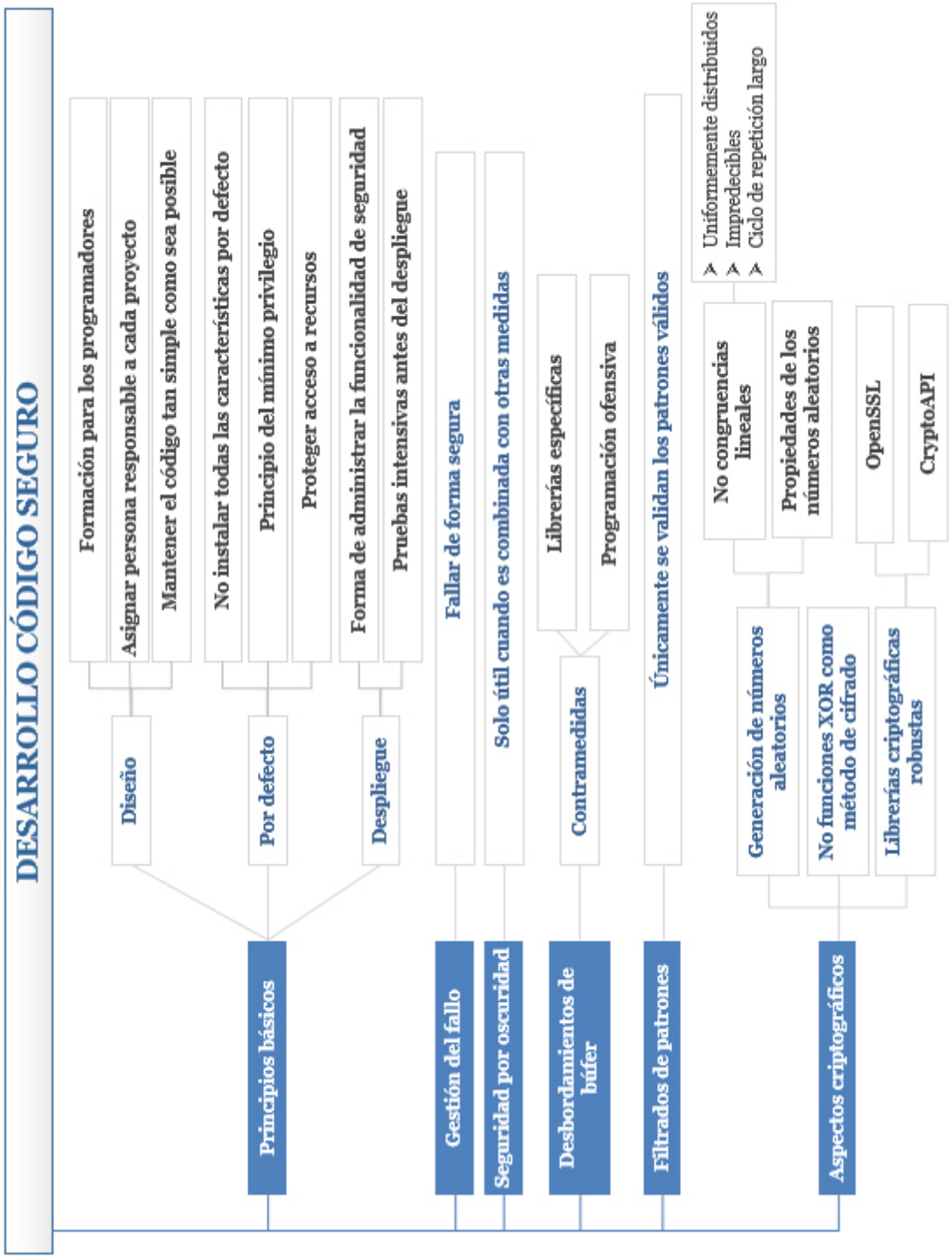
Desarrollo de código seguro

TEMA 8

Esquema


DESARROLLO CÓDIGO SEGURO
Diseño
Formación para los programadores
Asignar persona responsable a cada proyecto
Mantener el código tan simple como sea posible
Por defecto
No instalar todas las características por defecto
Principio del mínimo privilegio
Proteger acceso a recursos
Despliegue
Forma de administrar la funcionalidad de seguridad
Pruebas intensivas antes del despliegue
Principios básicos
Gestión del fallo
Fallar de forma segura
Seguridad por oscuridad
Solo útil cuando es combinada con otras medidas
Contramedidas
Librerías específicas
Programación ofensiva
Desbordamientos de búfer
Filtrados de patrones
Únicamente se validan los patrones válidos
Generación de números aleatorios
No congruencias lineales
Uniformemente distribuidos
Impredecibles
Ciclo de repetición largo
Propiedades de los números aleatorios
No funciones XOR como método de cifrado
Librerías criptográficas robustas
OpenSSL
CryptoAPI
Aspectos criptográficos

Ideas clave

8.1. ¿Cómo estudiar este tema?

El estudio de este tema se realiza a través de los contenidos desarrollados en las **Ideas clave** expuestas a continuación.

En este tema abordaremos un tema esencial en la seguridad informática pero al que se le concede habitualmente muy poco (si alguno) espacio en los planes de estudio y libros sobre el área. Y es sin embargo, realmente importante, pues buena parte de las vulnerabilidades que te encontrarás en tu carrera profesional, sean en la red, en una aplicación o en cualquier otro elemento de una infraestructura informática, tiene su origen en **el software**.

En este tema estudiaremos los **principales y más habituales errores que los programadores cometen en sus desarrollos** y por supuesto, **cómo evitarlos**. Hablaremos de **cómo gestionar los fallos** que inevitablemente, se producirán en todas las aplicaciones, y del concepto de ***seguridad por oscuridad***.

A continuación, analizaremos uno de los grandes tótems de la seguridad: **los desbordamientos de búfer (*buffer overflows*)**, que han sido los responsables de algunos de las más grandes vulnerabilidades de la historia de la informática. Por último, abordaremos brevemente **cómo integrar la criptografía en los desarrollos de software**, cuáles son los **principales pecados** y **cómo evitar cometerlos**.

8.2. Principios de seguridad básicos: D^3

Si debemos hacer ver al alumno la importancia del desarrollo de código seguro, veamos las consecuencias de la ausencia del mismo: ataques de desbordamiento de búfer, inyección SQL, cross Site Scripting (XSS), rotura de autenticación y administración de sesiones, referencias directas a objetos inseguras, cross Site Request Forgery (CSRF), configuración defectuosa de seguridad, almacenamiento criptográfico inseguro, falla de restricción de acceso por URL, protección insuficiente en la capa de transporte, redirecciones y reenvíos no validados. Suficiente, ¿verdad?

Es de vital importancia el desarrollo de código seguro para poder encontrar puntos débiles en las aplicaciones que permiten que un atacante comprometa la integridad, disponibilidad o confidencialidad de la misma.

Estos principios, aunque especialmente aplicables al desarrollo de código seguro, **pueden aplicarse** y, de hecho se hace**, a toda el área de la seguridad informática**. Son fáciles de recordar a través de su acrónimo de **las 3 D's**, o **D³**: **seguridad en el Diseño**, **por Defecto y en el Despliegue**.

Veamos cada uno de estos conceptos con mayor detalle.

Seguridad en el diseño

Este principio hace referencia a que **la seguridad en el código debería comenzar en la etapa más temprana de su desarrollo**, es decir, en el diseño. Hacerlo en esta etapa, en contra de lo que pueda parecer, es fácil y económico.

En general, pueden seguirse los siguientes **pasos**:

- **Realiza una formación adecuada para todo tu personal de desarrollo.** Si no se conocen los principios de seguridad en el código, difícilmente se podrán implementar.
- **Asigna una persona responsable de la seguridad en el código**, que se responsabilice específicamente de esta tarea.
- **Trata de mantener el código tan simple como sea posible.** La seguridad y la complejidad no se llevan nunca bien.

Seguridad por defecto

Este aspecto **se refiere a la seguridad del código que este proporciona por defecto, sin ninguna modificación o adaptación específica**. Puede conseguirse a través de los siguientes **principios generales**:

- No instalar todas las características y capacidades por defecto. Es preferible que el usuario active específicamente aquellas que va a necesitar a que estén todas activas por defecto.

- Seguir, también en el desarrollo de código, el ***principio del mínimo privilegio***. A veces, por comodidad, los programadores o administradores de sistemas asignan máximos privilegios a las aplicaciones para que puedan ejecutarse sin problemas. Obviamente, este es un riesgo de seguridad muy importante. Por tanto, es preferible que el código no necesite permisos de administrador (si no es estrictamente necesario, claro).
- Proteger adecuadamente el acceso a los recursos. Particularmente, los datos sensibles o críticos deberían estar especialmente protegidos (con el uso de, por ejemplo, una contraseña u otro mecanismo similar).

Seguridad en el despliegue

Por último, este principio pe**rmite que un código sea mantenible una vez que los usuarios instalan o comienzan a utilizar el mismo**. Podría crearse una aplicación bien diseñada y escrita, pero si es difícil de instalar o administrar, entonces será difícil de mantener la aplicación segura cuando lleguen nuevas amenazas de seguridad.

Para ello, es conveniente:

- Asegurarse de que la aplicación ofrece **una manera de administrar su funcionalidad relacionada con la seguridad**. Obviamente, si el usuario no conoce la configuración de seguridad de la aplicación, tampoco podrá saber si ésta es segura o la está utilizando correctamente.
- Relacionado con el punto anterior, es importante también **proporcionar suficiente información al usuario**, de forma que éste sepa cómo utilizar la aplicación de forma segura. Podría ser suficiente con ayuda online, documentación o información en la pantalla.
- **Realiza pruebas intensivas antes del despliegue.** Designa un equipo de gente, que debe ser diferente del de desarrollo, instala un servidor de prueba y pide, directamente, a este equipo que intente romper la aplicación de cualquier manera que se les ocurra.

Como resumen de estos principios, recuerda que la seguridad no es algo que pueda ser aislado en alguna porción del código. Como el rendimiento, la escalabilidad, el buen diseño o la legibilidad, la seguridad es algo que todo desarrollador de código debería conocer.

De acuerdo a la propia experiencia práctica, podemos afirmar que siguiendo estos tres grandes principios de seguridad que hemos analizado, pueden construirse sistemas seguros. Además, de estos principios se derivan una serie de **buenas prácticas** como las siguientes:

- **Aprende siempre de los errores cometidos.** Mantén una base de conocimiento actualizada con todo lo que vayas aprendiendo.
- **Minimiza tu superficie de ataque.**
- **Usa la defensa en profundidad.**
- **Usa el principio del mínimo privilegio.**
- **Asume que**, siempre y por defecto, **cualquier otro sistema externo es inseguro**.
- **Gestión del fallo:** contempla siempre el fallo de la aplicación, y que éste se produzca de forma segura.
- **Nunca (de verdad, nunca) basar la seguridad de tu aplicación en algún secreto embebido en el código.** Éste siempre puede ser desensamblado.

Muchos de estos puntos son auto-explicativos, pero otros merecen que nos detengamos algo más en ellos.

8.3. Gestión del fallo

Por muy buen programador que seas, cualquier aplicación terminará fallando, siempre...En el caso de un equipo mecánico, el problema puede surgir simplemente del envejecimiento o por una filtración de agua al interior. En el caso del software, las causas pueden ser también muy numerosas, algunas imprevisibles, como el fallo del hardware o del propio sistema operativo.

Por esta razón, asume que el fallo se producirá y pasa a planificar las acciones que se realizarán en ese caso. Ante la pregunta, « ¿qué ocurriría si...? » la respuesta incorrecta es «Eso nunca va a suceder».

Es como tener un «plan de evacuación en caso de incendio». Ojalá no tengas que utilizarlo nunca, pero te alegrarás enormemente de tenerlo si se da realmente el caso.

Fallar de forma segura

¿Fallar de forma segura? ¿No es eso una contradicción? Bueno, no realmente. Fallar de forma segura se refiere a que la aplicación, ante un error importante, **no desvele más información de la que mostraría normalmente** o que esta información no pueda ser modificada.

Si esto no es así, un atacante sabe que puede hacer que un código falle, entonces podrá evitar los mecanismos de seguridad porque el código falla de forma insegura. Por esta razón, cuando se produzca el error es preferible no proporcionar demasiada información sobre el fallo, únicamente la suficiente para que el usuario sepa qué está ocurriendo, y guardar los detalles en un sistema de log seguro (como el Gestor de Eventos de *Windows*, por ejemplo).

Veamos un ejemplo. Mira la siguiente porción de código y comprueba si eres capaz de detectar dónde existe una debilidad:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED) {
        // Comprobación de acceso errónea.
       // Informar al usuario de que su acceso ha sido denegado.
} else {
       // Comprobación de acceso OK
       // Realizar la tarea solicitada
}
```

A primera vista, el código parece correcto, ¿verdad? Realiza su función, que es comprobar las credenciales de acceso de un usuario. Pero, ¿qué ocurre si la función *IsAccessAllowed* falla? Por ejemplo, qué pasa si el sistema se queda sin memoria, o sin manejadores de objeto cuando se llama a dicha función. Pues que el usuario puede ejecutar una tarea privilegiada porque la función podría devolver un código de error como, por ejemplo, **`ERROR_NOT_ENOUGH_MEMORY`**, y la ejecución se iría por la rama de la autenticación correcta.

Por esta razón, la forma correcta de escribir este código es, en realidad, la siguiente:

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
	// Comprobación de acceso errónea.
	// Informar al usuario de que su acceso ha sido denegado.
} else {
	// Comprobación de acceso OK.
	// Realizar la tarea solicitada.
}
```

Así, si la llamada a *IsAccessAllowed* falla por cualquier razón, el acceso será negado al usuario, como debe ocurrir.

¡No a la seguridad por oscuridad!

El lema de **seguridad por oscuridad** (del inglés *security through obscurity*) es un viejo conocido de la seguridad informática. En esencia, se trata de un muy mal modo de proceder que algunos se empeñan en seguir utilizando, y que se basa en creer que porque el atacante no tiene acceso al código fuente y que, por tanto, sus secretos o malas prácticas de programación están a salvo.

La historia ha demostrado que esto nunca es cierto (de hecho, es fácil desensamblar un código ejecutable), por lo que lo recomendable es diseñar e implementar como si el atacante tuviera pleno acceso al código fuente y a todos los diseños. En cualquier caso, sí es cierto que la «oscuridad» puede ser útil, siempre y cuando no sea la única defensa. Es decir, **lo recomendable es que la oscuridad sea una parte (pequeña) de una defensa más potente basada en la estrategia de defensa en profundidad**.

8.4. Enemigo público número 1: el desbordamiento de búfer

En este apartado presentamos uno de los grandes problemas de la seguridad informática, y del **desarrollo de código seguro** en particular. De hecho, son una vieja amenaza, pues uno de los primeros y más conocidos ejemplos data del famoso gusano creado por Robert Morris en 1988 (y que estudiamos en el primer tema de la asignatura).

Este exploit paralizó prácticamente toda Internet, mientras los administradores se afanaban en desconectar sus máquinas de la red para tratar de contener el daño.

El impacto de este tipo de ataques es realmente elevado. Por ejemplo: el Centro de Respuesta a Incidentes de Microsoft estima que el coste económico de lanzar una alerta de seguridad y el parche de código asociado es de alrededor de 100.000 dólares. Y ese dinero es sólo el principio. Literalmente miles y miles de administradores de sistemas tienen que utilizar muchas horas de su tiempo para poder aplicar rápidamente el parche.

Pero por mucha prisa que se den, invariablemente algunos sistemas serán comprometidos. Y en este caso, el coste puede llegar a ser astronómico, si por ejemplo, el atacante consigue acceso a un sistema que gestione información como tarjetas de crédito. Así que un pequeño error de programación se puede traducir en millones de dólares de coste, por no mencionar las repercusiones en la imagen pública de la empresa.

Las razones para que los **desbordamientos de búfer** sigan siendo un problema hoy en día, tantos años después, es debido a la falta de formación de los programadores y al hecho de que los lenguajes como C o C++ carecen de funciones de manejo de cadenas seguras o fáciles de manejar, por lo que resulta relativamente fácil cometer este tipo de errores.

Pero no esperemos más. Veamos **en qué consisten los desbordamientos de búfer** exactamente y **por qué son tan peligrosos**.

Desbordamientos de búfer (de pila)

En pocas palabras, un desbordamiento de búfer basado en pila ocurre cuando un búfer declarado en la pila de ejecución se sobrescribe copiando en él más datos de los que puede alojar por su tamaño. Recordarás de otras asignaturas, que las variables locales de una función se sitúan junto a la dirección de retorno a la función que llamó a la que se está ejecutando. Echa un vistazo a la parte izquierda de la **Figura 1** para recordarlo.

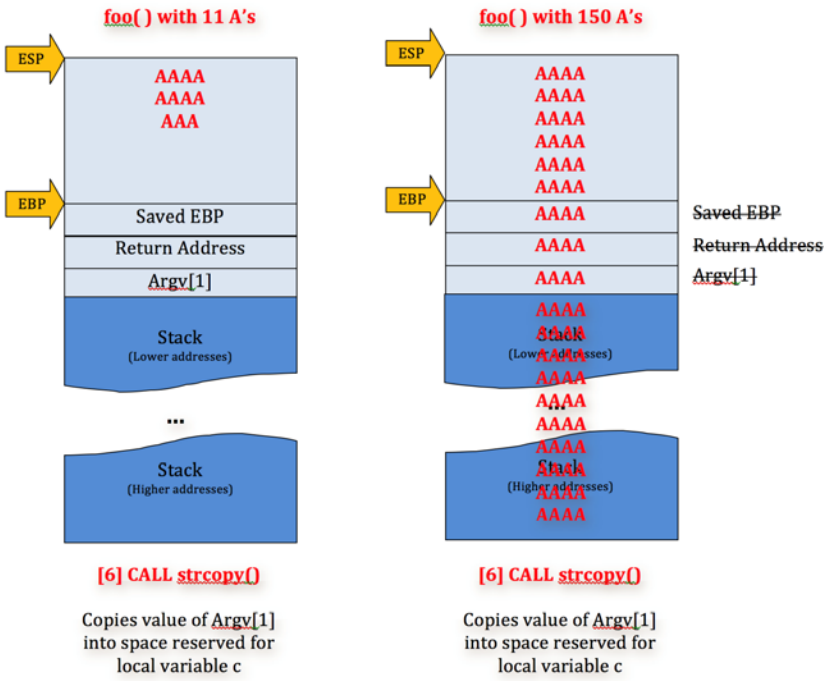


Figura 1. Ejemplo de ejecución de un desbordamiento de búfer. (Fuente: www.securitysift.com)

Como se puede observar, **si el búfer se desborda, los datos comienzan a sobrescribir el resto de elementos de la pila, entre ellos la dirección de retorno**. Llegado a este punto, ¿entiendas el problema? Si los datos que se envían se ajustan con cuidado, es posible hacer coincidir los que nos interesen para que sobrescriban justo la dirección de retorno. De esta forma, cuando la ejecución de la función acabe, haremos que le procesador salte a donde queramos, « ¡pudiendo tomar el control de la máquina! »

Veamos un ejemplo concreto de un ataque de este tipo. En este caso, cómo un simple error de un desbordamiento de un único byte puede ser explotado. Revisa el siguiente código:

```
-..,lyhyyryhnnñlk´.
```

¿Ves dónde está el problema? Nuestro programador utilizó la función *strncpy()* para copiar el búfer, y la función *sizeof()* para determinar el tamaño del mismo. El error radica en la línea `buf[sizeof(buf)] = '\0';` , debido a que esta sentencia sobreescribe, en realidad, el búfer en una posición.

Muy bien, **¿y cómo es posible explotar esta vulnerabilidad?** Es decir, **¿cómo hacen los hackers para explotar estos errores?** Bueno, una forma sencilla es comenzar compilando el código con información de depuración. Concretamente, si se está utilizando Visual Studio. NET, desactiva las opciones **`/GS`** y **`/RTC`** o esta demostración no funcionará. Después, establece un punto de ruptura (*breakpoint*), ejecuta el programa con una cadena muy larga de A's como argumento de entrada y veamos qué pasa:

1. Abre la ventana de *Registros*, y anota el valor del registro EBP que aparece allí. Veremos en un momento que este valor se volverá muy importante.
2. Continua la ejecución y entra en la función *func()*. Abre una ventana de *Memoria* y busca la localización en memoria de la variable *buf*. La llamada a *strncpy* llenará esta variable con letras A, y el siguiente valor a *buf* es el puntero EBP que hemos guardado.
3. Nos vamos acercando... A continuación continúa hasta la siguiente línea que guarda el carácter nulo en *buf*, y anota cómo cambia el valor del registro EBP. En mi caso, cambió de **`0x0012FF80`** a **`0x0012FF00`** (pero en cada sistema, tendrá valores diferentes).
4. Ahora recordemos cuál es nuestro objetivo, que es controlar el valor almacenado en la posición de memoria 0x0012FF00 (que actualmente es lleno de valores **`0x41414141`**, que es el código ASCII hexadecimal de la letra 'A').
5. Ejecutemos también con un solo paso la función *printf*, y cambiemos a modo ensamblador. Justo antes de la instrucción *ret*, que finaliza la función, vemos otra línea que es *pop ebp*. Si ahora volvemos a la función principal, *main()*, donde empezamos a salir, vemos que la última instrucción antes de acabar el programa es *mov esp,ebp* , que toma los contenidos del registro EBP y los copia en ESP, ¡que es nuestro punto de pila!
6. Si finalizamos ya del todo la ejecución, vemos que este puntero está corrupto y se intenta transferir el control del programa a la dirección **`0x41414141`**, donde no hay código ejecutable y el programa falla.

Hasta el momento hemos visto cómo es posible sobrescribir el puntero de pila, haciendo que la aplicación falle, habitualmente con un mensaje similar al de la **Figura 2**. Por supuesto, en este punto, nos preguntaremos, ¿cómo podemos explotar esta vulnerabilidad y ejecutar un código de nuestra elección?

Habitualmente el mejor método es, como otras tantas veces, el método de la prueba y error, hasta que consigamos que el error deje de producirse. Para ello, cualquier lenguaje que permita escribir un programa rápido puede servir, como *Perl* o *Python*. En este caso, un *script* en Perl sencillo para conseguir esto es:

```
$arg = “AAAAAAAAAAAAAAAAAAAAAAAAAAAA”."\x40\x10\x40";
$cmd = “off_by_one “.$arg;
system($cmd);
```

Figura 2. Mensaje habitual de Windows ante un problema en una aplicación

Y la salida que su ejecución produce es:

```
La dirección de func es 00401000, dirección de func_hackeada es 00401040
AAAAAAAAAAAAAAAAAAAAAAAAAAAA@?@
¡Upsss. Nos han hackeado!
```

Como puedes ver, hemos conseguido hackear el programa, cambiando su flujo de ejecución y consiguiendo llamar a la función ***func_hackeada***.

En este caso concreto, hay **dos condiciones** que deben cumplirse para que el problema pueda ser explotado. La primera es que **el número de bytes en el búfer necesita ser divisible por 4, o el desplazamiento por un solo byte no sobrescribirá el EBP almacenado**. La segunda es que, como **necesitamos controlar el área a la que apuntará el nuevo valor de EBP**, si el último byte de EBP fuese 0xF0 y el búfer fuera menor de 240 bytes (el valor de 0xF0 en decimal), no seríamos capaces de cambiar dicho valor.

En cualquier caso, ha habido muchas vulnerabilidades de desbordamiento de un único byte que han sido explotados en el mundo real. Dos de los casos más conocidos pueden ser, quizás, los de **“Apache mod_ssl off-by-one”** y **wuftpd ‘glob**.

Evitando los desbordamientos de búfer

Aunque resulte evidente, **«la mejor y primera línea de defensa es escribir buen código»**. A pesar de que algunos aspectos sobre escribir código seguro pueden ser un poco arcanos, pero, en general, evitar este tipo de vulnerabilidades es una cuestión de ser cuidadosos, y validar siempre todas las entradas. Como ya hemos comentado, el mundo exterior debe ser siempre tratado como si fuera hostil y estuviera constantemente tratando de atacar nuestra aplicación.

Sin embargo, siempre existen trucos, por supuesto. Uno poco conocido pero muy efectivo es uno que podríamos llamar ***programación ofensiva***. Por ejemplo, si una función recibe un búfer y su tamaño como argumentos, podríamos utilizar una porción de código como la siguiente:

```
#ifdef _DEBUG
memset(dest, 'A', buflen); //buflen = tamaño en bytes
#endif
```

De esta forma, cuando alguien llama a nuestra función y se las apaña para enviar un argumento erróneo para la longitud del búfer, el código fallará y el problema se hará patente rápidamente, antes de continuar la ejecución del resto del código.

Otra buena práctica para evitar los desbordamientos de búfer es **utilizar librerías específicas**, que incluyen protecciones para este problema. Una de ellas, al menos para el entorno *Windows*, es el conjunto de funciones *Strsafe.h* para los lenguajes C/C++, disponibles en el SDK de *Windows* desde *Windows XP SP2*. Entre otras, sus ventajas son las siguientes:

- **El tamaño correcto del búfer de destino siempre se pasa como argumento a la función**, para asegurarse de que esta no lo sobrescribe.
- Se garantiza que los búfers **terminan siempre con un carácter no nulo**.
- Todas las funciones devuelven un valor `HRESULT`, con un único valor posible `S_OK`.

¡No te fíes de ninguna entrada!

Si un desconocido llamara a la puerta de tu casa y te ofreciera algo de comer, ¿lo aceptarías? Probablemente no, ¿verdad? Entonces, ¿por qué tantas aplicaciones aceptan datos de entrada sin validarlos primero?

Es por esta razón por la que la mayoría de los exploits remotos contra aplicaciones funcionan. Por tanto, déjame repetirlo: **ninguna aplicación debería confiar en ningún dato de entrada hasta que haya sido validado**.

¿Y cuál es la forma adecuada de llevar a cabo esta validación? En general, es conveniente seguir esta regla: buscar lo que se considera datos válidos y rechazar cualquier otra cosa. Esto se relaciona con el ***principio de fallar de forma segura***, lo que significa que se deberías denegar cualquier acceso hasta que se determine explícitamente que la petición es válida.

Este **enfoque es el más adecuado** por varias razones:

- Podrían existir más de un patrón para representar los datos válidos.
- Sería fácil dejar pasar datos inválidos.

El primer punto hace referencia a que **es común escapar caracteres de forma que resulten válidos**, pero también es posible hacerlo para que datos inválidos pasen como válidos. Por ejemplo, imagina que tu aplicación busca explícitamente por caracteres '*' que no considera válidos (por la razón que sea). Si esta aplicación es Web, bastaría con codificar estos caracteres con el código ASCII en hexadecimal 0x2A.
El segundo punto es, de hecho, bastante común. Lo explicaremos de nuevo con un ejemplo. Imagina una aplicación que acepta peticiones de usuarios para subir ficheros, cuyos parámetros, entre otros, incluyen el nombre del fichero. Esta aplicación está diseñada para no aceptar ficheros que sean código ejecutable, pues podría resultar peligroso.

Un código de ejemplo que podría implementar esta aplicación sería el siguiente:

```
bool IsBadExtension(char *szFilename) {
       bool fIsBad = false;
       if (szFilename) {
             size_t cFilename = strlen(szFilename);
             if (cFilename >= 3) {
                   char *szBadExt[] = {“.exe", “.com", “.bat", “.cmd"};
                   char *szLCase = _strlwr(_strdup(szFilename));

                   for (int i=0;
                         i < sizeof(szBadExt)/sizeof(szBadExt[0]);
                         i++)
                         if (szLCase[cFilename-1] == szBadExt[i][3] &&
                               szLCase[cFilename-2] == szBadExt[i][2] &&
                               szLCase[cFilename-3] == szBadExt[i][1] &&
                               szLCase[cFilename-4] == szBadExt[i][0])
fIsBad = true;}
}
             return fIsBad;
}

bool CheckFileExtension(char *szFilename) {
      if (!IsBadExtension(szFilename))
             if (UploadUserFile(szFilename))
                   NotifyUserUploadOK(szFilename);
}
```

Antes de continuar, analiza el código y trata de encontrar en qué puntos pueden surgir problemas. **¿Qué te parece la función *IsBadExtension*?** Realmente hace un buen trabajo comprobando los posibles errores y es razonablemente eficiente.

El problema radica en la lista de extensiones «inválidas». Obviamente, está lejos de ser una lista exhaustiva, pues un usuario podría subir muchos tipos de ficheros ejecutables, como scripts de **Perl (.pl)**, de **Python (.py)** o algún tipo de **Windows Scripting Host (.wsh, .js and .vbs)**. Por esta razón, se decide actualizar el código para incluir este tipo de ficheros. Sin embargo, una semana después te vuelves a dar cuenta de que los ficheros de *Microsoft Office* pueden contener macros (.doc, .xls y otros muchos), lo que técnicamente los convierte en código ejecutable. Así que, de nuevo, actualizas la lista de ficheros ejecutables, para descubrir poco después nuevos tipos de ficheros.

Obviamente este proceso del gato y el ratón, puede continuar casi indefinidamente y, como hemos visto, no es la forma adecuada de enfocar el problema.

En lugar de tratar de detectar los patrones inválidos (extensiones de ficheros ejecutables en este caso) es siempre mejor dejar pasar sólo aquellos que sepamos con seguridad son válidos, y rechazar todos los demás.

8.5. «Chapuzas» criptográficas

A lo largo de vuestra carrera profesional oiréis en muchas ocasiones algo parecido a *«No, no, nosotros estamos seguros, porque usamos criptografía»*. Para eso hay un dicho entre los expertos (reales) en criptografía: *«Si crees que la criptografía puede solucionar tu problema, entonces es que no entiendes tu problema»*.

Aunque obviamente **la criptografía es importante**, por eso le hemos dedicado tantos capítulos en esta asignatura, existe la creencia en la industria de que a veces es la panacea para todos los problemas de la seguridad. Lo es sin duda, para muchos, pero **no te protege** por ejemplo, **contra errores de programación como los desbordamientos de búfer**.

En esta sección analizaremos algunos de los **errores más comunes** que la gente comete **cuando utiliza criptografía en sus desarrollos de código**, entre los que se incluyen generadores de números aleatorios defectuosos, mala derivación de claves criptográficas de contraseñas y, sobre todo, la creación de sus propias funciones criptográficas.

Números (poco) aleatorios

Es muy común que las aplicaciones necesiten **números aleatorios**, a menudo para usos criptográficos o de seguridad. Por ejemplo para crear una clave criptográfica o para ser usados en procesos de autenticación de reto/respuesta.

Por esta razón, elegir un buen algoritmo de generación de números aleatorios es esencial al escribir aplicaciones seguras. Y una de las elecciones que la mayoría de programadores hace, y no es buena, es la de la función *rand()* disponible en muchos sistemas operativos.

El problema con esta función es que es **predecible**. Como *rand()* es una función sencilla que usa el último número generado como la semilla para el siguiente, puede hacer que una contraseña, por ejemplo, que haya sido creada con esta función sea fácil de predecir.

Veamos el **código** de esta función, que siempre es la mejor manera de entender en profundidad el problema:

```
int __cdecl rand (void) {
       return(((holdrand =
       holdrand * 214013L + 2531011L) >> 16) & 0x7fff);
}
```

Como ves, se trata de una función extremadamente sencilla, que se basa en un concepto matemático llamada **congruencia lineal**. ¿Y por qué esta función no es buena?

Para ello hay que entender que un buen **generador de números aleatorios** debe poseer **tres propiedades**: **generar números distribuidos de forma uniforme**, **los valores deben ser impredecibles** (es decir, conocer uno de ellos no debe dar ninguna pista sobre el siguiente) **y tener un ciclo de gran longitud** (el ciclo es el momento en el que el generador comienza a repetir los números que ya había generado previamente).

El problema con los generadores basados en congruencias lineales es que cumplen con el primer requisito, pero fallan estrepitosamente con el segundo. Por esta razón, en entornos de una cierta seguridad, es necesario llamar a otras funciones del SO, como *CryptGenRandom()*. Esta función, cuyo esquema de funcionamiento puedes observar en la **Figura 3**, cumple con las dos primeras de las tres propiedades que hemos visto anteriormente.

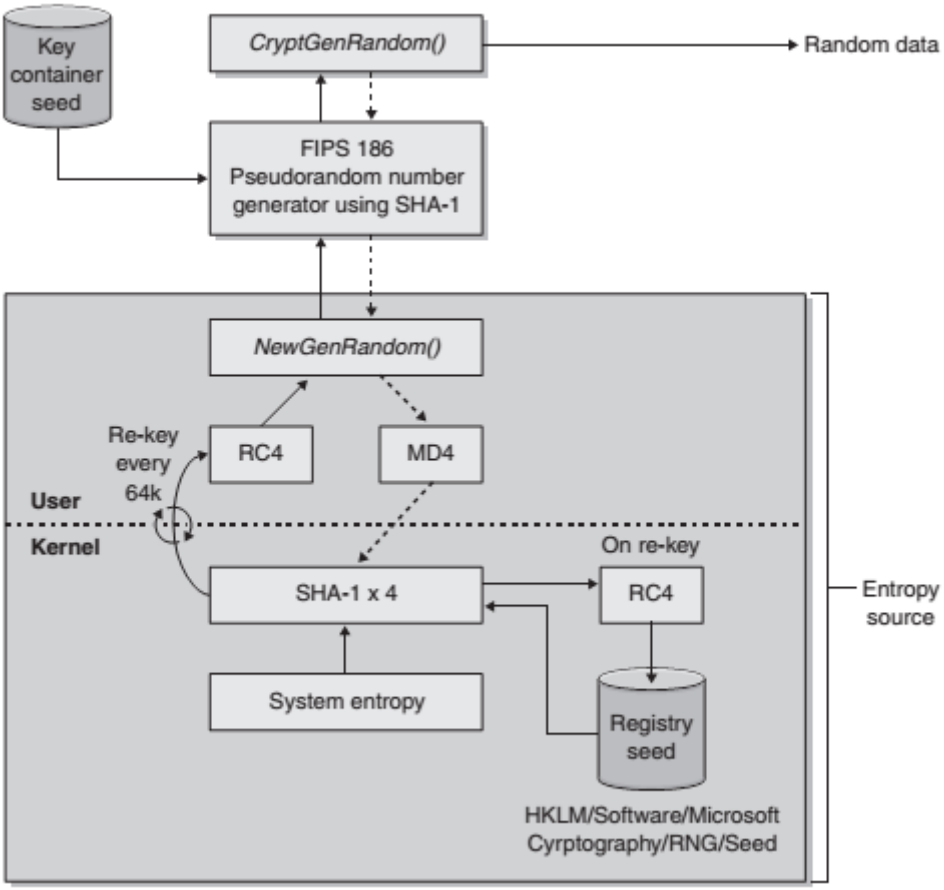


Figura 3. Esquema de funcionamiento de la función *CryptGenRandom()*

Funciones criptográficas «caseras»

No hay nada que ponga más nervioso a un criptógrafo que oír una frase del estilo *«Sí, sí, utilizamos criptografía, por supuesto. Hemos creado nuestro propio algoritmo. No confiábamos en ninguno de los algoritmos existentes, porque son todos conocidos, así que hemos creado el nuestro. Como sólo nosotros los conocemos, es mucho más seguro»*.

La experiencia demuestra que crear un buen algoritmo criptográfico, que no sea trivialmente débil, es una tarea muy difícil que lleva muchos años de estudio y que, incluso así, es fácil cometer errores. Por tanto, **nunca se debe crear ningún algoritmo criptográfico y utilizarlo en entornos de producción**.

Ilustremos este riesgo con un ejemplo. **¿Qué te parece el siguiente algoritmo de cifrado?**

```
void EncryptData(char *szKey,
       DWORD dwKeyLen,
       char *szData,
       DWORD dwDataLen) {
              for(inti=0;i< dwDataLen; i++) {
                    szData[i] ^= szKey[i % dwKeyLen];
       }
}
```

Este es un sencillo código basado en la ***función XOR*** que, aunque parezca difícil de creer, se utiliza incluso en algún producto comercial. ¿Ves por qué es trivial romper el algoritmo?

Imagina que eres un atacante que no tiene acceso a este código. Una aplicación que lo utilizara podría funcionar recibiendo el *texto claro* del usuario, «cifrándolo» con este algoritmo y guardando el resultado en un fichero, por ejemplo. Todo lo que tendría que hacer el atacante es volver a hacer XOR del texto cifrado con los datos en claro originales y «¡voilà, se obtiene la clave de cifrado!».

Como recordarás, la *función XOR* que se suele denotar por el símbolo ⊕, tiene una propiedad interesante, y es que `A ⊕B ⊕A = B`. Es por esta razón por la que se utiliza a menudo en algoritmos de cifrado «caseros», porque si calculas el XOR del *texto en claro* con una clave, se obtiene el *texto cifrado*, y si se calcula el XOR de éste con la clave, se recupera el *texto en claro*. Pero también, **si conoces el texto en claro y cifrado, ¡se puede recuperar la clave!**

Por estas razones, repetimos de nuevo que nunca intentes crear tu propio algoritmo de cifrado, ni utilizar este tipo de funciones sencillas basadas en XOR. Lo más conveniente es **utilizar algoritmos oficiales**, **bien probados y confiables**, **definidos en librerías criptográficas serias**, como *OpenSSL* o la *CryptoAPI* incluida por defecto en *Windows*.

Lo + recomendado

No dejes de leer...

Seguridad por oscuridad

En la página Web indicada más abajo, podrás encontrar un caso real de aplicación del *principio de seguridad por oscuridad*, en esta ocasión en la configuración del protocolo SSH. Observa la estrategia utilizada para cambiar el servicio SSH del habitual puerto 22. ¿Qué ventajas e inconvenientes te parece que tiene este enfoque?

Accede al documento a través del aula virtual o desde la siguiente dirección web:
http://www.securitybydefault.com/2010/06/seguridad-por-oscuridad-el-caso-del-ssh.html

Guías de desarrollo seguro

El *Proyecto OWASP* (Open Web Application Security Project) contiene cantidad de recursos interesantes relacionados con la seguridad. En este caso, nos interesan unas guías de desarrollo seguro, que contienen algunas de las directrices que hemos analizado en este capítulo, junto con otras más. Están disponibles además en español en las siguientes URLs.

Accede a los recursos a través del aula virtual o desde las siguientes direcciones web:
http://tecnologiasweb.blogspot.com.es/2010/10/guias-de-desarrollo-seguro-de-owasp.html
http://www.owasp.org/images/b/b2/OWASP_Development_Guide_2.0.1_Spanish.pdf

Buffer Overflow Attack

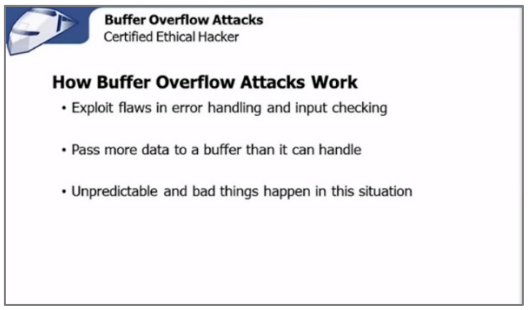
Este documento incluye gran idea e información sobre el desbordamiento de búfer como historial de vulnerabilidades, búferes, pila, registros, vulnerabilidades y ataques de desbordamiento de búfer, búfer actual sobre el flujo, código de Shell y problemas de desbordamiento de búfer.

Accede a los recursos a través del aula virtual o desde las siguientes direcciones web:
http://www.iosrjournals.org/iosr-jce/papers/vol1-issue1/B0111023.pdf?id=23

No dejes de ver...

Ethical Hacking - How Buffer Overflow Attacks Work

En el siguiente vídeo se explica cómo funcionan los ataques de desbordamiento de búfer.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:
https://www.youtube.com/watch?v=iZTilLGAcFQ

+ Información

A fondo

Writing Secure Code

Howard, M. & LeBlanc, D. (2002). *Writing Secure Code*. Microsoft Press. ISBN: 0735617228.

Este libro es sin duda uno de los mejores escritos sobre el tema del desarrollo de código seguro. Escrito de forma amena, contiene muchos ejemplos concretos sobre cada problema y sobre todo, sobre cómo evitarlos y codificarlos de la forma adecuada.

Test

1. Una de las siguientes medidas suele incluirse en la fase de seguridad en el diseño del código. ¿Sabrías identificar cuál?

A. Realizar una formación adecuada para el personal de desarrollo
B. Proteger adecuadamente el acceso a los recursos
C. Asegurarse de que la aplicación ofrece una manera de administrar su funcionalidad relacionada con la seguridad
D. Realizar pruebas intensivas antes del despliegue

2. El principio del mínimo privilegio, en lo relativo al código, hace referencia a:

A. No instalar todas las características y capacidades por defecto
B. Proteger adecuadamente el acceso a los recursos
C. El código debería necesitar el conjunto de permisos imprescindibles para su funcionamiento
D. Tratar de mantener el código tan simple como sea posible

3. Otra de las características de todo buen código es la gestión del fallo. ¿En qué consiste exactamente?

A. En prevenir el fallo de una aplicación, y evitar que se produzca
B. Aprender de los errores cometidos, para no volver a cometerlos
C. En la gestión del fallo de una aplicación, una vez que éste se ha producido
D. Tratar de minimizar la superficie de ataque de la aplicación

4. La seguridad por oscuridad es otro principio muy conocido, y utilizado, en el área. ¿Saber a qué hace referencia exactamente?

A. Usar el principio de defensa en profundidad
B. Minimizar la superficie de ataque
C. Asumir que cualquier otro sistema externo es inseguro siempre por defecto
D. Tratar de ocultar los detalles sobre la implementación del código

5. El desbordamiento de búfer es una vulnerabilidad de consecuencias potencialmente muy graves. ¿En qué consiste esencialmente?

A. Tomar el control de la ejecución de una máquina remota sobrescribiendo un búfer mal dimensionado

B. Provocar el mal funcionamiento de una aplicación a través de entrada no esperada

C. No filtrar adecuadamente los patrones de datos válidos

D. No filtrar adecuadamente los patrones de datos inválidos

6. ¿Cuál de las siguientes opciones son válidos para defenderse ante un posible ataque de desbordamiento de búfer?

A. El uso de librerías de tratamiento de cadenas especialmente diseñadas para ser seguras

B. Utilizar el principio del mínimo privilegio

C. La programación ofensiva

D. Las opciones A y C son correctas

7. A la hora de filtrar la entrada del usuario, es siempre mejor:

A. Buscar los patrones inválidos, para denegarlos

B. Aceptar sólo aquellos patrones válidos, y considerar todos los demás como inválidos

C. Buscar ambos patrones, pero aceptar sólo los válidos

D. Buscar primero los inválidos, y entre los que pasen la prueba, los válidos

8. Una de las características imprescindibles de un buen generador de números aleatorios es:

A. Rapidez

B. Eficiencia

C. La imprevisibilidad

D. Ciclo largo

9. Las congruencias lineales son un tipo de generadores de números aleatorios. ¿Es recomendable su uso en aplicaciones criptográficas?

A. Sí, siempre y cuando la semilla de la misma provenga de otro generador

B. No, excepto en aquellos casos en los que la semilla de la misma sea de calidad

C. No, en ningún caso

D. Sí, es apta para este tipo de usos

10. Entre las siguientes librerías, ¿podrías identificar aquellas que se consideran seguras para su uso en aplicaciones criptográficas?

A. OpenSSL y CryptoAPI
B. CryptoAPI y funciones XOR
C. Strsafe.h de Windows
D. Sólo OpenSSL