Ingeniería Inversa de Código en la Seguridad Informática como forma de detección de vulnerabilidades

La Ingeniería Inversa, no es solo lo que se conoce como Reversing. Es una ciencia que aplicada al servicio de la Seguridad Informática, sirve para detectar vulnerabilidades y riesgos de seguridad que no podrían ser detectados ni siquiera mediante herramientas automatizadas de penetración /Pentesting/. ¿Qué es, cómo se utiliza y para qué nos puede servir la Ingeniería Inversa de Código en Seguridad?

El pasado 12 y 13 de diciembre de 2013, tuvo lugar en Buenos Aires, el evento de seguridad informática ***A&D Security Conference***. El mismo se llevó a cabo en las instalaciones del Instituto de Tecnología ORT, cito en la calle Río de Janeiro 509 de la Ciudad Autónoma de Buenos Aires (República Argentina).

Durante la jornada del viernes 13, estuvo a mi cargo una charla sobre Ingeniería Inversa de Código aplicada en la Seguridad Informática para detectar vulnerabilidades, cuyo resumen transcribo en las siguientes líneas.

Introducción: Ingeniería de Software vs Reversing

Antes de comenzar a explicar el tema en sí mismo, hice una diferenciación entre el tema de la charla en sí mismo y lo que se conoce como *Reversing*.

Como *Reversing* se conoce a una de las tantas técnicas de la Ingeniería Inversa y muchas veces, se cree erróneamente que dicha técnica (o mejor dicho el término en sí), es sinónimo de Ingeniería Inversa.

Sin embargo, **el *reversing* como técnica, consiste en manipular binarios encapsulados para obtener el código fuente.**

Esto es, por ejemplo, el uso que se da en la Ingeniería Inversa de Hardware, cuando lo que se desea es obtener el firmware -generalmente privativo- y se utilizan herramientas de desensamblado.

Lo mismo se da en la Ingeniería inversa de Software, cuando lo que se desea, es obtener el código fuente sin compilar (generalmente, se efectúa sobre Software privativo utilizando herramientas de *descompilación*).

Luego, comenté que dentro de lo que se denomina Ingeniería Inversa de Software, no todo termina en la *descompilación*.

También existe la Ingeniería Inversa sobre el código fuente obtenido (ya sea el que se logró *descompilar* o, sobre código no compilado ya disponible) y también (y muy importante), se aplica la ingeniería inversa para GENERAR código, es decir, para desarrollar Software, mediante técnicas como TDD *(Test-Driven Development)*.

Ingeniería Inversa y Pentesting ¿Cuál y cuándo?

Entre el *Pentesting* y la Ingeniería Inversa existe una gran diferencia: mientras el primero busca, de forma automatizada, fallas que implican riesgos de seguridad conocidos, la Ingeniería Inversa de Código recurre pura y exclusivamente a al razonamiento lógico deductivo en busca de hallazgos que permitan no solo corregir el problema sino también, tomar decisiones.

Las herramientas de ***Pentesting*** basan su búsqueda en lo conocido, es decir, en la clasificación de vulnerabilidades que todos conocemos. Pero ¿qué sucede cuando comprobamos que la aplicación ha sido vulnerada y sin embargo no logramos detectar el punto de inflexión mediante test de penetración? Esto se conoce como **falsos negativos** y es una de las "excusas perfectas" para **implementar la Ingeniería Inversa de código**.

Vulnerabilidades no detectables mediante test de penetración, suelen ser ciertos errores de diseño. Éstos son contemplados por la herramientas de *pentesting*, pero sin embargo, la mayoría de las veces, un mal diseño (o mala decisión), no es un error si éste, es generado de forma ex profesa. Un claro ejemplo de malas de decisiones de diseño que no pueden contemplarse como error, son los ***backdors***, puertas traseras (o accesos ocultos) que se facilitan en un sistema informático, que incluso, cuando son programados con las mejores y más sanas intensiones, continúan siendo **una de las vulnerabilidades que mayores riesgos implica**.

> *La Ingeniería Inversa de código* ***es el único método mediante el cual, se puede detectar el 100% de las vulnerabilidades*** *de un sistema, incluso aquellas vulnerabilidades no detectables con pentesting*

Paradójicamente, un *backdor* a pesar de ser uno de los riesgos de seguridad más importantes, puede encontrarse resumido en un simple y sencillo algoritmo que no podría ser detectado siquiera con la herramienta más cruda, robusta y avanzada de *pentesting*. Un buen ejemplo, sería el siguiente algoritmo en Python, que simplemente comprueba que el *hash* MD5 resultante de la unión de el *hash* MD5 del nombre de usuario y el *hash* MD5 de la contraseña, sea la combinación de la llave que abre la puerta trasera:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from getpass import getpass
from hashlib import md5

usuario = raw_input('Username: ')
clave = md5(getpass('Password: ')).hexdigest()

backdor_hash = "11a9c1b608ab1cc155777537fa63948f"

backdoor_user = md5(
    "%(userhash)s|%(pwd)s" % dict(
        userhash=md5(usuario).hexdigest(),
        pwd=clave)
    ).hexdigest()

ADMIN = True if backdoor_user == backdor_hash else False
```

Más allá de lo descriptivo del nombre de las variables (elegidas de forma ex profesa a fin de facilitar el entendimiento en términos académicos), **el *backdor* del código anterior, solo podría ser detectado mediante el análisis humano**.

Las malas decisiones de diseño tampoco son errores de diseño (un error es una acción no esperada) cuando la **inyección de código** se permite de manera intencional.

> *La **inyección de código por diseño**, ha sido la vulnerabilidad más explotada por los propios fabricantes de **aplicaciones web privativas**, para **atacar instalaciones con licencias no autorizadas***

Algoritmos como el siguiente, son capaces de permitir la inyección de código HTML que impida, por el ejemplo, el acceso al panel de control de un CRM:

```
if(strpos($_POST['email'], '!foobar0075-12a70') === False) {
    $email = SafeData::validar_email($_POST['email']);
    WebForm::send_mail($email);
} else {
    $dashboard_content = $_POST['email'];
    WebSiteConfig::save('dashboard', $dashboard_content);
}
```

El código anterior, podría encontrarse en la función de envío de un simple formulario para suscripción a boletín de novedades. Nada más inocuo. Sin embargo, si la cadena **`!foobar0075-12a70`** se insertase en el campo de formulario que espera recibir una dirección de correo electrónico, dicho *input* no sería filtrado, y código como el siguiente, podría almacenarse en una base de datos e impedir el acceso a un panel de control:

```
<div style="position: absolute;
            top: 0;
            left: 0;
            background: #000;
            color: red;
            font-size: 4em;
            text-align: center;
            min-height: 100%;"
>Copia de Software NO AUTORIZADA</div>
```

Nuevamente, ninguna herramienta de *pentesting*, podría deducir que ingresando la combinación de los caracteres `!foobar0075-12a7` y en ese mismo orden, todo un sistema informático podría quedar inutilizado.

Restricciones

La Ingeniería Inversa de código, tiene un gran enemigo: la **ofuscación**.

Un código ofuscado puede ser desde un código poco estándar e ilegible escrito por un programador poco avezado, hasta instrucciones que de forma intencional son escritas de manera “rebuscada” como es el caso del siguiente fragmento en JavaScript:

```
$easteregg=String.fromCharCode(+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[],+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[],+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[],+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[],+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
```

```
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[],+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]
+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!
+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[],+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+
[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+!+[]+
```

Fragmento de **código JavaScript ofuscado** que de forma inocua, pretende ocultar el **EasterEgg** para Chromium, que programé en el **sitio Web de la revista Hackers & Developers Magazine**. El código completo puede obtenerse en http://www.hdmagazine.org/static/js/eastereggs.js

Aplicando la Ingeniería Inversa de Código: ¿cómo leer el código fuente de un Software?

Muy frecuentemente, cuando se entrega a un programador el código fuente de un programa, éste, comienza a explorar el directorio abriendo archivos y carpetas de forma indiscriminada, siendo el orden lógico elegido, el orden alfabético de los archivos, en el mejor de los casos.

Sin embargo, esa no es la forma correcta (ni mucho menos óptima) de analizar el código fuente de un programa.

> *El código fuente de un programa solo puede ser analizado mediante Ingeniería Inversa.*

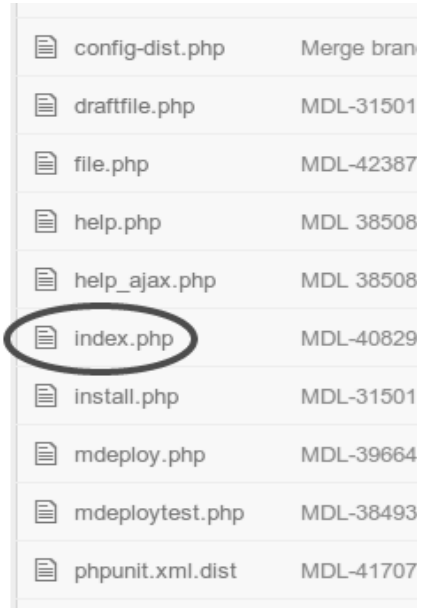
La Ingeniería Inversa de Código, se aplique con el fin que sea, consiste básicamente en **encontrar un punto de partida y seguir el rastro hasta un punto de quiebre (*breakpoint*)**.

1. Localización de un archivo de inicio válido

El punto de partida, siempre, consistirá en la localización de un archivo de inicio válido, válido tanto para la aplicación como para los fines del análisis.

Se intenta localizar el archivo principal (*main*) de la aplicación para posteriormente ubicar la instrucción de arranque que nos facilite el seguimiento de los rastros de código. Esto permitirá conocer cómo es el **mecanismo de acción de la aplicación** frente a las solicitudes del usuario.

> *Preferir lo **obvio frente a lo dudoso** y lo **simple frente a lo complejo***



La imagen izquierda muestra parte de los archivos de la carpeta raíz de ***Moodle***, un Software Libre para *e-Learning* desarrollado en PHP, cuyo repositorio oficial puede localizarse en:
https://github.com/moodle/moodle

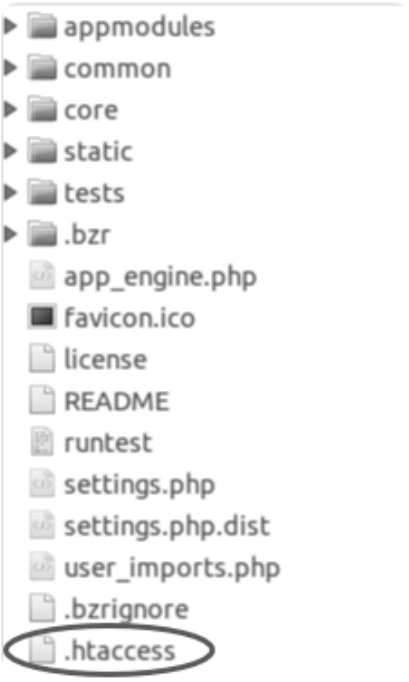
Resaltado se observa el archivo **`index.php`** puesto que, como expone la cita de la página anterior, ***"lo obvio es preferible a lo dudoso"***.

> *Frente a la duda, ir a lo seguro*

En este otro ejemplo, se elige lo seguro frente a la duda en general.

El archivo `.htaccess` es "seguro": probablemente allí se describan reglas tanto de reescritura de la URL como de archivos de inicio o raíz.

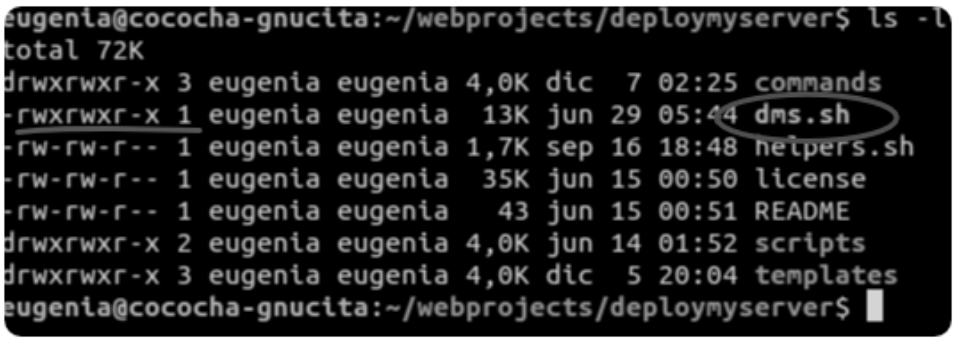
En este caso en particular, una regla de reescritura nos indica al archivo `app_engine.php` como siguiente paso.




```
1  RewriteEngine On
2  RewriteRule !(^static|favicon) app_engine.php
3  AddType image/x-icon .ico
4  RewriteRule ^favicon favicon.ico [NC,L]
5
6
```

Código fuente del archivo .htaccess

En los casos de las aplicaciones (o herramientas) desktop o de línea de comandos, siembre deben priorizarse los archivos ejecutables:



2. Localizar la instrucción de arranque

El código fuente no debe leerse de arriba hacia abajo. De hecho, las instrucciones de arranque suelen estar al final del archivo. Las únicas excepciones, son los guiones de código (o archivos de instrucción procedural directa).

Solo los archivos de instrucciones procedurales se leen siguiendo el orden de las instrucciones, un claro ejemplo, es el archivo `index.php` de *Moodle*:

```
18  /**
19   * Moodle frontpage.
20   *
21   * @package    core
22   * @copyright  1999 onwards Martin Dougiamas (http://dou
23   * @license    http://www.gnu.org/copyleft/gpl.html GNU
24   */
25
26      if (!file_exists('./config.php')) {
27          header('Location: install.php');
28          die;
29      }
30
31      require_once('config.php');
32      require_once($CFG->dirroot .'/course/lib.php');
33      require_once($CFG->libdir .'/filelib.php');
34
35      redirect_if_major_upgrade_required();
```

Si se observa el código anterior, las líneas 26 y 27 ya deberían hacernos sospechar de la primera vulnerabilidad.

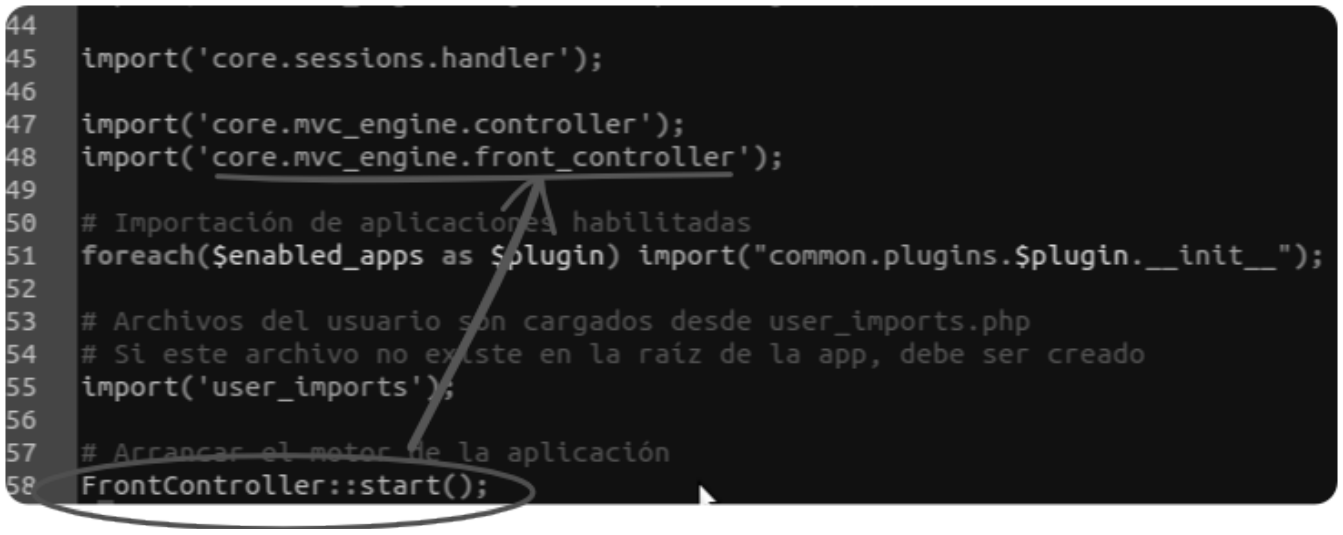
El código está diciendo: “si el archivo config.php no está en la carpeta raíz de la aplicación, matar el script

redirigiendo al usuario al archivo `install.php`". Por deducción e incluso antes de leer el archivo `install.php`, deberíamos sospechar: muy probablemente el archivo `install.php` sea el encargado de crear el archivo `config.php`.

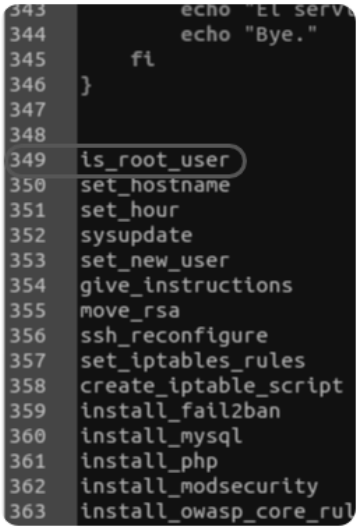
Si dicho archivo debe crearse en la carpeta raíz de la aplicación, el directorio raíz de la aplicación, entonces, deberá tener permisos de escritura más allá del propietario (de hecho, es un requerimiento de *Moodle* y efectivamente, es una vulnerabilidad).

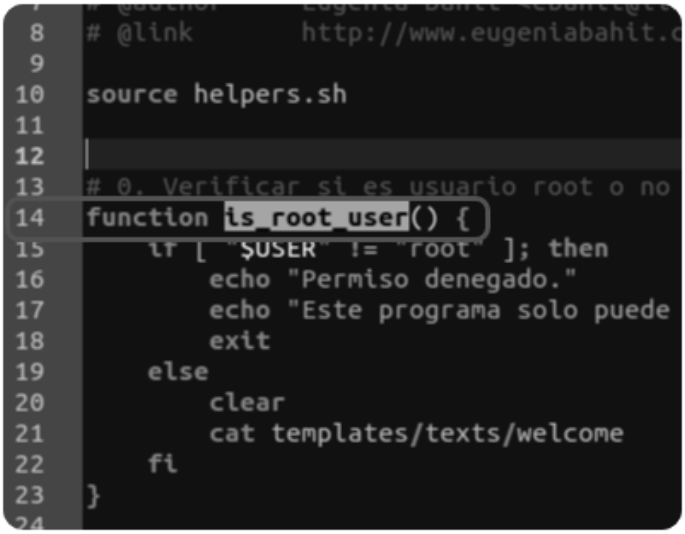
Como comenté anteriormente, **las instrucciones de arranque suelen estar al final de los archivos** y seguir el rastro, consistirá en localizar dicha instrucción y saltar hacia la definición de la misma.

Esto último, muchas veces puede requerir localizar previamente el archivo que la define, como se muestra a continuación:



Por ese motivo, el orden de lectura del código lo darán los rastros de las propias instrucciones y no, el orden en el cual éstas, hayan sido escritas:





Problemas conocidos

Cuando el problema ya se conoce, se debe intentar reproducir y en forma paralela deben examinarse los logs y depurar el código a fin de identificar a los actores (funciones, archivos, etc.), localizarlos y desde allí, ir siguiendo el rastro. Cuando alguna de estas acciones no sea posible, se deberán ejecutar los métodos habituales descritos en los pasos 1 y 2 anteriores.

Las diapositivas de la charla /en formato LibreOffice/ pueden descargarse desde www.originalhacker.org/IngenieriaIversaAyD2013.odp