

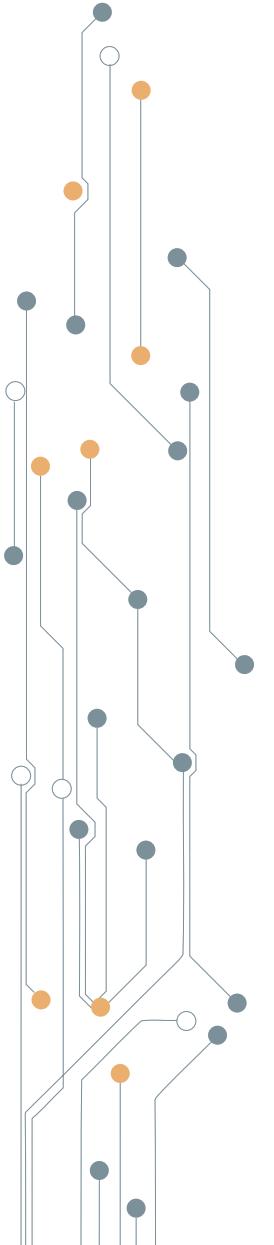


Integración con  
herramientas externas

Telefónica

EDUCACIÓN DIGITAL

# Índice



- 1 | Introducción
- 2 | Nmap
- 3 | Nessus
- 4 | Metasploit Framework
- 5 | Latch

3  
4  
7  
9  
12

# 1. Introducción

En muchas ocasiones, en la labor del día a día de un Pentester le surge la necesidad de realizar una prueba muy concreta y automatizarla mediante un script. La primera idea siempre es implementar la prueba entera manualmente, pero en muchas ocasiones esto significa “reinventar la rueda” ya que existen en el mercado un montón de herramientas muy potentes donde ya está implementada esta funcionalidad y que permiten la integración de su operativa desde programas externos en varios lenguajes de programación.



## 2. Nmap

Nmap es una herramienta de código abierto diseñada para realizar escaneos de puertos y de máquinas utilizando múltiples algoritmos de búsqueda. Actualmente esta herramienta es multiplataforma, pero en sus orígenes solo estaba disponible en entornos Linux.

Esta herramienta tiene gran popularidad, se usa habitualmente para evaluar la seguridad de sistemas informáticos, así como para descubrir servicios y servidores en una red determinada. Las principales características son:

- Descubrimiento de servidores.
- Identifica puertos abiertos.
- Determina qué servicios se están ejecutando.
- Determinar qué sistema operativo y versión utiliza dicho servidor.

Además, Nmap cuenta con la posibilidad de incorporar scripts a las búsquedas, ampliando las capacidades de búsqueda añadiendo sistemas de detección avanzados, detección de vulnerabilidades, etc.

Para poder utilizar la herramienta Nmap desde los scripts escritos en Python existen 2 alternativas:

- Invocar directamente al ejecutable de Nmap con un subprocess de Python.
- Utilizando la librería Python-nmap.

La mejor forma para interactuar con Nmap es utilizando la librería ya que abstrae toda la integración con la herramienta y el tratamiento de los resultados.

El sitio web oficial de esta librería donde se pueden encontrar la documentación y los **archivos fuente** es: <http://xael.org/pages/python-nmap-en.html>

Pero se recomienda que para su instalación se utilice la herramienta PIP:

```
pip install python-nmap
```

Esta librería permite ejecutar la búsqueda tanto de forma síncrona como de forma asíncrona. Por lo que se puede dejar en segundo plano ejecutando un escaneo mientras en el script se realizan otras labores. Para ello solo es necesario pasarle una función de callback que se ejecutará cada vez que se devuelva un resultado.

Para su invocación de forma síncrona se utilizará el método PortScanner, el cual lanzará un escaneo de Nmap y bloqueará la ejecución del script hasta que haya acabado.

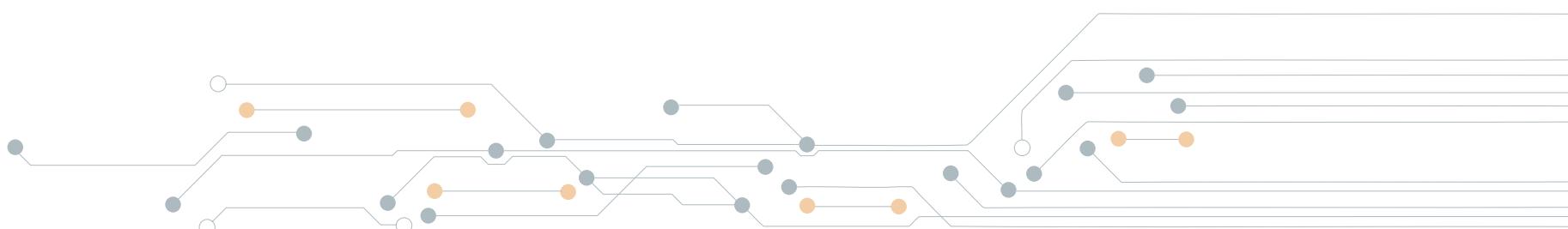
En el siguiente ejemplo se va a mostrar el funcionamiento de esta librería, su importación e instanciación, y su configuración de forma simple para realizar un escaneo de puertos sobre un host objetivo:

```
>>> import nmap
>>> nm = nmap.PortScanner()
>>> r = nm.scan('192.168.1.134', '21,22,80,110,135,139,455,808
0')
>>> nm.all_hosts()
['192.168.1.134']
>>> nm.scaninfo()
{'tcp': {'services': '21-22,80,110,135,139,455,8080', 'method':
'syn'}}
>>> nm.command_line()
'nmap -oX - -p 21,22,80,110,135,139,455,8080 -sV
192.168.1.134'
```

Como se puede apreciar, si no se configura el tipo de escaneo que se desea, por defecto utiliza el tipo "Stealth" además de incorporar a la petición de búsqueda la opción "-sV" que determina la versión de los servicios.

El resultado de este escaneo se puede obtener en formato CSV el cual puede ser útil para exportarlo y visionarlo por separado:

```
host;hostname;hostname_type;protocol;port;name;state;pro-
duct;extrainfo;reason;version;conf;cpe
192.168.1.134;;;tcp;21;ftp;closed;;;reset;;3;
192.168.1.134;;;tcp;22;ssh;closed;;;reset;;3;
192.168.1.134;;;tcp;80;http;closed;;;reset;;3;
192.168.1.134;;;tcp;110;pop3;closed;;;reset;;3;
192.168.1.134;;;tcp;135;msrpc;open;Microsoft Windows RP-
C;;syn-ack;;10;cpe:/o:microsoft:windows
192.168.1.134;;;tcp;139;netbios-ssn;vopen;Microsoft Windows
netbios-ssn;;syn-ack;;10;cpe:/o:microsoft:windows
192.168.1.134;;;tcp;455;creativepartnr;closed;;;reset;;3;
192.168.1.134;;;tcp;8080;http-proxy;closed;;;reset;;3;
```



Pero para poder trabajar con los datos obtenidos desde el propio script, es recomendable utilizar la estructura que nos devuelve la librería pudiendo indexar cada dato específico:

```
>>> nm['192.168.1.134']
{'tcp': {80: {'conf': '3', 'reason': 'reset', 'extrainfo': '', 'name': 'http', 'version': '', 'cpe': '', 'state': 'closed', 'product': ''}, 8080: {'conf': '3', 'reason': 'reset', 'extrainfo': '', 'name': 'http-proxy', 'version': '', 'cpe': '', 'state': 'closed', 'product': ''}, 21: {'conf': '3', 'reason': 'reset', 'extrainfo': '', 'name': 'ftp', 'version': '', 'cpe': '', 'state': 'closed', 'product': ''}, 22: {'conf': '3', 'reason': 'reset', 'extrainfo': '', 'name': 'ssh', 'version': '', 'cpe': '', 'state': 'closed', 'product': ''}, 135: {'conf': '10', 'reason': 'syn-ack', 'extrainfo': '', 'name': 'msrpc', 'version': '', 'cpe': 'cpe:/o:microsoft:windows', 'state': 'open', 'product': 'Microsoft Windows RPC'}, 139: {'conf': '10', 'reason': 'syn-ack', 'extrainfo': '', 'name': 'netbios-ssn', 'version': '', 'cpe': 'cpe:/o:microsoft:windows', 'state': 'open', 'product': 'Microsoft Windows netbios-ssn'}, 110: {'conf': '3', 'reason': 'reset', 'extrainfo': '', 'name': 'pop3', 'version': '', 'cpe': '', 'state': 'closed', 'product': ''}, 455: {'conf': '3', 'reason': 'reset', 'extrainfo': '', 'name': 'creativepartnr', 'version': '', 'cpe': '', 'state': 'closed', 'product': ''}, 'status': {'reason': 'reset', 'state': 'up'}, 'hostnames': [{'type': '', 'name': ''}], 'addresses': {'ipv4': '192.168.1.134'}, 'vendor': {}}
>>> nm['192.168.1.134']['tcp'][135]
{'conf': '10', 'reason': 'syn-ack', 'extrainfo': '', 'name': 'msrpc', 'version': '', 'cpe': 'cpe:/o:microsoft:windows', 'state': 'open', 'product': 'Microsoft Windows RPC'}
>>> nm['192.168.1.134']['tcp'][135]['state']
'open'
```

Si se quiere realizar un escaneo que no sea el estándar por defecto, es posible pasarle argumentos al método Scan de la librería:

```
>>> r = nm.scan('192.168.1.134', arguments='--sSV -A -n -T5 -p 21,22,80,110,135,139,455,8080')
>>> nm.command_line()
'nmap -oX - --sSV -A -n -T5 -p 21,22,80,110,135,139,455,8080
192.168.1.134'
```

Como se comentaba anteriormente, la librería también dispone de una forma de ejecución asíncrona. Para ello solo es necesario crear un método de CallBack y pasárselo al método scan() para que lo invoque cada vez que tenga un resultado:

```
import nmap
nma = nmap.PortScannerAsync()
def callback_method(host,result):
    print("Host: ", host)
    print("Result:", result)
r = nma.scan('192.168.1.134', arguments='--sSV -A -n -T5 -p 21,22,80,110,135,139,455,8080', callback=callback_method)
while nma.still_scanning():
    nma.wait(2)
```

Cuando se termine el escaneo de un puerto, se invocará al método callback\_method() ejecutando las acciones que se hayan configurado.

## 3. Nessus

Nessus es un escáner de vulnerabilidades muy potente el cual las categoriza por sus diferentes niveles de criticidad. Esta herramienta cuenta con un daemon (nessusd) que se queda ejecutando en el sistema operativo y es el encargado de realizar los escaneos. Aparte, esta herramienta cuenta con un cliente web donde está el cuadro de mando (generar un nuevo escáner, ver el estado, ver los informes, etc).

Esta herramienta no cuenta con un API al uso para que se puedan integrar scripts y programas, pero si cuenta con un API Rest para la comunicación entre el daemon y el cliente web.

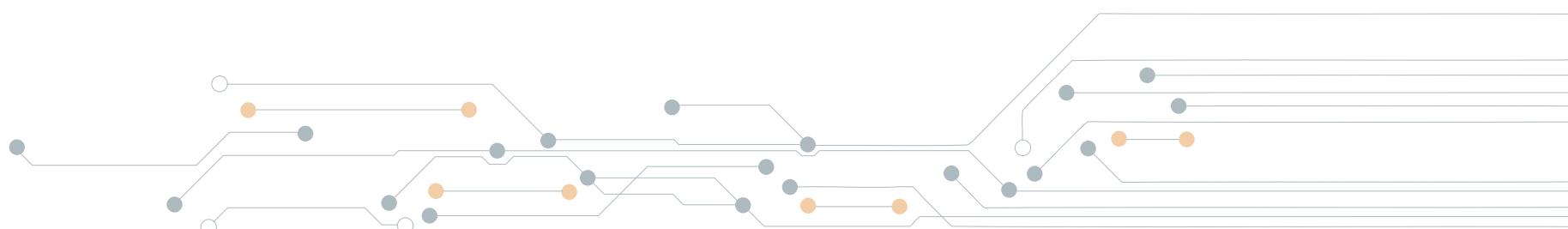
En base a esta API Rest, ha nacido una librería para poder integrar los scripts de Python con esta herramienta. El nombre del proyecto es 'pynessus-rest' y puede encontrarse en su **Github oficial**: <https://github.com/Adastra-thw/pynessus-rest>

Esta librería se encarga de integrar todas las funcionalidades disponibles en la herramienta de Nessus, y además de tratar los resultados devueltos por esta (en formato JSON e XML) a una estructura de objetos Python más amigable.

A continuación se muestra como instanciar la librería desde un script en Python:

```
from pynessus.rest.client.NessusClient import NessusClient
client = NessusClient('127.0.0.1', '8834')
client.login('ness', 'ness')
```

A partir de este momento, se puede interactuar con todas las funcionalidades de Nessus. Por ejemplo, se puede listar todas las políticas del servidor, descargar el contenido de una política en base a su identificador, subir o bajar ficheros con las nuevas políticas (estando en formato Nessus-XML):



```
client.policyList()
policy = client.policyDownload(1)
client.policyFileUpload("New.nessus", policy)
client.policyFilePolicyImport("New.nessus")
```

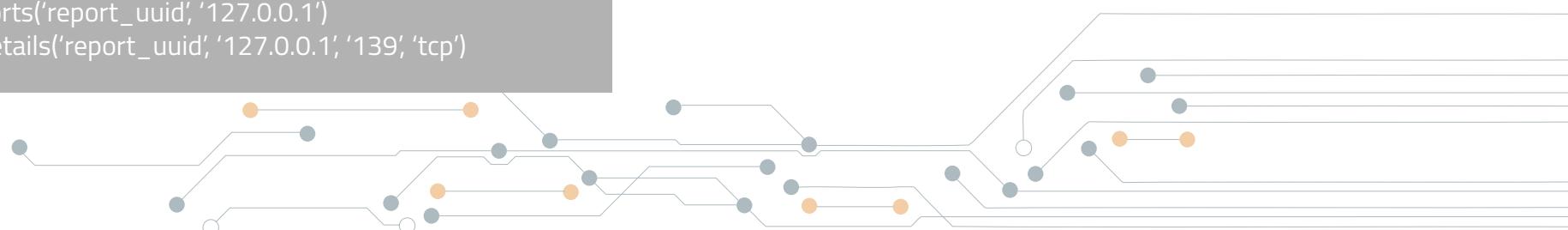
También se puede administrar las plantillas de escaneo, tanto crearlas, editarlas como borrarlas. Aparte, en base a una plantilla de escaneo se puede lanzar el mismo de forma programática:

```
client.scanTemplateNew('1', '127.0.0.1', 'NewTemplate')
client.scanTemplateEdit('templateID', 'Edited', '1', '127.0.0.1')
client.scanTemplateDelete('templateID')

client.scanTemplateLaunch('templateID')
```

Finalmente, desde la librería se puede interactuar con los reportes generados por Nessus:

```
client.reportList()
client.reportDelete('report_uuid')
client.reportHosts('report_uuid')
client.reportPorts('report_uuid', '127.0.0.1')
client.reportDetails('report_uuid', '127.0.0.1', '139', 'tcp')
```



## 4. Metasploit Framework

Este Framework es una de las herramientas más famosas y utilizadas para la explotación de vulnerabilidades y generación de Shellcodes. Es utilizado por los Pentester para agilizar los procesos de los test de intrusión, facilitando la labor de ejecución de exploits contra una máquina remota.

Este proyecto escrito en Ruby es Open Source, y fue diseñado para que fuese muy fácil la integración de nuevos módulos, exploits, etc al Framework, por lo cual es una herramienta muy potente y configurable.

Para poder integrarse a este Framework desde scripts desarrollados en Python existe una librería denominada `pyhton-msfrpc` que facilita esta labor. Se puede encontrar en su **Github oficial**: <https://github.com/SpiderLabs/msfrpc>

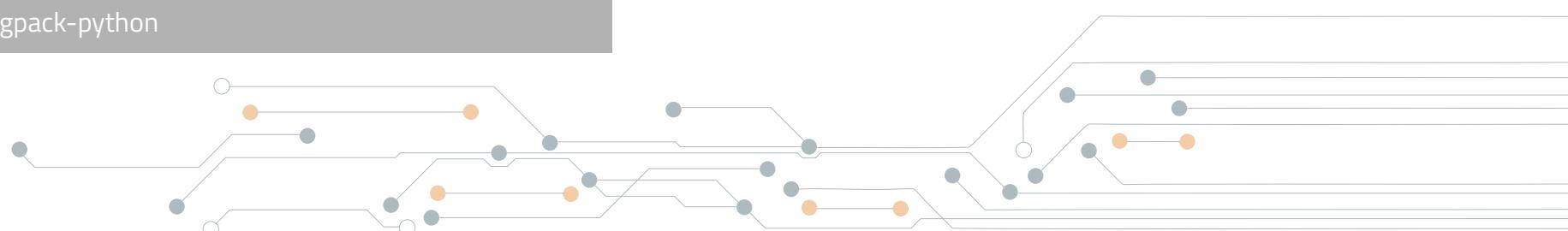
Aparte de esta librería, es necesario descargar e instalar un plugin para Python para realizar comunicaciones mediante el protocolo MessagePack, ya que Metasploit Framework permite la integración de Scripts utilizando únicamente este protocolo. Este plugin se llama `msgpack` y se puede instalar fácilmente utilizando PIP:

```
#pip install msgpack-python
```

Una vez instalado el plugin se puede continuar con la descarga e instalación de la librería `python-msfrpc`. Para ello habrá que descargar la rama Master del Github y ejecutar:

```
#python setup.py install
```

Nota: Esta librería aún no está portada a Python 3.x, por lo que será necesario realizar estos ejemplos utilizando las versiones anteriores.



El siguiente paso es crear una instancia en Metasploit para poder recibir peticiones mediante el plugin msgrpc. Para ello podemos realizarlo de dos formas diferentes

```
#msfconsole
msf > load msgrpc User=msfpython Pass=123abc.
[*] MSGRPC Service: 127.0.0.1:55552
[*] MSGRPC Username: msfpython
[*] MSGRPC Password: 123abc.
[*] Successfully loaded plugin: msgrpc

#msfrpcd -U msfpython -P 123abc. -p 8000 -u /msfrpc -n -f
[*] MSGRPC starting on 0.0.0.0:8000 (SSL):Msg...
[*] URI: /msfrpc
[*] MSGRPC ready at 2016-06-04 04:23:49 +0200.
```

Una vez está el servicio en ejecución, se podrán realizar peticiones desde un script de Python utilizando la librería msfrpc:

```
>>> import msfrpc
>>> import msgpack
>>> client=msfrpc.Msfrpc({'uri':'/msfrpc', 'port': '8000', 'host':
'0.0.0.0', 'ssl': True})
>>> client.login('msfpython', '123abc')
True
```

En el siguiente ejemplo se mostrarán las funcionalidades básicas que se pueden realizar contra Metasploit Framework. Todas las llamadas se realizarán mediante el método *call()*, el cual permite especificar la función en Metasploit a ejecutar y sus consiguientes parámetros:

```
print(client.call('core.version'))
print(client.call('core.thread_list', []))
print(client.call('core.setg', ['RHOST', '192.168.1.22']))
print(client.call('job.list', []))
print(client.call('module.exploits', []))
print(client.call('module.auxiliary', []))
print(client.call('module.post', []))
print(client.call('module.payload', []))
print(client.call('module.encoders', []))
print(client.call('module.nops', []))
print(client.call('module.info', ['exploit', 'unix/ftp/vsftpd_234_'
backdoor']))
```

Por otro lado, también es posible interactuar mediante la librería como si se ejecutases los comandos directamente en msfconsole. Para ello es necesario crear una instancia de la consola mediante 'console.create' y utilizar el identificador devuelto para realizar el resto de llamadas:

```
console = client.call('console.create')
command = """
    use auxiliary/scanner/ftp/ftp_version
    set THOST 10.0.2.6
    exploit
"""

client.call('console.write', [console['id'], command])
while True:
    data = client.call('console.read', [console['id']])
    if len(data['data']) > 1:
        print(data['data'])
    if data['busy'] == True:
        time.sleep(1)
        continue
    break
```

## 5. Latch

Latch es una herramienta que tiene como objetivo el proteger la información y las identidades digitales de una persona. Es una capa extra de seguridad que se añade a todos los servicios que el usuario utiliza para protegerlos ante el uso malintencionado de un tercero.

Como su nombre indica, Latch es un 'pestillo' digital para restringir el acceso a los servicios con la cuenta del usuario cuando este no va a hacer uso de los mismos. Visto de otra forma, desactiva los servicios para que no puedan ser usados con la cuenta del usuario hasta que este lo requiera. Su funcionamiento es simple y es muy similar a la autenticación en dos pasos. Para poder iniciar sesión, se ha tenido que desbloquear el cerrojo de este servicio desde la aplicación de los SmartPhones.

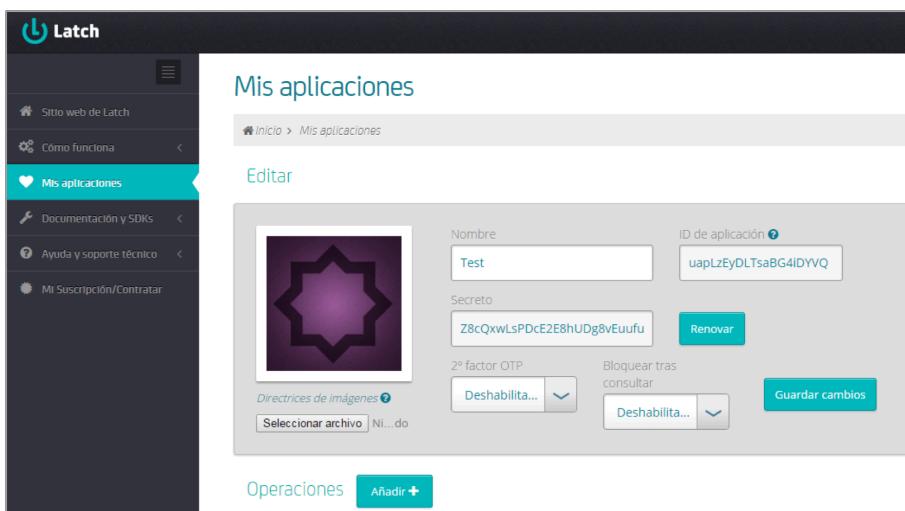
Estos servicios pueden ser cualquier cosa: páginas web, cuenta de bancos, redes sociales, login de servidores, servidores SSH, servidores VPN, hasta el cerrojo de una casa.

La potencia de Latch es que cuenta con librerías para integrarlo en cualquier lenguaje de programación y plataforma, por lo que se puede añadir a cualquier desarrollo que se esté realizando.



Para poder utilizar Latch desde un script de Python, es necesario abrir una cuenta en la página de **Latch para desarrolladores** (<https://latch.elevenpaths.com/www/>) y dar de alta una nueva aplicación.

Una vez autenticado, desde el área de **Mis Aplicaciones** (<https://latch.elevenpaths.com/www/developers/editapplication>) hay que seleccionar 'Añadir una nueva aplicación'.

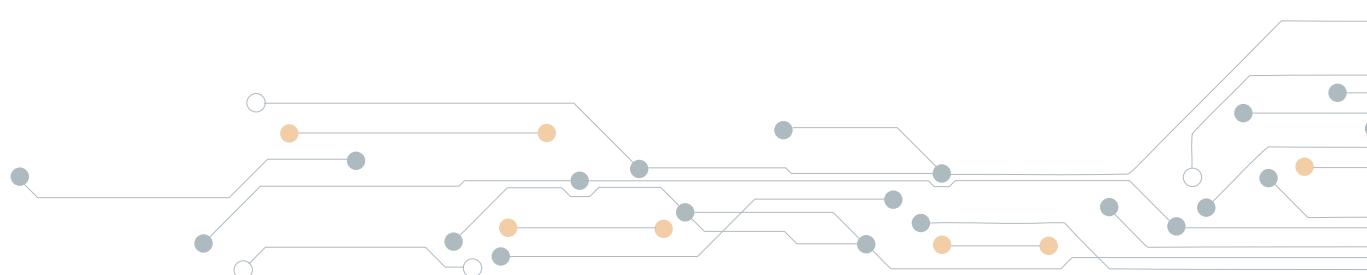


Para poder utilizarlo desde un script, es necesario tomar nota del ID de la aplicación y del secreto, ya que se usarán desde el script para autenticar a la aplicación.

Una vez se tiene creada la aplicación, se tendrá que descargar el SDK de Latch para Python desde el **Github de ElevenPaths** (<https://github.com/ElevenPaths/latch-sdk-python>). Habrá que colocar los ficheros Python descargados junto con los ficheros del script que se está desarrollando. En este momento ya se podrá importar la librería de Latch.

El primer ejemplo se centrará en el proceso de pareado. Un usuario generará desde su dispositivo móvil un código temporal el cual se asociará con la aplicación mediante este script

```
>>> import latch
>>> AAP_ID="uapLzEyDLTsaBG4iDYVQ"
>>> SECRET_TOKEN="Z8cQxwLsPDcE2E8hUDg8vEuufutzrzfh-N3YKdTxV"
>>> latch = latch.Latch(AAP_ID, SECRET_TOKEN)
>>> PAIR_TOKEN="q9gTw2"
>>> r = latch.pair(PAIR_TOKEN)
>>> r.get_data()
{'accountId': 'XXXXXX'}
```



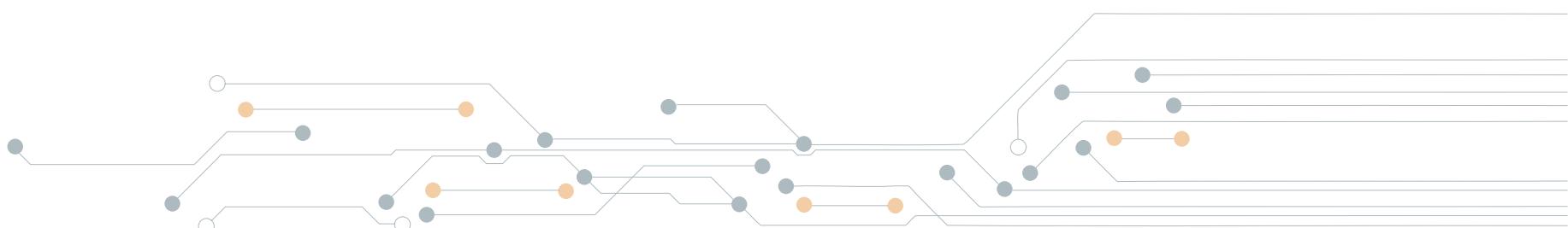
Una vez ejecutado el script se devuelve el accountID que es el identificador único para este usuario para comprobar si su Latch está activo o no. Esto se puede comprobar con el método *status()* pasándole como parámetro el accountID:

```
>>> r2 = latch.status("XXXX")
>>> r2.get_data()
{'operations': {'uapLzEyDLTsaBG4iDYVQ': {'status': 'on'}}}
>>> r2 = latch.status("XXXX")
>>> r2.get_data()
{'operations': {'uapLzEyDLTsaBG4iDYVQ': {'status': 'off'}}}
```

Entre la primera y la segunda ejecución, se ha cerrado el cerrojo desde la aplicación móvil. Por ese motivo en la 2º petición aparece con el status OFF. Cabe destacar, que al realizar la segunda petición se ha notificado a la aplicación móvil del intento de acceso a la aplicación.

Finalmente, también se puede desparear a un cliente desde el propio script:

```
>>> r3 = latch.unpair("XXXXX")
```



*Telefónica* **EDUCACIÓN DIGITAL**