the original Hacker

creado por EUGENIA BAHIT

jugando con la inteligencia

número 4

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

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, The Linux Foundation linuxfoundation.org e integrante del equipo de Debian Hackers debianhackers.net.

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, 25 de Febrero de 2014

ÍNDICE DE LA EDICIÓN NRO4

Bash Scripting Avanzado: sustitución de parámetros y manipulación de variables

De forma realmente simple y sencilla, es posible manipular variables en bash, para resolver necesidades tan frecuentes como el control de argumentos pasados al guión hasta acciones como renombrar archivos en masa.

En la mayoría de los casos, tan solo se requiere una línea de código y sin embargo, por lo general, nos olvidamos de utilizarla u optamos por complejos algoritmos.

Es notable también, la cantidad de veces que recurrimos a lenguajes interpretados más recientes que bash, creyendo que los mismos cubren ausencias que realmente no son tales. Recurrir a lenguajes como Perl o Python en la creación de guiones de *Shell* es tan frecuente que hasta llegamos a olvidarnos de la grandeza de bash.

También es muy frecuente recurrir al empleo de comandos para resolver necesidades que el lenguaje contempla en su propia sintaxis.

Manipulación de variables: Búsqueda y reemplazo

En bash, es posible efectuar búsquedas y sustituciones de manera rápida y flexible sin recurrir una lista interminable de comandos concatenados. En este aspecto, la necesidad más frecuente con la que nos encontramos, es **sustituir determinadas apariciones en una cadena de texto**. Esto es posible con una simple instrucción:

```
${variable//búsqueda/reemplazo}
```

Por ejemplo:

```
cadena='Hola mundo; Adiós mundo'
buscar='mundo'
reemplazar='¡mundo!'
echo ${cadena//$buscar/$reemplazar}
```

Arrojará el siguiente resultado:

```
Hola ¡mundo!; Adiós ¡mundo!
```

Vale aclarar que el código anterior es equivalen a:

```
cadena='Hola mundo; Adiós mundo'
echo ${cadena//'mundo'/'¡mundo!'}
```

Aunque en este caso en particular, en la búsqueda no serían necesarias las comillas.

Notar que la doble barra diagonal // es la que indica que la sustitución debe hacerse en todas las apariciones

Es posible además, **sustituir solo la primera aparición** de la búsqueda en la cadena, con tan solo omitir una de las barras diagonales del comienzo:

```
cadena='Hola mundo; Adiós mundo'
echo ${cadena/'mundo'/'¡mundo!'}
# Imprime: Hola ¡mundo!; Adiós mundo
```

Los signos #, ## y %, %% en las sustituciones

La almohadilla (#) se utiliza para identificar la búsqueda al comienzo de la cadena mientras que el porcentaje (%), para identificarla al final.

Cuando alguno de estos símbolos está presente, **se eliminará de la cadena** aquella fracción coincidente con **el patrón de búsqueda**:

```
${cadena#patron}
Retorna "cadena" SIN "patron"
```

Una sola almohadilla o porcentaje, encontrará en la cadena la parte más corta coincidente con el patrón de búsqueda mientras que **dos almohadillas o porcentajes, lo harán con la mayor fracción** coincidente. De esta forma, si el patrón a buscar fuese: *mun (el asterisco se leería como *"cualquier cosa"* siendo la expresión completa: *"cualquier cosa hasta la sílaba mun"*), una sola almohadilla encontraría coincidencia en *"Hola mun"*, mientras que dos extenderían la coincidencia hasta *"Hola mundo; Adiós mun"*:

```
cadena='Hola mundo; Adiós mundo'
echo ${cadena#*mun}
do; Adiós mundo

echo ${cadena##*mun}
do
```

Lo mismo sucede con el signo del porcentaje:

```
cadena='Hola mundo; Adiós mundo'
echo ${cadena%mun*}
Hola mundo; Adiós

echo ${cadena%%mun*}
Hola
```

Resumiendo, podríamos decir que:

- El porcentaje busca desde el final de la cadena
- La almohadilla busca desde el principio
- Ambos eliminan de la cadena, el patrón que coincida
- Un solo signo, busca la menor coincidencia mientras que dos, buscan la mayor coincidencia

La parte de la cadena que es suprimida, puede ser sustituida con una sintaxis sumamente simple:

```
sustitución${cadena#patron}
${cadena%patron}sustitución
```

Por ejemplo:

```
cadena='Hola mundo; Adiós mundo'
echo "*******"${cadena#*mun}
*******do; Adiós mundo

cadena='Hola mundo; Adiós mundo'
echo ${cadena%mun*}"*******"
Hola mundo; Adiós *******
```

Este último sistema, nos puede servir para, por ejemplo, **cambiar la extensión de archivos en masa**:

```
for archivo in $HOME/*.gif; do
    echo -n 'Renombrando '
    basename $archivo
    mv $archivo ${archivo%*gif}jpg
done
```

O por qué no, para **crear una copia de seguridad** de los mismos:

```
for archivo in $HOME/*.gif; do
    echo -n 'Creando copia de seguridad para '
    basename $archivo
    cp $archivo ${archivo%*gif}'gif.backup'
done
```

Subcadenas y sustitución

En bash, también es posible **extraer una porción de una cadena** utilizando la siguiente sintaxis:

```
${cadena:posicion}
${cadena:posicion:longitud}
```

Con la primera, obtenemos la porción desde `posicion` hasta el final mientras que con la segunda, la obtenemos desde `posicion` hasta alcanzar la longitud indicada:

```
cadena='Hola mundo; Adiós mundo'
echo ${cadena:5}
mundo; Adiós mundo

echo ${cadena:5:5}
mundo
```

Utilizando el mismo método anterior, también podríamos simular una sustitución:

```
cadena='Hola mundo; Adiós mundo'
echo '*****'${cadena:5}
*****mundo; Adiós mundo

echo '***'${cadena:5:5}'***'
***mundo***
```

Teniendo en cuenta que con **${#cadena}** se obtiene la **longitud total de una cadena**, podemos utilizar: **${#cadena}-cantidad** para obtener una porción calculando una determinada posición desde el final:

```
cadena='Hola mundo; Adiós mundo'
echo ${cadena:${#cadena}-11}
Adiós mundo
```

Control de argumentos

Cuando escribimos guiones cuyos argumentos pueden ser opcionales, a veces establecemos valores por defecto para los mismos. Seguramente, en más de una oportunidad te habrás encontrado haciendo lo siguiente:

```
if [ $1 ]; then
    file_path=$1
else
    file_path=$DEFAULT_PATH
fi
```

Sin embargo, bash dispone de una sintaxis mucho más simple para esto:

```
variable=${argumento:-VALOR_POR_DEFECTO}

Ejemplo:
file_path=${1:-$DEFAULT_PATH}
```

Pero lo anterior, es solo una de las tantas posibilidades que tenemos. A continuación, veremos todas ellas con mayor detalle.

Uso y Asignación de Parámetros por defecto

En bash, se pueden **utilizar valores por defecto** en diferentes casos:

```
${variable-valor a usar si la variable no fue definida}
${variable:-valor a usar si la variable no fue definida o su valor es nulo}

Algunos Ejemplos:

echo ${foo-25}              # foo no está definida. Imprime: 25
foo=                        # se define foo con valor nulo
echo ${foo-30}              # No imprime nada porque foo ya fue definida en la línea anterior
echo ${foo:-55}             # Imprime: 55 ya que foo fue definida pero su valor es nulo
```

Este método es ideal para utilizar en el control de argumentos, cuando éstos, no son pasados al guión:

```
declare -r DEFAULT_PATH=/etc/apache2/sites-available
apache_path=${1:-$DEFAULT_PATH}
```

En el ejemplo anterior, si no se pasa un argumento al guión, la variable `apache_path` toma como valor el de la constante `DEFAULT_PATH`.

Vale aclarar que en todos los casos anteriores, el valor por defecto NO ES ASIGNADO a la variable, solamente es UTILIZADO.

Eso significa que (retomando el ejemplo con la variable `foo`) si posteriormente se quisiera imprimir la variable $foo, arrojaría un valor nulo ya que nunca se le ha asignado uno:

```
echo ${foo-75}  # Imprime: 75
echo $foo       # no imprime nada
```

Pero de ser necesario, también es posible **asignar valores por defecto** utilizando el signo de igualdad en vez del guión medio:

```
${variable=valor de variable si no fue definida}
${variable:=valor de variable si no fue definida o su valor es nulo}


echo ${foo=25}   # foo no está definida. Imprime: 25 asigna a foo el valor 25
echo $foo        # Imprime: 25

foo=             # se define foo con valor nulo
echo ${foo=30}   # No imprime nada porque foo ya fue definida. El valor de foo no cambia
echo ${foo:=43}  # Imprime: 43 ya que foo fue definida pero su valor es nulo.
                 # El valor de foo se establece en 25
```

Resumen de búsquedas y sustituciones en Bash

```
================================================================================
BÚSQUEDA Y REEMPLAZO
================================================================================

${VAR/A/B}        Reemplaza la primera aparición de A en VAR, por B
${VAR//A/B}       Reemplaza todas las apariciones de A por B en VAR


================================================================================
CONTAR CARACTERES
================================================================================

${#VAR}           Retorna la longitud de VAR


================================================================================
EXTRACCIÓN
================================================================================

${VAR#PATRON}     Elimina la coincidencia más corta de PATRON del comienzo de VAR
${VAR##PATRON}    Elimina la coincidencia más larga de PATRON del comienzo de VAR
${VAR%PATRON}     Elimina la coincidencia más corta de PATRON del final de VAR
${VAR%%PATRON}    Elimina la coincidencia más larga de PATRON del final de VAR


================================================================================
SUBCADENAS
================================================================================

${VAR:POS}        Retorna la porción de VAR desde POS hasta el final
${VAR:POS:LEN}    Retorna la porción de VAR desde POS hasta alcanzar la longitud LEN


================================================================================
VALORES POR DEFECTO
================================================================================

${VAR-VALOR}      Usar VALOR si VAR no fue definido
${VAR:-VALOR}     Usar VALOR si VAR no fue definido o su valor es nulo
${VAR=VALOR}      Asignar VALOR a VAR si VAR no fue definido
${VAR:=VALOR}     Asignar VALOR a VAR si VAR no fue definido o su valor es nulo
```

Seguridad Informática: EuropioCode, un sistema de codificación de caracteres basado en el modelo SAC de 3 capas

¿Cuántas veces escuchamos y repetimos la frase "no existen los sistemas invulnerables"? Lo que nunca nos detenemos a pensar es que si no existen, es solo porque no nos detenemos a crearlos. Entonces ¿por qué no comenzar a desarrollar sistemas invulnerables? Aquí los primeros pasos para asegurar el 60% de un sistema informático.

Hace poco más de un mes, liberé la versión beta 1 de **EuropioCode**, un simple y sencillo **sistema de codificación de caracteres basado en secuencias alfanuméricas**, el cual presenté en el sitio Web de **Debian Hackers**[1].

Codificar caracteres con secuencias alfanuméricas, es la forma más confiable de permitir al usuario, el ingreso de caracteres no alfanuméricos, sin poner en riesgo la seguridad de la aplicación ni limitar las funcionalidades y prestaciones de la misma.

EuropioCode posee una doble capa de seguridad:

1) A nivel cliente (librería JavaScript);
2) A nivel aplicación (librería PHP);

Estas dos capas de seguridad, son el complemento ideal para **ModSecurity**, conformando así un **modelo de seguridad SAC** *(Server-Application-Client)*, **único capaz de alcanzar el mayor nivel de seguridad**, permitiendo sistemas informáticos invulnerables.

ModSecurity, el módulo de seguridad para Apache, debe estar instalado con las reglas de núcleo /core rules/ creadas por OWASP.

1 http://www.debianhackers.net/prevenir-ataques-xss-e-inyecciones-de-codigo-y-sql-con-europiocode

Modelo de Seguridad SAC

El modelo de seguridad SAC está basado en una triple capa de seguridad de 3 niveles: 1) **Servidor**; 2) **Aplicación**; y 3) **Cliente**.

Como modelo de seguridad, no solo es uno de los más antiguos sino que además es el único que tradicionalmente ha sido capaz de otorgar **garantías de invulnerabilidad sobre los sistemas informáticos** -siempre que el modelo sea implementado de la manera correcta-.

Un error frecuente en seguridad

A pesar de todo lo anterior, **el modelo de seguridad SAC**, generalmente solo es implementado por grandes compañías pero **subestimado por la mayoría de empresas y personas independientes**.

El **desarrollador Web** que da sus primeros pasos en la programación, suele hacerlo desde una óptica de usuario, **centrándose solo y únicamente en la seguridad del lado del cliente**, puesto que es la única que el usuario final (su cliente y en definitiva, él mismo) puede percibir sin necesidad de realizar los esfuerzos que el razonamiento deductivo (al cual no está acostumbrado) le demanden. Lamentablemente, este tipo de incipientes profesionales **desarrollan aproximadamente entre el 70% y el 80% de los sitios Web y *blogs* del mercado**.

> *Más del 70% de los sitios Web (excluyendo aplicaciones) solo implementan seguridad del lado del cliente.*

Luego nos enfrentamos al promedio de los **programadores de nivel intermedio y avanzado**, que confiados de «haberlo visto todo», **solo se concentran en la seguridad de la aplicación**, subestimando el control del lado del cliente (puesto que saben positivamente que el mismo es perfectamente vulnerable pero olvidan que no por ello es inútil) y desconociendo los mecanismos de abarcar la seguridad del lado del servidor, por lo que la misma, la delegarán en terceros que generalmente también la desconocen o no ponen en ella el esfuerzo mínimo necesario. Nuevamente debemos lamentarnos, ya que el promedio de los programadores **desarrolla aproximadamente el 80% de las aplicaciones del mercado**.

> *El 80% de las aplicaciones Web no cuenta con la seguridad necesaria del lado del cliente ni del servidor.*

Finalmente nos encontramos con los **administradores de sistemas**, quienes por lógica, **centrarán su atención en la seguridad a nivel servidor**, delegando la tarea de programación en aplicaciones prefabricadas que por lo general, han sido desarrolladas por programadores promedio de los que hablábamos en el párrafo anterior. Aquí nos encontramos con que prácticamente el 100% de los administradores de sistemas, recurre a aplicaciones prefabricadas (CMS) pero la buena noticia es que por lo general, no lo hacen como servicio a terceras personas o empresas y si lo hacen, **no superan el 5% de los blogs del mercado** (entiéndase que me refiero a los blogs que los administradores de sistemas montan para las empresas para las cuales trabajan y que no me estoy refiriendo a los blogs que montan para sí mismos).

Con suerte, podremos encontrar **un 20% de aplicaciones** que **implementan el modelo de seguridad SAC** y sin embargo, una tercera parte de estas, no lo implementa de la forma más óptima.

ModSecurity Core Rules y el fin de los falsos positivos

Hace poco tiempo, conversando con mi colega de *Debian Hackers*[2] **David Hernández**[3], surgía en medio de la charla el tema sobre *«los falsos positivos que el módulo* ***ModSecurity***[4] *de* ***Apache*** *suele arrojar»*. Hablando informalmente, comentábamos nuestro «enojo» con los mismos pero sin embargo, **la mayoría de las veces los falsos positivos de ModSecurity son producto de errores de diseño a nivel de la aplicación**. Es decir que **la mayoría de las veces, no se trata de falsos positivos**.

Los falsos positivos de ModSecurity por intentos de ataques del tipo SQL Injection o Cross Site Scripting se acaban evitando los errores de diseño en la aplicación y esto se logra, codificando las entradas del usuario mediante algún sistema alfanumérico.

A raíz de esto, que me propuse hacer algo al respecto y así nació **EuropioCode**.

EuropioCode en pocas palabras

EuropioCode no es más que un **algoritmo para codificar cadenas de caracteres en código alfanumérico** fácilmente decodificable y reversible. Esto, permite brindar al usuario la capacidad de utilizar campos de formulario con texto enriquecido -hasta incluso, escribir código fuente puro en cualquier lenguaje- y enviar el mismo codificado alfanuméricamente, evitando así los falsos positivos de ModSecuity por supuestos intentos de ataques de tipo ***SQL Injection*** o ***Cross Site Scripting (XSS)***.

De hecho, **esta técnica de codificación alfanumérica NO se utiliza en sí para evitar falsos positivos en ModSecurity** sino que por el contrario, **es la forma correcta de transferir datos de forma segura para prevenir ataques** por diversos tipo de inyecciones de código.

Si por ejemplo, se quiere permitir al usuario el ingreso de comillas dobles, incluso aunque se las reemplazase por su entidad HTML correspondiente (`&quot;`), sería un error de diseño permitir su envío en estado puro (ya sea en su carácter correspondiente como en su entidad HTML) puesto que **técnicamente se estaría permitiendo la inyección de código en la aplicación**, pues la forma correcta de permitir el ingreso de caracteres especiales, jamás puede contemplar la inyección de código.

Sin embargo, si dejásemos que el usuario ingresara esas comillas dobles pero nuestro sistema las transformase de forma inmediata en código alfanumérico (que luego tuviese la capacidad de interpretar y mostrar como al usuario como comillas dobles), estaríamos solucionando un requerimiento sin poner en riesgo la seguridad de nuestro sistema.

ECODQUOT podría ser el equivalente a las comillas dobles unicodes " y al " de HTML para nuestro sistema.

2 www.debianhackers.net

3 www.daboblog.com

4 www.modsecurity.org

Y eso, es lo que hace **EuropioCode**: transformar los caracteres no alfanuméricos en secuencias alfanuméricas que luego sea capaz de interpretar.

Codificación Europio

Antes de utilizar EuropioCode es muy importante saber que aún se encuentra en fase beta y solo contempla un número limitado de caracteres *unicode*, como se muestra en la tabla de la siguiente tabla.

Tabla de codificación de caracteres unicode

CARACTER UNICODE	EUROPIOCODE			CARACTER UNICODE	EUROPIOCODE		
	PREFIJO	CÓDIGO	SUFIJO		PREFIJO	CÓDIGO	SUFIJO
!	ECODG	33	ECODC	"	ECODG	34	ECODC
#		35		$		36	
%		37		&		38	
'		39		(		40	
)		41		*		42	
+		43		,		44	
-		45		.		46	
/		47		:		58	
<		60		=		61	
>		62		?		63	
@		64		[		91	
\		92		]		93	
^		94		_		95	
`		96		{		123	
|		124		}		125	
~		126		€		128	
(espacio)		160		¡		161	
£		163		«		171	
´		180		·		183	
»		187		¿		191	
Ç		199		ç		231	
Á		193		É		201	
Í		205		Ó		211	
Ú		218		Ü		220	
Ñ		209		á		225	
é		233		í		237	
ó		243		ú		250	
ü		252		ñ		241	
(tabulado)		09		;		59	
(salto de línea)		ECODS					

Tabla de codificación alfanumérica de caracteres

Como se puede apreciar, la codificación EuropioCode se basa en un **prefijo** (ECODG) y un **sufijo** (ECODC) predeterminados y entre medio, el **número que conforma el código hexadecimal** del carácter correspondiente. La única excepción es el salto de línea el cual es representado por el código alfabético ECODS.

La combinación **PREFIJO + NÚMERO DEL CÓDIGO HEXADECIMAL + SUFIJO** es la que permite diferenciar al intérprete, los caracteres alfanuméricos intencionales del código Europio.

Tabla de codificación de etiquetas para texto con formato enriquecido

La codificación Europio está preparada para admitir el formato de texto enriquecido, facilitando la codificación -y consiguiente decodificación- de etiquetas HTML. Para ello, todo texto introducido que sea *preformateado* mediante etiquetas HTML, siempre que se permita, será convertido sobre la base del siguiente esquema:

ETIQUETA HTML	EUROPIOCODE		ETIQUETA HTML	EUROPIOCODE	
	PREFIJO	CÓDIGO		PREFIJO	CÓDIGO
	pFt	00		pFt	12
		01			13
<i>		02	<h1>		14
		03	<h2>		15
<u>		04	<h3>		16
<strike>		05	<h4>		17
<sub>		06	<h5>		18
<sup>		07	<h6>		19
<p>		08	<code>		20
<blockquote>		09	<pre>		21
<hr>		10	 		22
		11	<small>		23

Tabla de código Europio para texto enriquecido

De la tabla anterior, se debe tener en cuenta que:

- Ninguna de las etiquetas anteriores debe aceptar atributos;
- Si una etiqueta HTML poseyera atributos, no sería interpretada como etiqueta de texto enriquecido, sino como caracteres no alfanuméricos a ser codificados;
- Aquellas etiquetas que requieren una etiqueta extra de cierre, se codificarán con los mismos códigos de la tabla anterior, pero antecediendo la vocal **e** al sufijo (`pFt00` será la codificación de la etiqueta mientras que `pFte00` lo será para)

Asimismo, se facilita una **codificación especial para etiquetas que deban indefectiblemente aceptar atributos**, como es el caso particular de la etiqueta **<a>** (ver tabla en la página siguiente).

Por favor, tener en cuenta que en caso de pretender codificar enlaces en formato HTML, solo será admitida la siguiente sintaxis:

```
<a href="url"[ target="_blank|_self|top"]>anchor text</a>
```

Ejemplos de **enlaces válidos** serían los siguientes:

```
<a href="pagina.html">Visitar enlace</a>
<a href="http://www.duckduckgo.com">Buscar en DuckDuckGo</a>
<a href="pagina.html" target="_self">Visitar enlace</a>
<a href="http://www.duckduckgo.com" target="_blank">Buscar en DuckDuckGo</a>
```

Sin embargo, **ninguno de los siguientes enlaces serán válidos como tales**:

```
<a target="_self" href="pagina.html">Visitar enlace</a>
<a href="http://www.duckduckgo.com" title="Buscar en DuckDuckGo">Buscar en DuckDuckGo</a>
<a href="pagina.html" class="mi_estilo">Visitar enlace</a>
```

Tabla de codificación de caracteres en hipervínculos

Dentro de los hipervínculos, etiquetas y atributos son codificados sobre la base de los códigos de la siguiente tabla:

CARACTER / GRUPO DE CARACTERES	EUROPIOCODE	CARACTER / GRUPO DE CARACTERES	EUROPIOCODE	CARACTER / GRUPO DE CARACTERES	EUROPIOCODE
<a href=	aH0n2	**target=_**	tG0n7	**-**	gN6n1
.	p01nt	**~**	nN0n5	**://**	pT7n3
/	bB0n1	**"**	*null*	**>**	fT0x1
****	eT0n1				

De esta forma, el enlace:

```
<a href="http://www.duckduckgo.com" target="_blank">Buscar en DuckDuckGo</a>
```

Enlace con espacios en blanco sin codificar (77 Bytes)

sería codificado como:

```
aH0n2httppT7n3wwwp01ntduckduckgop01ntcomECODG160ECODCtG0n7blankfT0x1BuscarECODG160ECODCenECODG160ECODCDuckDuckGoeT0n1
```

Enlace con espacios en blanco codificado (118 Bytes)

Un ejemplo similar, pero con enlaces sin espacios en blanco, se ve más compensado en la codificación:

```
<a href="http://www.duckduckgo.com">DuckDuckGo</a>
```

Enlace sin espacios en blanco antes de ser codificado (51 Bytes)

```
aH0n2httpT7n3wwwp01ntduckduckgop01ntcomfT0x1DuckDuckGoeT0n1
```

Enlace sin espacios en blanco ya codificado (61 Bytes - un 20% más)

Como observación, una forma de optimizar la codificación (que me apunto como un TODO), sería que el código **tG0n7**, incluyera al espacio en blanco previo al atributo *target* en la tabla de equivalencias para **target=_**

Uso de EuropioCode

Para que codificar cadenas de texto mediante EuropioCode no se convierta en una tarea tediosa, se puede recurrir a las librerías JavaScript (cliente) y PHP (aplicación) que codifican mediante EuropioCode teniendo resueltas de antemano, todas las funcionalidades necesarias.

Se puede obtener la última versión del paquete completo de librerías, haciendo un `branch` del repositorio oficial en Launchpad[5]:

```
bzr branch lp:~eugeniabahit/europiocode/trunk
```

Por favor, notar que **Launchpad** utiliza **Bazaar**[6], el sistema de control de versiones desarrollado por **Canonical** que forma parte oficial del **proyecto GNU**[7].

EuropioCode JS

europio_code.js https://bazaar.launchpad.net/~eugeniabahit/europiocode/trunk/view/head:/europio_code.js
La librería cliente, es un JavaScript capaz de codificar grandes cadenas de texto plano y texto enriquecido.

Su **uso**, está especialmente indicado para la codificación de grandes áreas de texto (`textarea`) en formularios HTML.

La forma de **implementación** sugerida, es mediante la codificación de dichos campos al ser invocado el envío del formulario, es decir, en el evento `onsubmit` tal como se muestra en el primer ejemplo de la página siguiente.

5 https://launchpad.net/
6 https://www.gnu.org/software/bazaar/
7 https://www.gnu.org/

```
window.onload = function() {
    europiocode = new EuropioCode();
    document.getElementById('id_del_formulario').onsubmit = function() {
        europiocode.encode('id_del_campo');
    };
};
```

La librería JavaScript posee dos **métodos de codificación**:

```
EuropioCode.encode()
Para codificar texto plano

EuropioCode.encode_preformat()
Para codificar texto enriquecido respetando etiquetas de formato
```

Cualquiera de las dos funciones puede ser llamada en el evento `onsubmit` y tras codificar el campo (o los campos) indicados, pasarán a un estado de solo lectura antes de ser enviados.

Vale aclarar que pueden codificarse tantos campos como sea necesario. No obstante se aconseja su uso con precaución, ya que **la codificación del texto no comprime el tamaño de las cadenas**, sino que actúan de forma contraria, insumiendo un mayor número de bytes.

El siguiente, podría ser ejemplo de implementación de EuropioCode JS para codificar el formulario utilizado por un CMS el cual es usado para agregar una nueva entrada a un *blog*. El mismo, supone tres campos de texto de tipo `textarea` con las ID: `etiquetas`, `keywords` y `entrada`:

```
window.onload = function() {

    europiocode = new EuropioCode();

    document.getElementById('nueva_entrada').onsubmit = function() {
        europiocode.encode('etiquetas');
        europiocode.encode('keywords');
        europiocode.encode_preformat('entrada');  // permite tags HTML
    };

};
```

EuropioCode PHP

europio_code.php

https://bazaar.launchpad.net/~eugeniabahit/europiocode/trunk/view/head:/europio_code.php

Por el momento, solo se dispone de una librería oficial a nivel aplicación, desarrollada en PHP. Por ello, me gustaría invitar a programadores y programadoras de diversos lenguajes como Python, Perl, Ruby, Java (entre otros), a motivarse y desarrollar una librería en el lenguaje de preferencia.

EuropioCode PHP es capaz no solo de decodificar sino también de decodificar, revertir, limpiar y purgar una cadena de texto. Pero veremos esto en detalle para poder comprenderlo mejor.

EuropioCode::encode()

Se aconseja su uso como refuerzo de seguridad en el envío de formularios. Asimismo, se aconseja la persistencia de cadenas codificadas mediante archivos de texto. Para persistencia en base de datos, ver más adelante el método clean.

Codifica una cadena de texto de la misma forma que lo hace la librería cliente.
Se puede utilizar como refuerzo de la seguridad a nivel aplicación, ya que todo lo que se implemente del lado del cliente, puede ser eludido con relativa facilidad.
Su uso no afecta a una cadena que ya se encuentre codificada.

Uso:
EuropioCode::encode(*string* $cadena)

Retorna:
La cadena codificada

Ejemplo de implementación:
```
$input = isset($_POST['input']) ? EuropioCode::encode($_POST['input']) : '';
```

Resultados obtenidos:

```
# Entrada del usuario: <b>Hola Mundo!</b>
$input = isset($_POST['input']) ? EuropioCode::encode($_POST['input']) : '';
print $input;
/*
Salida:
ECODG60ECODCbECODG62ECODCHolaECODG160ECODCMundoECODG33ECODCECODG60ECODCECODG47ECODCbECODG62E
CODC
*/
```

EuropioCode::decode()

Su uso está indicado solo para la lectura y decodificación de texto previamente codificado, preferentemente almacenado en un archivo de texto.

Decodifica una cadena previamente codificada. La decodificación NO se realiza al estado natural de la cadena antes de codificarse, sino que todos los caracteres no alfanuméricos previamente codificados, son convertidos a sus entidades HTML hexadecimales correspondientes.

```
Uso:
    EuropioCode::decode(string $cadena)

Retorna:
    La cadena decodificada a entidades HTML

Ejemplo de implementación:
    $encode_content = file_get_contents('entrada-1573');
    $decode_content = EuropioCode::decode($encode_content);
```

Resultados obtenidos:

```
$encode_content = file_get_contents('entrada-1573');
print $encode_content;
/*
Salida:
ECODG60ECODCbECODG62ECODCHolaECODG160ECODCMundoECODG33ECODCECODG60ECODCECODG47ECODCbECODG62E
CODC
*/

$decode_content = EuropioCode::decode($encode_content);
print $decode_content;
/*
Salida:
&#60;b&#62;Hola Mundo&#33;&#60;&#47;b&#62;
*/
```

EuropioCode::decode_preformat()

Se aconseja su uso para lectura y decodificación de archivos de texto enriquecido previamente codificados con encode_preformat

```
Decodifica una cadena de texto enriquecido, previamente codificada con encode_preformat.
La decodificación NO se realiza completamente al estado natural de la cadena. Solo, se
decodificarán a estado natural, aquellas etiquetas HTML permitidas.

Uso:
    EuropioCode::decode_preformat(string $cadena)

Retorna:
    La cadena enriquecida decodificada

Ejemplo de implementación:
    $encode_content = file_get_contents('entrada-1574');
    $decode_content = EuropioCode::decode_preformat($encode_content);
```

Resultados obtenidos:

```
$encode_content = file_get_contents('entrada-1574');
print $encode_content;
/*
Salida:
pFt00EtiquetaECODG160ECODCadmitidapFte00ECODSECODG60ECODCdivECODG62ECODCEtiquetaECODG160ECOD
CnoECODG160ECODCadmitidaECODG60ECODCECODG47ECODCdivECODG62ECODC

En estado natural, se hubiese visto como:
<b>Etiqueta admitida</b>
<div>Etiqueta no admitida</div>
*/

$decode_content = EuropioCode::decode_preformat($encode_content);
print $decode_content;
/*
Salida:
<b>Etiqueta admitida</b>
&#60;div&#62;Etiqueta no admitida&#60;&#47;div&#62;
*/
```

EuropioCode::clean()

Se aconseja su uso para almacenamiento persistente en bases de datos.

Elimina caracteres no alfanuméricos de una cadena decodificada, reemplazando acentos, cedillas y virgulillas de eñes de vocales, ce y eñes respectivamente.
Su uso es especialmente útil para almacenar resúmenes de archivos en una base de datos.

Uso:

```
EuropioCode::clean(string $cadena)
```

Retorna:

La cadena sin caracteres no alfanuméricos

Implementación:

```
$encode_content = file_get_contents('entrada-1575');
$decode_content = EuropioCode::decode($encode_content);
$clean_content = EuropioCode::clean($decode_content);
```

Resultados obtenidos:

```
$encode_content = file_get_contents('entrada-1575');
print $encode_content;
/*
Salida:
UnECODG160ECODCECODG241ECODCandECODG250ECODCECODG160ECODCqueECODG160ECODCvivECODG237ECODCaEC
ODG160ECODCenECODG160ECODCBraziliaECODG160ECODCseECODG160ECODCmudECODG243ECODCECODG160ECODCa
ECODG160ECODCValeECODG160ECODCdoECODG160ECODCAECODG231ECODCo

En estado natural, se hubiese visto como:
Un ñandú que vivía en Brazilia se mudó a Vale do Aço
*/
```

```
$decode_content = EuropioCode::decode($encode_content);
print $decode_content;
/*
Salida:
Un &#241;and&#250; que viv&#237;a en Brazilia se mud&#243
; a Vale do A&#231;o
*/

$clean_content = EuropioCode::clean($decode_content);
print $clean_content;
/*
Salida:
Un nandu que vivia en Brazilia se mudo a Vale do Aco
*/
```

EuropioCode::purge()

Se aconseja su uso para extracción de palabras claves.

Limpia una cadena decodificada y luego elimina palabras repetidas y de longitud menor a 3 caracteres. Su uso es especialmente útil para obtener las palabras claves de un archivo o contenido determinado.

Uso:

```
EuropioCode::purge(string $cadena)
```

Retorna:

La cadena limpiada y sin palabras repetidas ni de longitud inferior a 3 caracteres

Implementación:

```
$encode_content = file_get_contents('entrada-1576');
$decode_content = EuropioCode::decode($encode_content);
$clean_content = EuropioCode::purge($decode_content);
```

Resultados obtenidos:

```
$encode_content = file_get_contents('entrada-1576');
print $encode_content;
/*
Salida:
UnECODG160ECODCECODG241ECODCandECODG250ECODCECODG160ECODCqueECODG160ECODCvivECODG237ECODCaEC
ODG160ECODCenECODG160ECODCBraziliaECODG160ECODCseECODG160ECODCmudECODG243ECODCECODG160ECODCa
ECODG160ECODCValeECODG160ECODCdoECODG160ECODCAECODG231ECODCoECODG160ECODCporqueECODG160ECODC
enECODG160ECODCBraziliaECODG160ECODCnoECODG160ECODChabitabaECODG160ECODCningECODG250ECODCnEC
ODG160ECODCotroECODG160ECODCECODG241ECODCandECODG250ECODCECODG46ECODC

En estado natural, se hubiese visto como:
Un ñandú que vivía en Brazilia se mudó a Vale do Aço porque en Brazilia no habitaba ningún
otro ñandú.
*/

$decode_content = EuropioCode::decode($encode_content);
print $decode_content;
/*
Salida:
```

```
Un &#241;and&#250; que viv&#237;a en Brazilia se mud&#243
; a Vale do A&#231;o porque en Brazilia no habi
taba ning&#250;n otro &#241;and&#250;&#46;
*/

$clean_content = EuropioCode::purge($decode_content);
print $clean_content;
/*
Salida:
nandu que vivia brazilia mudo vale aco porque habitaba ningun otro
*/
```

EuropioCode::reverse()

Se aconseja su uso SOLO Y ÚNICAMENTE, previa modificación del tipo MIME como texto plano en los encabezados HTTP. Especialmente útil, para ofrecer descargas de código fuente en estado puro. POR FAVOR, NO UTILIZAR ESTE MÉTODO EN PRODUCCIÓN, SI SE CARECE DE CONOCIMIENTOS SOBRE SEGURIDAD INFORMÁTICA A NIVEL SERVIDORES.

Vuelve una cadena codificada a su estado puro tal como se encontraba antes de la codificación.

Uso:

```
EuropioCode::reverse(string $cadena)
```

Retorna:

La cadena en estado original puro

Implementación:

```
$encode_content = file_get_contents('sourcecode-2718');
$decode_content = EuropioCode::decode($encode_content);
header("Content-Type: text/text; charset=utf-8");
print EuropioCode::reverse($decode_content);
```

Resultados obtenidos:

```
$encode_content = file_get_contents('sourcecode-2718');
/*
Contenido del archivo: sourcecode-2718
ECODG60ECODCECODG63ECODCphpECODSECODSfunctionECODG160ECODCfooECODG40ECODCECODG36ECODCparamEC
ODG61ECODCNULLECODG41ECODCECODG160ECODCECODG123ECODCECODSECODG160ECODCECODG160ECODCECODG160E
CODCECODG160ECODCifECODG40ECODCECODG33ECODCisECODG95ECODCnullECODG40ECODCECODG36ECODCparamEC
ODG41ECODCECODG41ECODCECODG160ECODCreturnECODG160ECODCsplitECODG40ECODCECODG36ECODCparamECOD
G44ECODCECODG160ECODCchrECODG40ECODC10ECODG41ECODCECODG41ECODCECODG59ECODCECODSECODG125ECODC
ECODSECODSECODG63ECODCECODG62ECODC
*/

$decode_content = EuropioCode::decode($encode_content);
/*
&#60;&#63;php
```

```
function foo&#40;&#36;param&#61;NULL&#41; &#123;
    if&#40;&#33;is&#95;null&#40;&#36;param&#41;&#41; return sp
lit&#40;&#36;param&#44; chr&#40;10&#41;&#41;&#59;
&#125;

&#63;&#62;
*/

header("Content-Type: text/text; charset=utf-8");
print EuropioCode::reverse($decode_content);
/*
Salida:

<?php

function foo($param=NULL) {
    if(!is_null($param)) return split($param, chr(10));
}

?>

*/
```

Ingeniería Inversa: solo el "qué" puede desvelar el camino hacia el "cómo"

La Ingeniería Inversa aplicada al desarrollo de Software no es ni fácil ni difícil. Su sencillez puede resultar complicada o su complejidad puede resultar sencilla, dependiendo de cada persona. Por lo tanto, se desaconseja la implementación de las técnicas aquí expuestas, si al estudiar este paper se experimenta la sensación de estar "descubriendo errores", puesto que la misma es indicio de una comprensión incorrecta que podría llevar a la adquisición de malas prácticas.

Cuando la ingeniería inversa se aplica para el desarrollo de software, suele ser frecuente que el programador comience el diseño de la aplicación anticipando el cómo la desarrollará, sin haber evaluado previamente qué es lo que de sea lograr. Sin dudas, esto es **lo que más trabajo cuesta a los programadores: entender «el qué» y diferenciarlo del «cómo»**.

A diferencia de la Ingeniería Inversa de hardware, **la fórmula de la Ingeniería Inversa para el desarrollo de Software** no consiste únicamente en seguir una serie de pasos. Por el contrario, cada paso a seguir **consiste en**, paradójicamente, analizar y **deducir cuáles son -justamente- los pasos a seguir**.

> **La mejor forma de organizar el desarrollo es efectuar el análisis de requerimientos por medio de Historias de Usuarios**

Organizar el análisis de requerimientos mediante Historias de Usuarios, es el primer paso para comprender primero **qué es lo que se desea lograr**.

> *Si no se sabe con exactitud qué resultado se espera ver, no hay forma posible de hacer una evaluación que indique si vamos o no por el camino correcto.*

Una Historia de Usuario, en sí misma es un objetivo a alcanzar. Sin embargo, **el programador confunde el objetivo en sí mismo, con el cómo ese objetivo debe ser alcanzado**. Pero para entenderlo mejor, veámoslo con un ejemplo.

Supongamos que tenemos la siguiente Historia de Usuario:

«Como administrador puedo agregar nuevas categorías al catálogo de productos»

No interesa si se está programando orientado a objetos o no o si es mejor implementar un paradigma u otro. Lo que nos interesa ahora es entender cuál es el objetivo. Y a diferencia de lo que la mayoría de los programadores creería, el único objetivo que se encuentra inicialmente es **«insertar nuevas categorías a un catálogo de productos»** el cual se supone implícitamente, que ya existe.

Entonces, lo que se espera ver es **una nueva categoría dentro del sitio donde el mecanismo de persistencia elegido** (una base de datos, por ejemplo) **almacene la información**.

Teniendo en claro qué es exactamente lo que se espera ver, recién allí es cuándo se debe comenzar a pensar en cómo lograrlo.

Esto necesariamente implica tener que **convertir un objetivo en el procedimiento necesario para alcanzar el siguiente objetivo**. De eso se trata la ingeniería inversa. Cada objetivo debe ser a la vez, convertido en su propio cómo y este último, debe arrojar el siguiente objetivo.

Por ejemplo, si insertar categorías es un objetivo, el cómo nos lo tiene que dar el propio objetivo. ¿Qué significa esto? Qué lo primero que debemos hacer es insertar esas categorías tan manualmente como sea posible. Esto es "transformar el objetivo en un cómo alcanzarlo". Suponiendo que trabajamos proceduralmente (a fin de no enredar a los menos expertos con cuestiones inherentes a la programación orientada a objetos) y la base de datos fuese nuestro mecanismo de persistencia, necesitaríamos una función que guardase una categoría en la base de datos.

Cuando te cueste hallar "el qué", escribe un caso de uso. Eso te ayudará a encontrarlo más fácilmente.

Por ejemplo, podemos citar como caso de uso, el administrador que agrega la categoría llamada «Electrónica». Entonces ¿qué debemos hacer? Escribir el mínimo código necesario para insertar la categoría «Electrónica» en la base de datos (recordemos que el objetivo fue transformado en el cómo alcanzarlo):

```
function agregar_categoria() {
    # Definir el query
    $sql = "INSERT INTO categorias (categoria_id, denominacion) VALUES (NULL, ?)";
    # Asignamos los datos manualmente (hardcodeamos)
```

```
        $data = array("s", "Electrónica");
        # Ejecutamos la consulta
        $categoria_id = DBLayer::ejecutar($sql, $data);
        # retornamos el resultado de la ejecución a fin de constatar que la misma haya sido efectiva
        return $categoria_id;
}


# Invocamos a la función y probamos que funcione, es decir, corroboramos que se agregue la denominación Electrónica
# en la tabla categorias
print agregar_categoria();
```

Como se puede observar, simplemente nos limitamos a:

1. **Identificar el primer objetivo:** agregar una nueva categoría;
2. **Alcanzar el objetivo:** convertimos el objetivo en una acción concreta que nos dice como alcanzar el objetivo (insertar un nuevo registro en la tabla categorías)

Una vez hecho esto, se verifica que funciona. Si falla, se corrige tantas veces como sea necesario hasta lograr que funcione.

Nunca intentes adivinar un error. Utiliza el log de errores para saber por qué ha fallado algo.

Ahora, se busca un nuevo objetivo. Antes, habíamos dicho que "el cómo de un objetivo también debería ser quien nos arroje el siguiente objetivo". **Siempre después de *hardcodear* algo, el objetivo siguiente es elegir una parte del código para dejar de *hardcodear***. El ciclo se repite hasta que ya no quede nada que necesite dejar de ser *hardcodeado*. En nuestro caso, estamos insertando a mano, la denominación de la categoría desde el propio código. Nuestro objetivo debería estar relacionado a "dejar de insertar a mano la denominación de la categoría".

Se debe ser sumamente cuidadoso ya que aquí suelen aparecer gran parte de las complicaciones. Obviamente nuestro objetivo es hacer que el valor del campo `denominacion` sea variable (por favor, notar que cito como objetivo que el campo SEA VARIABLE pero en ningún momento menciono algo como "que valor sea ingresado mediante un formulario", es decir, no hablo de cómo lograrlo ni mucho menos, me planteo objetivos ambiciosos con respecto a lo que ya he alcanzado).

El cómo lograrlo, puede variar de programador en programador y cada uno tendrá ideas diferentes. Pero siempre debe establecerse como regla principal, **escribir el mínimo código necesario**.

En este caso puntual, se me ocurre una sola alternativa: para que un valor sea variable, lo defino en una variable (recurro a lo más simple, es decir, busco lo obvio que en definitiva no es más que convertir al objetivo en un cómo). Ahora mi objetivo, es definir el valor de una categoría en una variable (el cómo se transforma en el siguiente objetivo). Y se me ocurren -en principio- dos formas de lograrlo:

1. Definir la variable dentro de la función;

2. Pasar el valor como parámetro;

Frente a dos o más alternativas, se debe aplicar siempre el principio de la Navaja de Ockham, el cual dice que: **la solución más simple suele ser la más probable** (aunque no necesariamente la correcta). Para ello, debo analizar y comparar las alternativas.

Si definiera la variable dentro de la función y quisiera, por ejemplo, insertar 3 categorías, debería editar el archivo 3 veces y ejecutarlo 3 veces, mientras que pasando el valor como parámetro, podría invocar a la función 3 veces. Como esta última requiere menor esfuerzo (y el esfuerzo refleja la complejidad de algo), la elijo:

```
function agregar_categoria($denominacion) {
        $sql = "INSERT INTO categorias (categoria_id, denominacion) VALUES (NULL, ?)";
        $data = array("s", "$denominacion");
        $categoria_id = DBLayer::ejecutar($sql, $data);
        return $categoria_id;
}


print agregar_categoria('Juguetes');
print agregar_categoria('Ferretería');
print agregar_categoria('Bazar');
```

A pesar de todo, sigo *hardcodeando* los datos (aunque ya un poco menos que antes). Ahora, lo que debo lograr es que el valor de la categoría, no sea escrito manualmente en el archivo. La única forma de lograr mi nuevo objetivo, es hacer que el usuario lo ingrese. Seguramente, esto sea uno de los criterios de aceptación de la HU. Incluso, hasta debe mencionar que deberá hacerse mediante un formulario. Pero aún, no he llegado a los criterios de aceptación, así que omitiré pensar en ellos (incluso cuando los conozca).

Omite pensar en aquellas restricciones a las que aún no haz llegado. Deja que el código te guíe. De lo contrario, no estarás implementando la ingeniería inversa sino la tradicional.

Para que el usuario ingrese dicho valor podría hacerlo mediante dos métodos (POST o GET). Sé que POST es el más seguro pero implicaría mucho esfuerzo: crear un formulario HTML y la función que lo muestre o crear una función utilizando cURL para enviar los datos, lo cual implicaría escribirlos a mano nuevamente o enviarlos mediante GET, recogerlos y enviarlos con cURL mediante POST. DEMASIADO complicado. Pasarlo por GET (y listo) resulta mucho más sencillo. Y como bien dijimos, no debemos (ahora) pensar en restricciones futuras. Aún no hemos llegado a la seguridad de la aplicación ni a los criterios de aceptación, pues entonces, adelante con utilizar el método GET.
La función no cambiará; solo lo hará la forma en la que ésta será invocada:

```
print agregar_categoria($_GET['categoria']);
```

Vuelvo a probar, simplemente modificando el valor del parámetro en la URI tantas veces como lo desee:

```
.....categorias.php?categoria=Indumentaria
.....categorias.php?categoria=Deportes
.....categorias.php?categoria=Jardinería
.....categorias.php?categoria=Farmacia
```

Una vez que ya se ha eliminado todo dato *hardcodeado*, puedo pasar a los criterios de aceptación.

Por favor, notar que hasta aquí, hemos comenzado el desarrollo por el final, es decir, por cumplir el primer objetivo el cual es la Historia de Usuario en sí mismo. Esto, justamente, es aplicar la Ingeniería Inversa.

Entre los criterios de aceptación de esta Historia de Usuario, por lo general, suelen encontrarse varios que hacen referencia a las restricciones sobre los datos a agregar. Desde algunos que mencionan cantidades mínimas y máximas de caracteres hasta otros que nos dicen qué hacer si se intentan incluir datos duplicados.

Ten en cuenta que cada criterio de aceptación será un objetivo

Esto significa que debemos tomar uno -y solo uno- incluso aunque parezca "poco". Por ejemplo, imaginemos que entre los criterios de aceptación encontramos los siguientes:

- *Categoría con menos de 3 caracteres retorna mensaje "El nombre de la categoría debe tener al menos 3 caracteres sin incluir los espacios en blanco"*
- *Categoría con más de 30 caracteres retorna mensaje "El nombre de la categoría no puede contener más de 30 caracteres incluyendo espacios en blanco"*
- *etc....*

Ejemplo de Criterios de Aceptación

Lógicamente habrá muchos criterios de aceptación más, pero **se debe elegir solo uno** de ellos -sin excepción- para continuar el desarrollo y sobre el elegido, se repetirán todos los procedimientos anteriores.

Para que no queden dudas sobre cómo aplicar Ingeniería Inversa al desarrollo de Software, propongo un ejercicio sencillo que a mis alumnos suele ayudarlos a esclarecer toda duda. Se trata de desarrollar una librería (o *script*) que retorne una cadena de texto aleatoria (a modo de contraseña) con las siguientes características:

- Debe tener al menos 1 letra mayúscula, 1 minúscula, 1 número y 1 carácter no alfanumérico

- Su longitud debe ser de 8 caracteres
- El orden de aparición de los caracteres debe ser aleatorio (es decir que no puede cumplir un patrón)

Como única ayuda diré que:

- Se debe comenzar *hardcodeando* el resultado. Por ejemplo, comenzar retornando una contraseña a modo de ejemplo, que cumpla con los requisitos anteriores: **aA1#A1a=**

- Del retorno anterior, ir sustituyendo los caracteres *hardcodeados* de a uno por vez respetando el patrón del retorno inicial y dejando para el final, la "mezcla" de caracteres para evitar patrones.

Si sigues los pasos mencionados y no te dejas llevar por suposiciones ni por información que no esté explícita, lograrás entender por completo el procedimiento para desarrollar Software aplicando Ingeniería Inversa.

Ingeniería de Software: wrappers y decoradores en Python

En el mundo de la ingeniería de sistemas, Python es uno de los lenguajes más temidos y respetados. El respeto, no es algo que deba perderse, pero el miedo, debería ser erradicado. Y eso, es lo que pretendo lograr con esta serie de dos entregas: otorgar una explicación clara, simple y definitiva sobre los 4 "monstruos" del lenguaje. En la entrega final, wrappers y decoradores.

En la edición anterior de The Original Hacker, se hizo una explicación profunda y detallada sobre las funciones *lambda* y los ***closures*** dejando pendientes los ***wrappers*** y ***decoradores*** para esta entrega. Y de poner a los *wrappers* y decoradores en blanco sobre negro, es que nos encargaremos ahora.

Antes de comenzar es necesario entender que:

1. No se puede hablar de *wrappers* sin haber entendido previamente qué son los *closures*. Por lo tanto, si no lo haz hecho aún, te recomiendo leer las **páginas 5 a 8** de **The Original Hacker Nº3**, edición que puedes obtener en http://library.originalhacker.org/biblioteca/revista/ver/19

2. No se puede hablar de decoradores sin hablar de wrappers, puesto que básicamente ambos términos hacen referencia a lo mismo, pero con una sutil diferencia a nivel teórico-conceptual.

Aclarado esto, estamos en condiciones de continuar con lo nuestro.

Lo primero que debemos recordar, es que **un *closure* es una función que dentro de ella contiene a otra función** la cual es retornada cuando el *closure* es invocado:

```
def closure(parametro1):
    def funcion(parametro2):
        print parametro1 + parametro2
    return funcion

foo = closure(10)  # foo ahora es la función interna del closure
print foo(200)     # Imprime:  210
```

```
print foo(500)     # Imprime:  510
```

No olvides repasar el artículo sobre closures publicado en The Original Hacker Nº3: http://library.originalhacker.org/biblioteca/revista/ver/19

Un decorador (*decorator*), **es aquel *closure* que como parámetro recibe a una función (llamada función "decorada") como único argumento**:

```
def decorador(funcion_decorada):

    def funcion():
        pass

    return funcion
```

Mientras que **un *wrapper*** no es más que la **función interna de un *closure* que a la vez sea de tipo decorador** (función a la que en los ejemplos anteriores llamamos "funcion" a secas):

```
def decorador(funcion_decorada):

    def wrapper():
        pass

    return wrapper
```

La **función decorada** deberá ser invocada por el *wrapper*:

```
def decorador(funcion_decorada):

    def wrapper():
        return funcion_decorada()

    return wrapper
```

El decorador no se invoca como una función normal. Éste es llamado con una sintaxis especial:

```
@decorador
```

La sintaxis anterior se debe colocar en la línea anterior a la definición de la función decorada. De esta forma, el nombre de la función decorada es pasado como parámetro de forma automática sin tener que invocar nada más:

```
@decorador
def funcion_decorada():
    print 'Soy una función decorada'
```

Hasta aquí entonces, tenemos que:

- Un ***closure*** es una función que dentro de ella define otra función.
- Un **decorador** es un *closure* que recibe una función como parámetro.
- Una **función decorada** es la que se pasa como parámetro a un decorador.
- Un ***wrapper*** es la función interna del decorador, encargada de retornar a la función decorada.

¿Cómo funcionan y se accionan los wrappers y decoradores?

Cuando una función es decorada, el decorador se acciona de forma automática en el momento que el *script* es ejecutado:

```
#!/usr/bin/env python


def decorador(funcion):
    print 'Soy el decorador()'

    def wrapper():
        print 'Soy el wrapper()'
        return funcion()

    return wrapper


@decorador
def funcion_decorada():
    print 'Soy la funcion_decorada()'
```

Al ejecutar este *script* SIN haber invocado ninguna función, podemos ver como el decorador ya ha actuado:

```
eugenia@cococha-gnucita:~/borrador$ ./foo.py
Soy el decorador()
eugenia@cococha-gnucita:~/borrador$
```

Cuando la función decorada es invocada, el decorador ya la habrá reemplazado por el *wrapper*, retornando a éste en lugar de la función original, es decir, retorna al *warpper* en lugar la función decorada, tal como muestra el siguiente ejemplo:

```
#!/usr/bin/env python
```

```
def decorador(funcion):
    print 'Soy el decorador()'

    def wrapper():
        print 'Soy el wrapper()'
        # return funcion()

    return wrapper


@decorador
def funcion_decorada():
    print 'Soy la funcion_decorada()'


funcion_decorada()
```

El resultado de la ejecución del *script* será el siguiente:

```
eugenia@cococha-gnucita:~/borrador$ ./foo.py
Soy el decorador()
Soy el wrapper()
eugenia@cococha-gnucita:~/borrador$
```

Como bien puede verse, al haber invocado a `funcion_decorada()`, no fue ésta quien se ejecutó sino que ha sido la función `wrapper()`.

Pero, tal y como se dijo al comienzo, el *wrapper* será quien tras su ejecución, invoque a la función decorada:

```
#!/usr/bin/env python


def decorador(funcion):
    print 'Soy el decorador()'

    def wrapper():
        print 'Soy el wrapper()'
        return funcion()

    return wrapper


@decorador
def funcion_decorada():
    print 'Soy la funcion_decorada()'


funcion_decorada()
```

El resultado ahora, será el siguiente:

```
eugenia@cococha-gnucita:~/borrador$ ./foo.py
Soy el decorador()
Soy el wrapper()
Soy la funcion_decorada()
eugenia@cococha-gnucita:~/borrador$
```

El **orden de ejecución en *wrappers* y decoradores** se puede establecer como el siguiente:

1. **decorador** (automáticamente al ejecutar el *script* reemplazando la función decorada por el *wrapper*)
2. ***wrapper*** (al invocar a la función decorada)
3. **función decorada** (luego de ejecutarse el *wrapper*)

> *Puede decirse entonces que conceptualmente,* ***un decorador es un closure que se encarga de reemplazar a la función decorada por su función interna, a la cuál se denomina wrapper***

¿Para qué usaríamos un wrapper?

En primer lugar, notarás que la pregunta NO ha sido para qué usar un decorador, sino para que usar un *wrapper*. Esto es debido a que la respuesta a para qué utilizar un decorador es fácilmente deducible: para implementar un *wrapper*.

Los *wrappers* o **envolturas**, suelen utilizarse cuando se tienen funciones que ANTES de ejecutar su verdadera funcionalidad, realizan acciones redundantes. Un ejemplo muy típico, es cuando se utilizan bloques `try` y `except`:

```
def get_template():
    try:
        with open('template.html', 'r') as archivo:
            return archivo.read()
    except:
        return 'ERROR INTERNO'


def calcular(partes=0, total=100):
    try:
        return total / partes
    except:
        return 'ERROR INTERNO'
```

Como podemos ver, tenemos dos funciones que utilizan bloques try y except (generalmente, se tienen muchísimas más). Una forma de manejar los errores en estas funciones, sería utilizar un wrapper:

```
def intentar(funcion):
```

```
    # Usamos *args y **kwargs ya que tenemos una función que necesita argumentos
    def wrapper(*args, **kwargs):
        try:
            return funcion(*args, **kwargs)
        except:
            return 'ERROR INTERNO'

    return wrapper
```

De esta forma, solo deberíamos decorar a las funciones que anteriormente utilizaban los bloques `try` y `except`:

```
@intentar
def get_template():
    with open('template.html', 'r') as archivo:
        return archivo.read()


@intentar
def calcular(partes=0, total=100):
    return total / partes
```

Si invocáramos a `calcular` con los argumentos por defecto, se produciría un error al intentar dividir por cero, pero sería manejado por el *wrapper*:

```
eugenia@cococha-gnucita:~/borrador$ ./bar.py
ERROR INTERNO
eugenia@cococha-gnucita:~/borrador$
```

Frecuentemente, lo mismo que sucede con el **manejo de errores** en cuanto a acciones repetidas en distintas funciones, suele darse con acciones como el **control de acceso** (usado por ejemplo, para verificar si un usuario tiene permisos suficientes antes de efectuar una determinada acción) o el **registro de actividades** (conocido como *logging*).

Un uso menos frecuente aunque paradójicamente podríamos mencionarlo como imprescindible, es para el **control de datos** mediante **filtros de saneamiento** o de **tratamiento**, como se muestra en el siguiente ejemplo Web para Python sobre Apache con WSGI:

```
def get_post_data(funcion):  # decorador

    def wrapper(environ):  # claramente este es el wrapper del decorador
        _POST = {}

        try:
            datos = environ['wsgi.input'].read().split('&')
            for par in datos:
                key, value = par.split('=')
```

```
                _POST[key] = unquote(value).replace('+', ' ')
                key, value = (None, None)
        except:
            pass

        return funcion(_POST)  # retorna a la función decorada una vez termina lo anterior

    return wrapper


@get_post_data      # llamada al decorador
def guardar(POST):  # función decorada
    nombre = POST['nombre']
    apellido = POST['apellido']
    # ...


@get_post_data         # llamada al decorador
def actualizar(POST):  # función decorada
    pass


@get_post_data       # llamada al decorador
def eliminar(POST):  # función decorada
    pass
```

Como puede verse en el ejemplo anterior, la llamada al decorador es una sintaxis abreviada equivalente a:

```
get_post_data(guardar)
def guardar(POST):
    nombre = POST['nombre']
    apellido = POST['apellido']
    # ...


get_post_data(actualizar)
def actualizar(POST):
    pass


get_post_data(eliminar)
def eliminar(POST):
    pass
```

Sin embargo, la nueva sintaxis propuesta por la incorporación de decoradores, hace la lectura del código fuente mucho más simple.

EuropioEngine Lab: SecurityLayer, una capa de seguridad inteligente

Este paper es la continuación de la investigación iniciada en el artículo "Capas de seguridad inteligentes en PHP y saneamiento automático del array POST" publicado en The Original Hacker Nº2. Todo lo explicado anteriormente, se ha implementado en una capa de seguridad para Europio Engine tal como se comenta a continuación.

Un par de ediciones atrás, escribía un artículo sobre cómo gestar una capa de seguridad inteligente que saneara de forma automática el *array superglobal* $_POST. Si deseas, puedes obtener mis notas sobre el tema visitando el siguiente enlace: http://library.originalhacker.org/biblioteca/articulo/ver/120

Parte de la investigación, consistió en ir intentando implementar el resultado obtenido de cada estudio, en una capa de seguridad para Europio Engine. La misma fue agregada como *plugin*.

Puede verse el código fuente completo (incluidos los test) ingresando en el **repositorio oficial de Europio Engine**: https://bazaar.launchpad.net/~eugeniabahit/europioexperimental/trunk/files/8/common/plugins/securitylayer

No obstante, antes de que decidas abarcar el código de dicha capa de seguridad, veamos cómo funciona para que puedas llegar al código con mayor información.

ATENCIÓN: para seguir los ejemplos, recomiendo trabajar con un branch actualizado del repositorio. Para obtener un branch, ejecuta:

```
bzr branch lp:europioexperimental
```

Para hacer las pruebas, creé un módulo llamado `pruebas`:

```
cd core/cli
./europio -c pruebas
```

Y dentro del módulo, agregué el modelo demo:

```
./europio -f pruebas Demo
```

Agregué un formulario simple en la vista utilizando el *plugin* `WebForm` y para no tener que completarlo manualmente, le inserté los valores de prueba por defecto:

```
# Archivo: appmodules/pruebas/views/demo.php

public function agregar() {
    $form = new WebForm('/prueba/demo/guardar');
    $form->add_text('email', NULL, 'juan&quot;perez&quot;@somemail.foo');
    $form->add_text('number', NULL, '0982725');
    $form->add_text('float_1', NULL, '09.827,25');
    $form->add_text('float_2', NULL, '09,827.25');
    $form->add_submit();
    print Template('Agregar Demo')->show($form->show());
}
```

Me aseguré de tener habilitada la capa de seguridad en el archivo `settings.php`:

```
# Archivo settings.php - Línea 55
const SECURITY_LAYER_ENGINE = 'On';
```

Y de tener todos los *plugin* habilitados:

```
# Archivo settings.php - Línea 62
$enabled_apps = array('securitylayer', 'webform', 'collectorviewer');
```

Finalmente, modifiqué el recurso **`guardar`** del controlador, agregando simplemente un `print_r()` del *array* `$_POST`:

```
# Archivo: appmodules/pruebas/controllers/demo.php

public function guardar() {
    print_r($_POST);
}
```

En el navegador, ingresé en `/pruebas/demo/agregar` y al **enviar el formulario con la capa de seguridad activada**, este fue el resultado de la depuración del *array superglobal* `$_POST`:

```
Array
(
    [email] => juanperez@somemail.foo
    [number] => 982725
    [float_1] => 9827.25
    [float_2] => 9827.25
)
```

Todos los datos han sido filtrados de manera inteligente y han llegado al controlador de forma segura.

Sin embargo, **desactivando la capa de seguridad**:

```
# Archivo settings.php - Línea 55
const SECURITY_LAYER_ENGINE = 'Off';
```

El resultado se ve de la siguiente forma:

```
Array
(
    [email] => juan"perez"@somemail.foo
    [number] => 0982725
    [float_1] => 09.827,25
    [float_2] => 09,827.25
)
```

Esta capa, aún se encuentra en una fase experimental ya que -entre otras cosas- resta implementar soporte para saneamiento de *arrays* (algo que ya estoy probando y será incorporado en breve).

Mantener el núcleo actualizado

Recuerda que puedes mantener tu versión de **Europio Engine** en constante actualización, clonando los archivos del **repositorio oficial**: https://bazaar.launchpad.net/~eugeniabahit/europioexperimental/trunk/files

Para poder acceder, necesitas tener instalado **Bazaar**, el sistema de control de versiones que integra el proyecto GNU:

```
apt-get install bzr
```

Para clonarlo, simplemente necesitas ejecutar:

```
bzr branch lp:europioexperimental
```

Y luego, podrás mantenerlo actualizado, ejecutando todos los días (antes de trabajar en el directorio) un:

```
bzr pull
```

Notas sobre el código fuente

Activación

La capa de seguridad ha sido preparada para actuar de forma autónoma sin necesidad de ser invocada por el programador.

Su activación y desactivación se maneja mediante la constante **SECURITY_LAYER_ENGINE** definida en el archivo `settings.php` de la aplicación.

El inicializador de la capa (archivo `__init__.php`) invocará al método de limpieza de la capa, solo si el valor de la constante ha sido establecido en `On`. **Cualquier valor distinto de `On`, provocará la desactivación** del arranque automático.

```
if(ucwords(SECURITY_LAYER_ENGINE) == 'On') ...
    ...
    SecurityLayer($strict)->clean_post_data($sanitize_array);
```

Solución a problemas comunes:
Si SECURITY_LAYER_ENGINE ha sido establecida en On y sin embargo la capa parece no estar actuando, verificar que el *plug-in* se encuentre habilitado en el archivo `settings.php`:

```
$enabled_apps = array('securitylayer');
```

Inteligencia de filtrado

La capa de seguridad utiliza dos criterios para decidir el tipo de filtro que debe ser aplicado:

1. por nombre de clave: solo aplica a los campos de formulario cuyo valor sea de tipo e-mail, considerando como tales, a cualquiera que incluya la cadena `mail` como parte del nombre. De esta forma, un campo denominado `mail_address` (dirección de correo) sería interpretado como campo de tipo e-mail. Por lo tanto, se aconseja ser específico con los nombres de los campos y si se desea una dirección postal (y los campos se nombran en inglés), debería utilizarse `postal_address`.

```
    if(strpos($key, 'mail') !== False) $this->purge_email($key);
```

2. por contenido: todo campo que responda `True` a **`is_numeric()`** -previo eliminado ficticio de la coma- será considerado numérico y evaluado para deducir si se trata de número con coma flotante o entero. Ver método **`sanitize_number()`** en línea 34. Todo campo que no sea de tipo e-mail ni numérico, será considerado de tipo ***string*** aplicando un filtrado genérico. Ver método **`encode_string()`** en línea 24.

Nota adicional sobre valores numéricos:
La capa de seguridad utiliza inteligencia artificial para distinguir el separador de miles del de decimales. Si el número posee coma y además punto, supone un número con miles y decimales:

```
$has_colon = ($pos_colon !== False);
$has_dot = ($pos_dot !== False);

if($has_colon && $has_dot) ...
```

Si la posición de la coma es mayor a la del punto, entonces la coma indica decimales. Caso contrario, indica miles:

```
$pos_colon = strpos($_POST[$key], ',');
$pos_dot = strpos($_POST[$key], '.');

if($pos_colon > $pos_dot) ...
```

Si posee solo una coma o solo un punto (pero no ambos), considerará al signo en cuestión como separador de decimales:

```
elseif($has_colon xor $has_dot) ...
```

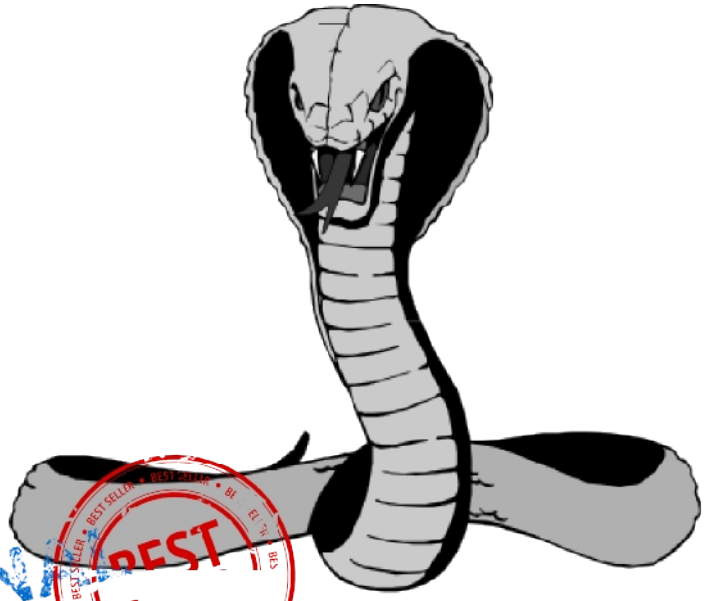
Todo caso no contemplado en los condicionales anteriores, será considerado un número entero.

Finalmente, aplica el punto como separador de decimales y suprime el separador de miles:

```
$this->helpernum('.', '', $key);  # reemplaza el punto por un valor nulo
$this->helpernum(',', '.', $key);  # reemplaza la coma por un punto
```

Filtrado en modo estricto

Si la constante **`SECURITY_LAYER_STRICT_MODE`** (definida en el `settings.php`) es `True`, aplicará un filtro adicional a todos los campos, aplicando simultáneamente las funciones `htmlentities` y `strip_tags`. Por defecto, el valor de *strict mode* se encuentra establecido en `False`.



The Original Hacker #_4__