Criptografía y Seguridad

25 Errores de Software Más Frecuentes

Trabajo Práctico Especial 1

Civile, Juan Pablo	50453
Crespo, Alvaro	50758
Susnisky, Dario	50592
Wassington, Axel	50124

22 de mayo de 2013

1. Introducción

La lista de los 25 errores de software más frecuentes es un proyecto impulsado por **SANS Institute**, **MITRE** y expertos en seguridad informática de EEUU y Europa. Se deriva de la experiencia del desarrollo del Top20[1] de ataques informáticos implementado por SANS, y la enumeración de debilidades comunes[2] de software desarrollado por MITRE. Este *ranking* se actualiza todos los años, y para la selección de los errores en cada año, se toman en cuenta evaluaciones realizadas por distintas empresas del rubro de seguridad informática, las cuales evalúan los errores según **prevalecencia**, **importancia** y **posibilidad de *exploit***. Luego, todas estas evaluaciones son las entradas para el sistema de puntaje diseñado por MITRE (*Common Weakness Scoring System*), de donde se obtienen los errores ordenados.

Los 25 errores están categorizados en los siguientes 3 grupos

- Interacción insegura entre componentes.
- Gestión riesgosa de recursos.
- Defensas porosas.

En este año, fueron 20 las organizaciones evaluadoras, y el resultado de la evaluación arrojó el *ranking* de errores que se muestra en la siguiente sección.

1.1. Los 25 errores más frecuentes del 2011

Se presentan resaltados los errores que se analizan en este documento.

1. SQL Injection
2. OS Command Injection
3. **Classic Buffer Overflow**
4. Improper Neutralization of Input During Web Page Generation (“Cross-site Scripting”)
5. Missing Authentication for Critical Function
6. Missing Authorization
7. Use of Hard-coded Credentials
8. **Missing Encryption of Sensitive Data**
9. Unrestricted Upload of File with Dangerous Type
10. Reliance on Untrusted Inputs in a Security Decision
11. Execution with Unnecessary Privileges
12. Cross-Site Request Forgery (CSRF)

[1] `http://www.sans.org/top20/`
[2] `http://cwe.mitre.org/`

13. **Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')**
14. Download of Code Without Integrity Check
15. Incorrect Authorization
16. Inclusion of Functionality from Untrusted Control Sphere
17. Incorrect Permission Assignment for Critical Resource
18. **Use of Potentially Dangerous Functionality**
19. Use of a Broken or Risky Cryptographic Algorithm
20. Incorrect Calculation of Buffer Size
21. Improper Restriction of Excessive Authentication Attempts
22. URL Redirection to Untrusted Site ('Open Redirect')
23. **Uncontrolled Format String**
24. Integer Overflow or Wraparound
25. Use of a One-Way Hash without a Salt

1.2. Otras vulnerabilidades

Otras vulnerabilidades que no fueron selecccionados para el *Top25* son:

26. Allocation of Resources Without Limits or Throttling
27. Improper Validation of Array Index
28. Improper Check for Unusual or Exceptional Conditions
29. Buffer Access with Incorrect Length Value
30. Inappropriate Encoding for Output Context
31. Use of Insufficiently Random Values
32. Untrusted Pointer Dereference
33. Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
34. Improper Cross-boundary Removal of Sensitive Data
35. Incorrect Conversion between Numeric Types
36. NULL Pointer Dereference
37. Improper Enforcement of Behavioral Workflow
38. Missing Release of Resource after Effective Lifetime
39. Information Exposure Through an Error Message
40. Expired Pointer Dereference
41. Missing Initialization

2. Errores

2.1. Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

2.1.1. Descripción del error

Esta vulnerabilidad ocurre cuando un programa copia un buffer de entrada a otro de salida, sin verificar que el tamaño del buffer de entrada sea menor que el tamaño del buffer de salida. Esto es lo que se conoce como **buffer overflow**.

Una condición de buffer overflow existe cuando un programa intenta poner más datos en un buffer de los que éste puede contener, o cuando intenta poner datos en un área de memoria fuera de los límites del buffer. El más simple de los errores, y la causa más común de los buffer overflows, es el típico caso en el que un programa copia un buffer sin restringir cuanto se está copiando efectivamente. Existen otros casos, pero la existencia de un overflow clásico sugiere fuertemente que el programador no está considerando las más básicas reglas de seguridad.

Previamente se conocía esta vulnerabilidad con el nombre de **Unbounded Transfer ('Classic Buffer Overflow')**. El 14 de Octubre de 2008 se le cambió el nombre por el actual.

2.1.2. Terminología

Términos alternativos

- **Buffer Overrun**
- **Unbounded Transfer**

Muchos casos que son llamados "buffer overflows" son susbstancialmente diferentes al overflow "clásico", incluyendo diferentes tipos de bugs que se basan en técnicas de overflow, como errores de signos de enteros, integer overflows y bugs de formatos de strings. Esta terminología imprecisa puede hacer que sea difícil determinar a cual variante se este refiriendo.

2.1.3. Detalles técnicos del error

Categoría	Manejo Riesgoso de Recursos
Plataforma	Lenguajes de Programación
Tiempo de Introducción	Implementación
Lenguaje	C, C++, Assembly
Probabilidad de *exploit*	Alta a Muy Alta

2.1.4. Ejemplos de código

Ejemplo en C

El siguiente código, escrito en C, le pide al usuario que ingrese su apellido y luego intenta guardar el valor ingresado en el array `last_name`.

```
    char last_name[20];
    printf ("Enter your last name: ");
    scanf ("%s", last_name);
```

El problema con este código es que no restringe o limita el tamaño del nombre ingresado por el usuario. Si el usuario ingresa "Very_very_long_last_name", que tiene 24 carácteres de largo, entonces ocurrirá un buffer overflow, ya que el array solo puede contener 20 carácteres en total.

Ejemplo en C

El siguiente ejemplo, también un fragmento de código en C, intenta crear un copia local de un buffer para hacer alguna manipulación de los datos.

```
    void manipulate_string(char* string){
    char buf[24];
    strcpy(buf, string);
    ...
    }
```

Sin embargo, el programador no se asegura que el tamaño de los datos apuntados por el string entren en el buffer local y copia ciegamente los datos, con la función potencialmente peligrosa *strcpy()*. Esto puede tranquilamente resultar en una condición de **buffer overflow** si un atacante puede influenciar los contenidos del parámetro string.

Ejemplo en C

El fragmento siguiente llama a la función *gets()* en C, que es inherentemente insegura.

```
    char buf[24];
    printf("Please enter your name and press <Enter>\n");
    gets(buf);
```

El programador intenta copiar ciegamente desde `STDIN` al buffer sin restringir cuanto se copia. Esto permite al usuario proveer una string que sea más larga que el tamaño del buffer, resultando en una condición de overflow.

Ejemplo en C++

En el siguiente ejemplo, escrito en C++, un servidor acepta conexiones de un cliente y procesa su pedido. Después de aceptar la conexión, el programa obtendrá la información del cliente usando el método *gethostbyaddr()*, copiará el hostname del cliente conectado a una variable local y también lo imprimirá en un archivo de log.

```
    struct hostent *clienthp;
    char hostname[MAX_LEN];

    // create server socket, bind to server address and
    // listen on socket
    ...

```

```
8     // accept client connections and process requests
9     int count = 0;
10    for (count = 0; count < MAX_CONNECTIONS; count++) {
11
12        int clientlen = sizeof(struct sockaddr_in);
13        int clientsocket = accept(serversocket, (struct
              sockaddr *)&clientaddr, &clientlen);
14
15        if (clientsocket >= 0) {
16            clienthp = gethostbyaddr((char*) &clientaddr.
                  sin_addr.s_addr, sizeof(clientaddr.sin_addr.
                  s_addr), AF_INET);
17            strcpy(hostname, clienthp->h_name);
18            logOutput("Accepted client connection from host ",
                  hostname);
19
20            // process client request
21            ...
22            close(clientsocket);
23        }
24    }
25    close(serversocket);
26    ...
```

Pero el hostname del cliente conectado podría ser más largo que el tamaño allocado para la variable local. En ese caso, ocurriría un buffer overflow cuando se copie el hostname usando el método *strcpt()*.

2.1.5. Métodos de detección

- **Análisis Estático Automatizado**
  Esta vulnerabilidad puede detectarse muy a menudo utilizando herramientas de análisis estático automatizado. Muchas herramientas modernas usan análisis de flujo de datos o técnicas basadas en *constraints* para minimizar el número de falsos positivos. En general, no toma en cuenta consideraciones ambientales al reportar overflows. Esto puede hacer que los usuarios encuentren difícil determinar cual *warning* investigar primero. Por ejemplo, una herramienta podría reportar buffer overflows originados de argumentos de línea de comandos en un programa que no se supone que corra con *setuid* u otros privilegios especiales. Tiene una alta efectividad. Las técnicas de detección para errores relacionados con buffer overflows están más maduras que para otras vulnerabilidades.

- **Análisis Dinámico Automatizado**
  Esta vulnerabilidad puede ser detectada usando herramientas y técnicas dinámicas que interactúan con el software usando largos tests con muchas y variadas entradas, como fuzz testing (*fuzzing*), *robustness testing*, y *fault injection*. Se supone que el software puede ralentizarse, pero no debería volverse inestable, *crashear*, o generar resultados incorrectos.

- **Análisis Manual**
  El análisis manual puede ser útil para encontrar esta vulnerabilidad, pero puede no lograr la cobertura de código deseada dentro los límites de tiempo deseados.

Esto se torna complicado para vulnerabilidades donde se deben considerar todas las entradas.

2.1.6. Nivel de vulnerabilidad

Los niveles de vulnerabilidad pueden ser

- *resultante*, lo que significa que la vulnerabilidad está típicamente relacionada con la presencia de alguna otra vulnerabilidad.
- *primario*, lo que significa que la vulnerabilidad existe independientemente de la existencia de otras vulnerabilidades.

2.1.7. Consecuencias más frecuentes

- Integridad, Confidencialidad, Disponibilidad
  Los buffer overflows pueden ser usados para ejecutar código arbitrario, lo que generalmente está por fuera de las políticas de seguridad de cualquier programa, y lo cual puede resultar catastrófico. El impacto técnico es entonces, la ejecución no autorizada de código o comandos. Esto representa una falla de integridad, confidencialidad y disponibilidad.
- Disponibilidad
  Los buffer overflows generalmente llevan a *crashes*. Otros ataques que llevan a la falta de disponibilidad son posibles incluyendo poner al programa en un loop infinito. El impacto técnico es variado: puede ser un Denial Of Service (DoS), crash, exit, restart, consumo de recursos (CPU). Esto representa una falla de disponibilidad.

2.1.8. Formas de mitigar el error

Existen múltiples maneras de prevenir este error. A continuación se presenta una lista con varias de estas opciones.

- Elegir cautelosamente el lenguaje de programación. Esta estrategia debe ser tomada en la fase de análisis de requerimientos del sistema. Basta con elegir un lenguaje de programación que no permita este error o que facilite maneras de evitarlo. Por ejemplo, varios lenguajes que tienen un propio manejo de la memoria, como Java o Perl, no causan overflows.
- Usar librerías o frameworks. El uso de librerías o frameworks que no permitan o ayuden a prevenir este error también es de buena práctica. Un buen ejemplo es la *Safe C String Library (SafeStr)*. Esto debe ser tenido en cuenta en la fase de arquitectura y diseño.
- Restricciones en la compilación. Una alternativa es correr o compilar el software usando prestaciones o servicios que tengan un mecanismo automático para prevenir este problema. Esta herramienta, usada en el momento de compilar no es una solución completa ya que no detecta todos los tipos de overflow, dejando posibilidades de un ataque.

- Buenas prácticas y costumbres. Como siempre a la hora de la implementación resulta útil seguir una guía de buenas prácticas y costumbres. De entre ellas se pueden destacar:
  - Volver a chequear (*double-check*) si nuestro buffer tiene el tamaño que deseamos.
  - Al copiar buffers tener en cuenta que el buffer al que se copia sea por lo menos tan grande como el original.
  - Al estar en un ciclo, chequear que los límites del buffer estén siendo respetados, sin escribir en áreas no permitidas.
  - Si es necesario, truncar strings a longitudes razonables.
- Validar las entradas. Considerar que toda entrada ajena es maliciosa es una solución al problema, haciendo que solamente sean aceptables las entradas que estrictamente cumplan ciertas especificaciones. Es necesario considerar en este caso todas las propiedades potencialmente relevantes, incluyendo tamaño y tipo de la entrada, entre otras. Si bien las *listas negras* pueden ser útiles para detectar potenciales ataques, no son del todo confiables, permitiendo en ciertos casos que alguna entrada mal formada o maliciosa pase los requerimientos.
- Chequeos desde el lado del servidor. Por cada chequeo de seguridad que se haga desde el lado del cliente, es una buena práctica volver a chequearlo desde el lado del servidor para evitar problemas.
- Reforzar el entorno. Usar alguna característica como *Address Space Layout Randomization (ASLR)* ayuda a prevenir el problema. Otra forma de reforzar el entorno implica usar una computadora y un sistema operativo que ofrezcan capacidades como *Data Execution Protection (NX)*. Estas herramientas tampoco son una solución completa ya que no solucionan todos los posibles ataques.
- Usar funciones que incluyan parámetros de longitud. Una buena práctica es sustituir aquellas funciones sin límites de longitud, por otras que sí lo reciban por parámetro.
- Reducir los permisos o crear entornos reducidos. Al operar con los permisos correctos se puede lograr minimizar un ataque efectivo. Un efecto similar se obtiene al crear un entorno de operación con límites estrictos.

2.1.9. Ejemplos observados

Referencia	Programa/Aplicación	Resumen de la vulnerabilidad
CVE-2000-1094	AOL Instant Messenger(prev 4.3.2229)	Buffer Overflow por usar comando con argumento largo.
CVE-1999-0046	rlogin	Buffer Overflow por usar una variable de entorno larga.
CVE-2002-1337	Sendmail(5.79 a 8.12.7)	Buffer Overflow por mal parseo de comentarios.
CVE-2003-0595	WiTango Application Server and Tango 2000	Buffer Overflow, por reemplazo de un valor de una cookie por un string extramadamente largo.
CVE-2001-0191	gnuserv (XEmacs)	Buffer Overflow, por reemplazo de un valor de una cookie por un string extramadamente largo.

2.2. Uncontrolled Format String

2.2.1. Descripción del error

Esta vulnerabilidad aparece cuando se le da la posibilidad al usuario de controlar el formato de un *string*, habitualmente en funciones del estilo de `printf()`. Este error se da normalmente en los sistemas con "logueo" o "internacionalización y localización" lo que no quiere decir que no pueda pasar en un sistema que no los tenga.

2.2.2. Detalles técnicos del error

Categoría	Manejo Riesgoso de Recursos
Plataforma	Independiente.
Tiempo de Introducción	Implementación
Lenguaje	Lenguajes que soportan texto formateado (Ej: C, C++, Perl).
Probabilidad de *exploit*	Muy Alta.

2.2.3. Ejemplos de código

Ejemplo en C

```
1 int main(int argc, char **argv){
2         char buf[128];
3         ...
4         snprintf(buf,128,argv[1]);
5 }
```

Este código permite a un intruso ver el contenido del *stack* y escribir al *stack* mediante un argumento de línea de comandos que contiene una secuencia de formatos (Ej: %d , %x). Con un %x se puede leer contenido del *stack*, mientras que con un %n, un atacante puede escribir en el *stack*.

2.2.4. Métodos de detección

- **Black Box**
  No es muy efectivo.
- **Análisis automatizado**
  Hay herramientas de análisis estático que permiten localizar este tipo de vulnerabilidad.

2.2.5. Nivel de vulnerabilidad

El nivel de vulnerabilidad es *primario*, lo que significa que la vulnerabilidad existe independientemente de la existencia de otras vulnerabilidades.

2.2.6. Consecuencias más frecuentes

Estos ataques pueden violar la confidencialidad, la integridad y la disponibilidad del sistema.

- Confidencialidad
  Se puede revelar información oculta.
- Integridad y disponibilidad
  Se puede ejecutar código arbitrario.

2.2.7. Formas de mitigar el errror

- Elegir un lenguage que no sea suceptible a este error.
- Asegurarse que todos los strings de formato sean estáticos.
- Prestar atención a los *warnings* del compilador y linkeditor.

2.2.8. Ejemplos observados

Referencia	Programa/Aplicación	Consecuencia de la vulnerabilidad
CVE-2002-1825	WASD (7.1 a 8.0).0	Ejecución de código arbitrario.
CVE-2001-0717	ToolTalk	Ejecución de código arbitrario.
CVE-2002-0573	RPC daemon, Solaris(2.5.1 a 8)	Ejecución de código arbitrario.
CVE-2002-1788	nn (6.6.0 a 6.6.3)	Ejecución de código arbitrario.
CVE-2006-2480	Dia 0.94	Denial of service (DoS) y posiblemente Ejecución de código arbitrario.

2.3. Missing Encryption of Sensitive Data

2.3.1. Descripción del error

Esta vulnerabilidad sucede en aquellos casos en donde se envían datos sensibles sin haberlos encriptado a través de un canal no seguro. Al omitir las garantías de seguridad e integridad no solo se le permite a un posible atacante tomar información de la comunicación, sino también la inyección de nuevos datos, sin permitirle a ninguna de las dos partes poder distinguir la información válida de la inválida.

2.3.2. Detalles técnicos del error

Categoría	Defensas porosas, problemas criptográficos.
Plataforma	Independiente.
Tiempo de Introducción	Arquitectura y Diseño. Operación.
Lenguaje	Independiente.
Probabilidad de *exploit*	Alta a Muy Alta.

2.3.3. Ejemplos en código

Ejemplo en PHP

El siguiente código, escrito en PHP, muestra este tipo de vulnerabilidad. En este caso se escribe la información de inicio de sesión de un usuario en una *cookie* para que el usuario no tenga que volver a iniciar sesión (*login*).

```
1  function persistLogin($username, $password){
2      $data = array("username" => $username, "password"=>
           $password);
3      setcookie ("userdata", $data);
4  }
```

La información guardada en la *cookie* en la computadora del usuario se encuentra como texto plano. Esto expone las credenciales del usuario si su equipo resulta atacado.

Ejemplo en Java

El siguiente ejemplo, en Java, intenta establecer una conexión con un sitio para intercambiar información sensible.

```
1  try {
2          URL u = new URL("http://www.secret.example.org/");
3          HttpURLConnection hu = (HttpURLConnection) u.
               openConnection();
4          hu.setRequestMethod("PUT");
5          hu.connect();
6          OutputStream os = hu.getOutputStream();
7          hu.disconnect();
8  }
9  catch (IOException e) {
10 //...
11 }
```

Aunque se realiza con éxito una conexión, esta conexión no está encriptada y es posible que toda la información enviada o recibida desde el servidor sea accedida por posibles atacantes.

Ejemplo en C

El siguiente ejemplo, en C, intenta establecer una conexión, leer una contraseña y luego guardarla en un *buffer*.

```
1  server.sin_family = AF_INET; hp = gethostbyname(argv[1]);
2  if (hp==NULL) error("Unknown host");
3      memcpy( (char *)&server.sin_addr,(char *)hp->h_addr,hp->
           h_length);
4  if (argc < 3) port = 80;
5  else port = (unsigned short)atoi(argv[3]);
6  server.sin_port = htons(port);
7  if (connect(sock, (struct sockaddr *)&server, sizeof server) <
       0) error("Connecting");
8  //...
9  while ((n=read(sock,buffer,BUFSIZE-1))!=-1) {
10          write(dfd,password_buffer,n);
11          //...
12 }
```

Mientras resulte exitoso, este programa no encripta la información antes de escribirla en el *buffer* haciendo posible la exposición de esta información.

2.3.4. Métodos de detección

- **Análisis manual**
  Definir que información es sensible y cuál no requiere un alto conocimiento del sistema, con lo cuál un análisis manual suele ser escencial para una solución efectiva. Sin embargo, un esfuerzo manual no asegura una solución total respecto a los posibles límites de tiempo.

  Los métodos de caja negra pueden producir artefactos que necesariamente requieran análisis manual.

  Este método tiene una alta efectividad.

- **Análisis automatizado**
  Un análisis automatizado puede alertar la falta de encriptación para ciertos datos. Sin embargo un análisis humano sigue siendo requerido para distinguir información que intencionalmente no está siendo encriptada.

2.3.5. Nivel de vulnerabilidad

El nivel de vulnerabilidad es *primario*, lo que significa que la vulnerabilidad existe independientemente de la existencia de otras vulnerabilidades.

2.3.6. Consecuencias más frecuentes

Los ataques basados en falta de encriptación de datos sensibles pueden violar la confidencialidad y la integridad del sistema.

- Confidencialidad
  La aplicación no utiliza un canal seguro, como SSL/TLS, para el intercambio información sensible. De este modo, un atacante puede acceder al tráfico de información tomando paquetes de la conexión y descubrir los datos.

- Integridad
  Omitir la encriptación de datos en cualquier aplicación puede ser considerado equivalente a enviarle la información a todos los miembros de las redes locala-les del emisor y el destinatario. Más aún, esta omisión permite que se agregue o modifique información haciendo para los usuarios imposible distinguir datos válidos de datos inválidos.

2.3.7. Formas de mitigar el error

En una primera instancia es necesario hacer un análisis profundo de los requerimientos de confidencialidad e integridad del sistema. Una vez pasada esta etapa basta con encriptar correctamente los datos considerados sensibles. Por último, el uso de un canal seguro como SSL es una buena práctica que ayuda a eliminar este problema.

2.3.8. Ejemplos observados

Referencia	Programa/Aplicación	Resumen de la vulnerabilidad
CVE-2009-2272	Huawei D100	Usuario y contraseña guardado como texto plano en una *cookie*.
CVE-2009-1466	Application Access Server 2.0.48	Contraseña guardada como texto plano en un archivo con permisos inseguros.
CVE-2009-0946	UserView_list.php en PHPRunner 4.2	Contraseña guardada como texto plano en la base de datos.
CVE-2007-5626	make_catalog_backup en Bacula 2.2.5	Rutina de *backup* envía la contraseña en texto plano por mail.

2.4. Path Traversal

2.4.1. Descripción del error

Esta vulnerabilidad se da cuando un programa hace uso de entrada externa para acceder a recursos del *file system*. Si no se valida y limita correctamente a qué recursos se puede acceder, un atacante puede obtener información privilegiada o vulnerar el programa y/o sistema en el que corre.

Todo sistema que haga manejo de recursos del *file system* se encuentra potencialmente afectado por esta vulnerabilidad. No depende del sistema operativo ni del lenguaje utilizado. Algunos lenguajes pueden ofrecer mecanismos para mitigar este ataque, pero deben ser correctamente configurados para esto.

2.4.2. Detalles técnicos del error

Categoría	Manejo Riesgoso de Recursos
Plataforma	Independiente.
Tiempo de Introducción	Arquitectura y Diseño. Implementación.
Lenguaje	Independiente.
Probabilidad de *exploit*	Alta a Muy Alta

2.4.3. Ejemplos de código

Ejemplo en PHP

```
1 <?php
2 include($_GET['file']);
3 ?>
```

Si este script se encuentra en un sistema *Linux* podemos obtener la lista de usuarios del sistema y sus contraseñas encriptadas haciendo un request a `/vulnerable.php?file=/etc/passwd`.

2.4.4. Métodos de detección

En lenguajes donde estén disponibles, herramientas de análisis estático pueden fácilmente detectar potenciales casos de *Path Traversal*. Es posible que se encuentre con falsos positivos, donde la funcionalidad sea deseada para administradores o usuarios con suficientes privilegios. En caso de no disponer de herramientas de análisis estático, el uso de *White Box testing* puede detectar ocurrencias de esta clase de vulnerabilidad.

2.4.5. Nivel de vulnerabilidad

Los niveles de vulnerabilidad pueden ser

- *primario*, lo que significa que la vulnerabilidad existe independientemente de la existencia de otras vulnerabilidades.
- *resultante*, lo que significa que la vulnerabilidad está típicamente relacionada con la presencia de alguna otra vulnerabilidad.

2.4.6. Consecuencias más frecuentes

Un programa afectado por esta vulnerabilidad podría ser manipulado para lograr los siguientes efectos:

- Modificar archivos críticos del sistema o programa para alterar el comportamiento del mismo o otros programas en el sistema.
- Otorgarle al atacante acceso a partes seguras del sistema alterando archivos de seguridad.
- El atacante podría obtener acceso de lectura a archivos con información sensible, potencialmente otorgándole información de otros usuarios del sistema o información de seguridad del mismo.

2.4.7. Formas de mitigar el error

Existen numerosas maneras de proteger un programa de este tipo de error. Como en toda vulnerabilidad de entrada externa maliciosa, se debe tratar como potencialmente hostil toda entrada, incluso si viene de una fuente confiada.

Se puede utilizar técnicas de *White listing* y *Black listing*. Es decir comparar la entrada contra una lista de valores conocidos que sean considerados seguros o no. En este caso, el uso de *Black listing* no alcanza para protegerse, ya que es posible que la lista no contenga todas las posibles combinaciones maliciosas.

Ejecutar el programa dentro de un *sandbox* puede ofrecer protección limitada. Pero no ofrece protección a los recursos dentro del alcance del programa. Un atacante podría obtener acceso privilegiado al programa, y usar este acceso para forzar al programa a salir del *sandbox*, obteniendo acceso al sistema.

La mejor forma de evitar este tipo de problema es hacer uso de funciones de *path canonicalization*. Si se resuelve el *path* resultante de la entrada del usuario, se puede verificar correctamente usando *white listing* que no se intente manipular recursos fuera de los permitidos.

2.4.8. Ejemplos observados

Referencia	Programa/Aplicación	Resumen de la vulnerabilidad
CVE-2009-4194	Golden FTP Server 4.30	Un servidor FTP permite el borrado de archivos arbitrarios usando ".." con en el comando `DELE`.
CVE-2009-4053	Home FTP Server 1.10.1.139	Un servidor FTP permite la creación de directorios arbitrarios usando ".." con el comando `MKD`.
CVE-2010-0012	Transmission 1.22	Sobreescritura de archivos mediante el uso de ".." en un archivo torrent.
CVE-2009-4581	RoseOnlineCMS 3 B1	Programa PHP permite la ejecución de código arbitrario usando ".." a archivos suministrados a la función `include()`.

2.5. Use of Potentially Dangerous Function

2.5.1. Descripción del error

Esta vulnerabilidad consiste en la invocación de una función potencialmente peligrosa, que podría introducir una vulnerabilidad si se usa incorrectamente, con algún conjunto de parámetros, pero que también puede ser invocada de forma segura.

2.5.2. Detalles técnicos del error

Categoría	Manejo Riesgoso de Recursos
Plataforma	Lenguajes de Programación
Tiempo de Introducción	Arquitectura y Diseño. Implementación.
Lenguaje	C, C++
Probabilidad de *exploit*	Alta

2.5.3. Ejemplos de código

Ejemplo en C

```
1    void manipulate_string(char * string){
2        char buf[24];
3        strcpy(buf, string);
4        ...
5    }
```

El fragmento anterior intenta crear una copia local de un *buffer* para manipular sus datos, pero no se asegura que los datos apuntados por el parámetro *string* entren en el *buffer* local y se copia ciegamente los datos, con la función potencialmente peligrosa *strcpy()*. Esto puede tranquilamente resultar en una condición de **buffer overflow** si un atacante puede influenciar los contenidos del parámetro string. En este caso, se podría evitar muy fácilmente haciendo una verificación del tamaño del parámetro string, o utilizando correctamente la función segura *strncpy()*.

2.5.4. Métodos de detección

La detección de este tipo de funciones resulta muy díficil, principalmente porque depende de un profundo conocimiento del programador que usa la función potencialmente peligrosa.

- Investigar y entender completamente el funcionamiento de una función, previo a su uso en un entorno de producción.
- Utilizar herramientas de análisis automático de código o compiladores que detecten el uso de este tipo de funciones.
- Mantenerse actualizado sobre las noticias y los parches de seguridad acerca del lenguaje o *framework* que se utiliza a diario.

2.5.5. Nivel de vulnerabilidad

El nivel de vulnerabilidad es *primario*, lo que significa que esta vulnerabilidad existe independientemente de la existencia de otras vulnerabilidades.

2.5.6. Consecuencias más frecuentes

Si la función se utiliza incorrectamente, es decir se le pasan ciertos parámetros que generan el comportamiento peligroso, entonces se podrían generar problemas de seguridad. Por ejemplo,

- Ejecución de código arbitrario por parte de un atacante.
- Que usuarios locales comunes, o atacantes remotos ganen privilegios que no deberían tener.

2.5.7. Formas de mitigar el error

- Identificar una lista de funciones de APIs prohibidas, y prohibir su uso a los programadores, proveyendo alternativas seguras.
- Utilizar o configurar herramientas de análisis automático de código o compiladores para detectar el uso de estas funciones. (Ej: *banned.h* de SDL de Microsoft).

2.5.8. Extras

Esta vulnerabilidad es diferente de la CWE-242 (Use of Inherently Dangerous Function), ya que esta última abarca funciones con tales problemas de seguridad que nunca podría ser considerada segura. Algunas funciones, si son usadas correctamente, no suponen directamente un riesgo de seguridad, pero puede introducir una debilidad si no son invocadas correctamente. Estas son las llamadas funciones potencialmente peligrosas. Un ejemplo muy conocido es la función *strcpy()*. Cuando se le provee un buffer de destino más largo que la fuente, *strcpy()* no causará un **buffer overflow**. Sin embargo, se utiliza de manera errónea tan frecuentemente, que algunos programadores prohiben por completo su uso.

2.5.9. Ejemplos observados

Referencia	Programa/Aplicación	Resumen de la vulnerabilidad
CVE-2007-1470	LIBFtp 5.0	Librería con multiples buffer overflows usando sprintf() and strcpy().
CVE-2009-3849	HP OpenView Network Node Manager(7.01-7.53)	Buffer overflow usando strcat().
CVE-2006-2114	SWS web Server 0.1.7	Buffer overflow usando strcpy().
CVE-2006-0963	STLport 5.0.2	Buffer overflow usando strcpy().
CVE-2011-0712	Linux kernel(2.6.38-rc4-next-20110215)	Uso vulnerable de strcpy() cambiado a uso seguro de strlcpy().
CVE-2008-5005	IMAP Toolkit(2002-2007), Alpine 2.00, Panda IMAP	Buffer overflow usando strcpy().

3. Conclusiones

Luego de realizar la investigación de cinco de los errores más frecuentes en el software se sacaron las siguientes conclusiones.

- Muchos de los errores pueden evitarse, si se los tiene en cuenta desde el momento en que se comienza a diseñar el sistema.
- Varios de los errores son fácilmente solucionables y no requieren de un desarrollo prolongado que requiera grandes costos.
- Deben analizarse qué errores considerar para el sistema que se está desarrollando, según su probabilidad de ocurrencia, y daños que puedan causar. Es muy fácil generar paranoia, y desarrollar técnicas para evitar ataques que nunca sucederían por el contexto en donde está el software, o puedan resolverse mediante la utilización de herramientas ya desarrolladas, y no tengan que ser contempladas en el desarrollo del sistema.
- Identificar aquellos ataques a los que se sabe que el sistema queda expuesto, y definir un plan de contingencia. Muchas veces, es menos costoso desarrollar una técnina para recuperarse de un ataque, que desarrollar una técnica para evitarlo.
- Muchos de los errores ya están contemplados en frameworks, por lo tanto, la elección de un framework adecuado es muy importante, para no reimpementar funcionalidades. No dejarse engañar por la belleza de las páginas de frameworks poco conocidos, o con pocos años de desarrollo.