Insecure Direct Object References

: 27/07/2022

Jul 26, 2022

9 min read
IDOR Web-Notes

IDORs happen when users can `**access resources that do not belong to them**` by directly referencing the object ID, object number, or filename.

References

GitHub - vavkamil/awesome-bugbounty-tools: A curated list of various bug bounty tools

PayloadsAllTheThings/Insecure Direct Object References at master · swisskyrepo/PayloadsAllTheThings

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

GitHub - alexbieber/Bug_Bounty_writeups: BUG BOUNTY WRITEUPS - OWASP TOP 10 🔴🔴🔴🔴✔

Insecure Direct Object References (IDOR)

HowToHunt/IDOR at master · KathanP19/HowToHunt

AllAboutBugBounty/Insecure Direct Object References.md at master · daffainfo/AllAboutBugBounty

IDOR

Web Application Penetration Testing Notes

pentest-guide/Insecure-Direct-Object-References at master · Voorivex/pentest-guide

Writeups Bug Bounty hackerone

Insecure Direct Object Reference [IDOR] - Pastebin.com

Mechanisms

For example, let's say that example.com is a social media site that allows you to chat with others. And there is a button to show all of `your messages:`.

What happen if we change the `user_id` to another number , we can see another user messages without and permisson

Example Code:

```
messages = load_messages(request.user_id)
display_messages(messages)
```

For example, let's say that users can submit a POST request to change their password. The POST request must contain that `user's ID` and `new password`, and they must direct the request to the `/change_password` endpoint:

```
POST /change_password
(POST request body)
user_id=1234&new_password=12345
```

If the applicaiton dosen't validate the user id , attacker can submit another user_id and change its password.

For example, this request allows users to access a file they've uploaded: `https://[example.com/uploads?file=user1234-01.jpeg](http://example.com/uploads?file=user1234-01.jpeg).`

we can easily deduce that user-uploaded files follow the naming convention of `USER_ID-FILE_NUMBER.FILE_EXTENSION`. Therefore, another user's uploaded files might be named `user1233-01.jpeg`.

If the application dosen't validate the user_id , an attacker can modify it like that :

Hunting for IDORs

The best way to discover IDORs is through a `**source code review**` that checks if all direct object references are protected by access control.

Step 1: Create Two Accounts

If users can have different permissions on the site, create two accounts for each permission level. For example, create two `admin accounts`, two `regular user accounts`, two group member accounts, and two non-group-member accounts.

Continuing the previous example, you could create two accounts on : user `1235` and user `1236`. One of the accounts would serve as your `attacker` account, used to carry out the `IDOR attacks`.

The message pages for the two users would have the following URLS: (`**Attacker**`)

](https://example.com/messages?user_id=1236) (`**Victim**`)

In addition to testing with two accounts, you should also repeat the testing procedure without `**signing in**`. See if you can use an unauthenticated session to access the information or functionalities made available to legitimate users.

Step 2: Discover Features

Use the `**highestprivileged**` account you own and go through the application, looking for application features to test.

Pay special attention to functionalities that `**return user information or modify user data**`. Note them for future reference. Here are some features that might have IDORs on :

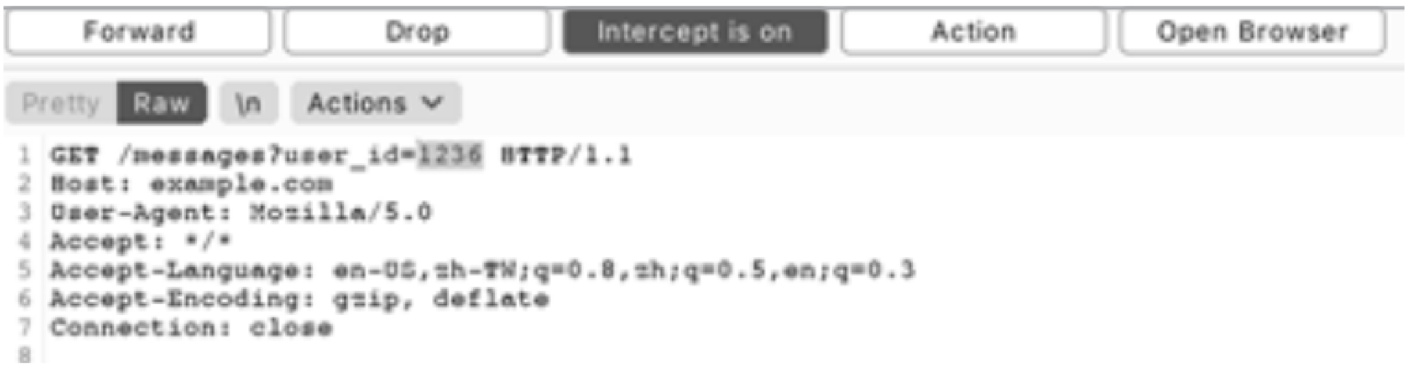
This endpoint lets you read user `messages`:

This one lets you read user `files`: This endpoint `deletes` user messages: POST /delete_message (POST request body) message_id=user1236-0111 This one is for accessing `group files`: This one `deletes a group`: POST /delete_group (POST request body) group=group3

Step 3: Capture Requests

Inspect each request carefully and find `**the parameters**` that contain `**numbers**`, `**usernames**`, or `**IDs**`. Remember that you can trigger IDORs from different locations within a request, like `**URL parameters**`, form `**fields**`, `**filepaths**`, `**headers**`, and `**cookies**`.

For example, let's say you create two accounts, `1235` and `1236`. Log into 1235 in `Firefox` and 1236 in `Chrome`. Use Burp to modify the traffic coming from `Firefox`. Turn on Intercept in the Proxy tab and edit requests in the proxy text window (Figure 10-1). Check if your attack has succeeded by observing the changes reflected on the victim account in `Chrome`.



```
GET /messages?user_id=1236 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: */*
Accept-Language: en-US,zh-TW;q=0.8,zh;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close

```

Figure 10-1

Step 4: Change the IDs

See if you can access the `victim account's information` by using the `attacker account`. And check if you can modify the second user's account from the first

For example, in this setup, you can try to access the functionalities that user 1236(chrome user) has access to via your Firefox browser: This endpoint lets you `read user messages`:

This one lets you `read user files`: This endpoint `deletes user` messages: POST /delete_message (POST request body) message_id=user1236-0111 This one is for `accessing group files`: This endpoint `deletes a group`: POST /delete_group (POST request body) group=group3

Bypassing IDOR Protection

Modern web applications have also begun implementing more protection against IDORs, and many now use more `**complex ID formats**`.

Encoded IDs and Hashed IDs

First, don't ignore encoded and `**hashed IDs**`. You should also learn to recognize the most common encoding schemes, like `**base64**`, `**URL encoding**`, and `**base64url**`. For example, take a look at the IDs of this endpoint: (base64URL)

You can use The [BurpSuite] Decoder to know the decoding scheme!

If the application is using a hashed or `**randomized**` ID, see if the ID is `**predictable**`.

In this case, try creating a `**few accounts to analyze how these IDs are created.**` You might be able to find a pattern that will allow you to `**predict IDs**` belonging to other users.

Leaked IDs

It might also be possible that the application leaks IDs via another `**API endpoint or other public pages**` of the application, like the profile page of a user.

The following request would return a list of conversation_ids belonging to that user: `GET /messages?user_id=1236` Since the user_id is publicly available on each `user's profile page`, I could read any user's messages by first obtaining their user_id on their profile page, retrieving a list of conversation_ids belonging to that user, and finally loading the messages via their conversation_ids.

Offer the Application an ID, Even If It Doesn't Ask for One

In modern web applications, you'll commonly encounter scenarios in which the `**application uses cookies instead of IDs to identify the resources a user can access.**`

For example, when you send the following GET request to an endpoint, the application will `deduce your identity based on your session cookie`, and then send you the messages associated with that user

```
GET /api_v1/messages
Host: example.com
Cookies: session=YOUR_SESSION_COOKIE
```

If no IDs exist in the application-generated request, try adding one to the request. `Append id, user_id, message_id`, or other object references to the URL query, or the POST body parameters, and see if it makes a difference to the application's behavior. For example, say this request displays your messages:

```
If it looklike that
GET /api_v1/messages
Just Try to Make this
GET /api_v1/messages?user_id=ANOTHER_USERS_ID
```

Keep an Eye Out for Blind IDORs

For example, imagine that this endpoint on allows users to email themselves a copy of a receipt:

```
POST /get_receipt
(POST request body)
receipt_id=3001
```

This request will send a copy of receipt 3001 to the `**registered email of the current user**`. Now, what if you were to request a receipt that belongs to another user, receipt 2983?

```
POST /get_receipt
(POST request body)
receipt_id=2983
```

While the HTTP response does not change, you may get a copy of `**receipt 2983 in your email inbox!**`

Change the Request Method

For example, if this `GET` request is not vulnerable to IDOR and doesn't return another user's resources: GET `

you can try to use the DELETE method to delete the resource instead. The `DELETE method removes the resource from the target URL`: `DELETE [] (http://example.com/uploads/user1236-01.jpeg)`

If `POST` requests don't work, you can also try to update another user's resource by using the `PUT` method. The P`UT method updates` or creates the resource at the target URL:

```
PUT example.com/uploads/user1236-01.jpeg
(PUT request body)
NEW_FILE
```

Another trick that often works is `switching between POST and GET requests`. If there is a POST request like this one

```
POST /get_receipt
(POST request body)
receipt_id=2983
```

you can try rewriting it as a GET request, like this: `GET /get_receipt?receipt_id=2983`

Change the Requested File Type

For example, applications commonly store information in the JSON file type. Try adding the `.json extension to the end of the request URL` and see what happens. If this request is blocked by the server

`GET /get_receipt?receipt_id=2983` Try thia instead: `GET /get_receipt?receipt_id=2983.json`

Escalating the Attack

For example, look for functionalities that handle `direct messages`, `personal information`, and `private content`. Consider which application functionalities make use of this information and look for IDORs accordingly.

You can also combine IDORs with other vulnerabilities to increase their impact. For example, a `write-based IDOR can be combined with self-XS`S to form a `stored XSS`. An IDOR on a `password reset endpoint combined with username enumeration can lead to a mass account takeover`. Or a write `IDOR on an admin account may even lead to RCE!`

Automating the Attack

The Burp extension `Autorize`) scans for authorization issues by accessing higher-privileged accounts with lower-privileged accounts, whereas the Burp extensions `Auto Repeater`) and `AuthMatrix`) allow you to automate the process of switching out cookies, headers, and parameters. For more information on how to use these tools, go to the Extender tab of your Burp window, then to the BAppStore tab to find the extension you want to use.

Finding Your First IDOR

1. Create `two accounts` for each application role and designate one as the attacker account and the other as the victim account.
2. Discover features in the application that might lead to IDORs. Pay attention to `features that return sensitive information` or modify user data.
3. Revisit the features you discovered in step 2. With a proxy, `intercept your browser traffic` while you browse through the sensitive functionalities.
4. With a proxy, intercept each sensitive request and `switch out the IDs` that you see in the requests. If switching out IDs grants you access to other users' information or lets you change their data, you might have found an IDOR.
5. Don't despair if the application seems to be immune to IDORs. Use this opportunity to try a `protection-bypass technique`!

   If the application uses an encoded, hashed, or randomized ID, you can try `decoding or predicting the IDs.`

   You can also try supplying the application with an ID when `it does not ask for one`. Finally, sometimes changing the request method type or file type makes all the difference.
6. `Monitor for information leaks in export files`, email, and text alerts. An IDOR now might lead to an info leak in the future.
7. Draft your first IDOR report!

Prevention

IDORs happen when an application fails at two things. First, it fails to `implement access control` based on user identity. Second, it fails to `randomize object IDs` and instead keeps references to data objects, like a file or a database entry, `predictable`.

Applications can prevent IDORs in two ways. **First**, the application can `**check the user's identity and permissions**` before granting access to a resource. For example, the application can check if the user's `**session cookies**` correspond to the user_id whose messages the user is requesting.

Second, the website can use a `**unique, unpredictable key**` or a hashed identifier to reference each user's resources.

If structured its requests as follows, attackers would no longer be able to access other users' messages, since there would be no way for an attacker to guess such a long, random `**user_key**` value:

But this method isn't a complete protection against IDORs. Attackers can still `**leak user information**` if they can find a way to steal these URLs or `**user_keys**`.