# Identifying IDORs

## URL Parameters & APIs

The very first step of exploiting IDOR vulnerabilities is identifying Direct Object References. Whenever we receive a specific file or resource, we should study the HTTP requests to look for URL parameters or APIs with an object reference (e.g. `?uid=1` or `?filename=file_1.pdf`). These are mostly found in URL parameters or APIs but may also be found in other HTTP headers, like cookies.

In the most basic cases, we can try incrementing the values of the object references to retrieve other data, like (`?uid=2`) or (`?filename=file_2.pdf`). We can also use a fuzzing application to try thousands of variations and see if they return any data. Any successful hits to files that are not our own would indicate an IDOR vulnerability.

## AJAX Calls

We may also be able to identify unused parameters or APIs in the front-end code in the form of JavaScript AJAX calls. Some web applications developed in JavaScript frameworks may insecurely place all function calls on the front-end and use the appropriate ones based on the user role.

For example, if we did not have an admin account, only the user-level functions would be used, while the admin functions would be disabled. However, we may still be able to find the admin functions if we look into the front-end JavaScript code and may be able to identify AJAX calls to specific end-points or APIs that contain direct object references. If we identify direct object references in the JavaScript code, we can test them for IDOR vulnerabilities.

This is not unique to admin functions, of course, but can also be any functions or calls that may not be found through monitoring HTTP requests. The following example shows a basic example of an AJAX call:

Code: javascript

```
function changeUserPassword() {
    $.ajax({
        url:"change_password.php",
        type: "post",
        dataType: "json",
        data: {uid: user.uid, password: user.password, is_admin: is_admin},
        success:function(result){
            //
        }
    });
}
```

The above function may never be called when we use the web application as a non-admin user. However, if we locate it in the front-end code, we may test it in different ways to see whether we can call it to perform changes, which would indicate that it is vulnerable to IDOR. We can do the same with back-end code if we have access to it (e.g., open-source web applications).

## Understand Hashing/Encoding

Some web applications may not use simple sequential numbers as object references but may encode the reference or hash it instead. If we find such parameters using encoded or hashed values, we may still be able to exploit them if there is no access control system on the back-end.

Suppose the reference was encoded with a common encoder (e.g. `base64`). In that case, we could decode it and view the plaintext of the object reference, change its value, and then encode it again to access other data. For example, if we see a reference like (`?filename=ZmlsZV8xMjMucGRm`), we can immediately guess that the file name is `base64` encoded (from its character set), which we can decode to get the original object reference of (`file_123.pdf`). Then, we can try encoding a different object reference (e.g. `file_124.pdf`) and try accessing it with the encoded object reference (`?filename=ZmlsZV8xMjQucGRm`), which may reveal an IDOR vulnerability if we were able to retrieve any data.

On the other hand, the object reference may be hashed, like (`download.php?filename=c81e728d9d4c2f636f067f89cc14862c`). At a first glance, we may think that this is a secure object reference, as it is not using any clear text or easy encoding. However, if we look at the source code, we may see what is being hashed before the API call is made:

Code: javascript

```
$.ajax({
    url:"download.php",
    type: "post",
    dataType: "json",
    data: {filename: CryptoJS.MD5('file_1.pdf').toString()},
    success:function(result){
        //
    }
});
```

In this case, we can see that code uses the `filename` and hashing it with `CryptoJS.MD5`, making it easy for us to calculate the `filename` for other potential files. Otherwise, we may manually try to identify the hashing algorithm being used (e.g., with hash identifier tools) and then hash the filename to see if it matches the used hash. Once we can calculate hashes for other files, we may try downloading them, which may reveal an IDOR vulnerability if we can download any files that do not belong to us.

## Compare User Roles

If we want to perform more advanced IDOR attacks, we may need to register multiple users and compare their HTTP requests and object references. This may allow us to understand how the URL parameters and unique identifiers are being calculated and then calculate them for other users to gather their data.

For example, if we had access to two different users, one of which can view their salary after making the following API call:

Code: json

```
{
  "attributes" :
    {
      "type" : "salary",
      "url" : "/services/data/salaries/users/1"
    },
  "Id" : "1",
  "Name" : "User1"

}
```

The second user may not have all of these API parameters to replicate the call and should not be able to make the same call as `User1`. However, with these details at hand, we can try repeating the same API call while logged in as `User2` to see if the web application returns anything. Such cases may work if the web application only requires a valid logged-in session to make the API call but has no access control on the back-end to compare the caller's session with the data being called.

If this is the case, and we can calculate the API parameters for other users, this would be an IDOR vulnerability. Even if we could not calculate the API parameters for other users, we would still have identified a vulnerability in the back-end access control system and may start looking for other object references to exploit.