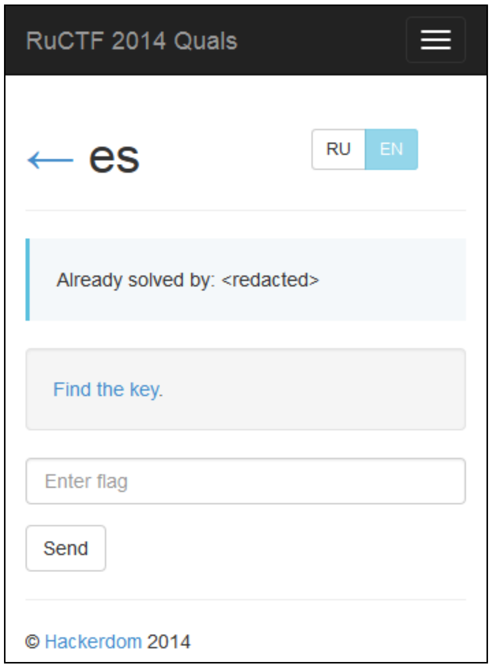
Munching Away at ElasticShortener

An overview of solving a cookie-based CTF web challenge

March 14, 2014
BY JASON COOPER, KNIGHTSEC

For this challenge belonging to the 2014 RuCTF Qualifiers, we are provided with a hyperlink that takes us to a simplified web application. The web challenge for 200 points is titled ES, which we soon learn stands for the web application's title, ElasticShortener.



Already solved by: <redacted>

Find the key.

Enter flag

Send

© Hackerdom 2014

Figure 1. *We are provided with a name-based link to the system hosting the ElasticShortener web application.*

ElasticShortener is a stripped-down link shortener that allows users to provide an extended or lengthy web address and in return receive a relatively short, easy-to-share web address, similar to services Bitly and Google URL Shortener.

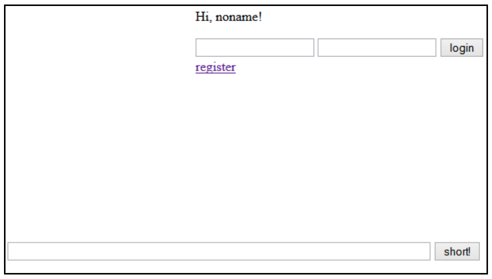
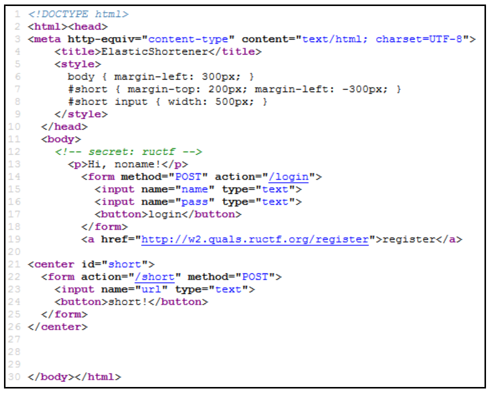


Figure 2. *The simple interface of ElasticShortener. The interface includes a user login region as well as a link submission region.*

At first glance, we see three input fields. Users can login, register, or create a shortened link immediately. Well, before we get too ahead of ourselves, we need to think about possible vulnerabilities—being a CTF challenge, we know there is some flaw with this web application. So many questions come to mind: Are we trying to access a specific user's account? How would we go about that? Could this be an SQL injection challenge?

Let us check out the HTML source code. This is often a good place to get started.



```html
<!DOCTYPE html>
<html><head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <title>ElasticShortener</title>
    <style>
      body { margin-left: 300px; }
      #short { margin-top: 200px; margin-left: -300px; }
      #short input { width: 500px; }
    </style>
  </head>
  <body>
    <!-- secret: ructf -->
      <p>Hi, noname!</p>
        <form method="POST" action="/login">
          <input name="name" type="text">
          <input name="pass" type="text">
          <button>login</button>
        </form>
        <a href="http://w2.quals.ructf.org/register">register</a>

<center id="short">
  <form action="/short" method="POST">
    <input name="url" type="text">
    <button>short!</button>
  </form>
</center>



</body></html>
```

Figure 3. *We examine the source code of the main web page.*

Surprise! The source code reveals something of great interest surrounded by comment tags: *secret: ructf*. What can we do with this? It is obviously some sort of hint. Let us keep it in mind; we might need it later.

We continue to examine the few pages of the web site and finally register ourselves a test user, *user000*. We authenticate as the user and take a look around the authenticated side. Screenshots of the registration page can be viewed under *Supplemental Screenshots*.

We quickly determine that the challenge does not involve any database injection. Instead, upon further examination, we come across the HTTP response headers:

```
HTTP/1.1 200 OK
Server: nginx/1.2.1
Date: Mon, 10 Mar 2014 01:30:55 GMT
Content-Type: text/html;charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie:
mojolicious=eyJuYW1lIjoidXNlcjAwMCIsI
mV4cGlyZXMiOjEzOTQ0MTg2NTB9--
d3c30da243daa9527e684ef4ae89b492eeb05
b1f; expires=Mon, 10 Mar 2014
02:30:50 GMT; path=/; HttpOnly
Content-Encoding: gzip
```

We are especially interested in the *Set-Cookie* header. It definitely looks like some base 64 and some sort of checksum—perhaps, SHA-1.

We find that the first-half of the cookie is indeed base 64 encoded; we decode it:

```
{"name":"user000","expires":139441865
0}
```

The cookie identifies the username of the active user. However, there is still a checksum. This checksum could be, and likely is, a mechanism that prevents us from tampering with the cookie.

The name of the cookie, *mojolicious*, looks worth looking into. We learn that Mojolicious (http://mojolicio.us/) is actually a Perl web framework. Upon further investigation, we learn that Mojolicious uses a cookie signature—that is the checksum we see appended to base 64. We find out that the signatures implemented by Mojolicious are composed of HMAC-MD5. The signature is calculated using the value of the cookie and a secret—oh, a *secret*—that is known by the server. Well, we know that the signature is definitely not an MD5 checksum, though; let us use SHA-1 instead.

So, we see that there are two hyphens that separate the encoded session information and the SHA-1 signature. Through various attempts, we discover that the equality symbol(s) appended to the base 64 encoded strings are replaced with hyphens as well because equality symbols are not valid characters for the cookie. For instance, a Mojolicious cookie with a base 64 value containing two trailing equality symbols would consist of a total of four hyphens: two for the base 64 equality symbol replacement and another two as the session-signature separator. Now, we want to verify our theory by regenerating the signature on our end, using *ructf*, as given in the HTML source code, as the secret:

```
HMAC-SHA-1(
    "eyJuYW1lIjoidXNlcjAwMCIsImV4cGly
         ZXMiOjEzOTQ0MTg2NTB9",
    "ructf"
) -> d3c30da243daa9527e684ef4ae89b492
         eeb05b1f
```

It is confirmed. Does this mean that we can simply change the value for the *name* property in the cookie to become another user? Can it be that simple? We want to try it…for possible user *admin*—what else? We create ourselves a cookie step-by-step:

```
{"name":"user000","expires":139441865
0}
```

```
{"name":"admin","expires":1394418650}
```

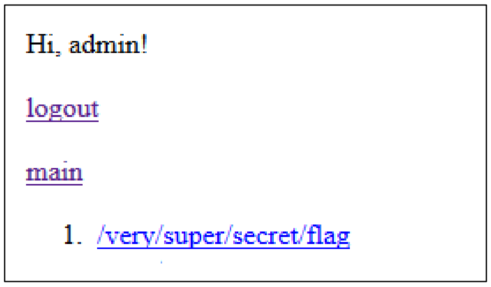
```
eyJuYW1lIjoiYWRtaW4iLCJleHBpcmVzIjoxM
zk0NDE4NjUwfQ==
```

```
eyJuYW1lIjoiYWRtaW4iLCJleHBpcmVzIjoxM
zk0NDE4NjUwfQ--
```

```
eyJuYW1lIjoiYWRtaW4iLCJleHBpcmVzIjoxM
zk0NDE4NjUwfQ----<HMAC-SHA-1("eyJuYW1
lIjoiYWRtaW4iLCJleHBpcmVzIjoxMzk0NDE4
NjUwfQ--", "ructf")>
```

```
eyJuYW1lIjoiYWRtaW4iLCJleHBpcmVzIjoxM
zk0NDE4NjUwfQ----891653312379843c58d2
d2c0f876bd830134d8b8
```

Now, we set the *mojolicious* cookie to the value we generated above using a base 64 string tailored for the *admin* user and signature using the HMAC variant of the SHA-1 algorithm. We refresh the web page:


Hi, admin!

logout

main

1. /very/super/secret/flag

Figure 4. *Once accessing admin's account, we see a very super secret flag link listed.*

There is a shortened link for us. We follow the link and locate the flag, earning us 200 points.

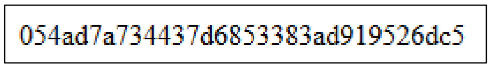
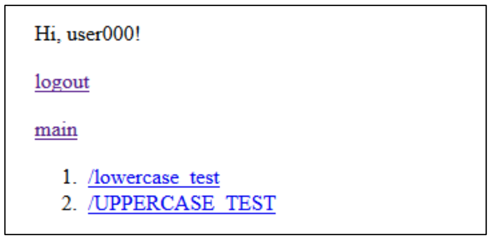

054ad7a734437d6853383ad919526dc5

Figure 5. *We have located the flag, thus solving the challenge.*

From this challenge, we learn how important it is for web developers to program securely. While an attempt was made to prevent the cookie, which ultimately controlled which user we were authenticated as, from being tampered with, we still found a way in. Keys are used to further secure data and obviously should not be left anywhere that is publicly accessible—even while or for purposes of debugging. Keys used for encryption, salting, or similar purposes should not be weak in strength or predictable. Should the key not have been so easily available, the application would be much more secure from the aspect of the challenge's goals. Keeping in mind what we have learned through this challenge, we can better protect our data and infrastructure from unauthorized access. ■

Bugs of Significance

Shortlink Associations. During our journey of solving this challenge, we came across a bug in the web application. If we authenticate as *user000*, for instance, we can create a shortened link—just as we could without being authenticated: *lowercase_test*. If we are authenticated, the shortened link is associated with our user. We then proceed to logout and authenticate as user *USER000*. Notice the only difference is the case. We create a shortened link: *UPPERCASE_TEST*. When we navigate to the page that lists our shortened links for this *USER000* account, we receive an empty list. However, upon accessing the *user000* account, we find both entries: *lowercase_test* and *UPPERCASE_TEST*.


Hi, user000!

logout

main

1. /lowercase_test
2. /UPPERCASE_TEST

Figure 6. *Identification of a bug in the web application. All associations with user accounts are linked to the lowercase version of the username.*

Our thoughts are that either the username associated with a shortened link is passed through a to-lowercase function before being stored in the database or queries to the database are being performed without sensitivity to case. In addition, we learn that this same effect occurs with invalid characters, such as symbols. Our first idea seems more reasonable. For example, user *user000-* can create shortened links that are associated with and appear only in the account for *user000*. We also find that *user0-00* can create shortened links that are associated with and appear only in the account for *user0*. This is an indication that some sort of right-end trimming is occurring—that is, trim everything after and including the first invalid character, which is a hyphen in this case. This bug, as a whole, could have been entirely innocent or even implemented in an intentional manner to distract us from the real solution.

Authentication as Invalid User. During this adventure, we determined that we can associate shortened links with an account that simply does not exist—at least, not until we make that association. For instance, we login to ElasticShortener with a username of *knightsec* and an empty password. Authentication succeeds. We can now create shortened links associated with account *knightsec*. This, however, does not work when we attempt to authenticate as a non-existent user with some arbitrary password.

So, we can continuously authenticate as *knightsec* using an empty password. If we attempt to officially register the account through the registration page, choosing to provide a password or entirely omit it, we can successfully register the account, even though there are active shortened links associated with the username. Furthermore, if we attempt to register the account again, we get the lovely Mojolicious error message, which is depicted under *Supplemental Screenshots*, and do not succeed.

There are a number of possible reasons as to why we are able to achieve this.

Supplemental Screenshots

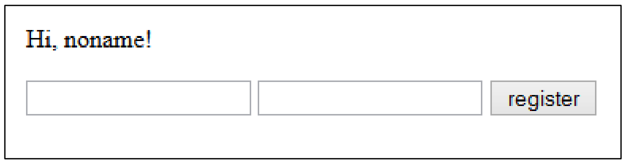
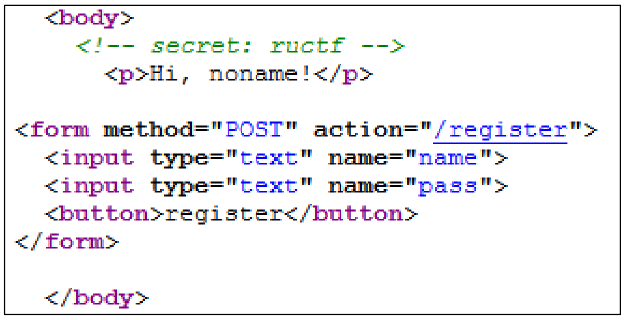

Hi, noname!

Figure 7. *The registration web page offers fields for a username and password.*


```
  <body>
    <!-- secret: ructf -->
      <p>Hi, noname!</p>

<form method="POST" action="/register">
  <input type="text" name="name">
  <input type="text" name="pass">
  <button>register</button>
</form>

  </body>
```

Figure 8. *The body source code of the registration page also offers the secret.*

Figure 9. *The wonderful error message for Mojolicious.*