the original Hacker

creado por EUGENIA BAHIT

jugando con la inteligencia

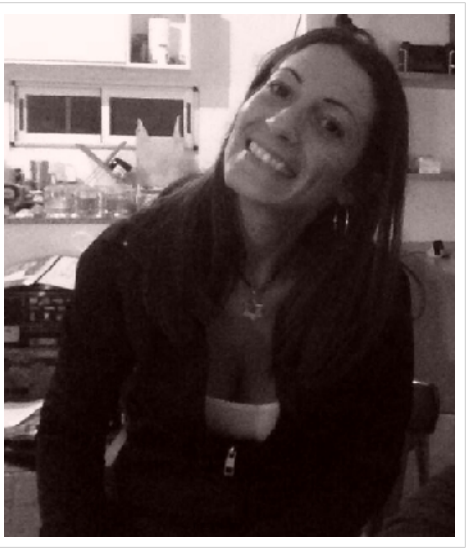
número 5

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, 29 de Abril de 2014

ÍNDICE DE LA EDICIÓN NRO5

Ingeniería de Software: Manipulación de Web Forms y Carga de Archivos con Python y WSGI sbre Apache

Trabajar con formularios HTML desde Python y sobre todo, manejar la carga de archivos, es una de las tareas menos sencillas a la hora de crear aplicaciones Web sin utilizar Frameworks. Sin embargo, que no sea la más sencilla no significa que sea imposible.

Desde que Python comenzó a implementarse en el diseño de aplicaciones Web, *frameworks* como **Django** o **Web2Py** han parecido ser la única alternativa posible para desarrollar Software accesible mediante un navegador.

El auge de estos *frameworks* y por sobre todo, la gran publicidad que los programadores menos experimentados en la Ingeniería de Software basado en Web le han hecho a Django, lograron acercar a miles de de usuarios que sin conocimientos de programación en Python, podían desarrollar aplicaciones Web con tan solo aprender a usar el *framework*.

Sabemos que **la humanidad se mueve a base de abusos** de todo tipo y la Ingeniería de Software, no es ajena a ello. Tanto es así que **los usuarios que desarrollan aplicaciones Web utilizando Django, desconocen por completo el lenguaje** y carecen de nociones básicas de programación, a un punto tal que en mucho casos, se llega a creer imposible que puedan desarrollarse aplicaciones Web en Python sin el uso de *frameworks*.

Hace un tiempo publiqué un *paper* en **Debian Hackers**[1], sobre cómo crear un sitio Web en Python bajo Apache sin utilizar *frameworks*[2]. Este artículo pretende continuar esta idea, otorgando al lector las herramientas necesarias para manipular todo tipo de formularios Web incluyendo la carga de archivos al servidor. Toda esta información puede complementarse con el material[3] del **curso de Desarrollo de Aplicaciones Web con Python y MySQL** que dicté durante 2012 y 2013 en cursos.eugeniabahit.com

1 www.debianhackers.net

2 http://www.debianhackers.net/una-web-en-python-sobre-apache-sin-frameworks-y-en-solo-3-pasos

3 http://www.cursosdeprogramacionadistancia.com/static/pdf/material-sin-personalizar-python.pdf

Declarando el enctype correcto

En los formularios HTML, el atributo `enctype` se define cuando el método ha sido establecido como POST y su finalidad es la de indicar en qué forma serán codificados los datos al enviarse el formulario.

```
<form id='something' method='post' action='/some/file' enctype='multipart/form-data'>
</form>
```

El atributo `enctype` puede tener 3 **valores** diferentes:

multipart/form-data

Codificación:
ninguna (los caracteres son enviados tal cual están, sin codificación previa)

Uso:
recomendado para manipular la carga de archivos

Delimitador de campos:
aleatorio (se obtiene mediante la clave CONTENT-TYPE del diccionario environ)

```
delimitador = environ['CONTENT-TYPE'].replace('multipart/form-data; boundary=', '')
# retornará algo como: -----------------------------175537584916979984255548673 82
```

Formato recepción de datos:
Ejemplo para un formulario con dos campos de textos (nombre y edad)

```
-----------------------------175537584916979984255548673 82
Content-Disposition: form-data; name="nombre"

Juan Pérez
-----------------------------175537584916979984255548673 82
Content-Disposition: form-data; name="edad"

90
-----------------------------175537584916979984255548673 82--
```

application/x-www-form-urlencoded

Codificación:
ASCII Hexadecimal. Los espacios en blanco se reemplazan por el signo +

Uso:
es el valor por defecto de todo formulario. Se recomienda para todos los formularios que no requieran manipular la carga de archivos.

Delimitador de campos:
Signo &

Formato recepción de datos:
Ejemplo para un formulario con dos campos de textos (nombre y edad)

```
nombre=Juan+P%C3%A9rez&edad=90
```

Notar que la vocal “e” acentuada ha sido codificada como %C3% mientras que el espacio en blanco fue sustituido por el signo +

text/plain
Codificación:
ninguna (los caracteres son enviados tal cual están, sin codificación previa)

Uso:
desaconsejado

Delimitador de campos:
retorno de carro

Formato recepción de datos:
Ejemplo para un formulario con dos campos de textos (nombre y edad)

```
nombre=JuanPérez
edad=90
```

El uso del valor text/plain se desaconseja ya que al emplear el retorno de carro como delimitador, dificulta la obtención de datos en bloques de texto que incluyan el salto de línea.

Para el común de los formularios, no es necesario declarar el enctype. Declararlo como multipart/form-data si se trabajará con la carga de archivos.

Manipulando datos desde Python y WSGI

Cuando se reciben datos mediante POST, siempre se necesita tener acceso a dos factores: el nombre de los campos y su correspondiente valor. En Python, la forma lógica de asociar un nombre de campo a un valor, es utilizar un diccionario:

```
datos =  dict(campo1='valor del campo 1', campo2='valor del campo 2')
```

Lo que se debe lograr entonces, es crear un diccionario con dicha información convirtiendo los datos recibidos en un diccionario.

La forma de lograr esta conversión, depende directamente del modelo de codificación con el que la información haya sido enviada, es decir que depende del `enctype` que se haya declarado en el formulario.

Cuando se trabaja con WSGI los datos enviados por `POST` se almacenan en la clave `wsgi.input` del diccionario `environ` (que WSGI entrega a la función `application`) y se obtienen leyendo dicho elemento con el método `read()` tal como se muestra a cotinuación:

```
datos = environ['wsgi.input'].read()
Retornatará algo como: campo1=valor1&campo2=valor2
```

APPLICATION/X-WWW-FORM-URLENCODED

Recordemos que éste, es el valor por defecto de todo formulario. Si no se ha asignado un `enctype`, los datos serán codificados siguiendo este formato.

Como el delimitador de campos es el signo &, podemos obtener una lista donde cada elemento sea un par `clave=valor`

```
datos = environ['wsgi.input'].read().split('&')
```

Luego, en cada elemento de la lista, el signo = separa al nombre del campo de su valor correspondiente con lo que ya tenemos todos los elementos necesarios para armar el diccionario:

```
datos = environ['wsgi.input'].read().split('&')

POST = {}

for par in datos:
    campo, valor = par.split('=')
    POST[campo] = valor
```

Recordemos que los datos son codificados en formato ASCII Hexadecimal y los espacios en blanco, sustituidos por el signo +. Podemos volver los valores a su estado puro, utilizando la función `unquote` del módulo `urllib2` y reemplazando el signo + por un espacio en blanco:

```
from urllib2 import unquote


datos = environ['wsgi.input'].read().split('&')
```

```
POST = {}

for par in datos:
    campo, valor = par.split('=')
    POST[campo] = unquote(valor).replace('+', ' ')
```

De esta forma, habremos obtenido un diccionario como el siguiente:

```
{
    'nombre': 'Juán Pérez',
    'edad': '65',
    'nacionalidad': 'uruguaya'
}
```

Al cual podremos acceder utilizando el nombre de los campos como nombre de clave:

```
nombre = POST['nombre'] if 'nombre' in POST else ''
edad = POST['edad'] if 'edad' in POST else ''
nacionalidad = POST['nacionalidad'] if 'nacionalidad' in POST else ''
```

RECOGER GRUPOS DE DATOS

En cualquier formulario Web, es frecuente tener grupos de opciones bajo un mismo nombre. Es el caso de los campos de tipo checkbox y los de tipo select múltiple:

```
<input type='checkbox' name='grupo_a' value='1' checked>Opción 1<br>
<input type='checkbox' name='grupo_a' value='2'>Opción 2<br>
<input type='checkbox' name='grupo_a' value='3' checked>Opción 3<br>

<select name='combo' size='3' multiple>
    <option value='1'>Opción 1</option>
    <option value='2'>Opción 1</option>
    <option value='3'>Opción 1</option>
    <option value='4'>Opción 1</option>
    <option value='5'>Opción 1</option>
</select>
```

Cuando éste es el caso, **los valores elegidos no son agrupados al momento de enviarse**. Por el contrario, el nombre del campo se repetirá tantas veces como opciones se hayan elegido.

Lo que se pretende entonces es, capturar las opciones elegidas dentro de una lista asignada a la misma clave del diccionario.

Un ejemplo de lo que se quiere obtener, sería como el siguiente:

```
{
    'grupo_a': [1, 3],
    'combo': [2, 4, 5]
}
```

Para lograrlo, si simplemente asignáramos el valor a la clave, el último valor sobre escribiría a los anteriores. Entonces, antes de asignar un valor a una clave, debemos verificar que la clave no exista.

```
POST = {}

for par in datos:
    campo, valor = par.split('=')

    if not campo in POST:
        POST[campo] = unquote(valor).replace('+', ' ')
```

Si la clave existe, podemos encontrar dos posibilidades:

1. Que su valor sea un único dato (el dato fue almacenado cuando `if not campo in POST` fue verdadero)
2. Que su valor sea una lista (cuando `if not campo in POST` fuese falso)

Esto habrá que verificarlo y:

1. En el primer caso, habrá que convertir al valor de dicha clave en una lista y agregarle el valor que ya tenía asignado más el nuevo valor;
2. y en el segundo, agregar directamente el nuevo valor.

Completaremos el código agrupándolo ya mismo en una función y haremos que esta retorne el diccionario con los datos para que pueda ser llamada desde cualquier función que necesite obtener datos enviados desde un formulario:

```
from urllib2 import unquote


def get_simple_form_data(environ):
    datos = environ['wsgi.input'].read().split('&')
    POST = {}

    for par in datos:
        campo, valor = par.split('=')
        valor = unquote(valor).replace('+', ' ')

        if not campo in POST:
            POST[campo] = valor
        else:
            if not isinstance(POST[campo], list):  # aún no es una lista
                POST[campo] = [POST[campo], valor]
            else:  # ya es una lista
                POST[campo].append(valor)
```

```
    return POST
```

Luego, podríamos llamarla desde cualquier otra función:

```
def enviar_form(environ):
    POST = get_simple_form_data(environ)
    # ...
```

MULTIPART/FORM-DATA

Manipular datos en este caso, es mucho más complejo ya que parte de esos datos, es la carga de archivos.
Para poder gestionar archivos cargados desde un formulario, se necesita recurrir a dos módulos adicionales: **tempfile** (para trabajar con archivos temporales) y **cgi** (para facilitar el acceso a todos los datos y obtener toda la información adicional sobre los mismos). De cada módulo, haremos uso de una sola clase: `TemporaryFile` y `FieldStorage` respectivamente.

```
from cgi import FieldStorage
from tempfile import TemporaryFile
```

La clase **FieldStorage** sirve para almacenar una secuencia de campos leyendo, justamente, los datos enviados desde un formulario codificado como `multipart/form-data`. Esta clase genera un diccionario como el que creamos anteriormente de forma manual, pero con una diferencia muy importante: cada elemento de este diccionario contiene información relevante como el tipo MIME del dato e incluye el contenido de archivos incluso cuando éstos sean binarios.
Pero esta clase requiere indefectiblemente que se le indique un puntero y no puede utilizarse `wsgi.input` para ser apuntado como archivo. Es necesario entonces, crear un archivo temporal para poder trabajar con `FieldStorage`, la clase que nos permitirá manejar la carga de archivos.

Lo primero que debemos hacer entonces, es grabar el contenido de `wsgi.input` en un archivo temporal ya que no nos será posible trabajar directamente con `wsgi.input` ya que solo podríamos acceder al nombre del archivo. Grabándolo en un archivo temporal, podremos utilizarlo como puntero desde `FileStorage` para crear el archivo en el servidor. Pero vamos de a poco. Comencemos por grabar el contenido de `wsgi.input` en un archivo temporal:

```
# Crea un archivo temporal y lo abre (por defecto) en modo w+r
archivo_temporal = TemporaryFile()

# Escribimos el contenido de wsgi.input en el archivo temporal
archivo_temporal.write(environ['wsgi.input'].read())

# Movemos al cursor al inicio ya que necesitaremos leer este archivo desde el comienzo
archivo_temporal.seek(0)
```

Una vez creado el archivo temporal, podemos inicializar un objeto `FieldStorage` indicando como puntero al archivo temporal que acabamos de crear:

```
datos = FieldStorage(fp=archivo_temporal)
```

Por defecto, `FieldStorage` obtiene el diccionario `environ` desde el módulo os, pero como estamos trabajando con WSGI, sabemos que el diccionario `environ` es modificado por WSGI, así que se lo pasaremos para que pueda obtener la información correcta:

```
datos = FieldStorage(fp=archivo_temporal, environ=environ)
```

Ahora, la variable `datos` se ha convertido en un diccionario sobre el que podremos iterar para obtener los pares de clave y sus valores correspondientes.

El nombre del campo (o clave para nuestro diccionario), lo obtendremos iterando entonces, sobre la variable datos, tratándola como si fuese un diccionario común y corriente:

```
for clave in datos:
    # ...
```

Antes de obtener el valor de un campo, necesitaremos saber si se trata o no de un archivo, porque de ser así, no solo necesitaríamos el contenido del archivo, sino otros datos como el tipo MIME y el nombre del archivo. Para saber si se trata o no de un archivo, debemos verificar si la propiedad `filename` es `None`:

```
for clave in datos:
    if datos[clave].filename is None:
        # NO ES un archivo
```

Si no se tratase de un archivo podríamos obtener el valor de este campo mediante la propiedad `value`:

```
for clave in datos:
    if datos[clave].filename is None:
        valor = datos[clave].value
```

Si en cambio se tratara de un archivo, quisiéramos que el valor fuese otro diccionario con los siguientes datos: nombre del archivo, tipo MIME y contenido (para poder crearlo posteriormente en el servidor). Para ello, necesitaremos acceder a las propiedades `filename`, `type` y `value` respectivamente:

```
for clave in datos:
    if datos[clave].filename is None:
        valor = datos[clave].value
    else:
        valor = dict(
            filename=datos[clave].filename,
            filetype=datos[clave].type,
            content=datos[clave].value
        )
```

Finalmente, solo resta utilizar las claves y valores para crear nuestro propio diccionario igual que hicimos anteriormente:

```
POST = {}

for clave in datos:
    if datos[clave].filename is None:
        valor = datos[clave].value
    else:
        valor = dict(
            filename=datos[clave].filename,
            filetype=datos[clave].type,
            content=datos[clave].value
        )

    POST[clave] = valor
```

RECOGER GRUPOS DE DATOS

Nuevamente nos podemos encontrar con que tenemos grupos de datos que pertenecen a un mismo nombre. En este caso, el tratamiento difiere del que hemos hecho antes ya que `FieldStorage` nos permite acceder a los grupos de datos como una lista, mediante el método `getlist()`. Lo que haremos entonces es verificar si se trata o no de un grupo de datos, llamando a este método y de ser así, el valor ya no será el obtenido mediante la propiedad `value`, sino el obtenido mediante este método. Al igual que en el caso anterior, vamos a aprovechar a colocar todo en una función que retorne el diccionario POST, para que pueda ser llamada desde cualquier otra que la necesite:

```
from cgi import FieldStorage
from tempfile import TemporaryFile


def get_multipart_form_data(environ):
    archivo_temporal = TemporaryFile()
    archivo_temporal.write(environ['wsgi.input'].read())
    archivo_temporal.seek(0)
    datos = FieldStorage(fp=archivo_temporal, environ=environ)

    POST = {}

    for clave in datos:
        if datos[clave].filename is None:
            lista = datos.getlist(clave)
            valor = datos[clave].value if len(lista) < 2 else lista
        else:
```

```
            valor = dict(
                filename=datos[clave].filename,
                filetype=datos[clave].type,
                content=datos[clave].value
            )

        POST[clave] = valor

    return POST
```

Como podemos ver, valor será igual a lo obtenido mediante la propiedad `value` siempre y cuando la lista tenga menos de dos elementos (es decir, uno o ninguno). De lo contrario, el valor será la lista de datos.

RECOGER GRUPOS DE ARCHIVOS

En el ejemplo anterior, se contempla que los grupos de datos (campos con el mismo nombre y distintos valores) no serán de tipo *file*. Sin embargo, puede suceder que un mismo formulario, tenga más de un campo de tipo *file* con el mismo nombre y aquí, la metodología de implementar `getlist()` para reconocer «grupos de información» ya no será viable. Por consiguiente, la primera condición a evaluar ya no sería preguntar si `datos[clave].filename` es nulo sino, si `datos[claves]` es una lista.

Para que la función no se haga repetitiva y engorrosa, crearemos una función adicional, encargada de retornar los valores como diccionario o *string*, según se trate de un campo de tipo file o no y luego, la llamaremos desde `get_multipart_form_data()`.

```
from cgi import FieldStorage
from tempfile import TemporaryFile


def get_multipart_form_data(environ):
    archivo_temporal = TemporaryFile()
    archivo_temporal.write(environ['wsgi.input'].read())
    archivo_temporal.seek(0)
    datos = FieldStorage(fp=archivo_temporal, environ=environ)

    POST = {}

    for clave in datos:
        if isinstance(datos[clave], list):
            POST[clave] = []
            for elemento in datos[clave]:
                POST[clave].append(traer_valor(elemento))
        else:
            POST[clave] = traer_valor(datos[clave])

    return POST


def traer_valor(dato):
    valor = dict(filetype=dato.type, filename=dato.filename, content=dato.value)
    return dato.value if dato.filename is None else valor
```

Un decorador que decida solo

Hemos logrado abarcar todas las posibilidades en cuanto a manipulación de formularios respecta desde Python con WSGI. Creamos dos funciones que contemplan todas las posibilidades que existen y podemos con ellas, crear un decorador para que mediante un simple llamado, envuelva a cada función que reciba datos desde un formulario.

Si te perdiste la edición anterior sobre wrappers y decoradores en Python, te recomiendo descargues The Original Hacker Nº4 y leas el artículo de página 30.

Para descargar The Original Hacker Nº4 ingresa en http://library.originalhacker.org/biblioteca/revista/ver/20

Crearemos un decorador genérico que identificando el `enctype` del formulario, se encargue de llamar a una u otra función desde el `wrapper`.

No necesitamos modificar las funciones que ya creamos aunque recomiendo hacerlas privadas y utilizar un módulo `.py` para las funciones y el decorador.

```
from cgi import FieldStorage
from tempfile import TemporaryFile
from urllib2 import unquote


def get_post_data(funcion):

    def wrapper(environ):
        _POST = {}

        try:
            if not environ['CONTENT_TYPE'].startswith('multipart/form-data'):
                _POST = __get_simple_form_data(environ)
            else:
                _POST = __get_multipart_form_data(environ)
        except:
            pass

        return funcion(_POST)

    return wrapper


def __get_simple_form_data(environ):
    datos = environ['wsgi.input'].read().split('&')

    POST = {}

    for par in datos:
        campo, valor = par.split('=')
        valor = unquote(valor).replace('+', ' ')

        if not campo in POST:
            POST[campo] = valor
        else:
            if not isinstance(POST[campo], list):
```

```
                POST[campo] = [POST[campo], valor]
            else:
                POST[campo].append(valor)

    return POST


def __get_multipart_formdata(environ):
    POST = {}

    archivo_temporal = TemporaryFile()
    archivo_temporal.write(environ['wsgi.input'].read())
    archivo_temporal.seek(0)
    datos = FieldStorage(fp=archivo_temporal, environ=environ)

    for clave in datos:
        campo = datos[clave]
        if isinstance(campo, list):
            POST[clave] = []
            for elemento in campo:
                POST[clave].append(__get_value(elemento))
        else:
            POST[clave] = __get_value(campo)

    return POST


def __get_value(campo):
    archivo = dict(filetype=campo.type, filename=campo.filename, content=campo.value)
    return campo.value if campo.filename is None else archivo
```

Luego, simplemente, deberás decorar a cualquier función que reciba datos desde un formulario, independientemente del `enctype` que se le haya declarado:

```
@get_post_data
def enviar_form(POST):
    pass
```

Carga de archivos desde formularios y almacenamiento en el servidor

Ya tenemos todo listo para poder manipular los archivos. Lo único que nos resta es algo tan simple como seguir unos cortos pasos.

Crear un directorio privado con permisos de escritura:

```
mkdir private_dir && chmod 777 -R private_dir
```

Grabar el contenido del archivo en uno nuevo dentro el directorio privado:

```
@get_post_data
def enviar_form(environ):
    carpeta = '/ruta/a/directorio/privado/'
    ruta_archivo = carpeta + POST['archivo']['filename']

    with open(ruta_archivo, 'w') as archivo:
        archivo.write(POST['archivo']['content'])
```

Por supuesto que si se debe validar el tipo MIME del archivo, se lo hará mediante `POST['archivo']['filetype']` antes del bloque `with` y solo se ejecutará éste si el tipo MIME es el esperado:

```
@get_post_data
def enviar_form(environ):
    carpeta = '/ruta/a/directorio/privado/'
    ruta_archivo = carpeta + POST['archivo']['filename']
    mime = POST['archivo']['filetype']
    if mime == 'application/pdf':
        with open(ruta_archivo, 'w') as archivo:
            archivo.write(POST['archivo']['content'])
        return 'Archivo guardado'
    else:
        return 'Solo se permiten archivos PDF'
```

Europio Engine Lab: Dict Object, un nuevo concepto en Objetos para las vistas en PHP

Europio Engine en la revisión 17 de su versión beta 3.4, incorpora un nuevo concepto que revoluciona el tratamiento de las vistas en PHP. Se trata de objetos de tipo diccionario que en estas notas, te cuento la motivación, su concepción y forma de implementarlos.

En varios *papers* y artículos en general, en los cuales hablo de MVC, no me canso de repetir que **las vistas en MVC son un arte**.

En MVC, todo el arte de las vistas se encuentra siempre en **convertir lo que se tiene en lo que se necesita**. Lo que se tiene, siempre es distinto. Sin embargo, siempre es un objeto (o colección de objetos) mucho más allá de su complejidad interna y niveles de dependencia. Y siempre, lo que se necesita es convertir todos esos objetos y sus propiedades en diccionarios que no son más que *arrays* asociativos para PHP.

Motivado por la necesidad de evitar la redundancia de código en las vistas, **Jimmy Daniel Barranco** me propuso crear un *helper* a nivel del *core* que eliminara la redundancia. Y así, entre ambos, creamos un nuevo concepto en PHP: objetos de tipo diccionario. **El objeto *Dict***.

Motivación

Como comenté al comienzo, la motivación principal fue la necesidad de suprimir la redundancia de código en las vistas. La misma, se genera frecuentemente en las siguientes circunstancias:

- **Objetos compuestos de otros que comparten propiedades con la misma denominación:** frecuentemente sucede que se tiene un objeto A compuesto de un objeto B y ambos poseen una propiedad denomina C. Esto produce que en las GUI, a fin de evitar el solapamiento de identificadores idénticos, se deban utilizar comodines compuestos que por lo general, se constituyen del nombre de las propiedades acompañadas por el nombre del objeto, adquiriendo formatos similares a `{objeto.propiedad}`. Por consiguiente, es muy habitual en las vistas, ver fracciones de código destinadas a crear nuevas propiedades (o claves en un diccionario).

- **Necesidad de visualizar datos sobre objetos con varios niveles de dependencia:** la denominación de las propiedades puede no ser igual pero sin embargo, tener un objeto A, que se compone de uno B que a la vez se compone de uno C, etc. y necesitar mostrar en las GUI los datos del objeto Z, es de lo más frecuente. Y esta necesidad, también genera en las vistas, fracciones de código destinadas solo a crear nuevas propiedades o claves en un diccionario.

- **Deformación y mal uso de objetos:** amparados por el hecho de saber que las vistas en MVC son más un arte que una ciencia, para satisfacer las dos necesidades anteriores, lo frecuente se transformaba en una deformación absoluta de los objetos, lo cual, tarde o temprano termina no «oliendo» bien.

Todo lo anterior, no hace más que generar grandes bloques de código, en la mayoría de las vistas, que no solo podría evitarse, sino que además, debería evitarse ya que **la simplicidad, siempre debería ser la única solución correcta**.

Un objetivo, una solución...

Teníamos que lograr entonces, crear un *helper* que no actuase de forma «desesperada» en pro de «no repetir código». Debía solucionar el problema de una forma correcta, es decir, solucionarlo empleando un argumento lógico válido, simpleza y prolijidad. Fue entonces que al pensar en que si aquello que se necesita en las vistas es un diccionario ¿por qué no crear el concepto de diccionario como objeto? Y de hacerlo ¿qué características debería tener éste como objeto? La solución hallada se explica a continuación.

Características que debe poseer un objeto diccionario

Como características del objeto, había que proponer algo simple, que no fuese necesario explicar y que pudiese deducirse por sí solo. Como únicas características de este objeto, se emplearon las siguientes:

- **PROPIEDADES:** Sus únicas propiedades serán denominaciones formadas por pares `objeto.propiedad` justificando éstas, en que un diccionario debe ser un mapa de las propiedades de uno o varios de los objetos que se desea *parsear* en una vista.

- **MÉTODOS:** haciendo honor a los objetos hallados en los inicios clásicos de la programación orientada a objetos tradicional, solo debería contar con un método `set`. El método `get` no tendría razón de ser ya que los diccionarios deberían ser volátiles y solo gestarse en base a objetos reales, persistentes por otras vías.

Resultados

Implementando los objetos de tipo Diccionario en las vistas se obtienen resultados sumamente elegantes y de esta forma, se pone fin a los complejos algoritmos y los largos bloques de código que habitualmente solemos tener.

El objeto `Dict` es un objeto *instanciable* que al invocar a su método `set()` se le pasa un objeto real como parámetro. De esta forma, tras invocar a `set()`, el objeto `Dict` tendrá disponible un mapa de propiedades compuesto por los nombres de los objetos y las propiedades de éstos.

Para entenderlo mejor, observar el siguiente ejemplo. Dado un objeto compuesto con los siguientes niveles de profundidad:

```
Foo Object
(
    [a] => 1
    [b] => 2
    [c] => 3
    [bar] => Bar Object
        (
            [a] => 1
            [b] => 2
        )

    [far] => Far Object
        (
            [a] => 1
            [b] => 2
            [c] => 3
            [d] => 4
        )

)
```

Obtendremos el siguiente diccionario:

```
Dict Object
(
    [foo.a] => 1
    [foo.b] => 2
    [foo.c] => 3
    [bar.a] => 1
    [bar.b] => 2
    [far.a] => 1
    [far.b] => 2
    [far.c] => 3
    [far.d] => 4
)
```

El cual nos permitirá, en las GUI, utilizar identificadores como en el siguiente ejemplo:

```
<h1>{foo.a}</h1>
<h2>{foo.b}</h2>
<blockquote>
    <b>Bar:</b> {bar.b}<br>
    <b>Far:</b> {far.d}
</blockquote>
```

Uso e Implementación

Utilizar el objeto `Dict` es suma simple. Solo debe ser instanciado y luego, se debe invocar al método `set()` enviando un objeto como parámetro y luego utilizar el objeto `Dict` para efectuar la sustitución de los HTML, los cuáles deberán implementar el nombre de los objetos además del de las propiedades.

Independientemente del nivel de profundidad al que se llegue con la composición de objetos, **los identificadores (comodines) de los HTML deberán emplear el formato:**

```
{objeto.propiedad}
```

Para **generar el diccionario**, solo se requieren dos instrucciones:

```
$dict = new Dict();
$dict->set($objeto);
print_r($dict)
```

Si se quieren **obtener diccionarios sobre una colección de objetos**, se debe apelar al objeto **`DictCollection`**. El mismo también cuenta con un método `set()` al que se le debe pasar una colección de objetos (propiedad colectora). El objeto `DictCollection` proveerá de una propiedad colectora `DictCollection->collection` compuesta de una colección de objetos `Dict`:

```
$dict_collection = new DictCollection();
$dict_collection->set($coleccion);
print_r($dict_collection->collection);
```

Seguridad Informática: Modelos de Seguridad permisivos como mecanismos de prevención de vulnerabilidades

Los modelos de seguridad permisivos, a pesar de lo complejo de su análisis, son los únicos capaces de generar aplicaciones invulnerables, ya que al no estar basados en la prevención de riesgos conocidos no dan lugar a descubrir «nuevos agujeros», pues los propios modelos, no son más que «ilusiones ópticas».

Cuando comenzamos a programar, una de las primeras cosas que aprendemos en materia de seguridad de aplicaciones, es a filtrar entradas del usuario, quitándoles todo aquel caracter prohibido. Hoy me pregunto **¿de verdad en algún momento nos hemos sentido más listos que el resto del mundo?**

Prohibir es una tarea tediosa e infinita; cuanto más se analizan e investigan los posibles «flancos de ataque» en una aplicación, la tarea de prohibir puede llegar a hacerse interminable, ya que en la Ingeniería de Software aplica, indirectamente, el mismo principio que en el derecho penal: «no existe prohibición sin pena».

En los sistemas informáticos, al igual que un código penal, cada prohibición debe especificarse minuciosa y detalladamente. Pero la peor parte -para la informática- es que al prohibir, necesariamente, se debe indicar «cuál será la pena», solo que para el Software, esto es metafórico. Y cuando hablamos de «establecer una pena», nos estamos refiriendo a **implementar complejos algoritmos que respondan frente a cada acto prohibido**.

Prohibimos el ingreso de comillas simples, entonces, al igual que en el derecho, no basta con decirle a un individuo que «está prohibido matar», porque en realidad, no está prohibido, está «penado». **Con colocar un cartel que diga «por favor, no ingrese comillas simples» no alcanza; debemos crear un algoritmo que se encargue de erradicarlas.** Pero esto no termina aquí y debe repetirse el mismo esquema de procedimientos, con cada «cosa» que deseemos prohibir en nuestro sistema.

> *Cuando implementamos modelos de seguridad por prohibición,* ***no estamos teniendo en cuenta que no somos omnipotentes***

En 1999, la mayoría de las aplicaciones Web que implementaban sistemas de registro de usuarios, buscadores basados en bases de datos y afines, no efectuaban ningún tipo de filtro como los de hoy en día. Hasta que algún día alguien habrá visto que un maldito % en un buscador, podía romper el LIKE de una sentencia SQL. Y eso mismo, sucede todos los días.

> *Lamentablemente, cada vez más gente invierte su tiempo en mejorar técnicas de pentesting y en crear herramientas de exploiting, en vez de ampliar sus capacidades cognitivas intentando* ***hallar verdaderas soluciones mediante investigación científica***

Todos los días se «descubren» nuevas formas de destruir los sistemas informáticos. Se les suele llamar «vulnerabilidades» pero desde mi punto de vista, todo sistema informático aunque se encuentre libre de toda vulnerabilidad, tendrá al menos una única vulnerabilidad si se implementa un modelo de seguridad prohibitivo: **la falsa omnipotencia de sus desarrolladores**.

> *El modelo de seguridad prohibitivo es la primera vulnerabilidad de un sistema informático*

La humanidad suele no tomarse la molestia de analizar y descubrir conexiones lógicas entre dos o más hechos que en apariencia no se encontrarían relacionados. Dicho de forma más simple, **la mayoría de las personas no se molesta en «atar cabos»**. Pues de hacerlo, la seguridad informática debería ser una ciencia exacta ya que **en nuestra vida cotidiana, tenemos sobrados ejemplos de cómo «el prohibir» siempre se queda detrás de los acontecimientos**, ya que como marca un viejo dicho popular *«hecha la Ley, hecha la trampa»*. Pero a diferencia del derecho, en el caso de la Ingeniería de Software, esto tiene solución y consiste en implementar **«modelos de seguridad permisivos»**.

¿Qué son y en qué consisten los modelos de seguridad permisivos?

No es nada difícil deducir. Se trata de modelos de seguridad que para establecer sus políticas, se basan en aquello que se permite en una aplicación en vez de invertir cientos y miles de bytes definiendo algoritmos centrados en lo que se prohíbe.

Características de los modelos de seguridad permisivos

Estos modelos suelen dar libertades al usuario que comúnmente se le niegan y hasta incluso, podrían considerarse impensables como sería el caso de permitirle elegir un nombre de usuario conformado por caracteres no alfanuméricos como podrían ser comillas simples. Es por ello, que **son considerados modelos de seguridad «liberales»** en los cuales la criptografía -tanto desde la codificación hasta la encriptación- juegan un papel protagónico, puesto que son el pilar del modelo.

> *La **criptografía** y la **codificación** son el pilar de los modelos de seguridad permisivos*

Es entonces, que como principales características podemos listar las siguientes:

1. Dan amplia libertad de acción y movimiento al usuario;
2. Emplean sistemas criptográficos con mucha frecuencia;
3. Utilizan la (re)codificación como base del modelo liberal.

Ventajas y desventajas de los modelos de seguridad permisos

Los modelos de seguridad permisivos (o liberales) suelen traer aparejadas muchísimas más ventajas y por consiguiente beneficios, que desventajas.

Las desventajas suelen ser más bien de índole social ya que las aplicaciones que implementan modelos de seguridad permisivos, suelen ser blanco fácil de los «aprendices de *crackers*» o de conductas delictivas semejantes ya que inicialmente, los futuros delincuentes suelen dejarse llevar por el entusiasmo de creer que «el exceso de libertades» se debe a una conducta negligente de los programadores.

Sin embargo, **el cese de los intentos de ataque suele producirse mucho más rápido** que en las aplicaciones que implementan modelos de seguridad restrictivos (o prohibitivos) ya que en su mayoría, **este tipo de delincuentes buscan satisfacer sus fantasías de forma inmediata y al no conseguirlo, abandonan para buscar una víctima más vulnerable que les facilite el placer inmediato**.

No obstante es válido recordar que si bien todo perfil criminal busca la satisfacción inmediata de sus fantasías, existe porción más reducida de delincuentes con una psiquis mucho más compleja que podrían permanecer años tratando de violar la seguridad de un sistema por el mero hecho de sentir el placer que no logran alcanzar en otros aspectos de sus vidas. Por consiguiente, **sería un error asumir que los modelos de seguridad permisivos evitan por completo los intentos de ataques a corto plazo**.

Si bien hasta aquí toda desventaja parece estar compensada por una ventaja o beneficio, existe un factor que convierte a los modelos de seguridad permisivos en algo difícil de ser implementados pues solo serán realmente seguros si la política de permisos es delineada por profesionales que poseen un conocimiento profundo del funcionamiento interno de un sistema informático completo y de la teoría de ordenadores, pues **una política permisiva mal pensada podría generar una aplicación mucho más vulnerable** que aquella que implemente un modelo restrictivo.

Ejemplos prácticos de políticas de seguridad basadas en modelos permisivos

Aunque todo parezca -en principio- demasiado novedoso, algunas políticas resultarán familiares ya que en algún momento se habrán puesto en práctica y seguramente, en más de una oportunidad. Sin embargo, **implementar políticas permisivas de forma aislada en un modelo restrictivo, puede ser tan riesgoso como no implementar ninguna.** Por lo tanto, los siguientes ejemplos solo se exponen a fin de alcanzar una mejor comprensión sobre los modelos de seguridad permisivos y no deben ser interpretados como una guía de seguridad única.

Sin duda, la política permisiva más implementada en aplicaciones es el paso de nombres de archivo por la URI, frecuentemente utilizado para la muestra o descarga de contenido estático. Sin embargo, **la misma política podría estar planteada tanto en un marco permisivo como en uno restrictivo, dependiendo de la óptica desde la cual se lo haya elaborado**.

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = file_get_contents($archivo);
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Hasta aquí, vemos el concepto sin ninguna medida de seguridad definida. Suponiendo que en la carpeta `SOME_PATH . "/html/"` todos los archivos con extensión `html` que se almacenan puedan ser accesibles por el usuario, a lo sumo se podría incluir una validación de existencia del archivo para evitar un fallo en el código que revelase información del sistema:

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
if(file_exists($archivo)) {
    $contenido_interno = file_get_contents($archivo);
} else {
    $contenido_interno = '';
}
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

E incluso, el algoritmo anterior podría optimizarse aún más definiendo un contenido interno por defecto y haciendo el algoritmo mucho más seguro y legible:

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = '';
if(file_exists($archivo)) $contenido_interno = file_get_contents($archivo);
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Conceptualmente, a pesar de estar «permitiendo» al usuario pasar el nombre de un archivo por la URI, le

estaríamos dando un enfoque «prohibitivo» ya que si la carpeta `SOME_PATH . "/html/"` almacenara un archivo que no se quisiese poder tratar como «contenido interno» habría que «prohibir su acceso»:

```
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = '';
$es_template = ($solicitud == 'template');
$es_menu_admin = ($solicitud == 'menu_admin');
if(file_exists($archivo) && !$es_template && !$es_menu_admin) {
    $contenido_interno = file_get_contents($archivo);
}
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Y en el mejor de los casos, el almacenamiento de archivos que no se quisiesen tratar como contenido interno, dentro de esa carpeta **se estarían restringiendo «solo por políticas de seguridad y no por cuestiones arquitectónicas»**. Esto es una clara consecuencia de las políticas restrictivas.

Sin embargo, no debe confundirse «permisivo» con «descuidado». **Una política permisiva, es aquella que «define lo que se puede hacer en vez de definir aquello que está prohibido»**. Entonces, definir cuáles archivos están permitidos, será mucho más *«limpio»* y seguro que crear algoritmos para custodiar «las prohibiciones»:

```
$archivos_permitidos = array('default', 'contacto', 'newsletter', 'noticias', 'login');
$solicitud = isset($_GET['page']) : strtolower($_GET['page']) : 'default';
if(!in_array($solicitud, $archivos_permitidos)) $solicitud = 'default';
$archivo = SOME_PATH . "/html/{$solicitud}.html";
$contenido_interno = file_get_contents($archivo);
print str_replace('{contenido}', $contenido_interno, SOME_TEMPLATE);
```

Como se puede ver en el último ejemplo, estableciendo una política permisiva obtenemos:

- **un código que no requiere de mantenimientos futuros** ya que el algoritmo se mantiene intacto incluso aunque se modifique el contenido de la carpeta en cuestión. De hecho, el *array* de «archivos permitidos» debería ser configurable desde un *settings*;

- **algoritmos con menos instrucciones** ya que no será necesario verificar la existencia del archivo en nombre de la seguridad. Si se decidiese efectuar dicha validación, sería por cuestiones de *usabilidad* pero no de seguridad.

Anteriormente, también había comentado que se podía llegar a extremos impensables como permitir todo tipo de caracteres en el nombre de usuario de un sistema de registro y autenticación. Ese es otro ejemplo de política permisiva. En este caso, no solo nos ahorramos largas validaciones en el código sino que además, evitamos tener que «limpiar» la entrada del usuario. Con solo encriptar (*hashear*) el nombre del usuario, guardarlo encriptado y luego, al momento de autenticar, compararlo mediante su *hash* al igual que se hace con las contraseñas, sería suficiente:

```
$username = md5($_POST['username']);
```

Aquí lo permisivo, a pesar de lo que se podría creer en un principio, no es el ingreso de caracteres «extraños». En realidad, de forma indirecta, estamos definiendo que solo se permiten caracteres alfanuméricos, puesto que un *hash* MD5 cifrará la cadena resultando en otra que solo es alfanumérica.

Un buen programador es aquel que mejor engaña al usuario

Convertirse en un timador suena trágicamente espantoso. Sin embargo, como todo lo «virtual» en nuestros queridos resultantes «ceros y unos», no necesariamente convertirá un «timo» en algo malo, sino que será aquello que otorgue gran libertad de movimiento a los usuarios generando una significativa reducción en la curva de aprendizaje que un usuario transita hasta entender la aplicación, que lo haga sentir cómodo pero que al mismo tiempo, no ponga en riesgo la aplicación. Se trata de **«ofrecer espejos de colores»** (pero sin el cruel objetivo de la colonización).

Me sucede con frecuencia que al momento de analizar requerimientos con mis alumnos, al igual que los usuarios se dejan llevar solo y únicamente por aspectos visuales. **Hace décadas atrás, era prácticamente impensable que un programador analizase un requerimiento con «ojos de usuario».** Sin embargo, desde la llegada de «las ventanas» en modo gráfico y dispositivos como el *mouse*, la informática se hizo accesible a todos nosotros mediante ordenadores personales y en definitiva, **la *«visión amigable e intuitiva»* que los gráficos otorgan al común de las personas, facilitaron el acceso a la programación, de gran parte de la población**.

Desde entonces, se ha convertido en «moneda corriente» observar a programadores confundir el problema a resolver (requerimiento visual en el 99% de los casos) con la solución del mismo. Y esto, es importante entenderlo ya que justamente, **los modelos de seguridad permisivos se basan en «mostrar al usuario lo que desea ver»** pero aquello que el usuario desea ver, debe ofrecérsele como una ilusión óptica, una fantasía. Para nosotros, lo que el usuario vea, deberá ser el resultado de convertir «aquello que permitimos» en lo que el usuario «crea estar viendo».

Por ejemplo, le hago creer al usuario que está viendo esto:

```
function add_confirm() {
    links = document.getElementsByTagName('a');
    for(i=0; i<links.length; i++) {
        if(links[i].title.indexOf('eliminar') == 0) {
            links[i].onclick = function() {
                partes = this.href.split('/');
                preg = "¿Confirma que desea eliminar este " + partes[4] + "?";
                if(confirm(preg)) {
                    location.href = this.href;
                } else {
                    return false;
                }
            };
        }
    }
}
```

Pero en realidad, lo anterior, es una ilusión óptica, resultado de haber convertido esto otro:

```
functionECODG160ECODCaddECODG95ECODCconfirmECODG40ECODCECODG41ECODCECODG160ECODCECODG123ECODCECODS
ECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODClinksECODG160ECODCECODG61ECODCECODG160ECODCdoc
umentECODG46ECODCgetElementsByTagNameECODG40ECODCECODG39ECODCaECODG39ECODCECODG41ECODCECODG59ECODC
ECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCforECODG40ECODCiECODG61ECODC0ECODG59ECODC
ECODG160ECODCiECODG60ECODClinksECODG46ECODClengthECODG59ECODCECODG160ECODCiECODG43ECODCECODG43ECOD
CECODG41ECODCECODG160ECODCECODG123ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCEC
ODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCifECODG40ECODClinksECODG91ECODCiECODG93ECODCECOD
G46ECODCtitleECODG46ECODCindexOfECODG40ECODCECODG39ECODCeliminarECODG39ECODCECODG41ECODCECODG160EC
ODCECODG61ECODCECODG61ECODCECODG160ECODC0ECODG41ECODCECODG160ECODCECODG123ECODCECODSECODG160ECODCE
CODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160
ECODCECODG160ECODCECODG160ECODCECODG160ECODClinksECODG91ECODCiECODG93ECODCECODG46ECODConclickECODG
160ECODCECODG61ECODCECODG160ECODCfunctionECODG40ECODCECODG41ECODCECODG160ECODCECODG123ECODCECODSEC
ODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160E
CODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECO
DG160ECODCpartesECODG160ECODCECODG61ECODCECODG160ECODCthisECODG46ECODChrefECODG46ECODCsplitECODG40
ECODCECODG39ECODCECODG47ECODCECODG39ECODCECODG41ECODCECODG59ECODCECODSECODG160ECODCECODG160ECODCEC
ODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160E
CODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCpregECODG160ECOD
CECODG61ECODCECODG160ECODCECODG34ECODCECODG191ECODCConfirmaECODG160ECODCqueECODG160ECODCdeseaECODG
160ECODCeliminarECODG160ECODCesteECODG160ECODCECODG34ECODCECODG160ECODCECODG43ECODCECODG160ECODCpa
rtesECODG91ECODC4ECODG93ECODCECODG160ECODCECODG43ECODCECODG160ECODCECODG34ECODCECODG63ECODCECODG34
ECODCECODG59ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECO
DCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG
160ECODCECODG160ECODCECODG160ECODCifECODG40ECODCconfirmECODG40ECODCpregECODG41ECODCECODG41ECODCECO
DG160ECODCECODG123ECODCECODG160ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG
160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECOD
CECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG1
60ECODClocationECODG46ECODChrefECODG160ECODCECODG61ECODCECODG160ECODCthisECODG46ECODChrefECODG59EC
ODCECODG160ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECOD
CECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG1
60ECODCECODG160ECODCECODG160ECODCECODG125ECODCECODG160ECODCelseECODG160ECODCECODG123ECODCECODSECOD
G160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECO
DCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG
160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCreturnECODG160ECODCfalseECODG59ECODCEC
ODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECOD
G160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECO
DCECODG160ECODCECODG125ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODC
ECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG12
5ECODCECODG59ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECODCECODG160EC
ODCECODG160ECODCECODG160ECODCECODG125ECODCECODSECODG160ECODCECODG160ECODCECODG160ECODCECODG160ECOD
CECODG125ECODCECODSECODG125ECODC
```

Como puede verse en el ejemplo anterior, es un simple e inocuo conjunto de caracteres alfanuméricos, que solo y únicamente del lado del cliente «adoptan forma», la forma que nosotros como programadores, le hacemos ver al usuario.

Conclusión

Implementar modelos de seguridad permisivos no es algo sencillo. Por el contrario, su estrategia requiere de cierta complejidad de análisis a la que tal vez, la mayoría de los programadores no está acostumbrado. Sin embargo, no es imposible. Las grandes ventajas y sobre todo, la innumerable cantidad de beneficios que los modelos de seguridad permisivos aportan a los sistemas informáticos, justifican la inversión de tiempo y esfuerzo que la definición, desarrollo e implementación de estos modelos, demandan al programador.

No existen fórmulas únicas y por consiguiente, mucho menos las hay mágicas, para definir estos modelos. Toda política de seguridad, toda medida a implementar, deberán ser el producto de un **análisis objetivo, abstracto y cuidado** de lo que realmente se necesita.

Es imposible implementar verdaderos modelos de seguridad permisivos si se analizan las aplicaciones con la misma óptica del usuario. Es importante entender que aquello que el usuario ve, no necesariamente será real. De la misma forma en la que nosotros hallamos 20 objetos complejos en lo que el usuario solo ve como un sencillo formulario con 10 campos en los cuáles escribir (o elegir) datos, debemos ver toda la aplicación. Basarnos en la premisa de que todo, absolutamente todo lo que el usuario vea, será una gran fantasía.

Insisto a mis alumnos de forma constante en que desarrollen sobre la misma plataforma en la que finalmente se ejecutará la aplicación y en que eviten tanto como les sea posible, el uso de herramientas gráficas para trabajar. Y no lo hago por «capricho», sino por el contrario, **lo hago para ir acostumbrando al cerebro a razonar cada vez de forma lo más abstracta, independiente y aislada de «fantasías ópticas» posible.** Porque cuanto más «amigable» sea el entorno de trabajo, menos esfuerzo mental requerirá por parte del programador, pues cuanto menos esfuerzo mental esté haciendo el programador en su trabajo, más estará siendo víctima él mismo, de las «ilusiones ópticas» producidas por un programador más astuto y entonces, la pregunta es:

> *¿cómo logrará un programador, hacer aplicaciones seguras si él mismo es «víctima» de «ilusiones ópticas» que no es capaz de entender ni descifrar?*

Un usuario que programa, no es lo mismo que un programador. Si te animas a ser programador, lograrás darte cuenta de que **en realidad, las aplicaciones invulnerables existen y son posibles**.

Bash Scripting: diversas formas de implementación de menús dinámicos

Desde simples lecturas de la entrada estándar hasta complejos algoritmos; con GNU/Bash, crear menús dinámicos puede convertirse en un entretenido arte.

Cuando el *scripting* en GNU/Bash no es nuestra actividad principal y solo lo utilizamos como herramienta complementaria para nuestros programas, es de lo más frecuente que le demos poca importancia al menú de nuestro *script*. Pero tarde o temprano nos encontraremos con la necesidad de mejorarlo y emplear un menú que realmente cumpla una función protagónica para el *script*.

Por suerte, GNU/Bash es un lenguaje muy completo que nos brinda un amplio abanico de opciones para crear menús que puedan satisfacer cada una de nuestras necesidades, cubriendo así una gran cantidad de alternativas posibles.

En este artículo veremos la forma de implementar **desde sencillos menús** que tan solo actúen leyendo la información de entrada estándar **hasta *hacks* que mediante el empleo de múltiples constructores del lenguaje, generen menús con una complejidad algorítmica que pueda contemplar hasta el más mínimo detalle**.

La forma más común y tradicional con la que la mayoría de los programadores suele trabajar en un *script*, es con la lectura de una entrada estándar:

```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"
echo $OPCIONES
echo
echo -n 'Elija una opción: '; read opcion
echo "Usted eligió $opcion."
```

Básicamente, todo consiste en imprimir en pantalla una cadena de texto de la cual el usuario «copiará» una palabra (qué hace las veces de «opción»), la ingresará, nuestro *script* la leerá y en base a ella, hará determinada acción. Simple, sencillo y resuelve de forma rápida los requerimientos más primitivos de la

interacción con un usuario. A lo sumo, como gran complejidad, se le suele agregar una estructura de control condicional:

```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"
echo $OPCIONES
echo
echo -n 'Elija una opción: '; read opcion

if [ "$opcion" = "Abrir" ]; then
    echo "Esto abre un archivo"
else
    echo "Usted eligió la opción $opcion"
fi
```

Sin embargo, **cuando la cantidad de opciones** a considerar en las estructuras de control condicionales **se van incrementando**, se comienza a **optar por constructores de selección más precisos**, como es el caso del **constructor case**:

```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"
echo $OPCIONES
echo
echo -n 'Elija una opción: '; read opcion

case $opcion in
    Abrir)
        echo "Esto abre un archivo"
        ;;
    Cerrar)
        echo "Eso cierra un archivo"
        ;;
    Salir)
        echo "Esto sale del programa"
        exit
        ;;
    *)  # Cualquier opción no contemplada
        echo "No eligió ninguna opción correcta"
        ;;
esac
```

Pero en GNU/Bash, el constructor case es tan solo uno de los tantos constructores provistos por el lenguaje. También podemos encontrarnos con el constructor `select`.

El constructor select

El constructor `select`, adoptado por primera vez por Korn Shell, nos da resuelto en un solo paso:

- La enumeración de opciones posibles;
- La impresión ordenada de opciones en pantalla;
- La lectura estándar de la opción elegida por el usuario;
- La repetición constante del menú, en pantalla.

La sintaxis de este constructor es la siguiente:

```
select VARIABLE_NAME [in lista_de_opciones]; do
    # instrucciones
    [break]
done
```

Los corchetes [ ] indican «sintaxis opcional»

El constructor `select` reproduce el menú de forma permanente incluso después de haber finalizado con las instrucciones internas. La instrucción `break` romperá ese bucle.

```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"

PS3="Elija una opción: "  # Si no se indica mostrará '#?' por defecto

select opcion in $OPCIONES; do
    echo "Usted Eligió la opción $opcion"
    break  # Rompe el bucle
done


# Ejecución en bucle (continuada) – no se indica el break
select opcion in $OPCIONES; do
    echo "Usted Eligió la opción $opcion"
done
```

El algoritmo anterior producirá una salida similar a la siguiente:

```
user@host:~$ ./menus-dnamicos.sh
1) Abrir
2) Cerrar
3) Editar
4) Borrar
5) Guardar
6) Salir
Elija una opción:
```

Como puede verse, nos ahorra muchísimo esfuerzo al momento de presentar las opciones en pantallas.

> *El constructor select es el generador de menús de opciones por excelencia de GNU/Bash*

Combinando el constructor con una estructura de control condicional simple, es posible determinar si el usuario ha ingresado -o no- una opción válida:

```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"

PS3="Elija una opción: "

select opcion in $OPCIONES; do
    if [ $opcion ]; then
        echo "Usted Eligió la opción $opcion"
        break
    fi
done
```

Esto nos da una gran ventaja ya que de tratarse de una opción válida *«switchear»* la elección del usuario no requeriría de validaciones adicionales:

```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"

PS3="Elija una opción: "

select opcion in $OPCIONES; do
    if [ $opcion ]; then
        case $opcion in
            Abrir)
                echo "Esto abre un archivo"
                ;;
            Cerrar)
                echo "Eso cierra un archivo"
                ;;
            Salir)
                echo "Esto sale del programa"
                exit
                ;;
        esac
        break
    fi
done
```

SELECT HACK: CAPTURAR LA OPCIÓN INVÁLIDA

Frecuentemente, cuando en la bibliografía se habla del constructor `select` -o del tan conocido case-, solo se hace referencia a un conjunto de opciones posibles. Sin embargo, muchísimas veces una opción `select` inválida podría requerir un tratamiento distinto al de otra opción `select` inválida.

Imaginemos este escenario: el usuario se encuentra con un menú de opciones delante y la imposibilidad de elegir la indicada, puesto desconoce la utilidad de cada una. El *script* espera una entrada del usuario pero el usuario necesita conocer el número de versión del *script* antes de decidir qué opción elegir. ¿Cómo se entera del número de versión sin salir del *script*? Podría ofrecerse la misma como una opción más del menú pero sin embargo, se estaría desvirtuando la finalidad del programa. En cambio, un argumento ingresado en la entrada estándar, podría solucionar el problema.

> *Se puede capturar la respuesta del usuario accediendo a la variable de contexto $REPLY*

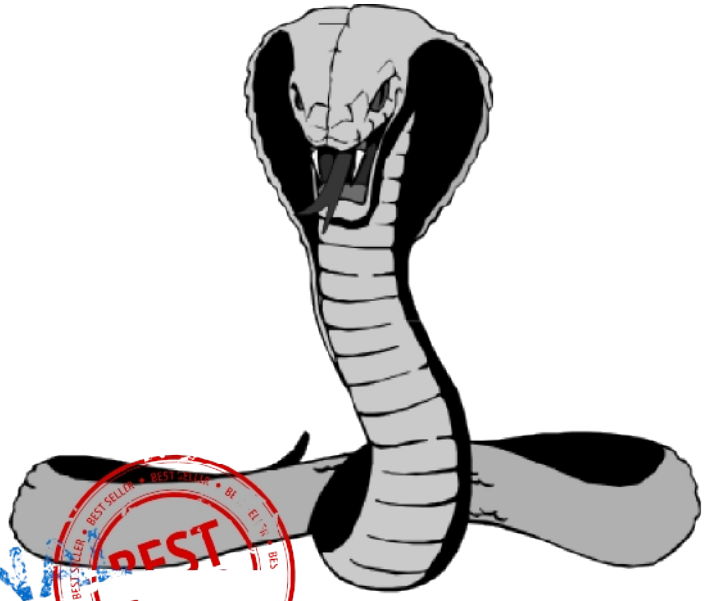
```
#!/bin/bash

OPCIONES="Abrir Cerrar Editar Borrar Guardar Salir"

PS3="Elija una opción: "

select opcion in $OPCIONES; do
    if [ $opcion ]; then
        case $opcion in
            Abrir)
                echo "Esto abre un archivo"
                ;;
            Cerrar)
                echo "Eso cierra un archivo"
                ;;
            Salir)
                echo "Esto sale del programa"
                exit
                ;;
        esac
        break
    else
        case $REPLY in
            -h|--help)
                echo "Ayuda sobre el programa"
                ;;
            -v|--version)
                echo "mi-programa versión 1.0.1"
                ;;
            -q|\q)
                exit
                ;;
            *)
                echo "Opción inválida"
        esac
    fi
done
```

ACERTIJO

¿Por qué un hombre nacido el 10 de agosto del año 15 AC a las 17:00 HS y fallecido el 10 de agosto del año 15 DC a las 16:59 HS, no llegó a cumplir los 30 años?

Ayuda: La respuesta no está en la matemática. Debes razonar de forma lateral.

Responde **ANTES** del **25/05/2014**
a través de **Twitter** utilizando el *hashtag* **#AcertijoTOH5**
El nombre de los ganadores será publicado en la siguiente edición
La respuesta correcta será publicada en The Original Hacker Nº6 junto con los ganadores



The Original Hacker #_5_