Introduction

Web services are components integrated into an application that allows storage and availability of data. As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks.

As an example, look at the Graph API used by Facebook which is available to developers. The Graph API allows other applications to read and write to the Facebook social graph. By using this web service, a developer can integrate his application with a Facebook web service. This functionality allows interoperability and extensibility between different organisations using different platforms. Web Services can be self-contained with availability only to certain applications or distributed APIs with similar behaviour to the mentioned Graph API. Furthermore, Web Services can be built to use different protocols and encodings such as SOAP, REST, HTTP and more.

SOAP also known as Simple Object Access Protocol is an XML-based messaging protocol which can be used for exchanging information among applications. By using the SOAP protocol, a developer can extend HTTP for XML messaging among web services. A SOAP encoded message consists of XML data contained within envelopes and headers. Furthermore, SOAP services include built-in rules for encoding any data types. This enables SOAP messages to successfully transfer any data types including integers and floats with corruption.

A REST service is a type of web service which consists of using HTTP Protocol for data communication. REST, an acronym for Representational State Transfer allows data to be transferred and viewed through URIs. Furthermore, any data transferred or viewed will be represented in JSON format. It is possible for a REST service to provide data through XML arrays but this practice is very rare. RESTful services generally allow numerous HTTP methods as part of its functionality. For example, to read data from a REST service, a user can use the GET method with the relevant URI resource. To create or update a resource, a user can use the POST or PUT method. Lastly, a user can use the DELETE method to delete a resource.

Other common web services used by applications include XML-RPC. XML-RPC is a remote procedure call that uses XML mark-up language to encode data and HTTP for data transfer. A client can use XML-RPC to send transmit, process and return data structures. XML-RPC considered simpler than SOAP because it allows one way serialisation of methods, whereas SOAP defines multiple different encodings. A variant of this remote procedure call protocol includes JSON-RPC which uses JavaScript Object Notation for data exchange. This web service is very similar to XML-RPC with only a small number of differences. JSON-RPC and XML-RPC calls are implemented to call methods which return data whereas REST allows resources to be accessed. This functionality is most commonly used by WordPress as part of its WordPress API. The WordPress API allows a user to access his site data through a variety of RPC calls.

Moreover, a developer can use less known web service architectures such as WCF and Web API to send from one service endpoint to another. It should be noted that the most of these web services mentioned use a variation of SOAP and REST architectural style hence this is not covered in great detail in this paper.

Attacking Damn Vulnerable Web Services

Damn Vulnerable Web Service (DVWS) is written PHP with MySQL and multiple back-end web services including SOAP, REST, XML-RPC and JSON-RPC. The application allows a user to test and exploit the following security vulnerabilities:

- WSDL Enumeration
- XML External Entity Injection
- XML Bomb Denial-of-Service
- XPATH Injection
- WSDL Scanning
- Cross Site-Tracing
- OS Command Injection
- Server Side Request Forgery
- REST API SQL Injection

The following section of the report will explain all vulnerabilities that can be found within the Damn Vulnerable Web Services application.

WSDL Enumeration

When analysing a SOAP service, it is very common to find a WSDL document. The WSDL 1.1 specification describes a WSDL as

"WSDL is an XML format for describing network services as a set of endpoints Operating on messages containing either document-oriented or procedure-oriented Information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. "

To an attacker, a WSDL document defines a web service, its location and all methods and services supported by the web service. WSDL documents are written in XML format and explain all methods supported by a service, which are enclosed in specific XML elements.

The below illustration is an example of a WSDL document with methods defined through a XML Schema. A WSDL file usually begins with a <definitions> element and the rest of the document details all methods and operations through predefined child elements.

```
−<definitions targetNamespace="dvwa:webservice">
  −<types>
    −<xsd:schema targetNamespace="dvwa:webservice">
       <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
       <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/"/>
     </xsd:schema>
  </types>
  −<message name="Return_priceRequest">
     <part name="name" type="xsd:string"/>
  </message>
  −<message name="Return_priceResponse">
     <part name="return" type="xsd:float"/>
  </message>
```

Figure 1.0 – Defines the methods supported by the server and with data type it accepts.

The <message> element defines all methods supported by the service which can be used by a user to send requests and perform user operations. The <message> element

is usually followed by part name and type element which describes the value parameter and its data type that needs to be sent to the service.

Other parts of the document will contain the <portType> element. The <portType> element explains all operations that can be performed on the service.

In the illustration below, the portType "DVWA Web Service PortType" defines a two-way operation called "Return_price".

The "Return_price" operation takes an input called "tns:Return_priceRequest" message with the input parameter "name" (as shown by <message> element) and outputs a “tns:Return_priceResponse”.

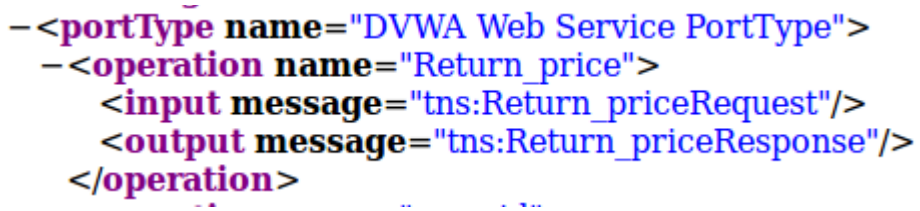

Figure 1.1 – All operations defined by the Return price function

The last part of a WSDL document is the <binding> element. The <binding> element shows details relevant to the SOAP Server used to process any sent requests. Furthermore, the <binding> element will detail the remote procedure call and the corresponding SOAP action that will performed.

In most cases, A WSDL document will need to be parsed in order to conduct any operations on the server. This can be done by taking all the XML elements defined by a WSDL and converting them using the correct XML schema. Fortunately, this can be done fruitlessly by using any generic SOAP Client which uses prewritten libraries for parsing.

To understand this vulnerability further, let’s take a look at the DVWS created example.

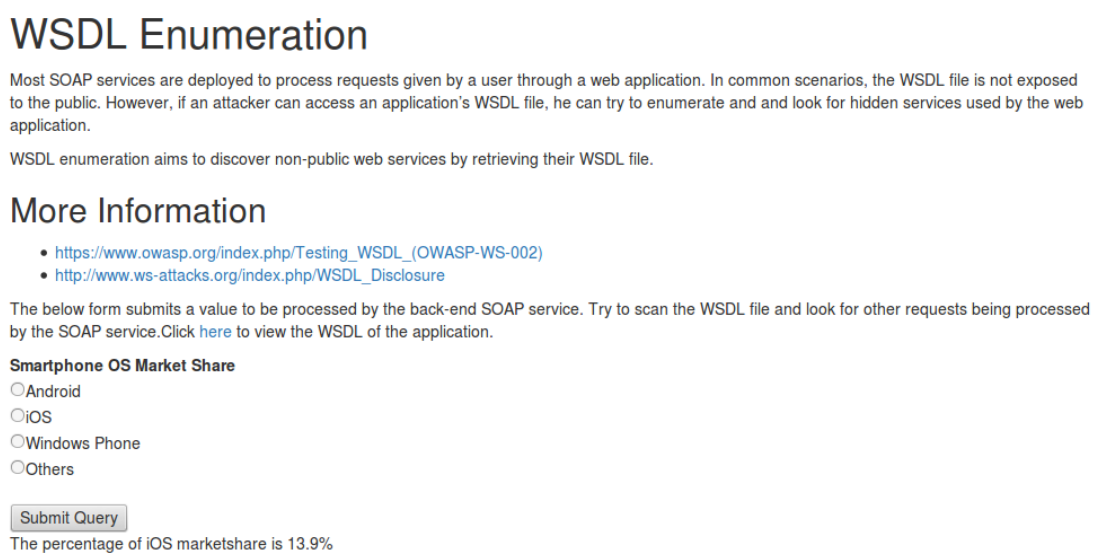

Figure 1.2 – The WSDL Enumeration exercise available on the DVWS application.

The above web page asks the user to choose a Smartphone OS of his choice. This value is sent to the web application's backend SOAP service. The SOAP service processes the chosen parameter and returns the associated value. This is intended behaviour and a user won't be able to use this functionality in any malicious ways.

But, this application publishes its WSDL document which is accessible to any user. This leaves the application vulnerable to WSDL Enumeration.

WSDL Enumeration is an information leakage flaw where the WSDL document of a web application is exposed to the public. By getting access to the WSDL file, an attacker can find hidden functions and services and get access to possible sensitive data.

To find a WSDL file, an attacker can spider the application or use wordlists and brute force the application find hidden directories. If the application is exposed to the internet, it is possible to find the WSDL by using the Google search function. By searching for files with the ending ".wsdl", an attacker can find all WSDL files indexed by the Google search engine.

The DVWS application discloses its WSDL which can be parsed and enumerated to find hidden functions.

There are numerous tools that are freely available for a user to parse WSDL. This includes

- SOAPUI
- SOA Cient (Firefox plugin)
- Boomerang (SOAP and REST Chrome client)
- Wsdler (Burp Suite Plugin)

Figure 1.3 – The SOA Client tool parses the WSDL file to show all operations and functions supported by the server.

Using SOA Client, the found WSDL can be enumerated to find the other methods available. By invoking and brute forcing any other available methods supported by the application, it is possible for an attacker to find additional information about the application. In the below illustration is an example of how the attacker can parse the WSDL file using the 'SOA Client' tool. Furthermore, the attacker can invoke the 'check_user_id' method to with an example user like root which returns the userid of root. This function wasn't implemented anywhere else in the application but it was found by analysing the WSDL file.

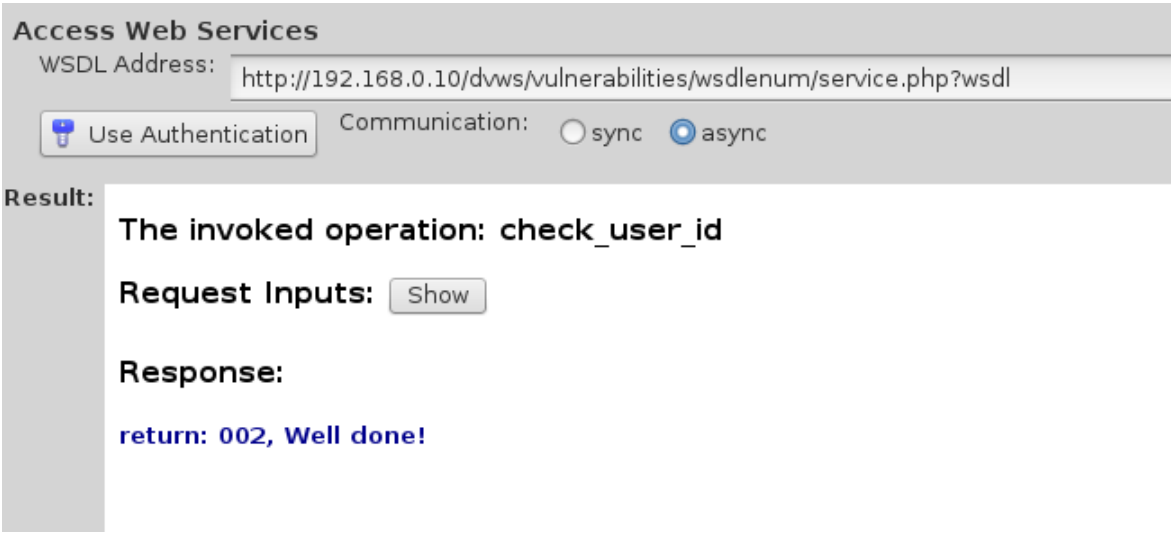

Figure 1.4– The WSDL file is parsed and used by SOA Client to invoke the check_user_id method.

XML Bomb Denial-of-Service

It is very common among web services to take XML data from a user and process it create arrays or store it in a backend database. There are a many programming libraries that can create this functionality such as simplexml, FastXML and more.

XML Bomb is a denial-of-service attack that can be used against exploitable XML parsers. XML Bomb is an exponential entity expansion attack which involves sending dangerously formed XML data with the intent of crashing a vulnerable server. When the XML parser tries to process a malicious XML data, the entity attributes runs itself and grows exponentially. If enough malicious XML is processed by the server, it can cause the server to crash.

XML Bomb attacks can lead to excessive CPU usage as well as memory leaks. In most servers, Denial of Service attempts are usually blocked by controlling all TCP traffic and looking for common attacks such as SYN flood or IP spoofed attacks. This usually allows traffic to be send through the web application since it is considered to be by a legitimate user.

The following payload is an example of how the denial of service attack works.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Figure 1.5 – Example of a XML external entity expansion payload.

This XML payload creates one root element called "lolz", that contains the text "&lol9;". However, "&lol9;" is a defined entity that expands to a string containing ten "&lol8;" strings. Each "&lol8;" string is a defined entity that expands to ten "&lol7;" strings and more. When a server processes the entity expansions, the XML data will take up to 2gb memory. By sending numerous payloads of this XML data, an attacker can take up valuable.

This vulnerability should not be confused with an XML external entity attack even though both attacks focus on vulnerable parsers.

The Damn Vulnerable Web Service application contains a vulnerable XML parser that takes any XML data and parses the data into an array structure. The parsed XML data is then showed as an array with system CPU usage and memory usage. To exploit this vulnerability, an attacker can use the payload shown on Figure 2.0. To exploit the DVWS XML parser.

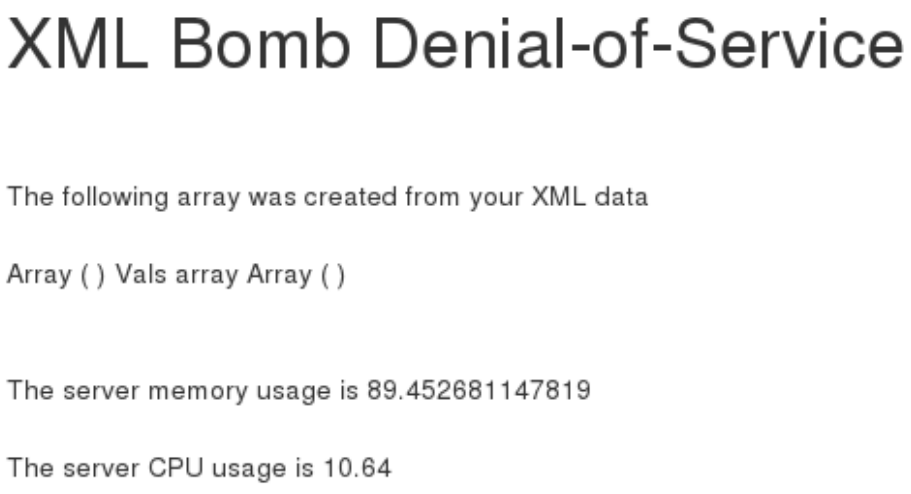

Figure 1.6 – The server usage of the server increased due to the processing of the XML entities.

To fully attack the server and take it offline, the payload should be sent multiple times within a short time window. This can be done through scripts or a fuzzer such as Burp Intruder.

XML External Entity Injection

An XML External Entity injection is a type of attack that can be used against weakly configured XML parsers. It is common for web applications use XML parsers to build documents based on different document types. This behaviour exposes applications to XML External Entity attacks.

Unless configured differently, most XML parsers allow for external entities to be used as part of the user input. This can be used by an attacker to access other resources stored on the server. XML External Entity (XXE) attacks can be used to perform a variety of attacks including

- Denial of service
- Access local system files
- Port scan remote machines hosted in the internal network.
- Remote Code Execution (in rare cases)

These types of attacks are common in web services and applications using XPath to retrieve a configuration setting from a XML file. E.g. file storage, authentication etc.

The Damn Vulnerable Web Service incorporates an exercise on XXE injection by exploiting the simplexml parser used in PHP applications. The application allows a user to input his name. This value in encapsulated within an XML element and sent to the server to be processed. The server processes the XML element and converts it into an object. This object is converted into a string and is reflected back on the page. The below illustration takes the value 'foo' encapsulated in within name element and prints it back on the page.

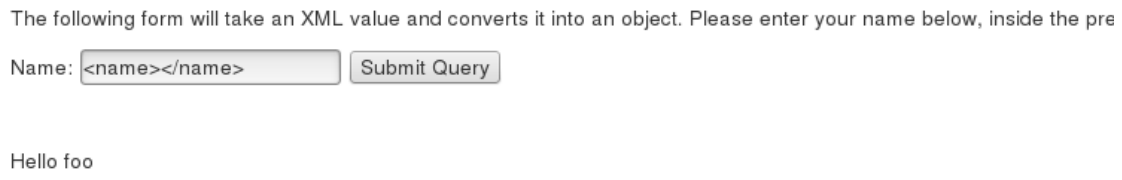

Figure 1.7 – The value 'foo' is printed back on the page.

To exploit this vulnerability, an attacker will need payload which defines an additional DOCTYPE and entity.

```
<!DOCTYPE test [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><test>&xxe;</test>
```

Figure 1.8 – The 'SYSTEM' value instructs the parser process entity value.

The above payload defines a DOCTYPE header called test with an entity called xxe. This entity is then defined using a system identifier that is present within the DOCTYPE header. The value 'SYSTEM' instructs the parser that the entity value should be read from the URI that it is associated with. Furthermore, the xxe entity value contains the location of the passwd file. This entity is then triggered by calling the entity encapsulated within the DOCTYPE header. When this payload is processed by the application, the application will access the passwd system resource stored on the local system. This file is then read and all values are printed back on the page.

By leveraging this payload, an attacker can read any system resource present on the vulnerable server.

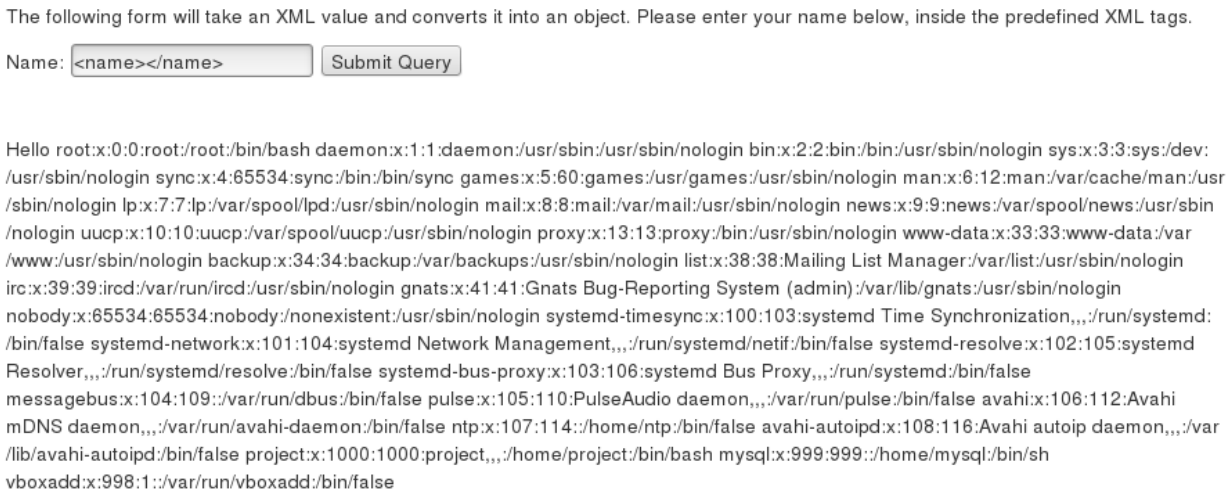

Figure 1.9 – The passwd file is fetched by the DVWS application and printed back onto the webpage

In real world scenarios, finding an XXE vulnerability can be trivial. The first objective a user should try to find is any endpoints that accept XML data as an input. It is common among web services to accept JSON encoded data as well as XML. It is also possible to exploit file parsers that parse word documents by inserting malicious XML in a

documents metadata. If the application parses the content correctly, then there is a scope for XXE. It should be noted that most XML parsers are vulnerable to this attack by default. Most libraries allow DOCTYPE DTDs due to the nature of its dependant libraries.

It is also possible for an attacker to exploit this vulnerability by giving different URI schemas. Instead of using the 'file://' URI to fetch a local resource, an attacker can use protocols such as 'http://' to fetch resources stored in arbitrary hosts across the internal network. This can be time consuming because an attacker has no direct interaction to the internal network and has no knowledge of back-end systems.

```
<!DOCTYPE test [<!ENTITY xxe SYSTEM "http://192.168.0.2:22 ">]><test>&xxe;</test>
```

Figure 2.0 – The entity is modified to include an internal IP range.

The payload mentioned in the beginning of this section can be modified to access any internal resources changing the URI schema. In some cases, the application might not return any data or system resource. To validate the payload is still being processed; an attacker can try to cause error exceptions by inserting carriage returns or null bytes. Furthermore, it may still be possible to cause a denial of service by reading a dangerous file steams such as /dev/random.

Lastly, it is possible for an attacker to retrieve all files stored on the server using direct and out of band methods like FTP or DNS exfiltration. It should be noted that these techniques only works if the server processing the XML input is allowed to make outbound connections to an attacker controlled server. Over the years, many tools like XXEinjector and xxeclient have been developed that can be used to automate this process.

XPath Injection

In certain occasions, web applications can store information in XML documents. This can later be transformed into other file formats such as HTML through XSLT. But, it can be difficult for application so search for data natively using programming languages due to XML formats. This problem was eventually solved by XPath query language. XPath language can be used to traverse through XML documents quickly and search elements or attribute with matching patterns.

When sensitive data is stored in XML forms which might have injection vulnerabilities, attackers can manipulate XPath queries to circumvent authentication, rewrite data and extract sensitive information.

As an example, take a look the following code

```
f(isset($_REQUEST["login"]) & isset($_REQUEST["password"]))

/take values from the HTML form
   $login = $_REQUEST["login"];
   $password = $_REQUEST["password"];


   // Loads the XML file saved in the same directory
   $xml = simplexml_load_file("accountinfo.xml");

   // Executes the XPath search
   $result = $xml->xpath("/users/user[login='" . $login . "' and password='" . $password . "']");

   if($result)
   {

       $message =  "<br>Accepted User: <b>" . ucwords($result[0]->login) . "</b><br>Your Account Number: <b>" .
       echo $message;
   }

   else
   {
```

Figure 2.1 – The result parameter executes an XPath search when it is called

This is an example of how XPath can be used for authentication. The example webpage takes a login and password input from the user. It then loads a saved XML document and executes an XPath query to search for the given account details. If the inputted values match the values stored in the xml file, the query will result as a true statement and the user is authenticated.

In this example, XPath uses a path expression to select login node from the users root node. So the web application takes the user values and searches all elements listed under the users elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user>
        <id>1</id>
        <login>admin</login>
        <password>$tr0ngpassw0rd</password>
        <accountd>06578368643</accountd>
        <country>United States</country>
        <birthday>19th March 1980</birthday>
    </user>
```

Figure 2.2 – The XML document with the stored user credentials.

But, if this user input is not validated properly, an attacker can modify the XPath query to be a true statement thus bypassing authentication. An attacker can bypass this authentication to entering a true statement in the login field such as

- foo' or 1=1 or 'a'='a

When this value is processed by the application, it evaluates as a true statement since 1=1 or a=a equates to true. This causes the application to return for all the users in the database and thus bypasses the security. In this scenario, the password value is irrelevant since the login field is executed as a true statement.

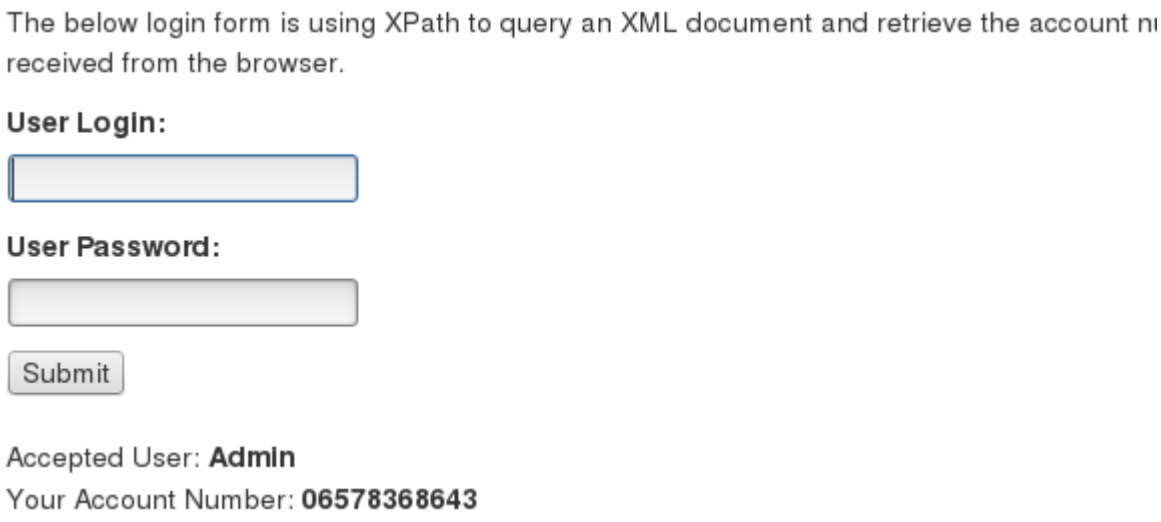

Figure 2.3 – A user is able to get admin access without knowing the username or password

In many cases, XPath injection is very similar to SQL injection because it follows the same logic as SQL Injection attacks. Unlike SQL injection, it is possible to access every part of an XML document using malicious XPath queries since no ACLs/stored procedures are enforced in the XPath language. Tools such as XCat can be used to automate XPath Injections. XCat can be used to retrieve the whole XML document being processed by a vulnerable XPath query, read arbitrary files on a target's system and utilize out of bound HTTP requests to make the server send data directly to XCat.

Command Injection

Most web servers support functionalities that allow data to interact with a server's operating
System. This feature can be useful when creating applications that does not need any additional features and can issue operating system commands with additional configuration.

If proper validation is not performed against user input, applications may be vulnerable to an attack known as Command Injection. Using this vulnerability, it may be possible for an attacker to supply input to the application that contains operating system commands that are executed with the privileges of the vulnerable application. Command Injection vulnerabilities are generally categorised as the following

- Results-based command injections - The vulnerable application outputs the results of the injected command.
- Blind command injections - The vulnerable application does not output the results of the injected command.

By leveraging this vulnerability, at attacker can get sensitive data such as

- Operating System Password Files,
- Operating System configuration files,
- Application Source Code

Furthermore, it is possible to get a command shell sending system commands to a reverse shell. Consider the following web application which shows the system uptime. The format this is shown is controlled by a user's input.

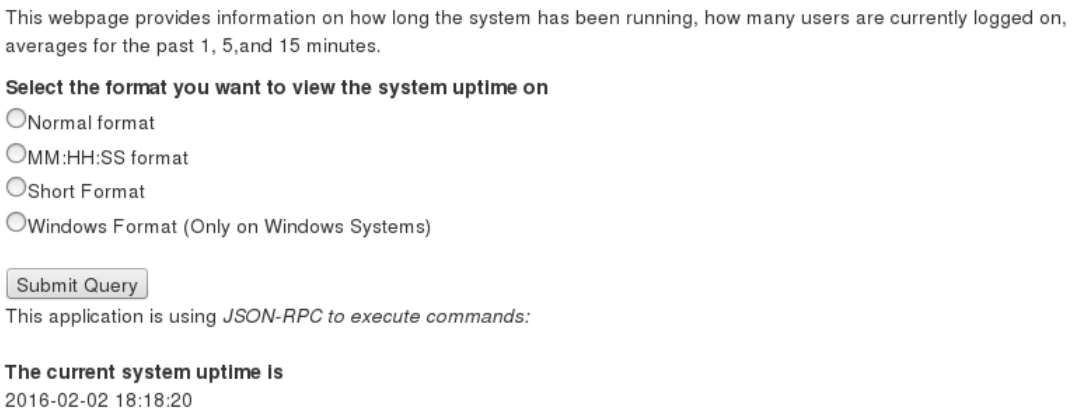

Figure 2.4 – The DVWS application shows the system's uptime based on a user's choice.

If this optioned is intercepted through a proxy, it is possible to deduce that the application is passing the uptime command with additional parameters directly to the web application. This is then interpreted as a system command.

```
POST /dvws/vulnerabilities/cmdi/client.php HTTP/1.1
Host: 192.168.0.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:43.0) Gecko/20100101 Firefox/43.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.10/dvws/vulnerabilities/cmdi/client.php
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

name=-s
```

Figure 2.5 – The application is sending the '-s' option which executes with the uptime command

For an attacker, the goal is the execution of arbitrary commands on the host operating system through a vulnerable application. This can be achieved by terminating the first command and adding a command such as 'ls' or 'ping'. The below illustration is an example of the Linux command 'ls' executed after terminating the uptime command with ';'.

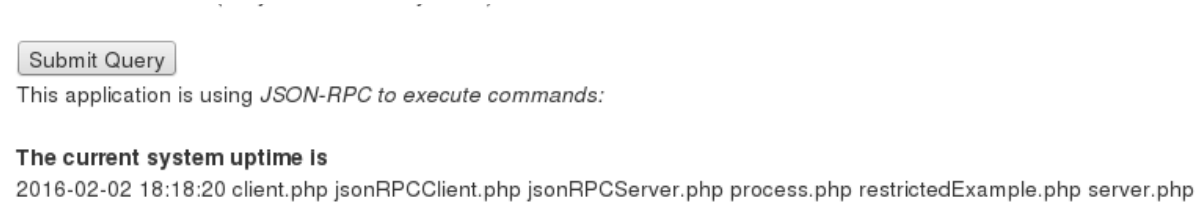

Figure 2.6 – The 'ls' command is executed and shows the current web directory.

It should be noted that any attacker-supplied OS commands are usually executed with the same privileges of the vulnerable application. Furthermore, command injection attacks are OS-independent and will differ between operating systems.

The vulnerability can be further leveraged to get a web shell on the server. This can be done using known network tools such as Netcat. Netcat is a simple utility which reads and writes data across a network connection using TCP or UDP. If Netcat is running on the vulnerable server, you could use it to set up a listener and then redirect the output of operating system commands into the listener.

To achieve this, an attacker will need to set up a Netcat listener on a controlled server. Then he can connect to the server to server and the port number used to execute the Netcat command.

Figure 2.7 – The nc command used to get a web shell

As the above illustration shows, the ‘nc’ command is used to setup a listener on the port 9090 and a bourne shell path. This starts an interactive shell on the DVWS application using Netcat. An attacker can then connect to the server using Netcat.

In real case scenarios, it can be hard to find command injection flaws in web services. This is commonly due to the way web services such as JSON-RPC handle data. It is common for web services to execute an injected command and not return any result back to the attacker. In these cases, the attacker can indirectly infer the output of the injected command using the following two techniques.

- Time-based (Blind) - An attacker injects and executes commands that introduce time delay.
- File-based (Semiblind) - When attacker is not able to observe the results of an executed command, he can write them to a file, which is accessible to the attacker.

This can be automated using tools such as Commix. Commix is a software tool aiming at facilitating web developers, penetration testers and security researchers to test web applications with the view to find bugs, errors or vulnerabilities related to command injection attacks.

Cross Site Tracing

Client-side scripts such as JavaScript are used extensively by modern web applications. They can perform simple functions up to full manipulation of client-side data. In certain cases, this can lead to a Cross Site Scripting flaw if the application is taking untrusted data and reusing it without performing any validation or encoding.

Cross Site scripting (also known as XSS) is a type of vulnerability where malicious client side scripts are injected into an application which can be executed by unsuspecting victims. In a XSS attack, the attacker can insert malicious JavaScript which will be reflected or stored by the application. This can then be triggered by an unsuspecting victim through phishing. By browsing to the vulnerable page on the server that renders the XSS payload, an attacker can gain access to a victim's session cookies.

But, this attack vector can be trivial if an HTTPOnly flag is set on a session cookie. This indicates that the cookie should not be accessible on the client. So if a malicious client-side script attempts to read cookies, the browser will return an empty string as the result. This can be considered a possible solution to Cross Site Scripting. But, this mitigation can be bypassed if a server supports the HTTP TRACE method.

To understand Cross Site Tracing vulnerability, take a look at the following example from DVWS. The DVWS application implements a NuSOAP server which is prone to a reflect cross-site scripting vulnerability. This may allow the attacker to steal cookie-based authentication credentials and to launch other attacks.

By injecting the following JavaScript code to the /dvws/vulnerabilities/wsdlenum/service.php, it is reflected back by the application without any sanitisation.

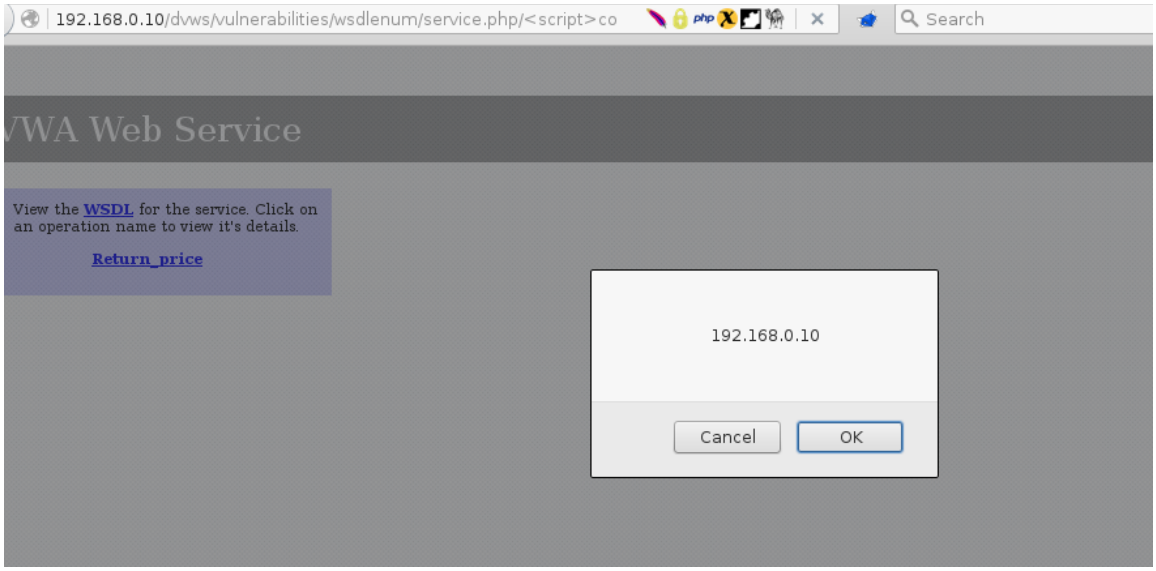

Figure 2.8 - The '<script>confirm(document.domain)</script>' payload is reflected by the NuSOAP 0.9.5 service page.

Even though the DVWS service page is vulnerable to a Cross Site Scripting flaw, an attacker will need to leverage the TRACE method allowed by the server to steal a user's cookie since HTTP Only flag is set on all DVWS cookies.

```
<script>
var xhr = new XMLHttpRequest();
xhr.open('TRACE','http://127.0.0.1:8000',
false);
xhr.send(null);
if(200 == xhr.status)
alert(xhr.responseText);
</script>
```

Figure 2.9 – XMLHttpRequest is used to send a victim's cookies to a malicious server.

The payload illustrated on Figure 2.9 can be used by an attacker to send a victim's credentials to an attacker controlled server. It should be noted that Cross Site Scripting flaws can be leveraged by an attacker to hook a victim's browser and exploit browser vulnerabilities but this left as an exercise for the reader.

Server Side Request Forgery

Web Applications and Web Services are often known to take user passed values and retrieve the contents of this value without any validation which can lead to known attacks such as SQL Injection, Local File Inclusion and more. But in rare cases, this can be exploited using an attack called Server Side Request Forgery.

SSRF (also known as XPSA) usually occurs when a web application attempts to connect to user supplied URLs and does not validate backend responses received from the remote server. SSRF can be used by an attacker to port scan any internet facing servers and services by creating requests from the vulnerable server. Furthermore, this attack vector can be leveraged to turn the application server against its hosted infrastructure.

An attacker can leverage this vulnerability to conduct the following attacks:

- Port scanning the affected server's internal network.
- Tunnel traffic through the vulnerable server and attack other external applications.
- Attack services running on the application server or on the internal Intranet.
- Denial of Service attacks on internal services.
- Access local files available to the application by using different URI schemes such as ‘file://’.

This vulnerability was mostly notably found in WordPress 4.0 applications using XML-RPC functions. XML-RPC is used in all versions of WordPress and is enabled by default. RPC calls allows users to access their site through the XML-RPC’s Pingback functionality. This function was found to be vulnerable to SSRF. This vulnerability gave an attacker the ability to create requests from the vulnerable server to the internet.

To understand a present SSRF vulnerability, take a look at the following example from DVWS. When clicking the Read button, The DVWS application uses the XML-RPC call which uses XML to read from usernames.txt saved on the server.

Figure 3.0 – The 'examples.stringecho' method is used to read usernames.txt

By modifying the 'string' parameter, it is obvious that the application is making outbound requests to other internet facing applications without any validation.

This vulnerability can be verified by forcing the application to perform outbound requests to a server you control.

Figure 3.1 – The DVWS application can perform outbound requests to a server you control

To exploit this vulnerability, an attacker can to conduct port scan on other servers by sending different requests with URI schemas and understanding error distinction given by the response. For example, an attacker can see if a port 22 of scanme.nmap.org is open sending 'http://scanme.nmap.org:22 as part of the string parameter. The server then sends the request to the given domain and returns the SSH banner in the server

debug message. This information can be decoded to get the SSH banner of the scanme.nmap.org which is ‘SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.3’.

Figure 3.2 – The base64 encoded server debug information is returned back in the response.

By enumerating through different ports, an attacker can distinguish if the port is open or not with error distinction. Furthermore, an attacker start port scanning the local host and looking for internal services. These type of attacks can usually bypass firewalls and web application firewalls since all traffic are proxied through the vulnerable application.

The automation of this vulnerability can be done through a tool called SSRF Proxy. SSRF Proxy is a multi-threaded HTTP proxy server designed to tunnel client HTTP traffic through HTTP servers vulnerable to Server-Side Request Forgery (SSRF). Using SSRF Proxy, an attacker can use scanning tools such as sqlmap and nikto and tunnel traffic through the SSRF vulnerable servers.

After setting up SSRF Proxy on an attacker machine, all traffic sent by the attacker machine will forward to the DVWS application.

Figure 3.3 – a proxy is setup using ‘ssrf-proxy -u " http://ip/dvws/vulnerabilities/ssrf/" --rules urlencode’

By attacking other hosts through the proxy, an attacker can remain anonymous while trying to exploit other hosts.

SQL Injection

SQL injection is possible when an application incorporates user input into a database query in an unsafe manner. An attacker is able to provide SQL commands as part of their input and circumvent the logic of the database query designed by the developer. This allows an attacker to gain access to the data held on the database server in use.

Contrary to popular belief, SQL Injection is very common among web services. It is common amongst developers to leave a web service unauthenticated while having the web application implement a user based permission model. This is a high security risk since an attacker can intercept and proxy all traffic being sent to the web application and therefore identity the web service.

The DVWS application has a REST service which is vulnerable to Error based SQL Injection. To find a SQL injection, the easiest attack technique is to insert a partial SQL query or syntax to cause the database to create an error.

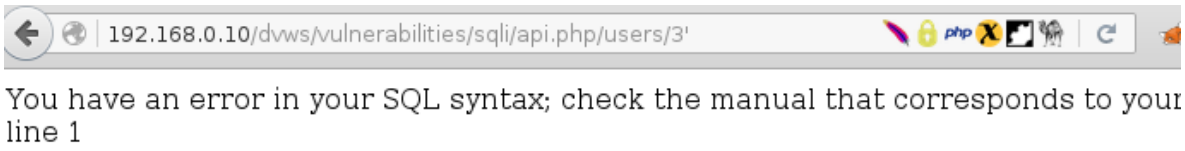

Figure 3.4 – Inserting a SQL query into the web service creates a syntax error in the SQL query.

By looking at the REST Service, an attacker can deduce the data being SQL interpreter is numerical. This can be confirmed by supplying a simple mathematical expression that is equivalent to the original numeric value. E.g. if the original value of the user value is 4, try submitting 2+2 or 5-1. If the application responds in the same way, it is vulnerable to SQL Injection.

To start extracting data, an attacker can start using the UNION operator. The UNION operator is used in SQL queries to combine the results of two or more SELECT statements into a single result set.

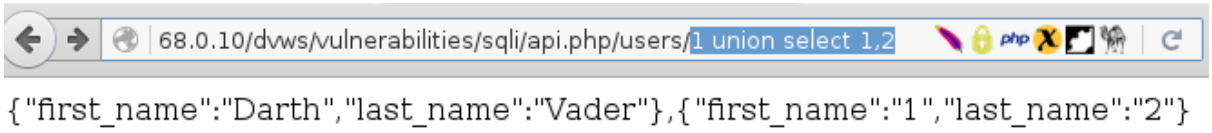

Figure 3.5 - The column names of the combined result set are returned by the SELECT query.

Furthermore, an attacker can try to get access to the information_schema.columns which will return the column names used by the database. By querying the information_schema, an attacker can get list all databases, columns and user defined tables. This data can then be queried by using the UNION operator.

Lastly, an attacker can try to get a command shell through SQL Injection. This is done by misusing the xp_cmdshell stored procedure. This stored procedure allows users with DBA permissions to execute operating system commands in the same way as the cmd.exe command prompt.

Numerous tools are available that automate exploiting SQL Injection vulnerabilities since it involves making large number of requests. Sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws. Sqlmap has a number of features which allows Boolean-based, time-based, error-based, UNION query-based, stacked queries and out-of-band injection techniques. Using Sqlmap, it is possible to dump the entire database of the DVWS application.

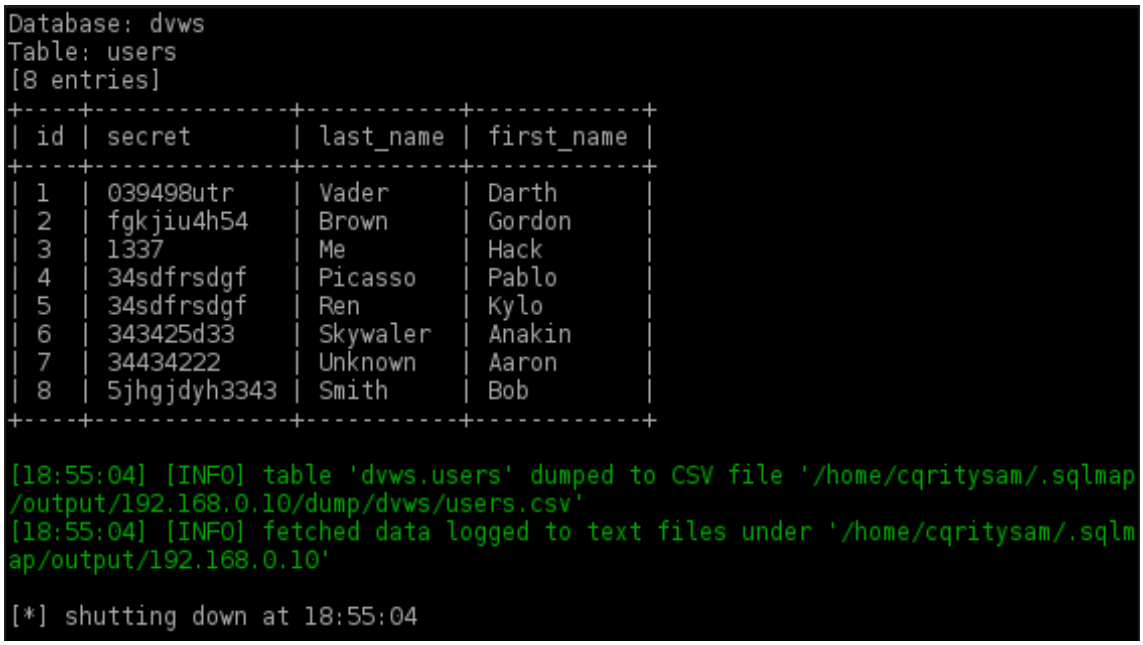

Figure 3.6- The entire contents of the database is retrieved by Sqlmap

It should be noted that since an REST URLs don't provide parameters, custom injection marking characters should be used to detect the SQL Injection flaw.