Criptografía y seguridad

TRABAJO PRÁCTICO ESPECIAL

Los 25 errores de Software más frecuentes

2 de Junio de 2014

Alan Pomerantz

Juan José Marinelli

Agustin Marseillan

1. Abstract

En el presente informe se expone la investigación de cuatro de los veinticinco errores presentes en el software más frecuentes, dentro del ámbito informático. Se procede a introducir el tema que contempla a dichos errores, y luego se brinda una descripción técnica del error, junto a ejemplos de código, métodos de detección, niveles de vulnerabilidad y consecuencias sobre un posible exploit sobre dicho error, y por último, formas de mitigar dichos errores. Los errores seleccionados fueron el 2 (dos) "Improper Neutralization of Special Elements used in an OS command ('OS Command Injection'), 7 (siete) "Use of Hard-coded Credentials" , 12 (doce) "Cross-Site Request Forgery (CSRF)" y 17 (diecisiete) "Incorrect Permission Assignment for Critical Resource".

2. Introducciòn

Existe una lista ordenada bajo un ranking, conteniendo los errores más frecuentes dentro del ámbito informático, de los que uno como programador debe tener cuidado de no generar, ya que un atacante puede sacar provecho de la situación, generando escenarios no deseables dentro del flujo general dentro de un programa. La lista, elaborada en el año 2011, se puede encontrar online en la siguiente direccion: *http://cwe.mitre.org/top25/*[1]
El presente informe selecciona 4 de estos 25 errores más frecuentes y propone un análisis detallado del mismo.

3. Errores seleccionados

Los errores seleccionados fueron elegidos por la cátedra para que los analice el grupo AliceBobMallory.

3.1 OS Command Injection

3.1.1 Descripción del error

El error OS Command Injection convive en la idea de que un atacante tome control del input con el cual un programa puede llamar a otro, dentro del mismo SO (Sistema Operativo). Un atacante entonces, podría

[1] Sitio web perteneciente a la CWE (Community Weakness Enumeration)

ejecutar sus propios comandos. Comparte las similitudes con cualquier tipo de inyección (Véase SQL Injection), en la que se hace aplicación de una debilidad a la hora de comprobar el input en un programa.

Facilidad de Detección del error: Sencilla
Consecuencias: Ejecución de código no deseado
Frecuencia del ataque : Alta
Costo de reparaciòn: Medio
Prevalencia de la debilidad: Medio
Conocimiento de causa del atacante: Alto

3.1.2 Detalles técnicos del error

El escenario generalmente sucede en web, donde un atacante no tiene acceso directo al sistema operativo, y puede ejecutar comandos que pueden ser no deseados, o en sistemas que tienen jerarquía de permisos, y donde un atacante puede ejecutar comandos para los cuales de otra forma no tendría permisos. Tambien es conocido como Shell Injection, ya que se aplica sobre terminales de comandos. El error es independiente del lenguaje.
Existen dos tipos de inyección de comandos:

- Una aplicación intenta ejecutar un programa del cual tiene control. El mismo utiliza un input que es externo, y el atacante logra interponerse para ejecutar un programa de su interès.
- Una aplicaciòn acepta un input externo, con el cual decide que programa va a correr luego, asi como tambien què comando utilizarà, de esta manera, la aplicaciòn retransmite el input externo, directamente al sistema operativo.

3.1.3 Ejemplos de código

Un ejemplo de código sería el siguiente:

system("nslookup [HOSTNAME]")
En donde el atacante, respetando la sintaxis de la terminal, podrìa poner en la variable HOSTNAME un comando de su propiedad, entonces luego de la ejecucion del nslookup, se ejecutarìa su comando.
Otros ejemplos:
(Lenguaje PHP)

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

Nada previene que un usuario malicioso coloque en $userName algun comando linux como ";rm –rf/". Lo que conduciria a que el comando final quede como: ls –l/home/;rm –rf/

3.1.4 Métodos de detección

Para poder detectar este error, se utilizan distintas técnicas. Entre algunas de ellas se destacan:

- Automated Static Analysis: Se usan herramientas de la jerarquía, analizando el data-flow o flujo de datos dentro del programa, en búsqueda de porciones de código en donde pueda llegar un usuario externo ingresar input al programa. Puede arrojar falsos positivos en la detección de casos en que API's de terceros utilizan llamadas al Sistema Operativo, y éstas son deseadas.
- Automated Dynamic Analysis: Se utilizan herramientas dinámicas y técnicas que interactúan con el código utilizando sets de testeo. Se aplica Fuzz Testing, robustness-testing y fault-injection.
- Manual Static Analysis: Se realizan tests de caja blanca, en dónde se pone a prueba el input del programa de manera manual, otorgando la mayor efectividad, frente a los otros métodos de detección y mitigación.

3.1.5 Nivel de vulnerabilidad y consecuencias posibles de un exploit sobre ese error

Un usuario malintencionado podría hacer uso de la ejecución de cualquier código de su autoría, o lograr ejecutar programas para los cuales no cuenta con los permisos o privilegios necesarios. Una vez que se toma el control del sistema, las consecuencias son inmediatas.

3.1.6 Formas de mitigar y/o evitar el error

- Utilizar un whitelist o lista blanca de comandos aceptados o esperados, y descartar todas las secuencias que no se correspondan con alguna marcada en esta lista.
- Analizar métricas del input, en base a conocimientos de las mismas. Ya sea como longitud esperada, tipo de input, sintaxis, etc.
- Literalizar, escapando ("Quote") al input, evitando caracteres especiales (Técnica conocida en todos los tipos de Injection)
- Utilizar librerías de terceros que proponen una solución transparente a lo que es pasaje de parámetros a una aplicación.
- En aplicaciones donde se realice una validación del lado cliente, esperar encontrar su par correspondiente del lado servidor.
- Limitar el acceso de la aplicación (su nivel de permisos o privilegios), para que cuando un atacante haga control de la misma, no pueda generar o desatar acciones, que superen al nivel de permisos que requiera la aplicación atacada.

3.2 Use of Hard-Coded Credentials

El principio de éste error, radica en la presencia de credenciales "hardcodeadas", ya sean contraseñas o claves criptográficas. dentro del mismo código.

Facilidad de Detección del error: Moderada
Consecuencias: ByPass a la seguridad
Frecuencia del ataque : Rara
Costo de reparación: Medio a Alto
Prevalencia de la debilidad: Medio
Conocimiento de causa por el atacante: Alto

3.2.1 Descripción del error

La práctica de hard-codear contraseñas y claves criptográficas (de aplicaciones) dentro de un programa, es una mala práctica que se suele cometer en etapa de desarrollo, ya que, es la manera más rápida (e insegura) y sencilla de poder operar con las mismas. El uso de la misma contraseña a lo largo de todo el sistema, para varios propósitos, también compromete a la seguridad de todo el sistema, cuando la misma es descifrada. Se espera de alguna manera, poder separar la presencia de credenciales y contraseñas, de lo que es el código de la aplicación en sí mismo.

3.2.2 Detalles técnicos del error

Existen dos variantes de lo que es el error:

Inbound: Un sistema de autenticación que chequea contra las credenciales "hardcodeadas" dentro del mismo sistema

Outbound: El software se conecta con otro sistema, ajeno al propio, utilizando estas credenciales "hardcodeadas" para autenticarse exitosamente

Este tipo de error, el número siete en el ranking según la CWE, es bastante frecuente en

aplicaciones mobile y web, y es totalmente independiente del lenguaje.

3.2.3 Ejemplos de código

A continuación se presentan dos ejemplos de código, uno desarrollado en C/C++, y otro en Java, en donde se hace uso de credenciales hardcodeadas (directamente en el código, o en un archivo de propiedades)

Ejemplo C/C++ :

```
int VerifyAdmin(char *password) {
  if (strcmp(password, "Mew!")) {

    printf("Incorrect Password!\n");
    return(0)
  }
  printf("Entering Diagnostic Mode...\n");
  return(1);
}
```

fig1. Ejemplo C/C++

Password Mew hardcodeada directamente en el código

Ejemplo Java:

```
# Java Web App ResourceBundle properties file
...
webapp.ldap.username=secretUsername
webapp.ldap.password=secretPassword
...
```

fig2. Ejemplo JAVA

Lectura de API Key & API Secret en un archivo de configuración

3.2.4 Métodos de detección

Se pueden realizar distintos métodos de detección para este problema, alguno de ellos:

- Black Box: Se buscan los archivos de configuración en búsqueda de credenciales que residan en el mismo.
- Automated Static Analysis: Técnicas automatizadas de test de caja blanca, en búsqueda de credenciales residentes en archivos de configuración de la aplicación.
- Manual Static Analysis: Se realiza análisis manual del código. Se pueden analizar archivos de configuración.
- Manual Dynamic Analysis: Se utilizan herramientas de monitoreo para ver como el programa interactúa con el Sistema Operativo y con la red (network). Ésta técnica se aplica generalmente cuando no se tiene acceso al código fuente.

3.2.5 Nivel de vulnerabilidad y consecuencias posibles de un exploit sobre ese error

Las consecuencias al obtener acceso a alguna de estas credenciales son triviales y se enumeran:

- Obtención de una cuenta no autorizada por un atacante en caso de que se almacenen contraseñas
- Uso indebido de servicios o features que pueden llevar a la obtención sensible o no autorizada de información relevante.

3.2.6 Formas de mitigar y/o evitar el error

Alguna de las maneras de intentar mitigar o evitar el error son las siguientes:

- Limitar desde donde se accede a los features
- Guardar credenciales en lugares fuera del código y protegerlos de una forma apropiada
- Credenciales con vencimiento que deban ser ingresadas por un administrador
- Limitar las acciones a llevar a cabo por el front-end desde el back-end

3.3 Cross-Site Request Forgery (CSRF)

3.3.1 Descripción del error

Los Cross-Site request forgery son un tipo de ataque en el que requests "maliciosos" son enviados por un usuario logueado y confirmado. Dentro del ranking se posiciona en el puesto número 12
La diferencia con cross-site scripting, es que en CSRF, se aprovecha la confianza que el sitio tiene en el usuario, mientras que en XSS, se toma provecho de la confianza del usuario por el sitio.

Entonces, categorizando un poco:

- Se da en sitios que dependen de la identidad del usuario
- Sacan ventaja de la confianza del sitio en la identidad del usuario
- Hacen que el browser del usuario mande requests HTTP a un sitio conveniente
- Estas requests tienen efectos colaterales o dañinos

Facilidad de Detección del error: Moderada
Consecuencias: Ejecución de código no deseado. Perdida de datos
Frecuencia del ataque : Frecuente
Costo de reparación: Alto
Prevalencia de la debilidad: Alto
Conocimiento de causa por el atacante: Medio

3.3.2 Detalles técnicos del error

Se puede ejecutar en cualquier browser o sitio que importe recursos directamente de otros usuarios y no los valide o que importe recursos de manera insegura.

3.3.3 Ejemplos de código

- Alice está en una sala de chat en la que se encuentra Mallory
Mallory le manda una imagen a Alice de la siguiente manera:
Mallory: Hola Alice! Mirá el perrito que me compré

<img src="http://bank.example.com/withdraw?account=Alice&amount=1000000&for=Mallory">

Example Language: **HTML**

```
<form action="/url/profile.php" method="post">
<input type="text" name="firstname"/>
<input type="text" name="lastname"/>
<br/>
<input type="text" name="email"/>
<input type="submit" name="submit" value="Update"/>
</form>
```

Example Language: **PHP**

```
// initiate the session in order to validate sessions

session_start();

//if the session is registered to a valid user then allow update

if (! session_is_registered("username")) {

  echo "invalid session detected!";

  // Redirect user to login page
  [...]

  exit;
}

// The user session is valid, so process the request
// and update the information

update_profile();

function update_profile {
  // read in the data from $POST and send an update
  // to the database
  SendUpdateToDatabase($_SESSION['username'], $_POST['email']);
  [...]
  echo "Your profile has been successfully updated.";
```

Example Language: **HTML**

```
<SCRIPT>
function SendAttack () {
  form.email = "attacker@example.com";
  // send to profile.php
  form.submit();
}
</SCRIPT>

<BODY onload="javascript:SendAttack();">

<form action="http://victim.example.com/profile.php" id="form" method="post">
<input type="hidden" name="firstname" value="Funny">
<input type="hidden" name="lastname" value="Joke">
<br/>
<input type="hidden" name="email">
</form>
```

fig. 3. Ejemplo del error en código PHP

En este ejemplo se ve la impostación de un formulario, con la ejecución de un script en el evento OnLoad del mismo, en donde se hace una llamada (en el action del formulario) a un

sitio ajeno a la aplicación en donde se enviará la información obtenida (en este caso se altera un perfil de usuario, modificando su email)

3.3.4 Métodos de detección

La mejor manera de detectar el error, es chequear todos los links y forms del sitio y verificar que sean seguros. Es muy difícil de automatizar ya que depende mucho de las reglas del negocio.

3.3.5 Nivel de vulnerabilidad y consecuencias posibles de un exploit sobre ese error

Nivel de vulnerabilidad: Alto, es un ataque sencillo si se encuentra un request que posea valor para el atacante.

Consecuencias: Se puede perder información, Dejar que se ejecute código no deseado en nuestro servidor. Además se pueden alterar, como se ve en el ejemplo, propiedades de un usuario dentro de una aplicación.

3.3.6 Formas de mitigar y/o evitar el error

Usar librerías o frameworks que no permiten que este error ocurra.

Asegurarse de que la aplicación esté libre de XSS

La mayoría de la gente lo soluciona poniendo un token en el form y en la cookie de session. De esta manera se envían los 2 tokens en cada request y se chequea que sean válidos. El token debe ser difícil de "adivinar".

BUEN DISEÑO WEB: No usar GET para acciones que modifiquen el estado del servidor.

3.4 Incorrect Permission Assigment for Critical Resource

Cuando a un recurso se le dan más permisos de los que se debería, este queda expuesto a ser modificado por estos actores indeseados. Esto es peligroso cuando este recurso es configuración del programa, datos de ejecución del mismo, o información de usuarios sensible.

3.4.1 Descripción del error

Facilidad de Detección del error: Sencilla

Consecuencias: Ejecución de código no deseado, perdida de información

Frecuencia del ataque : Frecuente

Costo de reparación: Bajo a Alto

Prevalencia de la debilidad: Medio

Conocimiento de causa por el atacante: Alto

3.4.2 Detalles técnicos del error

No depende del lenguaje de programación, se puede dar en cualquier sistema operativo.

3.4.3 Ejemplos de código

Todos los ejemplos de código son básicamente crear archivos con los permisos mal, por lo que otro usuario podría venir y leer datos críticos y modificar datos de configuración.

Example Language: **PHP**

```
function createUserDir($username){
  $path = '/home/'.$username;
  if(!mkdir($path)){
    return false;
  }
  if(!chown($path,$username)){
    rmdir($path);
    return false;
  }
  return true;
}
```

fig.4. Ejemplo del error en código PHP

En este error observamos que se crea un directorio, omitiendo el parámetro opcional, entonces, asume permisos por default (0777) para el directorio.

Este default permite a cualquier usuario leer y escribir el directorio

3.4.4 Métodos de detección

Análisis estático/dinámico automatizado: Sirve para detectar llamadas a funciones que pueden tener permisos inválidos, pero como los permisos para cada cosa dependen del programa, puede devolver falsos positivos.

Análisis manual: Se puede detectar con este análisis, sobre todo cuando el error tiene reglas de negocio asociadas.

Análisis manual estático/dinámico: Puede ser efectivo para detectar el uso de ciertas funciones y permisos, luego el analista puede ver estas funciones y permisos y hacer evaluarlas.

Caja negra: Se usa cuando no se tiene el código fuente, puede servir para detectar errores de privilegios y acceso no permitido.

3.4.5 Nivel de vulnerabilidad y consecuencias posibles de un exploit sobre ese error

Vulnerabilidad: Alta.

Consecuencias: Perdida de información, ejecución de código indeseado.

3.4.6 Formas de mitigar y/o evitar el error

Cuando se usa un recurso crítico, chequear que este tenga los permisos correctos.

A nivel arquitectura de aplicación, definir áreas o roles de usuario, definir estrictamente los permisos, grupos y privilegios de cada rol/área y que estos estén definidos en un solo lugar.

Para que menos archivos estén en peligro, correr la aplicación en un ambiente virtualizado, aunque los archivos que cree la aplicación en este ambiente serán igualmente sujetos a errores.

Cuando se instala o arranca el programa, setear las máscaras de archivos a lo más restrictivo posible para el que programa corra.

Para todos los archivos ejecutables, de configuración y librerías, hacer que solo se puedan leer y escribir por el administrador del sistema.

No asumir que el que ejecute el programa va a hacer los cambios listados en la documentación, hacerlo uno mismo.

4. Referencias

** CWE/SANS Top 25 Most Dangerous Software Errors http://cwe.mitre.org/top25/*

** Computer Security – Art and Science, Matt Bishop, Addison-Wesley, 2004*

** How to Break Code, Greg Hoglund, Gary Mc Graw, Addison-Wesley, 2004*

**https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project*