the original Hacker

creado por EUGENIA BAHIT

Jugando con la inteligencia

número 3

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.

Buenos Aires, 7 de Enero de 2014

ÍNDICE DE LA EDICIÓN NRO3

Ingeniería de Software: De lambdas, closures, wrappers y decoradores. Desmitificando Python – Parte I

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 de este mes, lambdas y closures.

Llevo años poniendo caras de sorpresa y asombro cada vez que escucho a mis colegas hablar sobre **funciones lambda**, ***closures***, ***wrappers*** y **decoradores** en Python. Y llevo la misma cantidad de tiempo tratando de dar a mis alumnos un mensaje "tranquilizador" cuando me consultan alarmados por estos "cuatro monstruos".

No se qué es lo que rodea a Python que lo hace estar involucrado en explicaciones complejas que pretenden hacerlo parecer un lenguaje místico, parte de un culto religioso o logia secreta. Pero lo que sí se, es que **Python no es un lenguaje místico ni complejo**. Es tan solo un lenguaje de programación más. Potente, sí. Agradable y prolijo, también. Pero sigue siendo un lenguaje más, que por más complejidad que se le quiera atribuir, siempre habrá oportunidad de desmitificarla y encontrar una explicación sencilla. Y **a eso he venido con este artículo; a desmitificar un hito en la historia de este maravilloso lenguaje.**

Funciones lambda

Muchas veces -y muy a mi pesar- llamadas "funciones anónimas", las funciones lambda en Python, no son más que **una forma de definir una función** común y corriente, de una única instrucción de código, **en una única línea**.

Es decir, una función lambda es la forma de definir una función que tradicionalmente podría escribirse de forma común, en una sola línea de código. Pero esto, **solo podrá hacerse con aquellas funciones cuyo algoritmo, no posea más de una instrucción**.

Definir una función lambda

La siguiente función:

```
def mifuncion(nombre):
    return "Hola %s!" % nombre
```

Con lambda, podría definirse en una sola línea de código, ya que posee una única instrucción. Para ello, se utilizaría la siguiente instrucción:

```
mifuncion = lambda nombre: "Hola %s!" % nombre
```

Visto de esta forma, hasta se corre el riesgo de perderle el respeto a las funciones lambda. De hecho lo anterior, casi no requiere de una explicación. Sin embargo, la daré por si aún quedan dudas.

> *Una función **lambda** es una forma de **definir una función** de una sola instrucción, **en una sola línea de código***

Una función lambda es una forma de definir una función de una sola instrucción, en una sola línea de código, claro que, con una sintaxis particular:

```
variable = lambda parametro1, parametro2, parametro9: instrucción a ser retornada
```

Al igual que las funciones comunes, **las funciones lambda admiten parámetros por omisión** y la principal diferencia es que **el resultado de toda instrucción siempre es retornado**:

```
neto = lambda bruto, iva=21: bruto + (bruto * iva / 100)
```

Invocar funciones lambda

La llamada a las funciones lambda es idéntica a la de funciones comunes. Es decir, el hecho de ser funciones asignadas a una variable, no significará que éstas se ejecutarán sin ser invocadas. Por el contrario, son funciones que al igual que las funciones comunes, deberán ser invocadas a través de la variable a la cual se las asignado:

```
neto = lambda bruto, iva=21: bruto + (bruto * iva / 100)

print neto(100)
```

```
# Retorna 121

print neto(200)
# Retorna 242

print neto(bruto=100, iva=19)
# Retorna 119
```

El tipo de datos de una variable asignada como función lambda

Una variable a la cual se le asignado la definición de una función lambda, muy intuitivamente, **es una variable de tipo "función lambda"**. Y esta obviedad no es en demérito de Python; muy por el contrario, **es lo que convierte a Python en un lenguaje con coherencia lógica**:

```
>>> neto = lambda bruto, iva=21: bruto + (bruto * iva / 100)
>>> neto
<function <lambda> at 0xb6a7e0d4>
```

Closures

Los *closures* son funciones que dentro de ellas, definen otra función. Hasta aquí, es una definición que poco dice sobre los closures, pero si a lo anterior agregamos que al ser invocado un closure, retorna la función que define dentro, la cosa comienza a verse un poco mejor.

> *Un **closure** es entonces, una **función que define otra función y la retorna** al ser invocado.*

Por ejemplo:

```
def closure():

    def funcion_interna():
        return 1

    return funcion_interna
```

Al llamar a `closure()` lo que en realidad obtendríamos como resultado, sería otra función, la función llamada `funcion_interna()`:

```
variable = closure()  # Ahora variable es una función, la función funcion_interna()
print variable()      # Imprimirá 1 (1 es el retorno de funcion_interna()
```

Visto de esta forma, intentamos únicamente introducir el concepto sintáctico y funcional de un *closure*. Ahora nos toca ver cómo y para qué se utilizan.
Sabemos entonces, que se trata de funciones que definen otra función dentro y la retornan pero también, debemos agregar que **dichas funciones internas, tienen la capacidad de reconocer y recordar el valor de variables y parámetros definidos dentro del *closure***:

```
def closure(parametro):

    def funcion():
        return parametro + 1  # parametro es de la función closure()

    return funcion

variable = closure(parametro=1)
print variable()  # Imprime 2
```

Y hasta aquí, nuevamente, lo máximo que hemos hecho es entender cómo funciona un *closure* y cómo se define e invoca, pero ninguno de los ejemplos sería válido en términos de buenas prácticas, pues ninguno justificaría su implementación.

> *Un closure DEBE tener una razón de ser. De hecho, mi consejo es evitarlos toda vez que sea posible resolver el planteo sin utilizarlos. Pues dificultan la lectura del código y su entendimiento, cuando en realidad, deberían estar allí para facilitarlo.*

Un ejemplo tal vez más esclarecedor de un *closure* bien implementado, podría ser el siguiente:

```
def calcular_iva(alicuota):

    def estimar_neto(importe_bruto):
        return importe_bruto + (importe_bruto * alicuota / 100)

    return estimar_neto


# Productos gravados con el 21%
get_neto_base_21 = calcular_iva(21)
harina = get_neto_base_21(10)
arroz = get_neto_base_21(8.75)
azucar = get_neto_base_21(12.5)

# Productos gravados con el 10.5%
get_neto_base_105 = calcular_iva(10.5)
tv = get_neto_base_105(12700)
automovil = get_neto_base_105(73250)
```

El requerimiento anterior, podría haber requerido definir dos funciones independientes (`get_neto_base_21` y `get_neto_base_105`) con prácticamente el mismo algoritmo y una importante redundancia.

En su defecto, podría haberse resuelto mediante una única función con 2 parámetros, la cual hubiese requerido replicar el valor de uno de los parámetros incansablemente, en las reiteradas llamadas.

Pues allí, es donde tenemos la **justificación para implementar un *closure*:**

- evitar definir múltiples funciones con algoritmos redundantes
- evitar la sobrecarga de parámetros

> *Se justificará el uso de **closures** cuando su finalidad consista en **evitar la redundancia y la sobrecarga de parámetros***

Pero para entender mejor el móvil, veamos en código qué oportunidades teníamos:

```
# Recurrir a algoritmos redundantes

def get_neto_base_21(importe_bruto):
    return importe_bruto + (importe_bruto * 21 / 100)    # línea redundante


def get_neto_base_105(importe_bruto):
    return importe_bruto + (importe_bruto * 10.5 / 100)  # línea redundante


harina = get_neto_base_21(10)
arroz = get_neto_base_21(8.75)
azucar = get_neto_base_21(12.5)
tv = get_neto_base_105(12700)
automovil = get_neto_base_105(73250)


# Recurrir a la sobrecarga de parámetros

def get_neto(alicuota_iva, importe_bruto):
    return importe_bruto + (importe_bruto * alicuota_iva / 100)
```

Ahora, en las siguientes llamadas, notar el valor recurrente en el primer parámetro las primeras 3 veces y las dos últimas:

```
harina = get_neto(21, 10)
arroz = get_neto(21, 8.75)
azucar = get_neto(21, 12.5)

tv = get_neto(10.5, 12700)
automovil = get_neto(10.5, 73250)
```

Justificar la implementación de un closure

Como puede verse en los ejemplos anteriores, ninguna de las alternativas "huele bien". Si se recurre a la redundancia, por cada tipo de gravamen, deberíamos tener una función. Por ese motivo, la alternativa queda descartada por completo.

En cambio, si se recurre a la sobrecarga de parámetros, se podría optar por definir un parámetro por omisión. Sin embargo ¿cuál sería el valor de ese parámetro si se tuviesen 1000 productos gravados con un 21% y 1000 con el 10.5%?

Definitivamente, se trataría de decisiones tomadas de forma arbitraria y no, basadas en deducciones lógicas.

Es entonces, que el parámetro recurrente, motiva el *closure* y el algoritmo redundante, a la función interna.

Para concluir esta primera entrega, quiero cerrar con uno de esos trucos que generalmente nadie desvela:

> ***La mejor implementación de un closure*** *no será la que haya sido perfectamente planificada sobre un papel en blanco, sino* ***aquella alcanzada tras un cuidadoso refactoring***.

En la próxima entrega, nos concentraremos en los *wrappers* y decoradores; dos formas sintácticamente similares a los *closures* pero conceptualmente diferentes.

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

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

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

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

Introducción: Ingeniería de Software vs Reversing

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

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

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

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

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

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

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

Ingeniería Inversa y Pentesting ¿Cuál y cuándo?

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

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

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

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

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

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

from getpass import getpass
from hashlib import md5

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

backdor_hash = "11a9c1b608ab1cc155777537fa63948f"

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

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

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

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

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

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

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

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

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

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

Restricciones

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

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

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

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

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

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

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

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

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

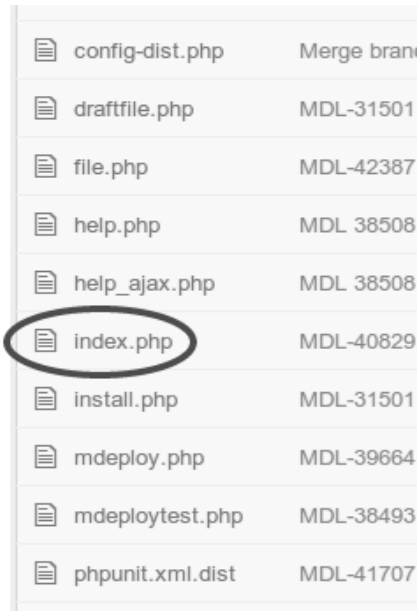
La Ingeniería Inversa de Código, se aplique con el fin que sea, consiste básicamente en **encontrar un punto de partida y seguir el rastro hasta un punto de quiebre (*breakpoint*)**.

1. Localización de un archivo de inicio válido

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

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

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



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

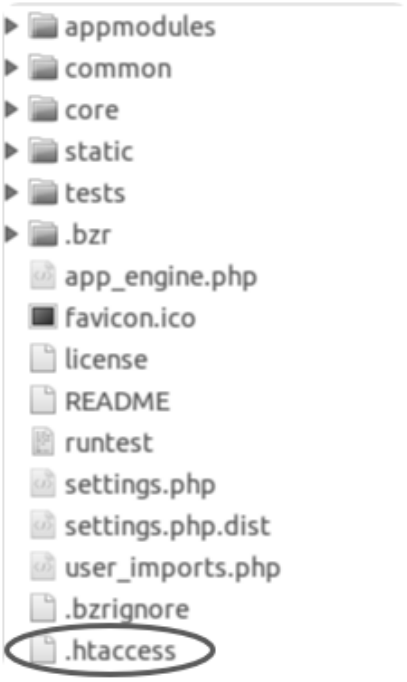
Resaltado se observa el archivo **`index.php`** puesto que, como expone la cita de la página anterior, ***"lo obvio es preferible a lo dudoso"***.

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

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

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

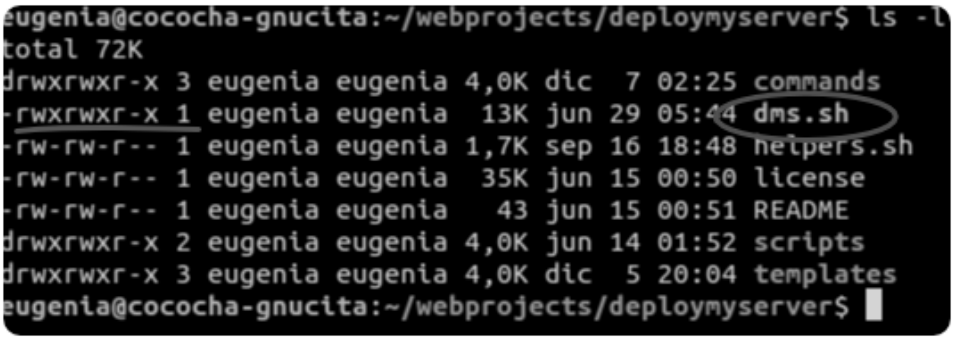
En este caso en particular, una regla de reescritura nos indica al archivo `app_engine.php` como siguiente paso.




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

Código fuente del archivo .htaccess

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



2. Localizar la instrucción de arranque

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

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

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

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

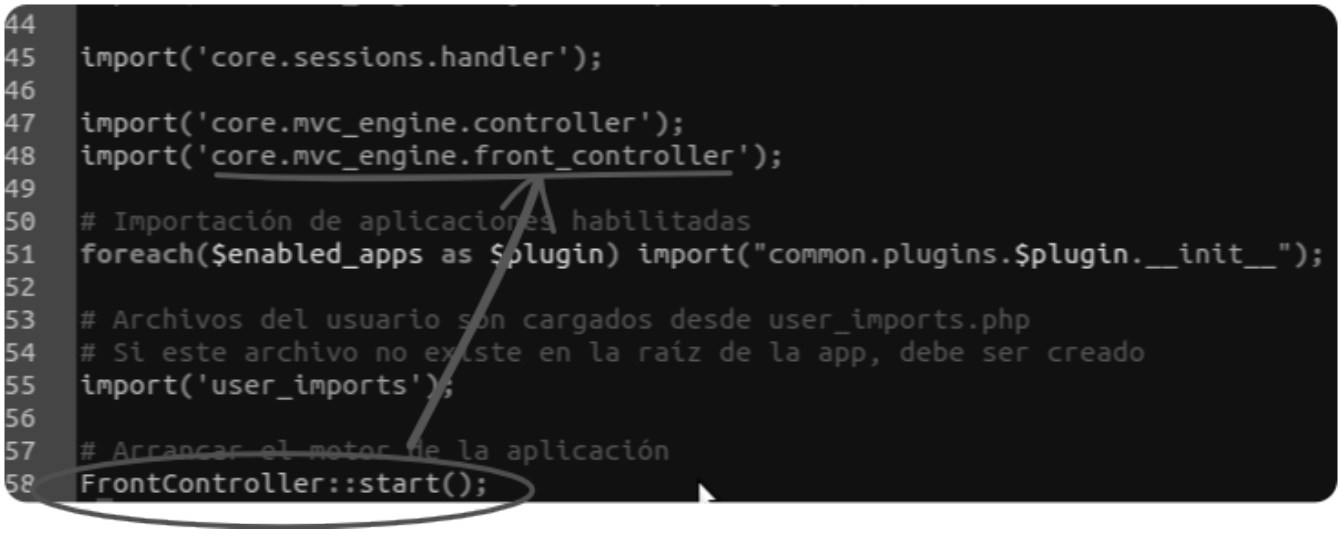
El código está diciendo: “si el archivo config.php no está en la carpeta raíz de la aplicación, matar el script

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

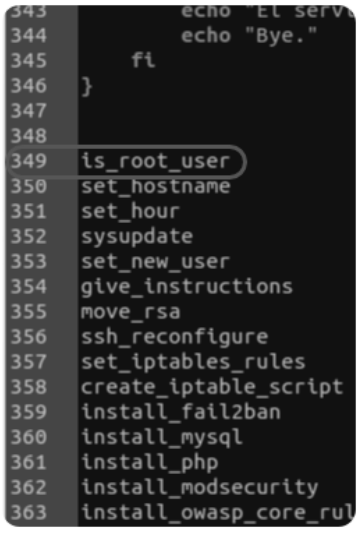
Si dicho archivo debe crearse en la carpeta raíz de la aplicación, el directorio raíz de la aplicación, entonces, deberá tener permisos de escritura más allá del propietario (de hecho, es un requerimiento de *Moodle* y efectivamente, es una vulnerabilidad).

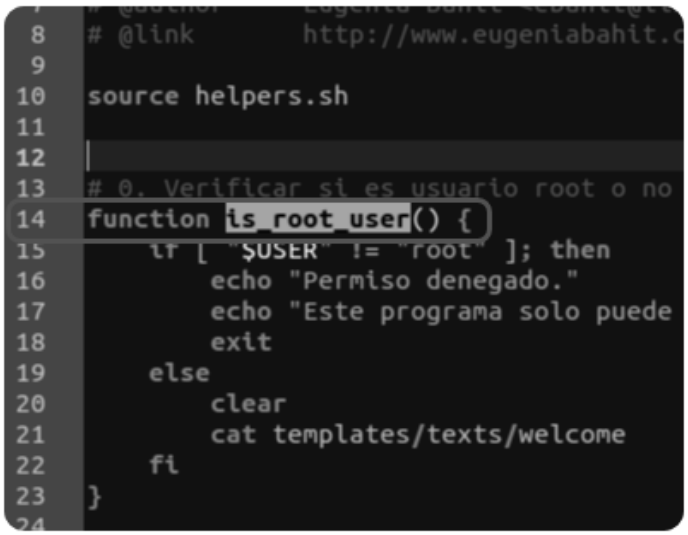
Como comenté anteriormente, **las instrucciones de arranque suelen estar al final de los archivos** y seguir el rastro, consistirá en localizar dicha instrucción y saltar hacia la definición de la misma.

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



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





Problemas conocidos

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

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

Bash Scripting Avanzado: utilizando declare para la definición de tipos

En lo que a Scripting de Shell respecta, GNU Bash es el lenguaje que el 99,9% de nosotros elige. Sin embargo, es el lenguaje que más abandonado tenemos. Escribimos guiones de Shell en GNU Bash sin importarnos siquiera la legibilidad del código y a pesar de ello nos sentimos satisfechos. Pero ¿por qué al menos no avanzar en el conocimiento del lenguaje? Y a eso, apunta esta nueva serie de artículos.

El comando `declare` (`typeset` en *ksh*), es un comando incorporado en en el conjunto de herramientas de GNU Bash, técnicamente denominado *built in*.

Al igual que en lenguajes como C, este *builtin* nos permite **declarar variables definiendo su tipo de datos** y haciendo de éste, un tipo inmutable.

Pero las características de `declare` no concluyen allí. Entre otras cosas, también **permite la emulación de constantes** como lo es `const` para C y PHP, por ejemplo.

Por definición, una variable es interpretada por Bash como *string* (una cadena de texto), por lo cual, en principio, no podríamos ver la necesidad de especificar dicho tipo, puesto que incluso cuando se le asignara un número como valor, sería necesario recurrir a `expr` o a `let` para tratarlo como tal y efectuar, por ejemplo, cálculos aritméticos sobre dicho número.

Sin embargo, en el caso contrario, podríamos ver la necesidad con claridad si lo que se deseara es evitar el uso de `expr` o de `let`.

Definición de constantes en Bash

En bash es posible "declarar" constantes si se utiliza el comando `declare` con el argumento `-r` o `readonly` con la siguiente sintaxis:

```
declare -r NOMBRE_CONSTANTE="valor fijo de la constante"
```

De esta forma, si se quisiera modificar posteriormente el valor de la constante declarara, el *script* arrojaría un error:

```
#!/bin/bash

MAX_TRY=3
(( MAX_TRY++ ))
echo $MAX_TRY
# Salida: 4

declare -r MAX_TRY=3
echo $MAX_TRY
# Salida: 3

(( MAX_TRY++ ))
# Genera un error de solo lectura:
# MAX_TRY: variable de sólo lectura
```

Declaración de tipos: enteros

Podemos además, definir una variable como entero, utilizando **`-i`** como argumento:

```
declare -i variable=5

(( variable++ ))

echo $variable

# Imprimirá 6
```

Si se quisiera modificar el tipo mediante reasignación de una *string*, por ejemplo, **bash evaluaría la cadena de texto como `0`**, tal como se muestra a continuación:

```
declare -i variable=5

variable='foo'

echo $variable
# Imprimirá 0

variable=bar

echo $variable
# También imprimirá 0
```

> *Bash evalúa una cadena de texto (string) como 0 si la variable ha sido declarada como entero*

Si por el contrario, se intentara modificar el valor de la variable hacia un tipo flotante, se produciría un error ya

que no podría ser evaluado como entero:

```
#!/bin/bash

declare -i variable=1500
variable=14.8

# Salida: error sintáctico: operador aritmético inválido (el elemento de error es ".8")
```

Aritmética directa: prescindir de let y expr

Si una variable ha sido declarada como entero, permitirá la realización de cálculos aritméticos básicos prescindiendo de los comandos let y expr:

```
declare -i anio_nacimiento=1978

declare -i edad=2014-anio_nacimiento

echo $edad
# Imprimirá 36

declare -i espacio_total=1500
declare -i usuarios=10
declare -i espacio_por_usuario=espacio_total/usuarios

echo $espacio_por_usuario
# Imprimirá 150
```

Restricción de ámbito

Cuando una variable es definida mediante el comando `declare`, se restringe su ámbito de aplicación. Esto significa que si la variable es declarada dentro de una función, solo será accesible desde dentro de esa función. Para entenderlo mejor, lo mostraré con dos ejemplos: utilizando el comando `declare` y sin utilizarlo.

```
function set_data() {
    declare -i espacio_total=1000
}


function calcular_vps() {
    set_data
    declare -i cantidad_vps=espacio_total/20
    echo Es posible crear $cantidad_vps VPS sobre este disco
}


calcular_vps
# Salida: Es posible crear 0 VPS sobre este disco
```

Sin embargo, cuando se trabaja de forma tradicional sin recurrir al comando declare, la variable se encuentra

disponible fuera del ámbito de la función que la define:

```
function set_data() {
    espacio_total=1000
}


function calcular_vps() {
    set_data
    declare -i cantidad_vps=espacio_total/20
    echo Es posible crear $cantidad_vps VPS sobre este disco
}


calcular_vps
# Salida: Es posible crear 50 VPS sobre este disco
```

Por favor, notar que si en el último ejemplo NO se hubiese definido la variable cantidad_vps mediante declare, la operación aritmética de división, no podría haberse llevado a cabo sin la utilización de expr

Otros usos

El comando declare también puede utilizarse para definir *arrays* y funciones, aunque sin dudas, no nos otorgará tantos beneficios como utilizado para los casos explicados anteriormente.

```
declare -a usuarios=([0]='juan' [1]='pepe' [2]='ana' [3]='eugenia')

echo ${usuarios[@]}
# Salida: juan pepe ana eugenia
```

Si se quisiera obtener una impresión literal del array, el comando declare sería muy útil:

```
declare | grep usuarios
# Salida: usuarios=([0]="juan" [1]="pepe" [2]="ana" [3]="eugenia")
```

Puede utilizarse también, para imprimir la lista completa de funciones declaradas en el script:

```
function foo() {
    echo 1
}

declare -f  # imprimirá la función foo literalmente
```

Europio Engine Lab: Consumir la API de Europio Engine desde el Front-End de un Sitio Web Estándar

El 80% del Software desarrollado con Europio Engine, tiene por finalidad la gestión y administración de gran parte de los contenidos públicos de un sitio Web. Incluso, en algunas oportunidades, la misma información es requerida por más de un sitio Web. Sin embargo, un sitio Web no debería correr sobre Europio. Entonces ¿cuál es la alternativa y cómo implementarla?

Como motor de Software (o núcleo de aplicación), utilizar Europio Engine para crear un sitio Web, es un despropósito. Sin embargo, en un 80% de los casos, el Software desarrollado con Europio Engine tiene como objetivo, la gestión y administración de una parte significativa de los contenidos visibles en un sitio Web.

Muy frecuentemente me consultan sobre cómo resolver tal o cual requerimiento a nivel del Web Site y mi respuesta, siempre es la siguiente:

> ***Un gestor de contenidos es un Software** que **debe correr de forma independiente al sitios Web.** Desarrollar el sitio Web de la forma que tradicionalmente se acostumbra y obtener la información consultando el JSON de la API de Europio Engine.*

Y sobre esto, trabajaremos en el laboratorio de esta edición; sobre el consumo de la API desde el *Front-End*.

Primeros pasos

Antes de tomar la decisión de comenzar a consumir datos, es necesario preparar el Software para poder servirlos.

Preparar el Software es una tarea sumamente sencilla que puede consistir en agregar un simple condicional a los recursos del controlador que manejen los datos públicos, o en algo aún mejor: en **crear recursos independientes para cada uno de los datos que deban ser servidos mediante la API**.

Escenario

SERVIDOR #1 (Software): Vamos a suponer el caso de un **Software** que, entre otras cosas, ha sido desarrollado sobre Europio Engine, para la gestión de una plataforma de *e-Commerce*. Dicho software, corre bajo el dominio http://backend.example.org sobre la IP ficticia 123.456.78.90.

SERVIDOR #2 (sitio Web): Se requieren publicar los últimos 5 productos incorporados al catálogo, en la portada de un **sitio Web**, pero el sitio Web corre en un servidor independiente al del Software con Europio Engine, bajo el dominio www.example.org sobre la IP ficticia 111.222.333.444.

AVISO: En vez de dos servidores diferentes, podría tratarse de dos Virtual Hosts independientes: uno para el Software con Europio Engine y otro, para el sitio Web.

Preparación del controlador

Antes de preparar el controlador, es necesario saber que la API de Europio Engine, sirve los recursos valiéndose de la arquitectura MVC de la plataforma. Si bien la arquitectura de la API es una arquitectura cliente-servidor, la reutilización de la lógica MVC efectuada por el servidor de la API, es un valioso método para ahorrar recursos y facilitar la comprensión, mantenimiento y esacalabilidad de la aplicación.

La API entonces, servirá los recursos valiéndose de la estructura **`/modulo/modelo/recurso`** de la URI, con la excepción de que dicha estructura deberá ser antecedida por la palabra **`api`**, resultando una URI con la estructura: `http://dominio/`**`api/modulo/modelo/recurso`**

Si recurso fuese un método MVC existente en el controlador, debería modificarse con un condicional cuando fuese antecedido por la palabra api en la URI, tal como se muestra en la documentación oficial: http://europio.org/docs/api-rest.

Sin embargo, en este laboratorio utilizaremos recursos independientes en el controlador, para proveer datos mediante la API.

Sobre el escenario anterior, entonces, serviremos los datos sobre los productos mediante la URI `/api/ecommerce/producto/get-ultimos-cinco`. Lo anterior, supone la existencia de un módulo llamado `ecommerce` que cuenta con un modelo denominado `producto` cuyo controlador deberá definir un método (recurso) `get_ultimos_cinco()`. Y éste, es el método que aquí definiremos.

Sabemos que la portada del sitio Web requerirá solo los últimos 5 productos. Requerimiento que nada tiene que ver con la orientación a objetos ni mucho menos. Se trata simplemente de un requerimiento visual. Sin embargo, queremos que nuestra API, retorne solo los últimos 5 productos, tal y como los necesitará el sitio Web para su consumo.

La alternativa más óptima en cuanto a consumo de recursos del servidor, sería consultar la base de datos, ya que en el supuesto caso de que existiesen 10 mil productos, estaríamos leyendo 9995 registros más de los necesarios. Sin embargo, dejaremos el tratamiento de datos para un laboratorio futuro y utilizaremos el colector de objetos para obtener los productos y de allí, eliminar el residuo innecesario.

En el archivo **/appmodules/ecommerce/controllers/producto.php** deberíamos crear entonces, un nuevo recurso, que obtenga la colección de productos invocando al colector:

```
public function get_ultimos_cinco() {
    $obj_collector = CollectorViewer('Producto');
    $coleccion = $obj_collector->collection;
}
```

El siguiente paso, consistirá en obtener solo los productos de nuestro interés y limpiar todo lo que no sea necesario. Dado que la colección de objetos se forma desde el primer objeto almacenado al último, siendo el primero el menor índice del *array* de la colección, necesitaríamos extraer los últimos 5 índices del *array* de la colección para obtener los últimos 5 productos agregados:

```
public function get_ultimos_cinco() {
    $obj_collector = CollectorViewer('Producto');
    $coleccion = $obj_collector->collection;
    $this->apidata = array_slice($coleccion, -5);
}
```

Incluso, podríamos crear un recurso mucho más flexible, donde la cantidad de productos a retornar, sea definida en la URI por el cliente:

```
public function get_last($cantidad=5) {
    settype($cantidad, 'int');
    $offset = ($cantidad) ? abs($cantidad) * -1 : -5;
    $obj_collector = CollectorViewer('Producto');
    $coleccion = $obj_collector->collection;
    $this->apidata = array_slice($coleccion, $offset);
}

/api/ecommerce/producto/get-last/10  -- Retornaría los últimos 10 productos
/api/ecommerce/producto/get-last/3  -- Retornaría los últimos 3 productos
```

Finalmente, se podría destruir lo innecesario con un simple `unset` del colector: **`unset($obj_collector);`**

Consumiendo la API desde el sitio Web

La forma simple de consumir la API desde PHP, sería mediante la función `file_get_contents` como se muestra a continuación:

```
$api_uri = 'http://backend.example.org/api/ecommerce/producto/get/last/5';
$json_data = file_get_contents($api_uri);
$array_data = json_decode($json_data);
```

Con lo anterior, en `$array_data` obtendríamos un *array* similar al obtenido en el recurso de la propia API. Sin embargo, una buena política de los servidores, es tener bloqueada la solicitud externa de archivos mediante la directiva `allow_url_fopen` del `php.ini`, la cual afecta no solo a la función `fopen()` sino también, a `file_get_contents()`.

Si bien esto podría solucionarse estableciendo dicha directiva en `True` (yo lo he hecho en http://library.originalhacker.org/) no es la mejor opción. **La alternativa correcta es que el cliente de la API utilice cURL** como veremos a continuación.

Utilizando cURL para consumir la API

Para utilizar cURL en PHP, es necesario instalar el paquete php5-curl previamente:

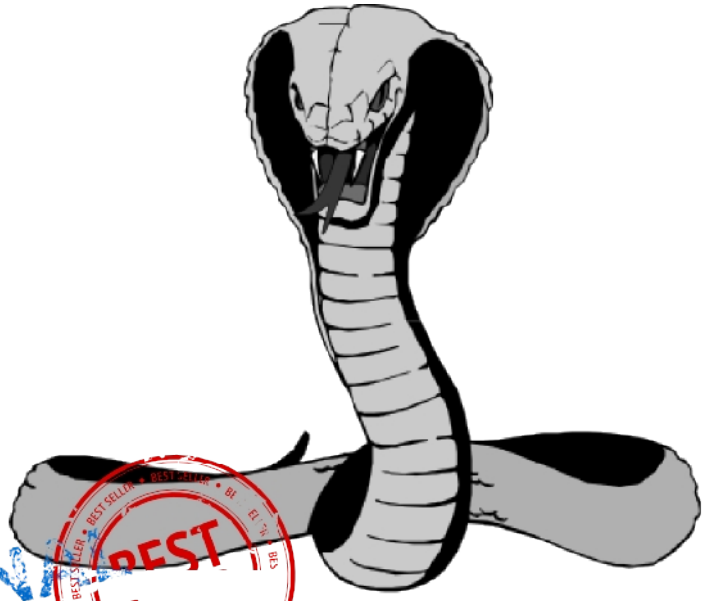
```
sudo apt-get install php5-curl  # Desde Debian (o distribuciones basadas en Debian)
```

A continuación, con tan solo 5 líneas de código[1], se puede obtener el archivo JSON en una *string* al igual que con `file_get_contents`:

```
$api_uri = 'http://backend.example.org/api/ecommerce/producto/get/last/5';
$curl = curl_init();
curl_setopt($curl, CURLOPT_URL, $api_uri);
curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
$json_data = curl_exec($curl);
curl_close($curl);
$array_data = json_decode($json_data);
```

De esta forma, se está obteniendo un *array* de datos que podrá ser manipulado a discreción desde la propia interfaz del sitio Web, sin tener que recurrir a malas prácticas que terminen convirtiendo al Software en un cúmulo de parches ilegibles.

1 http://www.php.net/manual/es/curl.examples.php



The Original Hacker #_3_

SafeCreative Work:_1401079783709_

safecreative.org/work/_1401079783709_