IDOR Prevention

We learned various ways to identify and exploit IDOR vulnerabilities in web pages, web functions, and API calls. By now, we should have understood that IDOR vulnerabilities are mainly caused by improper access control on the back-end servers. To prevent such vulnerabilities, we first have to build an object-level access control system and then use secure references for our objects when storing and calling them.

Object-Level Access Control

An Access Control system should be at the core of any web application since it can affect its entire design and structure. To properly control each area of the web application, its design has to support the segmentation of roles and permissions in a centralized manner. However, Access Control is a vast topic, so we will only focus on its role in IDOR vulnerabilities, represented in `Object-Level` access control mechanisms.

User roles and permissions are a vital part of any access control system, which is fully realized in a Role-Based Access Control (RBAC) system. To avoid exploiting IDOR vulnerabilities, we must map the RBAC to all objects and resources. The back-end server can allow or deny every request, depending on whether the requester's role has enough privileges to access the object or the resource.

Once an RBAC has been implemented, each user would be assigned a role that has certain privileges. Upon every request the user makes, their roles and privileges would be tested to see if they have access to the object they are requesting. They would only be allowed to access it if they have the right to do so.

There are many ways to implement an RBAC system and map it to the web application's objects and resources, and designing it in the core of the web application's structure is an art to perfect. The following is a sample code of how a web application may compare user roles to objects to allow or deny access control:

Code: javascript

```
match /api/profile/{userId} {
    allow read, write: if user.isAuth == true
    && (user.uid == userId || user.roles == 'admin');
}
```

The above example uses the `user` token, which can be `mapped from the HTTP request made to the RBAC` to retrieve the user's various roles and privileges. Then, it only allows read/write access if the user's `uid` in the RBAC system matches the `uid` in the API endpoint they are requesting. Furthermore, if a user has `admin` as their role in the back-end RBAC, they are allowed read/write access.

In our previous attacks, we saw examples of the user role being stored in the user's details or in their cookie, both of which are under the user's control and can be manipulated to escalate their access privileges. The above example demonstrates a safer approach to mapping user roles, as the user privileges `were not be passed through the HTTP request`, but mapped directly from the RBAC on the back-end using the user's logged-in session token as an authentication mechanism.

There's a lot more to access control systems and RBACs, as they can be some of the most challenging systems to design. This, however, should give us an idea of how we should control user access over web applications' objects and resources.

Object Referencing

While the core issue with IDOR lies in broken access control (`Insecure`), having access to direct references to objects (`Direct Object Referencing`) makes it possible to enumerate and exploit these access control vulnerabilities. We may still use direct references, but only if we have a solid access control system implemented.

Even after building a solid access control system, we should never use object references in clear text or simple patterns (e.g. `uid=1`). We should always use strong and unique references, like salted hashes or `UUID`'s. For example, we can use `UUID V4` to generate a strongly randomized id for any element, which looks something like (`89c9b29b-d19f-4515-b2dd-abb6e693eb28`). Then, we can map this `UUID` to the object it is referencing in the back-end database, and whenever this `UUID` is called, the back-end database would know which object to return. The following example PHP code shows us how this may work:

Code: php

```php
$uid = intval($_REQUEST['uid']);
$query = "SELECT url FROM documents where uid=" . $uid;
$result = mysqli_query($conn, $query);
$row = mysqli_fetch_array($result));
echo "<a href='" . $row['url'] . "' target='_blank'></a>";
```

Furthermore, as we have seen previously in the module, we should never calculate hashes on the front-end. We should generate them when an object is created and store them in the back-end database. Then, we should create database maps to enable quick cross-referencing of objects and references.

Finally, we must note that using UUIDs may let IDOR vulnerabilities go undetected since it makes it more challenging to test for IDOR vulnerabilities. This is why strong object referencing is always the second step after implementing a strong access control system. Furthermore, some of the techniques we learned in this module would work even with unique references if the access control system is broken, like repeating one user's request with another user's session, as we have previously seen.

If we implement both of these security mechanisms, we should be relatively safe against IDOR vulnerabilities.

Previous Next Mark Complete & Next