Land of Magical Cats

Yet another security blog. Only difference is, I'm a cat.

Wednesday, August 29, 2012

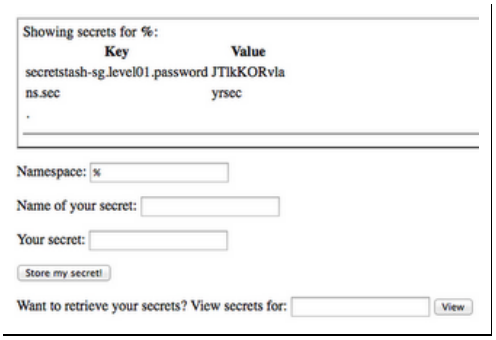
Stripe CTF Writeup

This week, I participated in Stripe CTF. What's different about this CTF is that it focused solely on web vulnerabilities. I'll be going over the challenges and my solutions.

Level 0

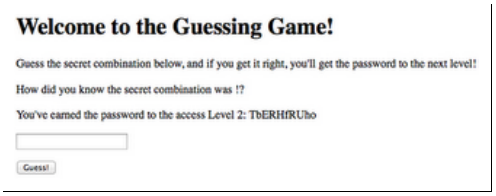
Level 0 was probably as basic as SQL injections get. So basic that it's not even really "injection" so much as "unexpected behavior." The application takes input, and performs a SELECT using LIKE. We can specify a wildcard statement for sql, %, and see everything in the table.

```
app.get('/*', function(req, res) {
  var namespace = req.param('namespace');

  if (namespace) {
    var query = 'SELECT * FROM secrets WHERE key LIKE ? || ".%"';
    db.all(query, namespace, function(err, secrets) {
      if (err) throw err;

      renderPage(res, {namespace: namespace, secrets: secrets});
    });
  } else {
    renderPage(res, {});
  }
});
```



Level 1

This level basically reintroduces an old issue with PHP - register_globals - but with a more obvious function call, extract(). What this function call does is take GET parameters and store each one in its own variable in the global scope. This allows us to overwrite any variables that have been set before the extract(). In our case, we will be overwriting $filename and point it to a file that doesn't exist. We can now compare an empty "attempt" variable to the contents of a nonexistent file, and read the password.



Level 2

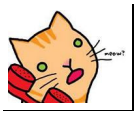
This is another basic vulnerability, which can be exploited like any other upload form that doesn't check files. All we need to do is upload a PHP file that reads the contents of the password file to us. In later levels, the code execution here becomes really helpful.

Level 3

Another straight forward SQL injection. This isn't the typical SQL injection, however.

Blog Archive

About Me



Cat

View my complete profile

When I first approached this challenge, I noticed right away that I could read data using a blind SQL injection. Half way through retrieving the hash, I realized I'd be wasting my time. First of all, that would mean having to brute force a hash, which would have made for a really lame challenge. Secondly, the secret isn't even stored in the database.

I then realized - if I'm not cracking the hash, can I replace the hash that they're checking? The answer is yes. My final query was as follows:

> bob' union select "3","4fc82b26aecb47d2868c4efbe3581732a3e7cbcc6c2efb32062c08170a05eeb8","1" from users limit 1,2 --

This set my id to 3, and gave me a hash of a password I knew. ("1")

Level 4

This was a weird level. You're given a site where your password is displayed to everyone who you send "karma". The goal of this challenge is to get karma_fountain's password, presumably through it sending you karma.

I took the wrong approach at first. Examining the cookies, I noticed a base64'd blob of text, one of which contained my username. I also noticed logging out didn't prevent your cookie from working, so I assumed modifying the cookie would be the challenge. I quickly realized the sha1 hash in the cookie was the hmac and that I was definitely going the wrong way.

Eventually after playing around with the inputs a bit, I noticed my password could have XSS. Normally, I wouldn't care, but here, our password gets shown to whoever we want! This would be the best way to get karma_fountain to send us something. I started off using the typical script tag containing a url with my payload. After a few minutes of no traffic to that page, I noticed in the description that the firewall permits connections to only stripe-ctf servers. I just changed my payload from pointing to an external JS file, to actually containing the code. This worked, and I managed to get karma_fountain to send me some karma. And its password.

One really big hint was that jquery was on every page, though it was never used.

[insert screenshot]

Level 5

This was a pretty fun level, considering the method I used was not the expected way. I'll go over the way I used since there will probably be writeups of the "correct" method anyway. The method has since been patched. (People probably reported it...)

This level use Rack, a session management library for Sinatra, a webserver written in ruby. This is the same setup used in level 4. Except here, I managed to trigger some very revealing Tracebacks. While playing around with inputs in an attempt to bypass authentication, I noticed certain symbols (in my case, "$") would cause Sinatra to throw a URI::InvalidURIError exception.

This exception was pretty nice. Not only did it help me understand was going on, it also showed me the secret used to sign the cookies! This secret is typically stored privately (in our case, entropy.dat) but the traceback shows it anyway.

Simply enough, I used this opportunity to modify my cookie such that "auth_host" pointed to a level5 stripe-ctf server. I then signed my cookie using the newly available secret (which was in some weird ruby "octal" format) and got the password for the next challenge. I later solved this challenge via the intended method but it wasn't as satisfying...

Level 6

Meh. This was really the same as level 4, just with more restrictions. This time, the password was on a different page, anything with quotes would trigger a server error, and the vulnerable parameter was the username - not the password. I solved this level using basic javascript obfuscation, namely String.fromCharCode and regex*.

Also worth noting, the message fields were not sanitized in the server side. This meant viewing the source would show that XSS is possible through messages, but in reality it is not because there is a javascript function which uses jQuery to encode all HTML.

* Did you know that /etc/.source is equal to "etc" as a string? Easy to use and no need to memorize ASCII character codes.

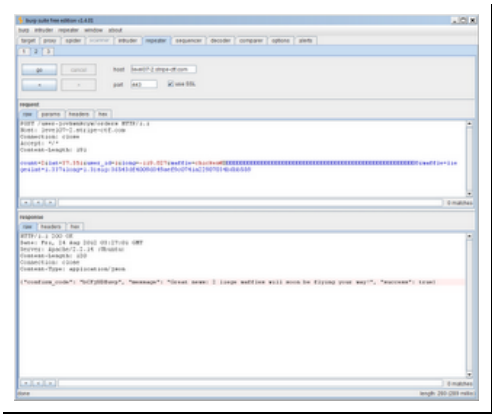
Note: The attack I used in the previous level would have worked here as well.

Level 7

This level is really easy if you know the type of vulnerability involved. The first clue was that you could view anyone's API access logs, whether or not you were logged in as them. The second clue was the API signing method - plain old sha1. If you know anything about using a MAC to sign requests, then you'll recognize this is a classic hash length extension vulnerability. A hash length extension attack basically allows us to use a known value and its hash to append our own data and calculate a new, valid hash. A long time ago, flickr had this vulnerability in its API when it used md5 to sign requests. [PDF]

After reading the source, it's clear that if there are parameters with the same name, the last one takes priority. This is the key detail in making our length extension attack work, as we can only append data to the data we already have. The first step is to obtain a working request made by someone who can purchase premium waffles.

To test this out, we can replay this request and confirm that we can indeed purchase premium waffles. However, we want a **liege** waffle. We can tack on "&waffle=liege&lat=1&long=1" to the request, and now we have a liege waffle to be sent to 1, 1. But wait! Our signature doesn't match any more. Using a length extension attack, we can use the existing signature and request body, we can create a new request body with a new signature, all *without* the owner's secret. There are many scripts available to perform length extension attacks on sha1 so I will leave that out. We now have a liege waffle order, with a signature that matches. We win!



One problem I ran into was url encoding. I had url encoded the non printable bytes in my extended hash, which kept causing the server to throw errors. After decoding these bytes in Burp, my requests were working fine again. Another issue is that I made the assumption that the key length was 14 simply because my secret was 14 characters. Although rare, providing different length secrets would not make the challenge any harder, just slightly more time consuming. And by "more time consuming" I mean "5 more minutes."

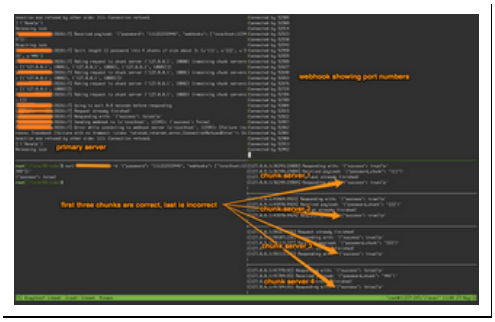
Level 8

This was a particularly tricky level. Unlike the other levels, this one actually requires you to set up a local version of the challenge or else you'd miss everything.

First, a brief description. There are five servers - one primary server, and four chunk

servers. The password (12 characters) is split into 4 chunks, and the primary server goes one by one verifying that your chunks are correct. You do not have direct access to any chunk servers, and the only output you really have is true or false. There are also "webhooks" which is just a fancy term for an http callback url, which is POSTed to if your password is correct or not. Timing attacks are hindered by calculated sleep() calls. The challenge is to retrieve the password, without having to brute force the entire keyspace, 10^12, which would take forever anyway. How can we retrieve the password in a timely manner if we can't verify our chunks separately?

Running the server, I noticed if I submitted a password, not all chunks were necessarily verified. I saw that if I submitted a password with the first two chunks correct, then the first three chunk servers would be contacted, and would respond with true true and false, respectively. The fourth chunk server was never contacted. If I submitted a password with only the first chunk correct, then the first two chunk servers would be contacted, and would respond with true and false, respectively. Now we know how to verify parts of our password. But wait - this doesn't work in production because we don't have access to the debug logs which show which chunk servers were contacted. A timing attack would have worked, if it were not for the sleep calls.



What we need are webhooks. Webhooks don't receive anything but true or false, but we do have some useful data - remote port. If we monitor the value of remote port of consecutive requests, we can actually determine how many requests the server had made in the mean time! More specifically, we can see how many chunk server requests were made. Earlier, I pointed out that a password with 2/4 correct chunks would mean 3 requests. With this data, we can conclude that if we have 2/4 correct chunks in our password, the next remote port in our webhook should increase by 3! By being able to confirm one chunk at a time through measuring the number of requests made, we can cut down a typical full keyspace brute force of 10^12 down to 4*10^3! That's a 250000000x decrease!!!!!

Now that we know what we want to do, we need to get ssh access. If you recall, we have code execution on a level2 machine from earlier. We were also told there is an sshd running on the machine. If you check out the sshd_config, you'll see the only way to log in is via public key authentication. To use this, we have to generate an rsa keypair using ssh-keygen, and place the public key into the ~/.ssh/authorized_hosts file.

Once on the machine, we can use a script to brute force chunks while monitoring the remote port. This becomes difficult once you have hundreds of people on the same machine. Not impossible though - we just need to be clever. As I stated earlier, 1 correct chunk means an increase of 3. With many people connecting to the server, a correct (or incorrect) chunk may lead to an even higher increase. Many people performed statistics, used histograms, etc to determine if their chunk was correct. I instead relied on something that was guaranteed - if our first chunk leads to only 2 requests being made, then we can actually guarantee the chunk was incorrect. Anything greater, and we risk false positives due to other users. I ran my code multiple times, eliminating any value that is guaranteed to be wrong, until there was just one left. I did this for the first three chunks. For the final chunk, I just submitted my code over and over until I saw "success: true".

To sum it all up:

1. Get SSH access by uploading your public key.
2. Brute force one chunk at a time, by measuring the number of total requests made to chunk servers.
3. Brute force the final chunk using the regular JSON endpoint.
4. ???
5. Profit!

My code for this solution can be found on github.

Conclusion

This was an interesting CTF. I liked the challenges because a strong background in security was not necessary to participate - anyone with decent programming skills could beat most of these challenges. I think that makes for a CTF that ends up being too easy, however. Most of the challenges were unrealistic, and the only reason the last 2 challenges were even included in this web CTF is because HTTP is the underlying protocols. Still a decent CTF, and as always, there are still things to learn.

Something Extra

Nothing particularly special, but I had found an XSS in the web interface. I was not the only one apparently, as the bug had already been patched when I reported it. Basically the regex used to validate the profile URL was really loose - anything was allowed as long as it contained a URL in it. I was able to submit a url as

```
javascript:alert('anything'/* http://www.url.com/ */)
```

and it would work. If you clicked my username, the alert would have gone off.

Posted by Cat at 2:59 PM

2 comments:

Kyle Willmon August 29, 2012 7:53 PM

The chunk-wise brute force of Level 8 is 4*10^3, not 4*10^4 (I didn't check the math on your reduction factor, but it might need adjustment too)

Reply

Replies

Cat August 29, 2012 8:44 PM

You are right. I've updated it.

Reply

Subscribe to: Post Comments (Atom)