permisos de archivo estrictos. Las rutinas de validación de entrada se pueden dividir en dos grupos principales:

* Comprobaciones de sintaxis que prueban el formato de la entrada y a menudo puede ser desacoplada de la lógica de aplicación y aplicada cerca del punto donde los datos entran en el programa.
* Comprobaciones semánticas que determinan si la entrada es apropiada, considerando la lógica de la aplicación y la función. Por lo general, tienen que aparecer junto a la lógica de aplicación, porque las dos están estrechamente relacionadas.

Se tendrá que realizar la validación de entrada no solo sobre la entrada de usuario, sino también sobre datos de cualquier fuente externa al código del sistema que está siendo desarrollado. Esta lista debería incluir, pero sin limitarse a ella, lo siguiente:

* Parámetros de línea de comando-
* Archivos de configuración.
* Datos recuperados de una base de datos.
* Subida de archivos.
* Variables de entorno.
* Importaciones de archivos planos.
* Servicios de red.
* Valores de registro.
* Propiedades de sistema.
* Archivos temporales.
* Variables de entorno.
* Direcciones URL e identificadores URI.
* Referencias de otros nombres de archivo.
* Encabezados Hyper Text Transfer Protocol (HTTP).
* Parámetros HTTP GET.
* Campos de formulario (especialmente los ocultos).
* Listas de selección.
* Listas desplegables.
* *Cookies* y comunicaciones Java *applets.*

A continuación, se verá un ejemplo de estos tipos de errores por ausencia de validación en la entrada que conduce una vulnerabilidad conocida como inyección de SQL. La Versión 2.1.9, Hiberne (Chess y West, 2007), un paquete popular *open source,* contiene un ejemplo excelente de qué no hacer con la entrada de línea de comando. La versión Java de la herramienta SchemaExport de Hibernate acepta un parámetro de línea de comando llamado -- delimiter , que se utilizaba para separar los comandos SQL en los *scripts* que generaba. El código se muestra ahora de una forma simplificada.

A close-up of a computer code

Description automatically generated

Figura 3. Ejemplo de no validación de la entrada que puede ocasionar una vulnerabilidad de inyección SQL. Fuente: Chess y West, 2007.

La opción existe para que un usuario pueda especificar el separador que debería aparecer entre sentencias SQL. Valores típicos podrían ser un punto y coma o un retorno de carro. Pero el programa no coloca ninguna restricción contra el valor del argumento; por lo tanto, a través de un parámetro de línea de comando se puede escribir cualquier cadena que se quiera en la sentencia SQL generada, incluyendo órdenes de SQL adicionales.

Por ejemplo, si una simple SELECT fue generada con , esto generaría un *script* para ejecutar la sentencia siguiente:

Pero si la misma consulta fue emitida con la opción malévolageneraría una sentencia que limpia a fondo la tabla *items*con la sentencia siguiente:

Establecer los límites de confianza

Un límite de confianza se puede pensar como una línea dibujada a través de un programa, fuera de la cual no se confía en los datos. Al otro lado de la línea, los datos se asumen seguros para alguna operación particular. El objetivo de la lógica de validación es el de permitir a los datos cruzar el límite de confianza, moverse de la zona no segura a la zona en la que se confía. El modo más fácil de cometer este error es el de permitir datos confiables y no confiables mezclados en la misma estructura de datos.

El ejemplo de la figura siguiente demuestra un problema común de frontera de confianza en Java. Los datos no confiables llegan a una petición HTTP y el programa almacena los datos en un objeto de sesión HTTP sin realizar primero una validación. Como el usuario directamente no puede tener acceso al objeto de sesión, los programadores típicamente creen que pueden confiar en la información almacenada en el objeto de sesión. Combinando datos validados y no validados en la sesión, este código viola un límite implícito de confianza.

A white background with black text

Description automatically generated

Figura 4. Ejemplo de violación de los límites de confianza. Fuente: Chess y West, 2007.

Realizar fuerte validación de entrada

Una correcta aproximación a la validación de entrada es comprobar la entrada frente a una lista de valores correctos conocidos. Una buena validación de entrada no intenta comprobar valores específicos no válidos. Se considera mejor la comprobación contra:

Whitelisting: lista de valores conocidos válidos.

Cuando el juego de valores de entrada posibles es pequeño, se puede usar la selección indirecta para conseguir que el *whitelist*sea imposible de evitar, es la primera opción cuando se realiza la validación de entrada. Al final, se pone:

Lista negra *(Blacklinting)*: intento de enumerar todas las entradas posibles inaceptables.

Realizar una buena validación de entrada por defecto

En vez de codificar una nueva solución al problema de validación de entrada cada vez, se deberá diseñar el programa de modo que haya un lugar claro, constante, y obvio para la validación de entrada. La aplicación debería hacer pasar toda la entrada por esta lógica de validación, y dicha lógica debería rechazar cualquier entrada que no pueda ser validada.

Esto requiere la creación de una capa de abstracción sobre las bibliotecas del sistema en la que el programa suele ser introducido. Hacer una buena validación de entrada por defecto, creando una capa de funciones o métodos, se denomina seguridad API. La siguiente figura muestra cómo se interpone el API de seguridad entre el programa y las librerías de sistema.

A diagram of a software development process

Description automatically generated

Figura 5. Las API de seguridad implican un amplio contexto para hacer la validación de entrada. Fuente: elaboración propia.

Comprobar la longitud de la entrada

La lógica de validación *front-end* siempre debería comprobar la longitud de la entrada contra un mínimo y máximo. Las comprobaciones de longitud son, por lo general, fáciles de añadir, pues no se requiere mucho conocimiento sobre el significado de la entrada. Se deberá tener cuidado cuando el programa transforma la entrada antes del tratamiento, pues esta puede hacerse más larga durante el proceso.

A white rectangular object with black text

Description automatically generated

Figura 6. Comprobación de la longitud de un campo. Fuente: Chess y West, 2007.

Comprobar el tamaño de los campos numéricos

Comprobar la entrada numérica tanto contra un valor máximo como contra un valor mínimo como parte de la validación de la entrada. Tener cuidado con las operaciones que podrían ser capaces de llevar un número más allá de su valor máximo o mínimo.

Cuando un atacante aprovecha la capacidad limitada de una variable de tipo entero, el problema es el desbordamiento de este tipo de datos. En C y C ++ el desbordamiento de número entero típicamente supone parte de la preparación para un ataque de desbordamiento de *buffer.* El desbordamiento de número entero en Java no conduce vulnerabilidades de desbordamiento, pero puede causar un comportamiento indeseable.

El mejor modo de evitar problemas de desbordamiento de un número entero es comprobar toda la entrada de este tipo de datos, tanto contra una cota superior como contra una cota inferior. Por ejemplo, malloc() acepta un entero y realiza una conversión implícita a entero si signo, si doAlloc() recibe un número negativo como argumento puede resultar en un intento de reservar una gran cantidad de memoria.

A white rectangular box with green text

Description automatically generated

Figura 7. Comprobación de tipos. Fuente: Chess y West, 2007.

Prevenir vulnerabilidades de meta-caracteres

Si se permite a los atacantes controlar los comandos que se envían a la base de datos, sistemas de ficheros, navegador u otros subsistemas, se podría cambiar la finalidad de este. Se pueden tener problemas serios como consecuencia de ello. Como ejemplos, tenemos los siguientes:

A blue and white checklist with white text

Description automatically generated

Tabla 1. Defectos de inyección. Fuente: elaboración propia.

* Inyección de SQL: la realización de sentencias SQL donde se combinan palabras clave con datos tiene el problema de que alguien malintencionadamente pueda alterar las estructuras de control y, por lo tanto, el significado de la sentencia, cuando la intención era solamente suministrar datos a la sentencia sin alterar las estructuras de control. Un atacante explota este tipo de vulnerabilidades especificando meta-caracteres que tienen un significado especial, como: comillas simples (‘) o dos puntos (..) —peligroso en sistema de ficheros—. Para comandos de *shell:* punto y coma (;), (&&), carácter de nueva línea (/n) para los ficheros de *log.*

En el ejemplo de la siguiente figura se muestra este tipo de vulnerabilidad en una sentencia SQL que se forma concatenando *strings*de tal forma que se deja abierta a un ataque conocido como inyección de SQL:

A screenshot of a computer code

Description automatically generated

Figura 8. Ejemplo de *bug* que deja abierto el código abierto a un ataque de inyección de SQL. Fuente: Chess y West, 2007.

El programador que escribió el código pretendía formar consultas como:

Pero un atacante puede asignar al campo itemname la cadena "name' OR 'a'='a", con lo que la sentencia se convierte en:

La adición de convierte la sentencia pretendida a esta: . Esta sentencia extrae todas las filas de la tabla ítems, lo que posibilita que el atacante las pueda visualizar. Este es un pequeño ejemplo de lo que un atacante puede hacer. Muestra que se podrían también borrar todos los datos de la tabla y ejecutar muchas más sentencias distintas de la pretendida, que solo mostrar una fila al usuario.

La forma de ayudar a prevenir este tipo de ataques es utilizar consultas parametrizadas que fuerzan a la distinción entre estructuras de control, como son las palabras clave de una sentencia SQL de los que son puramente datos de entrada por parte del usuario de la aplicación. Los programadores pueden, explícitamente, especificar a la base de datos lo que debe ser tratado como comando y lo que debe ser tratado como datos. Debe evitarse formar consultas construidas con sentencias parametrizadas y cadenas concatenadas:

A computer code with red text

Description automatically generated

Figura 9. Ejemplo de consulta SQL construida con el uso de sentencias parametrizadas y concatenadas. Fuente: Chess y West, 2007.

* Manipulación de rutas: este tipo de error tiene lugar cuando se permite en las entradas de un usuario incluir meta-caracteres de sistemas de ficheros como: *slash*(/), *backslash* (\) y punto (.).

Donde se espera una ruta relativa, un atacante puede convertirla en una ruta absoluta o recorrer el sistema de ficheros a una posición no planeada subiendo en el árbol de directorio. Se denomina a un acceso al sistema de ficheros no autorizado de este tipo manipulación de ruta. El código del ejemplo de la figura siguiente muestra la entrada a la aplicación desde una petición HTTP para crear un nombre de archivo. El programador no ha considerado la posibilidad de que un atacante podría proporcionar un nombre del archivo como , que hace que la aplicación suprima uno de sus propios archivos de configuración.

A close-up of a text

Description automatically generated

Figura 10. Ejemplo de vulnerabilidad de manipulación de ruta. Fuente: Chess y West, 2007.

Las vulnerabilidades de manipulación de ruta son relativamente fáciles de prevenir con *whitelists.*

* Inyección de comandos: si se permite al usuario especificar comandos de sistema que su programa ejecuta, los atacantes podrían ser capaces de hacer que el sistema ejecute comandos maliciosos en su nombre. Se denomina inyección de comandos a la ejecución de comandos no autorizada.

El código en el ejemplo de la siguiente figura es para ejecutar un *backup* de una base de datos ORACLE en Windows. Se acepta un parámetro que especifica un tipo de *backup* con la utilidad rman. Se necesita un acceso privilegiado para poder acceder a la BD; por lo tanto, la aplicación se ejecuta con un usuario privilegiado. Un atacante podría pasar como parámetro "&& del c:\\dbms\\\*.\*”. Cualquier comando que se inyectara se ejecutaría con los privilegios que se tenían para trabajar con la BD.

A computer code with red text

Description automatically generated with medium confidence

Figura 11. Ejemplo de inyección de comandos. Fuente: Chess y West, 2007.

* Falsificación de *logs:* los *logs* son un objetivo para atacantes, pues son un recurso valioso para administradores de sistema y desarrolladores. Si los atacantes pueden falsificar el valor que es escrito en los *logs,* podrían ser capaces de fabricar eventos en el sistema mediante la inclusión de entradas corrompidas. Los archivos de *log* falsificados o, de otro modo, archivos de *log* corrompidos, pueden usarse para seguir las pistas de un atacante o implicar a otra parte en la comisión de un acto malicioso.

El ejemplo de la siguiente figura muestra un trozo de código de una aplicación web que intenta leer un número entero desde un objeto *request.* Si el valor, al ser comprobado, no es entero, se escribe el evento de *log* correspondiente indicando lo ocurrido.

A white background with black text

Description automatically generated

Figura 12. Ejemplo de falsificación de log. Fuente: Chess y West, 2007.

Si un usuario envía a la aplicación *twenty-one* en el parámetro val, se registra la siguiente entrada:

Sin embargo, si un atacante envía:

Se registra la siguiente entrada:

### **4.4. Desbordamiento de buffer**

Un desbordamiento de buffer ocurre cuando un programa escribe datos fuera de los límites de la memoria asignada.

Las **vulnerabilidades por desbordamiento de** buffer, por lo general, son explotadas con el objetivo de superponer valores en la memoria en provecho del atacante. Los errores de desbordamiento buffer están muy extendidos y normalmente dan a un atacante un control alto del código vulnerable. Este tipo de vulnerabilidades se puede clasificar en los siguientes tipos:

A diagram of a string

Description automatically generated

Figura 13. Tipos de vulnerabilidades de desbordamiento de buffer. Fuente: elaboración propia.

La mejor forma de prevenir el desbordamiento de buffer es utilizar un lenguaje de programación que **fuerce la comprobación de tipos y de memoria** de forma que su gestión sea segura. C#, Java, Python, Ruby o dialectos de C como CCured y Cyclone son lenguajes de este tipo. Se usa el término safe para referirse a lenguajes que automáticamente realizan comprobaciones en tiempo de ejecución para impedir a los programas violar los límites de la memoria asignada.

Los **lenguajes seguros** deben proporcionar **dos propiedades** para asegurar que los programas respetan límites de asignación:

* **Seguridad de memoria.** Requiere que el programa no leerá o escribirá datos fuera de los límites de la memoria asignada.
* **Seguridad de tipo.** En los lenguajes que cumplen esa propiedad se asigna un tipo a los objetos, como, por ejemplo, un **tipo int,** puntero a int, puntero a función, etc. Esta propiedad asegura que las operaciones en el objeto son siempre compatibles con su tipo. Sin esta característica, cualquier valor arbitrario podría usarse como una referencia en la memoria. **C y C++** son lenguajes inseguros y ampliamente utilizados y, por lo tanto, el programador es el responsable de prevenir que las operaciones que manipulan la memoria puedan resultar en **desbordamientos** de buffer. Un ejemplo de este tipo de error que se pude cometer en este lenguaje es cuando se declara un puntero a un número entero (int) y luego se utiliza como un puntero a una función, tal y como se muestra en ejemplo siguiente:

A white background with black text

Description automatically generated

Figura 14. Error de seguridad de tipos. Fuente: elaboración propia.

La máquina virtual de java **(JVM),** en sí misma, está escrita en C para una plataforma dada. Esto quiere decir que la JVM, en sí misma, puede ser susceptiblea **problemas de desbordamiento.**

A computer code with black text

Description automatically generated

Figura 15. Llamada a un método nativo JNI. Fuente: Chess y West, 2007.

El código de la figura siguiente implementa el método nativo definido en la clase **Echo.** El código es vulnerable a buffer overflow causada por una llamada ilimitada a .

A close-up of a computer code

Description automatically generated

Figura 16. Vulnerabilidad de buffer overflow en un método nativo. Fuente: Chess y West, 2007.

### **Ataque de desbordamiento de buffer basado en el stack**

En un ataque clásico por desbordamiento de buffer, el atacante envía los datos que contienen un segmento de código malévolo a un programa que es vulnerable a este error, basado en el stack. Además del código malévolo, el atacante incluye la dirección de memoria del principio del código.

El código malévolo suele ser un ***shellcode:*** conjunto de instrucciones de programación, generalmente en lenguaje ensamblador y trasladadas a **opcodes** que suelen ser inyectadas en la pila (o stack) de ejecución de un programa para conseguir que la máquina en la que reside se ejecute la **operación de tipo malicioso.**Deben ser de longitud corta para poder ser inyectadas dentro de la pila en un espacio reducido y se utilizan para ejecutar código malévolo. En la figura se muestra un ejemplo:

A white rectangular object with black text

Description automatically generated

Figura 17. Ejemplo de shellcode. Fuente: es.wikipedia.org (s. f.)

Esta shellcode,que ejecuta la shell/bin/sh, realiza una llamada al sistema **execve** para realizar la ejecución de la shell(ventana de comandos) contenida dentro del array scode.

Cuando el desbordamiento de buffer ocurre, el programa escribe los datos del atacante en el buffer y sigue más allá de sus límites hasta que superponga la dirección de vuelta de la función con la dirección del principio del **código malicioso.** Cuando la función devuelve el control, salta al valor almacenado en su dirección de retorno.

Normalmente, esto lo devolvería al contexto de la función llamada, pero debido a que la dirección de retorno ha sido superpuesta, el control salta el buffer y comienza a ejecutar el código malicioso del atacante. Para aumentar la probabilidad de adivinar la dirección correcta del código malicioso, los atacantes típicamente rellenan el principio de su entrada con una serie de **instrucciones NOP** (ninguna operación).

El código del ejemplo de la siguiente figura define el problema de función, que asigna un buffer**char** y un int en el stack(pila) y lee una línea de texto de**stdin** con en el buffer. Comosigue leyendo la entrada hasta que se encuentra un carácter de**final-de-línea,** un atacante puede desbordar el buffer line con datos maliciosos. En el ejemplo, esta función declara dos variables locales y usa para leer una línea de texto en el buffer line del stackde 128 octetos.

La pila es un lugar en la memoria de un equipo en el que todas las variables que se declaran e inicializan antes de pasarlas a tiempo de ejecución se almacenan.

Otro ejemplo mostrado en la siguiente Figura 18, expone lo que ocurre en un clásico ataque de buffer overflow.

* El **primer marco de** stack representa el contenido de la memoria después de llamar a , pero antes de que sea ejecutada. La variable localline es asignada sobre el stackque comienza en la dirección 0xNN. La variable local a está justo encima de line en la memoria; la dirección de vuelta está encima de **a**. Asumir que apunta a la función que llamó a .
* El **segundo marco de** stackilustra un argumento en el cual el se comporta normalmente. **¡Se lee la entrada! ¡EJEMPLO BUENO!** y retorna. Se puede ver ahora que line está parcialmente llena con un string de entrada y que otros valores almacenados sobre el stack son inalterados.
* El **tercer marco de** stackilustra un escenario en el cual un atacante explota la vulnerabilidad de desbordamiento de buffer en y hace que se ejecute el código malévolo en vez de retornar normalmente. En este caso, line ha estado llena de una serie de **NOPs,** el código del exploit, y la dirección del principio del buffer, .

A close-up of a card

Description automatically generated

Figura 18. Empleo incorrecto de gets() que puede incurrir en desbordamiento de buffer. Fuente: Chess y West, 2007.

A diagram of a diagram

Description automatically generated with medium confidence

Figura 19. Ejemplo de ataque de buffer overflow. Fuente: Chess y West, 2007.

### **Ataques relacionados con la reserva dinámica de memoria (heap overflow)**

Uno de los errores de concepto que se tienen sobre las vulnerabilidades de buffer overflow, es que son solo explotables cuando el buffer está en el stack. Los ataques basados en el heappueden sobrescribir datos importantes almacenados en el mismo y cambiar el flujo del programa. Por ejemplo, se podría sobrescribir el valor de un puntero a una función, de tal forma que cuando el programa invoque la función referenciada por el puntero a la función se ejecutará el exploit.

Heap es la sección de la memoria de un equipo donde todas las variables creadas o inicializadas en tiempo de ejecución se almacenan.

Gran parte de las vulnerabilidades relacionadas con la reserva de memoria dinámica se corresponden a fallos de programación que encajan en alguna de las siguientes categorías:

A close-up of a blue and white background

Description automatically generated

Figura 20. Errores de heap overflows.Fuente: elaboración propia.

* **Memoria no utilizada, sin desasignar** (memory leaks). Favorece de forma potencial ataques de denegación de servicio. Se produce cuando una aplicación **adquiere la memoria y falla al liberarla** de nuevo al sistema operativo. La siguiente función en **C** deliberadamente pierde memoria al no liberar el puntero a la memoria asignada, una vez que el **puntero a** está fuera de ámbito, una vez ejecutada la function\_which\_allocates() finaliza sin liberar **a.**

A screenshot of a computer code

Description automatically generated

Figura 21. Ejemplo de memory leaks. Fuente: [en.wikipedia.org](http://en.wikipedia.org), s. f.

* **Uso de memoria después de ser liberada** (use after free). Ocurre cuando un programa continúa usando un puntero que ha sido previamente liberado. Si esa memoria se vuelve a reservar, un atacante puede lanzar un ataque de buffer overflowcatalogado con el CWE-416.

A computer error message

Description automatically generated

Figura 22. Ejemplo de use after free.Fuente: Chess y West, 2007.

* **Acceso a una variable después de liberarla** (dereference after free). Caso concreto de use after free que ocurre cuando intentamos acceder a la memoria dinámica previamente liberada como consecuencia de un . En el ejemplo de abajo (Merino, 2011), el puntero a es liberado, aunque más adelante se vuelve a referenciar desde la función strncpy. Como consecuencia, es probable que se corrompan datos asignados posteriormente en dicha dirección de memoria tras haber liberado la misma, o que se produzca un crash de la aplicación al intentar sobrescribir «memoria aleatoria» en la que el proceso no tiene permiso.

A computer screen shot of a code

Description automatically generated

Figura 23. Ejemplo de reference free. Fuente: Merino, 2011.

* **Liberar la memoria más de una vez** (double free). Cuando un programa libera un trozo de memoria más de una vez ,las estructuras de datos para gestión de la memoria pueden llegar a corromperse, lo que puede ocasionar que el programa falle, o causar que se llame dos veces a . Esto último puede dar el control a un atacante sobre los datos que se escriben en esa memoria doblemente reservada. Entonces, se tiene un programa potencialmente expuesto a ataque de buffer overflow.

A white rectangular object with black text

Description automatically generated

Figura 24. Ejemplo de double free. Fuente: Merino, 2011.

* **Uso de un puntero con valor NULL** (null dereference). El uso de un puntero con valor null puede ocasionar fallos de segmentación en la gestión de la memoria.

A computer code with text

Description automatically generated with medium confidence

Figura 25. Ejemplo de null dereference.Fuente: elaboración propia.

### **Manipulación de strings**

La estructura de datos de strings de C básica serie de caracteres terminada por el carácter nulo, es propensa a **error** y, además, las **funciones de biblioteca** para la manipulación de estos solo empeoran el panorama.

En el presente apartado se introduce al alumno en el estudio de los problemas de seguridades relativas a las funciones de manipulación de stringsy las de segunda generación; que son más seguras, pues limitan el número de datos copiados por parámetro.

A blue and white card with black text

Description automatically generated

Figura 26. Problemas con funciones que manipulan string.Fuente: elaboración propia.

* **Funciones inherentemente peligrosas**

Específicamente, hay que **evitar** el uso de funciones como o

* Esta función lee, de la entrada estándar, una corriente de bytes, y la almacena en el array apuntado por **s** hasta que se encuentra el carácter de nueva línea o de fin de fichero. Puede producir desbordamiento de buffer, que se le pasa como argumento, si es más pequeño que la fuente de entrada, que en este caso es la entrada estándar: el teclado. La función se comporta de la misma forma.



Figura 27. Llamada insegura a gets() similar a la explotada por el gusano Morris. Fuente: Chess y West, 2007.

* Función que lee de la entrada estándar los caracteres que corresponden al formato que se le especifica. Por ejemplo, si se especifica , se leerán los caracteres de la entrada en el buffer hasta que llegue un carácter , lo que puede ocasionar un **desbordamiento de** buffer. También se puede especificar y limitar a ese número de caracteres los que se pueden leer de la entrada y usarse la función de un modo más seguro. Funciones de similar comportamiento son

A computer code with green and white text

Description automatically generated

Figura 28. Código de w3-msql 2.0.11 vulnerable a un desbordamiento de búfer remoto causada por una llamada insegura a scanf(). Fuente: Chess y West, 2007.

* A diferencia de las funciones anteriores, opera en datos ya almacenados en una variable de programa, lo que hace que su manejo suponga obviamente menos riesgo de seguridad. Esta función copia el contenido de un buffer en otro hasta que se encuentra un carácter nulo en el buffer fuente; por lo tanto, hay que tener cuidado de que el buffer fuente sea más pequeño que el destino y que esté terminado por el carácter nulo. Funciones con similar comportamiento son

A computer screen shot of a computer code

Description automatically generated

Figura 29. Código del programa php.cgi en PHP / FI 2.0beta10 vulnerable a un desbordamiento de buffer remoto causada por una llamada insegura para strcpy(). Fuente: Chess y West, 2007.

* Para usarla de forma segura, hay que comprobar que el buffer de destino pueda acomodarse a la combinación de todos los argumentos de la fuente de entrada, controlando los tamaños **fuente-destino**y las conversiones de formato que se especifican. Si la longitud del argumento fuente es mayor que la del destino, se producirá un buffer overflow. Funciones de comportamiento similar son

A white background with black text

Description automatically generated

Figura 30. Código de la Versión 1.0 del daemon de Telnet Kerberos 5 que contiene un desbordamiento de bufferdebido a que la longitud variable de entorno TERM, que nunca se valida. Fuente: Chess y West, 2007.

* **Operaciones con** strings**de tamaño limitado.**

Muchas de las primeras vulnerabilidades descubiertas en C, C++ eran causa de operaciones con strings.Cuando se revisó el estándar de C se introdujeron nuevas funciones equivalentes a las que se han analizado en el apartado anterior con un parámetro que limita la longitud del buffer de destino de los datos. Conceptualmente, el uso de estas funciones con limitación de buffer destino se realiza reemplazando las antiguas funciones con las nuevas:

Para asegurarse que ningún desbordamiento de buffer ocurre aquí, una herramienta tiene que asegurarse solo de que dest\_size no es más grande que la cantidad de espacio asignado para dest. El tamaño y el contenido de srcno importan. En la mayoría de los casos, es mucho más fácil comprobar que dest y dest\_size están de acuerdo el uno con el otro.

A table with text on it

Description automatically generated

Tabla 2. Funciones ilimitadas con sus correspondientes limitadas. Fuente: adaptado de Chess y West, 2007.

El código de la figura siguiente muestra una vulnerabilidad de desbordamiento de buffer que es resultado del empleo incorrecto de funciones de manipulación de strings**ilimitadas** y en Kerberos 5 Versión 1.0.6, CERT CA-2000-06, [15]. Dependiendo de la longitud de cp y copy, con cualquiera de las llamadas final ao a podría desbordarse

La vulnerabilidad ocurre en el código responsable de manipular un string de un comando, que inmediatamente sugiere que pudiera ser explotable debido a su proximidad a la entrada de usuario. De hecho, esta vulnerabilidad públicamente ha sido explotada para ejecutar órdenes no autorizadas sobre sistemas comprometidos.

A screenshot of a computer program

Description automatically generated

Figura 31. Función ilimitada, vulnerabilidad en Kerberos 5. Fuente: Chess y West, 2007.

* **Errores de truncado**

Incluso cuando se usan correctamente, las funciones de strings**limitadas** pueden introducir errores, porque truncan los datos que exceden el límite especificado.

El código de la figura siguiente muestra un **error de truncamiento de** string que se convierte en un problema de terminación. El error está relacionado con el empleo de la función que no termina con carácter nulo su buffer destino y puede devolver hasta el número de octetos especificados en su tercer argumento, el código de la figura cae en la trampa demasiado común de terminar a mano con carácter nulo la ruta expandida, buf, en este caso, un octeto más allá del final del buffer.

Este error por un byte, denominado off by one**,** podría ser inconsecuente dependiendo de lo que se almacena en la memoria justo más allá del buffer, pues permanecerá con eficacia terminado por el carácter uno hasta que otra posición de memoria sea superpuesta. Es decir, devolverá solo el tamaño real del buffer más uno, , en este caso.

Sin embargo, cuando buf posteriormente sea copiado en otro buffer con el valor de vuelta de , como el límite pasado a , los datos en buf están truncados y el buffer de destino se deja sin terminar con el carácter nulo. Este «error de por uno», probablemente causará un serio desbordamiento de buffer.

A computer code with text

Description automatically generated

Figura 32. Llamada a strncpy() que podría causar error de truncado. Fuente: Chess y West, 2007.

Hay que evitar truncar los datos. Si la entrada proporcionada es demasiado grande para una operación dada, hay que intentar manejar el problema redimensionando dinámicamente los buffers, o directamente rehusar realizar la operación e indicar al usuario lo que tiene que ocurrir para que la operación tenga éxito.

* **Mantenimiento del carácter nulo de terminación**

En C, los stringsdependen de la terminación nula apropiada; sin ello, su tamaño no puede ser determinado. Los **errores de terminación de** strings fácilmente pueden conducir a desbordamientos y errores lógicos. Estos problemas, normalmente se hacen más insidiosos porque ocurren, aparentemente, de manera**no determinista,** dependiendo del estado de la memoria cuando el programa se ejecuta.

Considerar el código de la figura siguiente, como , no termina con carácter nulo su buffer de salida, lee la memoria hasta que encuentra un carácter nulo y por lo que a veces produce valores incorrectos de longitud, dependiendo del contenido de memoria después de buf.

A white rectangular frame with blue border

Description automatically generated

Figura 33. String no terminado en carácter nulo producido por readlink(). Fuente: Chess y West, 2007.

Nunca hay asumir que los datos del mundo exterior están correctamente terminados con carácter nulo. Si el string no se termina correctamente, la adición de un octeto nulo en el último octeto del buffer impedirá a otras operaciones como calcular mal la longitud del string o desbordar el buffer.



Figura 34. Ejemplo. Fuente: elaboración propia.

### **Errores de formato de cadena (format strings)**

Los errores de formato de string se explican en el siguiente video:

Una de las vulnerabilidades que se suelen cometer a la hora de programar en lenguaje C y C++ es la de format string. En este vídeo (Vulnerabilidades de format string) se profundiza en el estudio de este tipo de vulnerabilidad.

El análisis estático de código fuente se considera la actividad más importante de entre las mejores prácticas de seguridad que se han de realizar en el curso del desarrollo de una aplicación. En la siguiente clase magistral (Herramienta de análisis de código fuente SCA Fortify) se realiza una introducción a una de las mejores herramientas de análisis estático de código del mercado, Fortify, de la compañía HP.