the original Hacker

creado por EUGENIA BAHIT

Jugando con la inteligencia

número 6

Woman Eyes creado por Mourad Mokrane - Silueta de Mujer creado por Leonardo B. Cunha

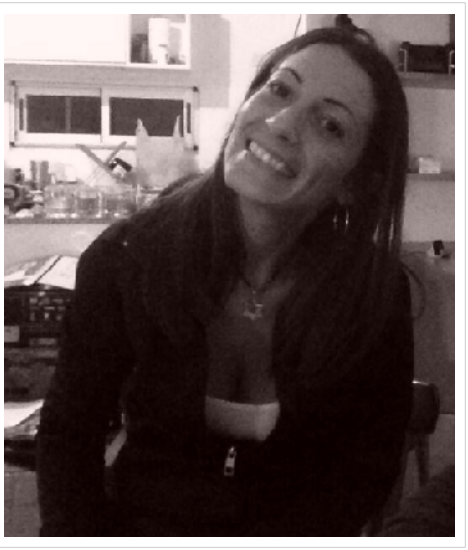
AUTHORS AGAINST DRM

SIN DRM

GNU run free run GNU

THE ORIGINAL HACKER
SOFTWARE LIBRE, HACKING y PROGRAMACIÓN, EN UN PROYECTO DE

EUGENIA BAHIT



@eugeniabahit

GLAMP HACKER Y PROGRAMADORA EXTREMA

Hacker especializada en Programación eXtrema e Ingeniería Inversa de código sobre GNU/Linux, Apache, MySQL, Python y PHP. eugeniabahit.com

Docente e instructora de tecnologías GLAMP cursos.eugeniabahit.com cursosdeprogramacionadistancia.com

Miembro de la Free Software Foundation fsf.org y The Linux Foundation linuxfoundation.org.

Creadora de python-printr, Europio Engine, JackTheStripper. Vim contributor. Fundadora de Hackers n' Developers Magazine y responsable editorial hasta octubre '13.

MATERIAL DE
LIBRE DISTRIBUCIÓN
HECHO EN LA REPÚBLICA ARGENTINA

Buenos Aires, 30 de Mayo de 2014

ÍNDICE DE LA EDICIÓN NRO6

Bloqueos mutuos y estados de carrera: Las vulnerabilidades más explotables generadas por hilos y Procesos en los sistemas informáticos

Días atrás, inicié un post sobre este tema en blog de DebianHackers y dado que hay mucho que investigar sobre la implementación de Hilos y Procesos en la ingeniería de sistemas y sus consecuentes riesgos de seguridad, decidí agregarlo en esta edición, para así poder dar inicio a una exhaustiva investigación al respecto.

No conozco programador **Python** que no alucine trabajando con **«threads»** y **«process»** y a pesar de ello, muy pocos son los que realmente saben de qué se trata. Pero esto no es lo peor. Lo peor, es que desconocen los **riesgos de seguridad que una mala implementación de hilos y procesos puede generar en todo un sistema informático**.

Si ya trabajas con **«threads»** y **«process»** este artículo te aclarará algunas dudas con respecto a la seguridad del sistema. Pero si aún no haz llegado hasta allí, **es una buena forma de comenzar, puesto que te lo explicaré de forma que lo entiendas, sin rollos ni tecnicismos inútiles**.

El principio ¿Qué es un proceso?

Primero que nada, vamos a tratar de entender qué es un proceso. Un proceso, no es más que una parte de un programa que se está ejecutando. Técnicamente es un **conjunto de instrucciones haciendo uso de un conjunto de recursos del sistema**.

> *Cuando hablamos de **recursos del sistema** nos referimos a uso de memoria RAM, Swap, CPU, tiempo de ejecución, entre otros recursos menos relevantes*

Por este motivo, cuando se dice que un programa hace un mal uso de los recursos del sistema, por lo general es debido a la ejecución desorganizada que hace de sus propios procesos.

¿Qué es un hilo?

Un hilo (*thread* en inglés) es similar a un proceso en cuanto a que también representa un conjunto de instrucciones. Sin embargo, los hilos se diferencian enormemente de los procesos, paradójicamente, por la administración que hacen de los recursos del sistema. Y esta última, es la característica que diferencia a los hilos de los procesos, tal como te lo explico en el siguiente párrafo.

Threads vs. Procesos: ¿cuál es la diferencia entre hilos y procesos?

La diferencia fundamental, es que ambos administran los recursos de formas diferentes. **Mientras que los diferentes procesos de un mismo programa ocupan diferentes espacios en la memoria, diferentes hilos comparten un mismo espacio**.

A primera vista, esto nos puede hacer suponer que los threads son una mejor alternativa que los procesos, puesto que utilizarán menor espacio de memoria. Esto es muy cierto, pero no son la panacea. Pueden acarrear múltiples problemas de seguridad tal como explicaré más adelante. No obstante, el uso de procesos en vez de threads, también solucionará problemas -a nivel seguridad de la aplicación- que estos últimos no tienen capacidad de resolver. Pero vayamos de a poco.

Problemas de seguridad generados por los threads

Todo conjunto de hilos o de procesos, probablemente necesite compartir información entre ellos. Sin lugar a dudas, será mucho más fácil compartir información si se comparte un mismo espacio de memoria, que hacerlo si hay que estar saltando de un puntero hacia otro. Por consiguiente, **compartir información será mucho más fácil para los threads que para los procesos**.

Sin embargo, compartiendo grandes cantidades de información y haciendo un uso significativo de tareas concurrentes a través de threads, **se pueden producir dos vulnerabilidades trágicas**: Una de ellas, la llamada **«bloqueo mutuo»** (*deadblock* en inglés); la otra y a la vez más peligrosa, la denominada **«condición de carrera»** (*race condition* en inglés).

> ***Bloqueo mutuo (deadlock):** Es el **bloqueo irreversible de un conjunto de hilos o procesos**.*

Un bloqueo mutuo es lo que sucede cuando un programa se te queda “colgado” y te ves en la obligación de “matar un proceso” ya que el conjunto de hilos o procesos bloqueados, no tiene solución.

> ***Condición (o estado) de carrera:*** *Es aquel que se produce cuando* ***varios hilos o procesos intentan modificar de forma simultánea a un mismo recurso.***

Si más de un hilo o recurso intenta modificar el estado o valor de un mismo recurso al mismo tiempo, los datos dejan de ser confiables y por consiguiente, es correcto afirmar que **los datos quedan corruptos**.

Por ejemplo, si Juan y Ana intentan modificar la contraseña del usuario admin al mismo tiempo. Juan decide colocar 123456 y Ana, 223344 ¿cuál de las dos claves será finalmente, la clave del usuario admin? La respuesta es que si el programador no contempló está posibilidad, será imprevisible conocer la clave. Incluso, podría quedar corrupta a tal punto de quedar sin clave. Esta concurrencia mal implementada es la que se denomina condición de carrera, donde los actores "compiten" por ver quien "gana" la modificación del dato.

Los estados de carrera en un sistema informático **son fácilmente explotables mediante *«exploits»* locales**. Por ese motivo, representan uno de los riesgos de seguridad más graves en un sistema.

Linux y GNU vs Sistemas Operativos privativos

En más de una oportunidad habrás escuchado decir que Linux (el kernel) y los sistemas operativos GNU que lo implementan, hacen una mejor administración de recursos que los sistemas operativos privativos comúnmente usados por el usuario promedio y que no se encuentran basados en Unix. Esto simplemente hace referencia al costo que la creación y destrucción de procesos, implica en los distintos sistemas.

Tanto en Linux y Unix a nivel kernel como en GNU a nivel sistema operativo, el costo de creación y destrucción tanto de hilos como de procesos puede ser casi imperceptible, mientras que en sistemas operativos privativos con núcleos diferentes, **el costo es muy elevado** y por ese motivo, en estos sistemas, se utilizan más los threads que los procesos. De allí que comúnmente se escuche decir que estos sistemas son más vulnerables que los que implementan Linux o Unix.

Uso de procesos que evitan errores de diseño explotables

Un buen uso de procesos y de preferencia de estos frente a los hilos, se da cuando una aplicación requiere efectuar tareas simultáneas con usuarios diferentes. Puede que no te hayas planteado emplear diferentes usuarios para diferentes tareas de tu aplicación y esta es una excelente oportunidad. **Apache** nos da un ejemplo concreto de ello.

Apache utiliza múltiples procesos con diferentes usuarios para la ejecución de sus tareas. El enlace a los puerto de bajo número -como el puerto 80, por ejemplo- lo realiza a través el usuario root mientras que el procesamiento de las solicitudes Web, lo efectúa con el usuario www-data quien tiene permisos muy inferiores a los de root.

Casos como el de Apache, son los que evitan que frente a un error producido en el código de las tareas ejecutadas por usuarios con permisos escuetos, ejecuten acciones solo permitidas al usuario root. La

única forma de que esto suceda, sería escalando privilegios para lo cual, otro error sería necesario de forma simultánea pero en el código de las tareas asignadas al usuario root.

Trabajo distribuido y portabilidad

Otra de las grandes ventajas que los procesos facilitan frente a los hilos, es la de permitir una mayor portabilidad en las aplicaciones y posibilidad de compartir el trabajo de forma simple (característica por excelencia en las arquitecturas cliente-servidor).

Esto significa que si lo que se necesita es una aplicación que permita distribuir/compartir el trabajo a través de una red local o externa, la ejecución de tareas mediante diferentes procesos, será la única alternativa. Caso contrario, la aplicación podría responder de forma impredecible facilitando así la corrupción de los datos y su consecuente explotación.

Nota final sobre threads: hilos del kernel vs hilos de usuario

Es muy importante que como programador o programadora, entiendas que **diferentes hilos comparten un mismo PID** (*process identifier* o identificación de proceso) mientras que diferentes procesos, poseen sus propios PID. De allí que frecuentemente puedas escuchar frases como *"los diferentes hilos de un proceso"*.
Sin embargo, **esto no es así a nivel del kernel**. Los hilos del kernel tienen sus propios PID. Esto es debido a la forma en la que el Kernel es ejecutado.

El kernel en sí mismo no se ejecuta como un proceso sino que sus tareas corren como parte de otros procesos. Al ser una cantidad de tareas significativas, **el kernel se ve obligado a implementar acciones alternativas que operen de forma similar a los procesos**, a fin de abaratar costos. Estas acciones alternativas **son los famosos demonios** (*daemon* en inglés) los cuales se ejecutan constantemente en segundo plano, asegurando de esta forma, un menor uso de los recursos ya que en caso contrario, de no estar disponibles de forma constante al usuario, éste debería invocar a dichos servicios cada vez que necesitara hacer uso de ellos. Esto implicaría nuevos procesos creándose y destruyéndose de forma permanente.
Cuando hablamos de caro o barato refiriéndonos al uso de recursos del sistema, estamos diciendo que **"a mayor uso de RAM y mayor tiempo de ejecución, más cara es la implementación"**.

En entregas futuras, procuraré avanzar en cuestiones más específicas sobre la implementación de hilos y procesos en la Ingeniería de aplicaciones. Pero mientras tanto, **si programas en Python, desde ahora en más, antes de decidir utilizar *threads*, piensa en lo que acabas de leer.**

Ingeniería de Software: Propiedades al vuelo, una forma de optimizar colecciones

Los objetos tienen la facilidad de poder crear propiedades una vez instanciados. Por lo general, esta característica se utiliza como una mala práctica. Sin embargo, puede emplearse como técnica de optimización de forma tal que permita una independencia modular absoluta.

Cuando se trabaja con objetos en estado puro, una colección puede terminar consumiendo un sinfín de recursos que para lo que se necesita controlar, suelen ser innecesarios.

Lamentablemente, en la mayoría de los casos los programadores terminamos optando por crear métodos que terminan manipulando datos dejando a un lado todos los principios de la programación orientada a objetos y junto a ellos, la gran cantidad de beneficios que dichos principios nos facilitan.

Estos casos que como bien dije antes nos llevan a lamentar la decisión, necesitan -sin lugar a dudas- de una solución no solo viable y que resuelva realmente el problema, sino que además, no se aparte de los principios de la programación orientada a objetos más pura.

El Problema: propiedades colectoras

Por lo general, el mayor problema con las **colecciones** de objetos, se da en aquellas cuyos compositores se encuentra a la vez, compuestos de colecciones. Es decir, en aquellos casos donde **los compositores poseen propiedades colectoras**. Sin lugar a dudas es lo peor que nos puede pasar al necesitar manipular colecciones de objetos en estado puro, sobre todo si lo que se pretende es efectuar un manejo básico.

En al menos el 98% de las aplicaciones orientadas a objetos, todo, absolutamente todo el diseño de objetos genera en este sentido, un cuello de botella a partir del objeto Usuario, del cual, más temprano que tarde, todos los objetos terminarán dependiendo, ya sea por cuestiones de herencia o por composición directa.

Y esto comienza a ser un problema visible, a medida que vamos agregando nuevos módulos al sistema. Es simple ver como cada uno de los módulos genera su propia figura de usuario pero sin embargo, en muchas ocasiones (en la mayoría me arriesgo a decir), no existe una verdadera diferencia entre los usuarios de un

módulo y los de otros, excepto por el tipo de objetos que lo componen. Por ejemplo, si en el mismo sistema conviviesen un módulo de *telemedicina* y otro para compartir código fuente (con lo son las aplicaciones *pastebin*, por ejemplo), en el primer caso los usuarios serían médicos mientras que en el segundo, programadores. Ambos tendrían un nombre de usuario y un nivel de acceso. Pero mientras que un médico podrá tener una colección de casos (o de registros médicos electrónicos), el programador, tendrá una colección de códigos fuente. ¿Para qué crear entonces dos tipos de usuarios diferentes? ¿Por qué mejor no reutilizar el mismo usuario y diferenciarlos solo por su rol dentro del sistema?

La solución

Después de mucho investigar, probar, hacer y rehacer, la idea de reutilizar al mismo objeto usuario en cada módulo, sin heredar de él y de hecho, sin siquiera crear nuevas figuras, ha demostrado solucionar no solo el problema aquí planteado, sino además, arrojó grandes beneficios, entre ellos, los principales son:

- Unifica conceptos y sentando un estándar de forma indirecta: todos los módulos utilizarán al mismo tipo de objeto como único tipo de usuario;
- Facilita la independencia y portabilidad de los módulos: al no ser necesaria la modificación del objeto usuario en el mismo objeto usuario, un mismo módulo podrá ser portado a cualquier aplicación que cuente con un objeto de tipo usuario que al menos posea una propiedad ID;
- Hace más rápido el entendimiento del código ya que el programador solo se centrará en los objetos troncales.

Las propiedades colectoras se crearían al vuelo y en tiempo de ejecución, en cada uno de los módulos.

De esta forma, el objeto usuario permanecería intacto y con un conjunto de propiedades estándar:

```
class Usuario {

    public function __construct() {
           $this->usuario_id = 0;
           $this->denominacion = '';
           $this->nivel = 0;
    }

}
```

Y los módulos de *telemedicina* y *pastebin* se encargarían de agregar sus colecciones respectivas:

```
# Recurso del controlador del modelo CasoClinico del módulo de telemedicina
# Lista los casos clínicos del usuario que solicita el recurso
# Antes de invocar al método get() del usuario, crea la propiedad colectora que almacenará
# dichos casos clínicos
public function listar() {
    $usuario = new Usuario();
    $usuario->usuario_id = isset($_SESSION['user_id']) ? $_SESSION['user_id'] : 0;
    $usuario->casoclinico_collection = array();
    $usuario->get();
}
```

```
# Lo mismo hará el recurso del controlador del modelo CodigoFuente del módulo pastebin
public function listar() {
    $usuario = new Usuario();
    $usuario->usuario_id = isset($_SESSION['user_id']) ? $_SESSION['user_id'] : 0;
    $usuario->codigofuente_collection = array();
    $usuario->get();
}
```

Como puede verse, es un simple artilugio que respetando los principios de la POO es capaz de optimizar el rendimiento de la aplicación de forma impensada.

Utilizando __call para crear pseudo métodos de agregación al vuelo

Finalmente, habrá que tener en cuenta que por cada propiedad colectora, un método de agregación será necesario e invocado por `get()` cada vez que éste sea llamado. Si en el modelo de usuario tuviésemos que crear tantos métodos de agregación como propiedades colectoras fuesen a crearse al vuelo, se perdería toda portabilidad y no solo volveríamos al principio, estaríamos haciendo un modelo inviable.

El método mágico `__call()` de PHP[1] ha sido diseñado para actuar cuando el método inaccesible de un objeto es invocado (notar que no sirve para cuando el método es invocado como función de clase. En ese caso, debe emplearse `__callStatic()`[2]). Definiendo este método en la clase usuario, su clase madre o en la que sea necesario crear propiedades colectoras al vuelo, estaríamos dando solución al problema.

Una implementación básica de este método, podría verse como la siguiente:

```
    function __call($call, $arg) {
        if(strpos($call, 'add_') === 0 && $arg) {
            $cls = str_replace('add_', '', $call);
            $property = "{$cls}_collection";
            if(!property_exists($this, $property)) $this->$property = array();
            $this->$property = array_merge($this->$property, $arg);
        }
    }
```

Ejemplo copiado literalmente del método `__call()` del objeto `StandardObject`[3] de Europio Engine

El método `__call()` recibe dos parámetros desde el intérprete: el primero, es el nombre del método que se intenta invocar. El segundo, un *array* con cada uno de los argumentos pasados al método inaccesible durante su llamada.

1 http://www.php.net/manual/es/language.oop5.overloading.php#object.call

2 http://www.php.net/manual/es/language.oop5.overloading.php#object.callstatic

3 https://bazaar.launchpad.net/~eugeniabahit/europioexperimental/trunk/view/head:/core/orm_engine/objects/standardobject.php#L60

El condicional del método se encarga de verificar que el nombre del método inaccesible comience por la cadena `add_` (esto le dará la pauta de que se trata de un método de agregación) y que el *array* `$arg` no esté vacío.

```
if(strpos($call, 'add_') === 0 && $arg)
```

Para definir el nombre de la propiedad colectora, simplemente eliminará `add_` del nombre del método y concatenará el resultado a `_collection`:

```
$cls = str_replace('add_', '', $call);
$property = "{$cls}_collection";
```

Finalmente, si la propiedad no existe (por si se olvidó crearla), la creará para luego, combinarla con la actual:

```
if(!property_exists($this, $property)) $this->$property = array();
$this->$property = array_merge($this->$property, $arg);
```

Europio Engine Lab: manipulación de datos con DataHandler. Un mundo más allá de los objetos...

Este artículo no solo pretende documentar al objeto DataHandler de Europio Engine, sino que además, servirá como fuente de referencia para el pseudo mapeo de tablas mediante information_schema.

Desde hace varios meses atrás, el tema más redundante en los cursos de capacitación que imparto sobre Europio Engine, era la manipulación de datos en el ámbito de los objetos. Pero no se trataba de algo caprichoso, sino de una necesidad real: ¿qué sucede cuando necesitamos efectuar estadísticas sobre datos, evitando adrede el uso de objetos?

Antes de preocuparnos por el «cómo» siempre debemos plantearnos «qué necesitamos» y a partir de allí, decidir qué es lo que tenemos que hacer, pero no antes. Y de esos «qué» fue que terminó surgiendo la necesidad de crear un objeto que permitiese trabajar con datos sin tocar la estructura de las tablas empleadas por los objetos.

Por este motivo, **aunque aún no hayas utilizado Europio Engine** como *core* de tus aplicaciones, **te recomiendo leer este artículo** ya que te servirá para ver cómo resolver el tratamiento de datos en tus aplicaciones orientadas a objetos.

Manipulación de datos vs. trabajo con objetos

Para que exista una **necesidad real de manipulación de datos** (y no de objetos) deben cumplirse ciertas características:

1. **Debe estar directamente relacionada a la base de datos.** Por ejemplo, decir «necesito localizar los *objetos X* que pertenezcan al *usuario Y*» no es una necesidad real ya que estamos hablando de objetos y cometiendo el error de tratar de manipular objetos como si fuesen datos. Siguiendo el ejemplo, obtener la colección de objetos X del usuario Y debería ser lo que suceda por defecto. Es

decir, se soluciona con un buen diseño de los objetos (el *usuario Y* tendrá una propiedad colectora *objeto_x* y recuperando al usuario Y se recuperará la colección de los *objetos X* que lo componen. Para saber cómo lograr un diseño de objetos de este estilo, referirse al artículo anterior sobre *«propiedades al vuelo»*);

2. **Debe tener una base sólida sobre la cual justificar la necesidad de trabajar con los datos.** Por ejemplo, la necesidad de crear una herramienta de análisis de rendimiento de la base de datos sería una justificación aceptable. Mientras que cualquier requerimiento visual sobre los objetos, no sería justificación suficiente para manipular datos directamente desde la propia base de datos;

3. **No debe existir una forma óptima y viable de resolver el requerimiento manipulando objetos.** Este ítem debe ir de la mano con el anterior. Por ejemplo, si lo que se desea es obtener estadísticas sobre uso del sistema, un requerimiento podría resolverse de forma viable «obteniendo N registros de una sola tabla» que obteniendo una colección de objetos y haciendo el análisis estadístico sobre toda la colección, ya que obtener una colección de 1000 objetos podría implicar efectuar consultas a decenas de tablas, algo que no justificaría para conocer cuántos datos se cargaron en la DB en los últimos 5', por ejemplo.

4. **Debe tratarse de necesidades de solo lectura.** Jamás debe escribirse sobre la base de datos destinada al almacenamiento de información sobre los objetos. Solo los modelos pueden escribir en su propia tabla y esto debe respetarse de forma estricta. De lo contrario, se pierde toda garantía sobre rendimiento y funcionamiento de la aplicación.

Habiendo aclarado estos puntos fundamentales, paso directo a hablar sobre DataHandler.

Data: un nuevo módulo incorporado al core de Europio Engine

Primero me enfocaré en cómo se utiliza este módulo (es verdaderamente muy simple) para luego explicar cómo trabaja de forma interna.

El objeto DataHandler

Como su nombre lo indica, se trata de un manejador de datos y es por el momento, el único objeto disponible del módulo.

Al ser instanciado, este objeto necesitará dos parámetros: el nombre de la tabla sobre la que se trabajará (obligatorio) y el formato en el que se desea obtener los resultados (opcional, por defecto retorna un array de datos).

```
DataHandler(string $tabla[, const $formato=DH_FORMAT_DATA])
```

El formato (opcional) debe ser una de las constantes predefinidas por Europio Engine. Estas pueden ser:

```
DH_FORMAT_DATA          Formato al que se refiere: array de datos.
                        Un array cuyos elementos serán arrays asociativos (nombre de clave
                        será nombre de campo). Utilizado como valor por defecto.

DH_FORMAT_OBJECT        Formato al que se refiere: array de objetos.
                        Un array cuyos elementos serán objetos.
                        Notar que DH_FORMAT_OBJECT no retorna los datos de una tabla
                        convertidos a objetos sino que en base a los datos obtenidos se
                        encargará de crear los objetos correspondientes.
```

Métodos

Por el momento, solo dispone de 2 métodos públicos.

```
get_latest(int $n)
Retorna los últimos N registros.


# EJEMPLO DE USO
$ultimos = DataHandler('comentarios')->get_latest(10);
# devolverá los últimos 10 registros de la tabla llamada comentarios
```

```
filter(string $condition[, const $filter=DH_FILTER_EQ])
Retorna los registros coincidentes con la condición pasada como primer parámetro.
```

El **tipo de filtro** `$filter` podrá ser cualquiera de las constantes predefinidas por Europio Engine:

```
-------------------+-------------------+------------
CONSTANTE          SIGNIFICADO         SIGNO
-------------------+-------------------+------------
DH_FILTER_EQ*      igual que           =
DH_FILTER_NOTEQ    distinto que        <>
DH_FILTER_LT       menor que           <
DH_FILTER_GT       mayor que           >

* valor por defecto
```

Por el momento, `filter()` admite una única condición cuyo operador sea cualquiera de los anterios.

```
$productos = DataHandler('productos')->filter('producto_id < 5', DH_FILTER_LT);
Retornará los primeros registros donde la ID sea menor que 5
```

DataHelper, el «backstage» del objeto DataHandler

DataHelper se encarga de «meter mano» en la base de datos. Su objetivo es el de sustituir la función de un ORM ya que si trabajamos realmente con datos no sería correcto mapear objetos. Y para lograrlo, debe evitar meterse en la tabla de los objetos de forma directa.

Entonces ¿cómo crear *queries* dinámicos sin mapear los objetos? Es muy simple: se pseudo-mapea la tabla ¿cómo? Consultando a la tabla **COLUMNS** de **INFORMATION_SCHEMA** de **MySQL**:

```
SELECT  COLUMN_NAME,
        IF(COLUMN_TYPE LIKE '%int%', 'i',
            IF(COLUMN_TYPE LIKE '%decimal%', 'd', 's')
        )
FROM    INFORMATION_SCHEMA.COLUMNS
WHERE   TABLE_SCHEMA = ?
AND     TABLE_NAME = ?
```

TABLE_SCHEMA será el nombre de la base de datos de la aplicación, mientras que TABLE_NAME el que se haya pasado como parámetro al crear el `DataHandler`. Para ver un ejemplo completo, recomiendo estudiar el código fuente de DataHelper[4].

4 https://bazaar.launchpad.net/~eugeniabahit/europioexperimental/trunk/view/head:/core/data/datahelper.php

Desmitificando el concepto de Sesiones. Todo es más simple de lo que parece...

Una de las consultas que mis alumnos me hacen en los cursos de MVC es qué hacer "con los login" en una aplicación. Realmente es algo muy sencillo, si se sabe con exactitud de qué se trata esto del «manejo de sesiones»

El concepto de «sesiones», básicamente es una entelequia. No existe un concepto de «sesión» como tal y que sea independiente al concepto de *cookies*.

Las sesiones, **son en realidad un modo de persistencia de datos** mediante la **interacción de archivos en el propio servidor** y en el ordenador del usuario (o sea, *cookies*).

El concepto de sesión, **se refiere a la mecánica utilizada para generar la persistencia temporal de ciertos datos del usuario**.

Dicha mecánica, consiste básicamente en:

Identificación del usuario: se identifica a cualquier usuario que ingresa a un sitio Web, mediante un identificador único (ID, que por cierto, nada tiene que ver con las bases de datos). Por ejemplo: imaginemos que estamos desarrollando un nuevo lenguaje de programación. Entonces, cada vez que una persona ingresa en el sitio Web, de forma automática, nuestro *script* le «inventa» un «código o cadena» de identificación, único para ese usuario.

Persistencia del identificador: el proceso de persistencia del identificador, es un proceso interno que debe realizarse en el servidor. Consiste en implementar una mecánica que permita almacenar de forma temporal, el identificador asignado al usuario, hasta que éste abandone el sitio. La mayoría de los manejadores de sesiones o de los lenguajes como PHP que implementan dicho concepto de forma nativa en el propio lenguaje, dividen esta mecánica en dos acciones:

1) Almacenamiento del identificador en el servidor, el cual puede imaginarse como un archivo temporal;

2) Almacenamiento del identificador en el ordenador del usuario, el cual no es más que una *cookie*.

El uso de *cookies* en el ordenador del usuario, podría evitarse si por ejemplo, en el archivo del usuario en el servidor, se guardara información diversa sobre el

usuario: desde su IP hasta la versión del sistema operativo. Sin embargo, esto podría traer aparejadas consecuencias indeseables: desde una sobrecarga del servidor, hasta la colisión de usuarios que mediante un *proxy*, pudieran utilizar la misma IP. Es por este motivo, que la mejor solución encontrada hasta ahora, ha sido el uso múltiple de archivos: en el servidor y en el cliente, a modo de *Cookies*.

Esta, es la base del manejo de sesiones y nada tiene que ver con un concepto tangible ni mucho menos, con los procesos de registro de usuarios e identificación mediante el ingreso de claves privadas.

Por este motivo, hay que aprender a ***diferenciar entre el manejo de sesiones y la gestión de usuarios***.

Gestión de Usuarios vs. Manejo de Sesiones

La gestión de usuarios, debe verse como la gestión de cualquier otro objeto del sistema. Es así que si se requiere gestionar usuarios se deberá contar con un módulo de gestión de usuarios.

Cuando trabajamos con objetos, el usuario en sí mismo, no es más que un objeto del sistema. Es otro componente como tantos. Y como tal, tendrá su clase con métodos y propiedades como cualquier otra y en una arquitectura MVC, también contará con su vista y controlador correspondiente.

Cuando se analiza el total de la aplicación, se puede observar que un modelo de usuarios, no puede relacionarse de forma directa, con ningún otro módulo del sistema. De allí, la necesidad de diseñar un módulo independiente para los usuarios. Incluso, aunque éste, tenga solo un modelo.

Si se describen las cualidades del objeto usuario, respetando de forma estricta la organización orientada a objetos de una arquitectura e incluso, su paradigma homónimo, el usuario no tendrá más que dos propiedades: la primera e inevitable, su ID y la segunda, su nombre (el usuario es de nombre Juan). A lo sumo, cuando se trabaja con diferentes niveles de acceso, tendrá un nivel (el usuario es de nivel 5).

El usuario como objeto, NO «es de» contraseña ni «tiene una» contraseña. El usuario, solo debe ingresar un dato privado para acceder al sistema. Dicho dato, es conocido como «contraseña» y no es más que un requerimiento gráfico a los ojos de una orientación a objetos estricta.

Por este motivo, la contraseña (la cual es un dato privado que solo el propietario debe conocer como si se tratase del PIN de su tarjeta de crédito), es el único requerimiento gráfico que requerirá un tratamiento especial en el modelo. Y éste, NO consistirá en agregar una nueva propiedad de clase, sino tan solo, en preparar al método `save()` para recibir como parámetro, dicho dato. Pues la forma más viable de hacer persistir dicho dato, es manipulando datos más allá del objeto. Y a fin de reutilizar recursos, no debe crearse una nueva tabla en la base de datos, sino, aprovechar la destinada a hacer persistir los objetos Usuario.

Por otra parte, debe hacerse notar que el objeto Usuario como tal, tendrá los mismos métodos que cualquier otro: `save()`, `get()` y `destroy()`.

Cuando un acceso restringido sea requerido, será el manejador de sesiones quien entre en juego, pero no ya manipulando objetos sino datos. Pues la restricción de acceso y el ingreso de contraseñas, no son más que requerimientos gráficos que recurren a «datos puros» para su tratamiento.

> ***Un módulo de usuarios es totalmente independiente de un manejador de sesiones.*** *Solo se aprovecha la base de datos utilizada para hacer persistir los objetos, a fin de evitar la sobre-saturación de recursos.*

Incluso, un módulo de usuarios podría no ser necesario, si el sistema no requiere su gestión y por el contrario, solo cuenta con el requerimiento gráfico de ingresar datos determinados que coincidan con los esperados. Es el caso de restricción de acceso por un único nombre de usuario y contraseña. Para el sistema y a nivel arquitectónico, éstos, son solo datos que el «usuario» (no el programador) conoce (o ubica) como nombre de usuario y contraseña. A nivel algorítmico, son tan solo, dos cadenas de texto que deben coincidir con las esperadas.

¿Cómo sería entonces una clase usuario si la clave no es una propiedad?

Si se desea entonces, contar con un módulo de gestión de usuarios, no habrá más que crear dicho módulo. Las sesiones, son un caso aparte e independiente, ya que ni siquiera interactúan con el o los objetos de este módulo. Repito: simplemente reutilizan la misma base de datos.

A tal fin, colocaré aquí un típico modelo Usuario, solo a modo de ejemplo por si la gestión de los mismos es requerida. Si se trabaja con MVC, las vistas y controladores, serán como las de cualquier otro modelo.

En Python:

```
from core.dblayer import run_query


class Usuario(object):

    def __init__(self):
        self.usuario_id = 0
        self.nombre = ''

    def save(self, pwd):
        if self.usuario_id == 0:
            sql = """INSERT INTO usuario (nombre, pwd)
                     VALUES ('%(u)s', '%(p)s')""" % dict( u=self.nombre, p=pwd)
            self.usuario_id = run_query(sql)
        else:
            sql = """UPDATE usuario
                     SET    nombre = '%(u)s', pwd = '%(p)s'
                     WHERE  usuario_id = %(uid)i""" % dict( u=self.nombre, p=pwd,
                        uid=self.usuario_id)
            run_query(sql)

    def get(self):
        sql = """SELECT usuario_id, nombre FROM usuario
                 WHERE usuario_id = %i""" % self.usuario_id
        fields = run_query(sql)[0]
        self.nombre = fields[1]

    def destroy(self):
        sql = """DELETE FROM usuario
                 WHERE  usuario_id = %i""" % self.usuario_id
```

```
        run_query(sql)


En PHP:
require_once 'core/dblayer.php';


class Usuario {

    function __construct() {
        $this->usuario_id = 0;
        $this->nombre = '';
    }

    function save($pwd) {
        if($this->usuario_id == 0) {
            $sql = "INSERT INTO usuario (nombre, pwd)
                    VALUES (?, ?)";
            $data = array('ss', "{$this->nombre}", "$pwd");
            $this->usuario_id = DBObject::ejecutar( $sql, $data);
        } else {
            $sql = "UPDATE usuario
                    SET nombre = ?, pwd = ?
                    WHERE usuario_id = ?";
            $data = array("ssi", "{$this->nombre}", "$pwd", "{$this->usuario_id}");
            DBObject::ejecutar($sql, $data);
        }
    }

    function get() {
        $sql = "SELECT usuario_id, nombre
                FROM usuario
                WHERE usuario_id = ?";
        $data = array("i", "{$this->usuario_id}");
        $fields = array( 'usuario_id'=>'', 'nombre'=>'');
        DBObject::ejecutar($sql, $data, $fields);
        $this->nombre = $fields['nombre'];
    }

    function destroy() {
        $sql = "DELETE FROM usuario
                WHERE  usuario_id = ?";
        $data = array("i", "{$this->usuario_id}");
        DBObject::ejecutar($sql, $data);
    }

}
```

Es necesario aclarar que en MVC, cuando el controlador llame al método `save()` deberá hacerlo pasándole como parámetro, el dato correspondiente a la contraseña ya cifrado.

```
En Python:
from hashlib import md5
self.model.save(md5(password).hexdigest())

En PHP:
$this->model->save(md5($password));
```

ACERTIJO

Por 1, Juan debía pagar $5 y lo mismo que paga por 1 lo paga por 5. Por 15, el precio era de $10 pero por 100, de $15. ¿Qué estaba comprando Juan?

Ayuda: que los números no te engañan. No hay nada que la aritmética básica no pueda resolver.

Responde **ANTES** del **20/06/2014** a través de **Twitter** utilizando el *hashtag* **#AcertijoTOH6**

El nombre de los ganadores será publicado en la siguiente edición

La respuesta correcta será publicada en The Original Hacker Nº7 junto con los ganadores

SOLUCIÓN AL ACERTIJO DE THE ORIGINAL HACKER Nº5

ESTADÍSTICAS. TOTAL DE RESPUESTAS: **22** / CORRECTAS: **11** / ACIERTOS: **50%** / DESACIERTOS: **50%**

Para morir el día de su cumpleaños número 30, el hombre tendría que haber nacido un año antes o muerto un año después ya que al no haber existido un año cero, murió el día de su cumpleaños número 29.

> *Como programadores, el razonamiento lateral -entre otras cosas- nos ayuda a **aislar el verdadero problema** del contexto **descartando** así, aquella **información** que a pesar de aparentar ser de interés, resulta **irrelevante** para la solución del problema. El acertijo nos lo demuestra con los datos sobre la hora de nacimiento y fallecimiento. Puede parecer interesante pero si sabemos que no es matemático el cálculo ¿es entonces un dato relevante?*

*...porque **«no existió el año 0»**, éste, es el*

RANKING DE GANADORES DEL #AcertijoTOH5

PRIMERA RESPUESTA CORRECTA

Usuario que dio la 1ª respuesta correcta en su primer intento

@micodwarft
(felicítalo por su respuesta)

RESPUESTA MEJOR ARGUMENTADA

Respuesta que mejor ha explicado el acertijo

@diegodelavega
(felicítalo por su respuesta)

OTROS GANADORES:

@3rn3st0, **@chuso**, **@neomathews**, **@Micky_Carrasco**, **@imonclus**, **@rpla**, **@mtapiag1**, **@benjasVazquez** y **@santararo** también merecen ser felicitados por haber acertado en la respuesta :)

«Programar es saber aislar el problema descartando lo irrelevante»

FREE AS IN FREEDOM LA VERDADERA ÉTICA HACKER

Necesitamos vivir en un mundo donde el acceso al **conocimiento** sea **LIBRE**; donde la **solidaridad** con el vecino NO sea **ILÍCITA**. Debe dejar de condenarse el compartir conocimiento con nuestros seres queridos y allegados. **El cambio está en tus manos, hay mucho que puedes hacer.**

DILE A MOZILLA QUE NO INCORPORE EL DRM DE ADOBE A FIREFOX

Sólo una semana después del Día Internacional contra el DRM, **Mozilla anunció que apoyará la Gestión Digital de Restricciones en su navegador Firefox**, el cual tendrá una utilidad incorporada que obtendrá e instalará automáticamente los DRM de Adobe.

ENVÍALE UN E-MAIL A ANDREAS GAL (CTO de Mozilla) haciéndole saber que te opones al DRM.

CANCELA TU CUENTA EN NETFLIX ¡YO LO HICE!

Hace un año atrás, enterada de que **Netflix solo permitiría la reproducción de vídeos en navegadores «*extensiones premium*» (así llamaron al DRM)** cancelé mi suscripción anual ¿por qué? Porque la libertad de toda una comunidad DEBE valer más que cualquier serie HD.

ACCEDE A TU CUENTA DE NETFLIX y **ENVÍALES UN MENSAJE CANCELANDO TU SUSCRIPCIÓN**. **Anuncia tu decisión en Twitter** y tus redes sociales.

NO USES DISPOSITIVOS QUE PROHIBEN COMPARTIR TUS PROPIOS ARCHIVOS CON LIBERTAD ¡YO NO LOS USO!

Amazon Kindle, iPad, iPod, iPhone y demás dispositivos de Apple incluyen una Gestión Digital de Restricciones que **daña la libertad de las personas**. Dispositivos con Android ¡también son peligrosos!

¡NO LOS USES!

UTILIZAR SOFTWARE PRIVATIVO O DISPOSITIVOS CON RESTRICCIONES

Y DECIR QUE APOYAS EL SOFTWARE LIBRE, ES CINISMO.

UN HACKER ÉTICO ES AQUEL QUE ES COHERENTE CON LO QUE DICE Y HACE.

DEFECTIVE BY DESIGN.org

HACKTIVISMO ÉTICO

LAS SIGUIENTES COMPAÑÍAS HACEN DAÑO A LA LIBERTAD DE LAS PERSONAS

amazon.com

Mediante sus plataformas Kindle y Prime. **Leer más »**

Apple utiliza el DRM para controlar iOS y a los usuarios de OS X. **Leer más »**

Microsoft

El DRM se incorpora en el corazón de Windows y muchos servicios como Silverlight, imponen el DRM en los usuarios. **Leer más »**

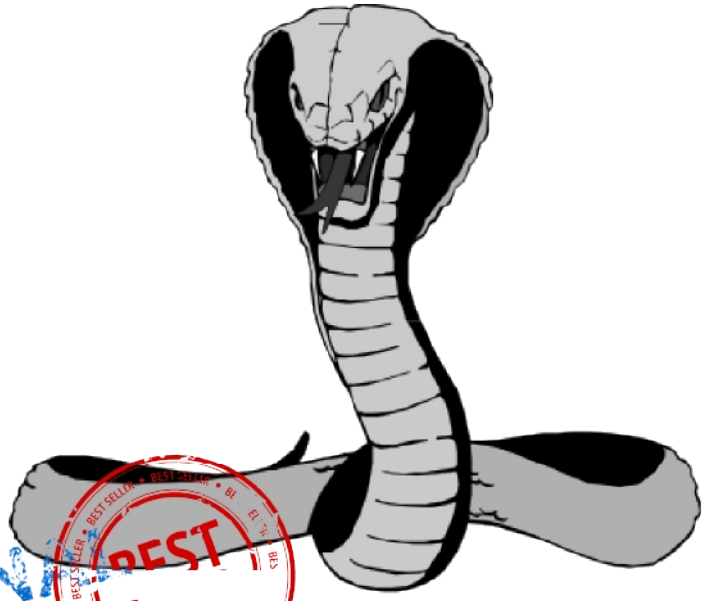
NETFLIX

No satisfecho con el uso de DRM para su servicio de *streaming*, ahora Netflix está tratando de imponer el DRM en la Web. **Leer más »**

SONY

Sony ha utilizado acciones legales para hostigar e intimidar a las personas que han modificado sus sistemas PS3. **Leer más »**

SI DE VERDAD APOYAS EL SOFTWARE LIBRE, ÚSALO, PROMUÉVELO Y ABANDONA LO PRIVATIVO SIN EXCUSAS. EL CAMBIO ESTÁ EN TUS MANOS.



The Original Hacker #_6_

SafeCreative Work:_1405301135816_

safecreative.org/work/_1405301135816_