Direct Object Reference or,

How a Toddler Can Hack Your Web Application

There is no point in denying that everyday software is steadily moving from desktop applications to Web applications. When you can check your mail, play games, create documents and file your tax report without ever leaving your browser, then you are indeed a citizen of the Web.

In this era, many miscreants have changed their game. It's easier for them to impersonate you or steal your private data from a vulnerable Web application than to take control of the *Extended Instruction Pointer* (EIP) register of your CPU. The reason is simple. As a software industry, we have more experience writing native applications in C and C++ than writing Web applications in PHP and JavaScript. People still write bugs in their code, but they are definitely harder to find and exploit than it was 10 years ago.

In this article we will investigate one type of Web application vulnerability, namely Direct Object Reference. A Direct Object Reference occurs when an identifier, used in the internal implementation of a Web application, is exposed to users. When this is done insecurely, it can lead to a lot of trouble. This vulnerability is probably one of the easiest to exploit but is so deadly and prevalent that it claims the 4th position in OWASP's Top 10 Web Application security risks [2]. Many institutions have fallen victim to it, with the most recent example of an Australian financial company which was vulnerable in a way that made it possible for anyone to access other peoples' private financial information [1].

Vulnerable Web Application

In order to make explaining easier, we will use as an example a dummy Web application that allows logged-in users to send personal messages to each other. All the messages exchanged between members are stored in a specific table in an SQL database as follows: see Table 1.

The `message_id` column contains auto-incremented values, unique for every message. The columns *to* and *from* contain the user identifiers of the sender and recipient of any given message. The *title* column contains the title of each message and lastly the *message* column contains the actual message exchanged between two users. Now lets look at some of the PHP functions used by the Web application to display a user's Inbox and allow him to read incoming messages (Listing 1).

Viewing the INBOX

The function `get_message_titles()` is responsible for providing logged-in users an overview of their Inboxes. The first thing that the function does is to check whether the user is logged-in. It does this by checking whether the superglobal `$_SESSION` array contains a key titled `user_id`. This key is set by the Web application when the user successfully logs in, and is typically a unique identifier in the *Users* table of the application much like the unique identifier of each message in the *Messages* table. We will not need that function in our discussion thus for the sake of brevity, it is not shown in Listing 1. If that key is not set, then the code redirects the user to the login page of the Web application and returns.

If the user is indeed logged-in then the user's `user_id` is extracted from his session. Note that the

Figure 1. *Table "messages" containing personal messages that users exchanged with each other*

Message_id	From	To	Title	Message
...	...	...	...	...
776	23	11	"Hey!"	"Hey man! What news?</br>"
777	11	25	"Foo..."	"U there?"
778	25	42	"No Title"	"Kthnxbye!"
779	23	11	"Welcome"	"Welcome to our site!..."
...	...	...		...

`user_id` is passed as a parameter to the `intval()` function. The `intval()` function is a built-in function of PHP that returns the integer value of the parameter that the programmer passes to it. This function is very useful for Web application developers since it allows them to easily filter out erroneous requests or stop malicious users who attempt SQL injections or Cross-Site Scripting attacks using fields that the programmer knows should be integers. Since we will be incorporating this value to an SQL query we want to make sure that it is an integer and nothing more than that.

In the next step, the Web application will query its database for all messages that have as a recipient the current user. For instance, if we assume that the current logged-in user corresponds to `user_id` value 11, the SQL query will be:

```
SELECT message_id, title FROM Messages where to = 11;
```

The database server will return two rows of results, specifically the messages with `message_id` equal to 776 and 779. For each of them the Web application will create an HTML link with the message's title.

Reading a Specific Message

When a user decides to read a specific message, he or she can click on the desired title, which will cause

Listing 1. *Code of vulnerable Web application*

```
<?php
session_start();
function get_message_titles(){

            /*Redirect users if they are not logged in*/
            if  ( !isset($_SESSION['user_id'])){
                        header("Location: http://www.example.com/login.php");
                        return;
            }

            $user_id = intval($_SESSION['user_id']);
$result = mysql_query("SELECT message_id, title FROM Messages where to =       {$user_id}");
            while($row = mysql_fetch_array($result)){
                        print "<a href='./read_message.php?id={$result['message_id']}"> {$result['title']}";
                        print "</a><br>";
            }
            return;
}

function read_message(){
            /*Redirect users if they are not logged in*/
            if (! isset($_SESSION['user_id'])){
                        header("Location: http://www.example.com/login.php");
                        return;
            }
            if  (! isset($_GET['id'])) return;
            $message_id = intval($_GET['id']);
            $result = mysql_query("SELECT * FROM Messages where message_id = {$message_id}");
$row = mysql_fetch_array($result);
            print "<b> From: </b>" . uid_to_username($row['from']) . "<br>";
            print "<b> Title :  </b> {$row['title']} <br>";
            print "<b> Message: </b> {$row['message']} <br>";
            return;
}
?>
```

the Web application to call the function `read_message()`. This function, like `get_message_titles()` first checks if the user who is requesting the reading of a message is logged-in and redirects the user if not. In the next step the value of the GET parameter named 'id' is retrieved (given that it exists) and filtered through the integer value function. Thus, continuing with our previous example, if the user clicks on the following link: *http://example.com/read_message.php?id=776*, the `$message_id` variable of the `read_message` function will contain the number 776. In the next step, the message id is used to retrieve the full message from the messages table and thus in this example case the SQL query will be the following:

```
SELECT from,title,message FROM Messages where message_id = 776;
```

The Web application uses the data and prints it out as HTML to the user. The user reads the message and is happy. Or maybe not ?

Exploiting IT

Those of you who have a security-oriented mindset [4] may feel a bit uncomfortable with the workings of the previously described Web application. Sure, they check for SQL injections whenever they use data coming-in from the user but you feel that something is not quite right...

Lets look closely at the `read_message()` function and at the resulting SQL query. We saw that if a user clicks on link *http://example.com/read_message.php?id=776*, the Web application will use the id parameter to find and retrieve the appropriate message. In fact, this is exactly where the vulnerability `lies--` the Web application uses ONLY the id that the user provided as a means of reading a message. For the user with `user_id` equal to 11, the Web programmer assumed that he or she can click only on the links provided by the `get_message_titles()` function, thus only click on one of the following links:

- *http://example.com/read_message.php?id=776*
- *http://example.com/read_message.php?id=779*

While it is true that only the these two links will be available in his INBOX, nothing is stopping the user from changing the id parameter to any value

Listing 2. *A simple Python script which downloads and saves the first 1024 messages from the*

```
import urllib
import os

os.mkdir("./messages")
for i in range(0,1024):
            current_message = urllib.urlopen("http://example.com/read_message.php?id=%d" % i)
            out_file = open("./messages/%d.txt" % i,"w")
            out_file.write(current_message.read())
            out_file.close()
```

Listing 3. *Adding authorization checks to the vulnerable SQL query*

```
<?php
function read_message(){
            /*Redirect users if they are not logged in*/
            if  (! isset($_SESSION['user_id'])){
                        header("Location: http://www.example.com/login.php");
                        return;
            }
            if  (! isset($_GET['id'])) return;
            $message_id = intval($_GET['id']);
$user_id = intval($_SESSION['user_id']);
            $result = mysql_query("SELECT * FROM Messages where message_id = {$message_id} and to= {$user_id}");
            /* The rest of the code is the same  with the original function*/
            [...]
}
?>
```

at all. Thus, the user with `user_id 11`, can ask for *http://example.com/read_message.php?id=778* and start reading a message that was sent to another user! Once the malicious user is convinced that the above works then just 10 lines of Python code obtains the full database of messages: Listing 2.

And that's it! So simple, that even a toddler could do it! The problem in the design of this Web application is that the user can *directly reference* messages in the database. If we generalize this, there is a problem when a user is allowed to directly reference objects without any authorization checks in place. Thus, this vulnerability falls under the class of *Direct Object Reference* (DOR) vulnerabilities. In the next section we will see how to repair this Web application and also give some general guidelines for other Web applications.

Defenses

There are more than one ways of defending against DOR vulnerabilities and it is up to the Web programmer to choose the appropriate one for his Web application. Lets review some of these ways:

Listing 4. *Adding an extra layer of indirection to defeat DOR attacks*

```
<?php
function get_message_titles(){

            /*Redirect users if they are not logged in*/
            if (! isset($_SESSION['user_id'])){
                        header("Location: http://www.example.com/login.php");
                        return;
            }
            $message_array = array();
            $array_index = 0;
            $user_id = intval($_SESSION['user_id']);
$result = mysql_query("SELECT message_id, title FROM Messages where to =       {$user_id}");
            while($row = mysql_fetch_array($result)){
                        $message_array[$array_index] = $result['message_id'];
                        print "<a href='./read_message.php?id={$array_index}'> {$result['title']}";
                        print "</a><br>";
                        $array_index += 1;
            }
            $_SESSION['message_array'] = $message_array;
            return;

function read_message(){
            /*Redirect users if they are not logged in*/
            if (! isset($_SESSION['user_id'])){
                        header("Location: http://www.example.com/login.php");
                        return;
            }
            if  (!isset($_SESSION['message_array'])) return;

            $message_array = $_SESSION['message_array'];
            $fake_id = intval($_GET['id']);
            if (!isset($message_array[$fake_id])) return;

            $message_id = $message_array[$fake_id];
/* The rest of the code is the same  with the original function*/
            [...]
?>
```

References

- "Financial company heavies researcher for reporting vulnerability" http://www.theregister.co.uk/2011/10/14/first_state_super_shoots_messenger/ [1]
- OWASP TOP 10, *https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project* [2]
- Nick Nikiforakis, Marco Balduzzi, Steven Van Acker, Wouter Joosen and Davide Balzarotti, "Exposing the lack of Privacy in File Hosting Services", in the Proceedings of the 4th USENIX Workshop of Large-scale Exploits and Emergent Threats, 2011 [3]
- Bruce Schneier, "The Security Mindset", *http://www.schneier.com/blog/archives/2008/03/the_security_mi_1.html* [4]

Authorization

If a Web application knows which objects can be accessed by which users then the Web programmer can straightforwardly integrate this knowledge into the code. For example, the messages database table contains a From column (who sent any given message) and a To column (who is the recipient of any given message). Thus anytime that a user requests to read a specific message, the programmer can verify whether the user was the recipient or the sender of that message. Since we have limited ourselves in the above examples to only the user's Inbox, the SQL query in function `read_message()` can be modified as shown in Listing 3.

As you can see in the modified lines, the SQL query now asks for all data given a specific `message_id` and a specific `user_id`. The `user_id` variable is out of the user's control thus even if a malicious user changes the `message_id` to a different value, the SQL query will not match any row in the table and thus return no results. Additional code could be added at this point to warn the administrator that someone is attempting to access resources that are owned by another.

Indirection Layer

Another way of tackling this problem is to add an extra level of indirection between the references that the user sees (and is in control of) and the references used in the back-end of the application. This technique is useful when the Web application programmer wishes to hide the implementation details from users. It is best explained with an example. In their vulnerable implementation, the function `get_message_titles()` reads the message identifiers from the database and gives the appropriate ones to the user. Then the function `read_message()` reads the message identifier from the user and queries the database directly using the user-provided value. The above protocol could be modified so that `get_message_titles()` transforms the database values before giving them to the user and then `read_message()` transforms them back before querying the database. For instance, `get_message_titles()` and `read_message()` can be re-written as shown in Listing 4.

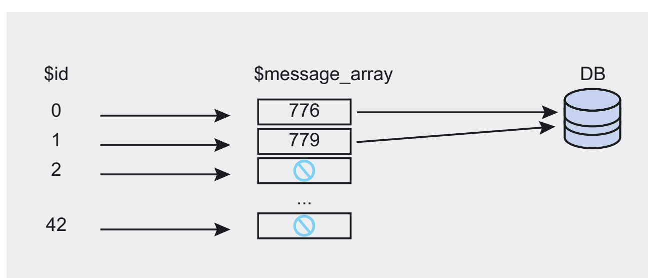

Figure 1. *The effects of adding an extra layer of indirection for user with user_id equal to 11*

Note the changes done to the code. Before querying the database we create an array that will hold our future transformations. For every valid `message_id` returned by the database, a new row is created in our array, that connects an incremented counter (`$array_index`) with that `message_id`. Each link given back to the user, instead of containing the `message_id`, holds the `array_index` with which the `message_id` was associated. Finally, the `message_array` is stored in the user's session so that it can be retrieved by `read_message()`. The `read_message()` simply reverses the procedure. What is interesting now is that the user no longer receives the actual values that the database contains but rather a set of values that correspond (at the server-side) with the actual values. Thus, the user with `user_id` equal to 11 will see the following links generated by `get_message_titles()`:

- *http://example.com/read_message.php?id=0*
- *http://example.com/read_message.php?id=1*

Even if a user changed the id to a new value, say `id=2`, there is no `2` in the `$message _ array` read by `read _ message()` thus the query will fail. Figure 1, shows this procedure graphically.

Randomized identifiers

There are cases in which for some reason, the Web application has no knowledge of which resource belongs to which user. While certainly this is not the case in our vulnerable Web application, there are plenty of real-life examples. One of them is a *File Hosting Service* (FHS), a Web service which a user can utilize to exchange large files with friends. In the usual scenario, the user visits a FHS, uploads a file and receives a link to the uploaded file. The link contains a unique identifier that the service generates

and associates with that specific file. The FHS user can then proceed to share this link with any friends or colleagues. These services typically do not require users to create accounts, thus they do not have the means to *remember* who uploaded a file nor can they know with whom the user shared the link to the uploaded file. In these cases, the only protection available is to use randomized identifiers so that users can not easily guess the identifiers of other resources. Thus, a FHS can assume that if one has a valid URL to a file that they should be given access to it simply because it should be *impossible* to recreate it [3]. In our Web application example where the `message_id` is an auto-incrementing number, a user can find the previous identifier simply by subtracting one from the current identifier. If instead, each message identifier was a random identifier, the user could no longer easily guess the identifiers of other users' messages. The only way to figure out other valid identifiers is through brute-force methods which usually take a great amount of time, hopefully much greater than the time resources of the attacker.

Conclusion

The evolution of Web sites, from simple static HTML pages to full-blown dynamic Web applications marked a new era, not only for users but also for attackers. Today, any Web application programmer has to keep in mind tens of ways that a Web application can fall victim to attackers and code defensively to avoid exploitations. In this article, you learned about the Direct Object Reference vulnerability that's easy to introduce in code and even easier to exploit. We saw an example of this vulnerability, the methods to attack it and finally three ways that a Web programmer can protect his applications from it.

NICK NIKIFORAKIS

Nick Nikiforakis is currently a PhD student at the Katholieke Universiteit of Leuven in Belgium. He is interested in all aspects of computer security, but he mostly focuses on Web security and on the protection and exploitation of low-level vulnerabilities in native applications. In the past, Nick has presented his work in well-known academic conferences (e.g. Usenix LEET and EuroSEC), local OWASP chapters, AppSecDev and BeNeLux OWASP events as well as top European hacking conferences (CONFidence, BruCON and AthCon). All of his work can be found online at http://www.securitee.org

HAKIN9

Join our
Exclusive and Pro club
and get:

Hakin9 one year subscription

Full page advertisement in
Hakin9 every month!

Information about your company
send to over 100,000
Hakin9 readers!

More information at
en@hakin9.org