

fuzzy notepad

this is really hard to write on

- [RSS](#)

- [Blog](#)
- [Archives](#)

Stripe CTF 2.0

Aug 29th, 2012

This is [a thing I did](#). It was a [cracking contest](#) held by [Stripe](#) (who run a pretty neat service, btw), and it ended today. I was third to beat level 7 and twentieth to beat level 8, so here is the tale of how I came upon the solutions.

I haven't reproduced the entirety of each puzzle below, because that would suck, but if you're lucky maybe you can still [sign up](#) and follow along. If not, Stripe has promised to release the puzzles (and solutions) tomorrow. I think.

Level 0: Secret Safe

This one was written in JavaScript and implemented really simple security-by-obscurity storage: you provide a namespace, and it either stores data for you under some key or tells you all keys and data stored under that namespace.

This was the intro level, so the solution was pretty obvious, but actually less obvious than I expected for a level called "0". The offending line is:

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

The key is actually stored as `namespace.key`. So the "exploit" is just to enter % as the namespace, and voilà, every secret is revealed. The db doesn't know the difference between a % in your literal query and a % in your bound parameter, so any key containing a period (i.e., all of them) is selected. I suppose you'd call this LIKE injection.

It's not vanilla SQL injection, but it relies on the same principle as all injections: dropping arbitrary data blindly into a structured format.

Level 1: Guessing Game

PHP this time, and a similar idea, really. Enter the password, receive the data, which is stored in a file.

This one relied on recognizing a hilariously awful standard PHP function:

```
1      $filename = 'secret-combination.txt';
2      extract($_GET);
```

`extract()` takes all the keys of a hash and dumps them into your local namespace, as variables. The line above implements the infamous `register_globals`.

That's just a low blow, Stripe. :)

Solution, then, is to use a query string of `?attempt=&filename=junk`. The file won't exist, PHP will cheerfully read it and return something falsey, and that'll compare equal to the empty string.

The vulnerability here is called "PHP". Yeah, whatever, PHP runs Facebook, I don't care, go away.

Level 2: Social Network

PHP again. Now we're getting into exploits I have, tragically, actually seen in the wild. The password is still stored in a file (that cannot be read directly), but now the only real entry point to the program is uploading an avatar.

So. Yeah.

```
my_avatar.php
1 <?php echo file_get_contents('../password.txt');
```

Upload that as your avatar and visit the URL. PHP injection.

Props to level 1 for reminding me that `file_get_contents` exists.

Level 3: Secret Vault

Level 2 was the last of the PHP puzzles. Now we're getting serious. This one is a [Flask](#) app—i.e., Python.

This is a sequel to the Secret Safe, but it's the same general idea, except that namespaces and keys have been replaced with genuine usernames and passwords.

A glance over the code, and something stands out:

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

This, then, is the obligatory SQL injection puzzle.

I am shamed to admit I took a few minutes on this—way longer than I should have. I tried to trick SQLite into running multiple statements here, or embedding an `INSERT/UPDATE` inside this `SELECT`. Those don't work, which is good.

I didn't get it until I rephrased the question as: how can I trick this query into retrieving data that's *not* really from the `users` table?

Oh, right. New username:

```
' UNION SELECT (select id from users where username = 'bob'), '2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae', '
```

Which produces the final query:

```
1 SELECT id, password_hash, salt FROM users WHERE username = ''
2 UNION
3 SELECT (select id from users where username = 'bob'), '2c26b46b68ffc68ff99b453c1d30413413422d706483bfa0f98a5e886266e7ae', ''
4 LIMIT 1
```

There are no usernames of `''`, so the first `SELECT` doesn't find anything. The second one pretends to be `bob`'s id, a constructed SHA1 hash, and an empty salt. Result is a single row that tricks the app into thinking my password of "foo" is correct.

Level 4: Karma Trader

A Ruby app, built on Sinatra. I can't beat the original description:

The Karma Trader is the world's best way to reward people for good deeds. 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.

This should sound more ludicrous than it is, but I genuinely believe there are people in the world who would think this is a great idea.

The gimmick here is that an existing user, `karma_fountain`, has both unlimited karma and also the password to level 5 as his own password. To get it, obviously I need to make it look like *he* has give *me* karma.

This was hard. Because there's a catch.

The database access all uses Sequel (a little db library) and bound params, so there's no SQL injection. There are no silly oversights like `extract`. I can create as many users as I want, but I can only make them send karma to each other; I can't make anything that looks like `karma_fountain`.

There are no obvious exploits. I can't fake anything on the server.

I was going slightly crazy until I noticed some hints.

Perhaps you read Encyclopedia Brown books, or similar kid mysteries. I loved those. I started tearing through them once I noticed that most of the solutions revolved around some minor detail that received undue importance in the story, like just how many quarts of water were given to dogs before a race.

And so it was here. Quite a lot of code, relative to the size of this dinky app, is dedicated to updating and displaying a `last_active` timestamp. That's utterly pointless; I'm the only one using this thing, and I know when I've been active.

But wait! From the app itself:

If you're anything like **karma_fountain**, you'll find yourself logging in every minute to see what new and exciting developments are afoot on the platform.

Below this is a list of all registered users, and their last-active timestamps.

`karma_fountain` did indeed have a very recent timestamp.

I refreshed the page.

The timestamp advanced by precisely one minute.

Brilliant.

That made the solution obvious: I created a new account with a password of:

```
<form id="x" method="post" action="transfer"><input type="hidden" name="to" value="eevee2"><input type="hidden" name="amount" value="100000"></form><scri
```

Then I sent `karma_fountain` a single karma. One minute later, the bot hit the page again, dutifully executed my XSS, and sent me ten thousand karma and the next password.

(The bot is still going; as of this writing i have 60000499 karma.)

Level 5: Domain Authenticator

Ruby and Sinatra again. The app implements a federated identity system: you provide a username, password, and URL. It posts your username and

password to the URL, and if the response is `AUTHENTICATED`, it considers you as logged in. If you log in with a URL hosted on a `level05-*.stripe-ctf.com` machine, it'll also tell you the password for level 6.

The trick, of course, is that nothing running on the machine actually *implements* this protocol, and in fact there is no implemented provided at all.

There is one more hint: the production app can only make requests to `*.stripe-ctf.com` machines, *but* someone “forgot” to firewall off the ports on the level 2 machines. You know, those machines that let me upload and run any code I want. So, that's nice.

This one was *good*. I actually didn't really solve it. I found most of the solution, but then I accidentally tripped over something else that was even better.

My solution

I entered a URL with letters in the port. The app crashed and showed me a generic Rack error page, with debugging information.

Now, much like Flask, Rack (and thus Sinatra) handles sessions by default by serializing a hash, tacking on a signature, and storing the whole shebang in a cookie. The upside is that this doesn't require any server-side setup or maintenance whatsoever. The downside is that this is fucking bozotic, because it let me do the following.

You see, the Rack debug page *exposes the key used to sign session cookies*.

With that, it wasn't particularly difficult to construct a fake cookie that claimed I had already been authenticated by `level05-2.stripe-ctf.com`. (I actually ran into a bit of trouble here: I took the original cookie from Firefox, but tried to inject it in Chromium, which already had a cookie editor installed. It took me a few minutes to notice that Rack also tracks your user agent in your cookie, and ignores it if the cookie and browser don't match. So, minor props there. Then I found out that Firebug can edit cookies; problem solved.)

[@kevinlange](#) later pointed out to me that this actually works for all three Rack puzzles: 4, 5, and 6. I believe he informed Stripe of the unintended exploit, and it was fixed the next day: errors now serve the generic Apache 500 page.

I wasn't quite sure whether this was intentional or not; after all, it was a legitimate exploit, but it didn't require mucking with level 2 at all. But I had access to level 6, so whatever.

The real solution

The username and password are, of course, an irrelevant distraction. There's no list of usernames and passwords anywhere, so they can't possibly matter.

The real puzzle here is tricking the app into thinking `level05` has verified you. And since there's only one thing running on `level05`, the puzzle is tricking the app into thinking *it* has verified you.

The first step is to try feeding the app to itself as the URL. That produces:

```
An unknown error occurred while requesting https://level05-2.stripe-ctf.com/user-abcdefghijklj/: 500 Internal Server Error
```

The app is calling itself, but the second call has no pingback URL, so it gets confused and dies. Hmm.

Lucky for me, the app examines `params`, which is a combined hash of *both* `GET` and `POST` data. So I can make it not crash, at least, by feeding it `https://level05-2.stripe-ctf.com/user-vmcscdesvlp/?pingback=www.google.com`.

```
Remote server responded with: Host not allowed: www.google.com (allowed authentication hosts are /\.stripe-ctf\.com$/). Unable to authenticate as foo@lev
```

A valiant start. I can keep this loop going as long as I please, but without an actual authentication somewhere, I won't get very far. And that's where the level 2 servers come in.

```
pingback.php
1 AUTHENTICATED
```

I don't know why they mentioned “high ports” not being firewalled off; the above is all you need. There's already an HTTP server running, after all.

Upload this guy, try to authenticate as `https://level02-2.stripe-ctf.com/user-zlbgqlkyoe/uploads/pingback.php`, and I get:

```
Remote server responded with: AUTHENTICATED. Authenticated as foo@level02-2.stripe-ctf.com!
```

Wrong server, but getting there.

This is actually as far as I got before accidentally breaking Rack, but the rest isn't too difficult. The actual check for authentication uses the following regex:

```
1 body =~ /^[^\\w]AUTHENTICATED[^^\\w]*$/
```

(That's why I have an extra space in `pingback.php`: to match the weird `[^^\\w]` atom. Should be `\\b`, but, whatever.)

Trouble is brewing: how can I trick the `level05` app into putting the word “AUTHENTICATED” at the end of a response? It always prints trailing literal text!

Well, it's an uncommonly known quirk (and hilarious potential source of [exploits](#)—like this one!) of Ruby regular expressions that they are treated as *multiline by default*. That means `^` and `$` don't match the beginning or end of a string; they match the beginning or end of a *line*.

Now the solution is easy peasy. In fact, I don't even have to do anything, because my `pingback.php` already contains a trailing newline. The final URL is `https://level05-2.stripe-ctf.com/user-vmcscdesvlp/?pingback=https://level02-2.stripe-ctf.com/user-zlbgqlkyoe/uploads/pingback.php` and we're off to the races.

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

The newline became a space in HTML land, of course.

Back to the main page, and the password is revealed.

Level 6: Streamer

The last of the Ruby/Sinatra puzzles. This is a little “stream of posts” app, built with Bootstrap and jQuery. Once again, I have an automated friend, except now he taunts me over time:

Streamer is *soo* secure

Yes, we’ll see about that.

Again, his password is the password to the next level. I’m told in advance that his password contains some number of quotation marks and apostrophes, and I can see from the code that any such characters anywhere in any request cause an immediate abort. Lame. I do know that his password appears on the user info page, but of course, only if you’re logged in as him.

XSS and CSRF seem to be no good here; the template is actually escaping things now. (Shame on someone for not making that the default.) But this is the first puzzle with client-side JavaScript. After some useless futzing with bogus usernames, that seems promising.

```
1      var username = "<%= @username %>";
2      var post_data = <%= @posts.to_json %>;
3
4      function escapeHTML(val) {
5          return $('<div/>').text(val).html();
6      }
7      function addPost(item) {
8          var new_element = '<tr><th>' + escapeHTML(item['user']) +
9              '</th><td><h4>' + escapeHTML(item['title']) + '</h4>' +
10              escapeHTML(item['body']) + '</td></tr>';
11          $('#posts > tbody:last').prepend(new_element);
12      }
13
14      for(var i = 0; i < post_data.length; i++) {
15          var item = post_data[i];
16          addPost(item);
17      };
```

There are only a few places user data gets put in here: the username and the posts. The username is easily breakable, but that only affects *me* here. The post elements are escaped by the hacky but correct `escapeHTML` function. So what does that leave?

It leaves JSON.

JSON is structured, yes, but it doesn’t know or care about HTML. After all, the JSON representation of `` is just `""`, and the JSON representation of `</script>` is just `"</script>"`, the one thing that can break out of an inline script tag.

So I can just write a post like this:

```
</script><script>alert("gotcha");
```

And the JS will execute.

But wait! I’m not allowed to send any data that contains quotes or apostrophes. This really *is* super secure!

Unless I make it:

```
</script><script>alert(String.fromCharCode(103, 111, 116, 99, 104, 97));
```

And that’s basically the solution. Take advantage of the provided jQuery to fetch `user_info` on my friend’s behalf, extract the password, and post it as a message. The final trick is remembering to somehow encode the password (as I’m told it also contains quotes); I just url-encoded the whole page and posted that. Easy peasy.

Level 7: WaffleCopter

Back to Python, but now we’ve moved on beyond websites: this is an API endpoint. There’s a very simple web interface that tells me my API key and shows a lot of my API requests.

I’m supposed to make a request for a privileged waffle, but requests are signed with my API key. They look like this:

```
count=1&lat=42.39561&user_id=5&long=-71.13051&waffle=dream|sig:30d0ca71b0bbe5e649628b8a7f2f88f90e17c27
```

The signature is a SHA1 hash of my API key plus the rest of the request.

So. Now what? The database is solid. There’s no other player here. Surely I’m not supposed to just crack SHA1. Surely.

Well, let’s see. The first thing I notice is that the server code does its own parsing of the query:

```
1 def parse_params(raw_params):
2     pairs = raw_params.split('&')
3     params = {}
4     for pair in pairs:
5         key, val = pair.split('=')
6         key = urllib.unquote_plus(key)
7         val = urllib.unquote_plus(val)
8         params[key] = val
9     return params
```

This is standard affair, with the lone exception that it doesn’t do anything special to handle multiple values for the same key. But that doesn’t help me; I only know how to sign my own requests, and adding more junk data to those isn’t helpful.

I flail around for a bit. Then I notice this.

```

1 @app.route('/logs/<int:id>')
2 @require_authentication
3 def logs(id):
4     rows = get_logs(id)
5     return render_template('logs.html', logs=rows)

```

This block only lets registered users see API logs, but it *doesn't check which user you are*. And sure enough, I can visit `/logs/2` and see some requests for privileged waffles! But, alas, nobody has requested the particular waffle I *need*, which rules out a replay attack. If I could take one of these requests and just run it with a different waffle...! But I cannot.

OR CAN I.

I couldn't think of anything to attack besides the hash itself. SHA1 has some theoretical weaknesses that make it simpler to attack than mere brute-forcing, but nothing practical, or that I could reasonably be expected to do here.

I *do* know that SHA1, like many hash algorithms, operates on blocks at a time. I also know that I can *add* to an existing request, because later keys overwrite previous ones.

And this was enough to shake loose an old memory: you can attack hashes by appending to the message. The googles swiftly told me that this is called a hash length extension attack (creative!). The googles also provided an [implementation for SHA1](#), saving me the trouble of figuring out how to recreate a hash function's last internal state.

All I need to know is the length of the key—which I have, because mine is the same length—and the hash of my existing message. The attack implementation tacked on a bunch of NULs until the end of the block, tacked on my extra `&waffle=liege`, and computed a new hash without ever knowing user 2's key. Chuck it at the API endpoint and I unlock level 8.

That last part ended up being surprisingly painful. I fucked around with curl for at least ten minutes trying to get it to send the request correctly, but it never worked quite right, even using `--data-binary` and a file. (I had to send *literal NULs* in the request; the server checks the signature *before* doing any url-decoding.) In the end I just hacked up `client.py` to send my forged request instead of computing a real one.

This level is a little unnerving, because the code *almost* does everything right. Ignoring all but one parameter in the presence of duplicates is reasonable and ubiquitous. SHA1 is still fairly solid. The API log leak was an oversight, but a pretty minor one. And using a hash with a "salt" as a signature sure sounds like it should be reliable. How many people have honestly heard of hash length extension (well, more have now) and will remember to think about it when writing code like this? A few very minor mistakes allowed me to break this application wide open. Imagine if this were a real ordering system; I could use someone else's request and replace the item and the delivery "address", then order whatever I wanted.

The Right Thing would be to use HMAC, which is built for exactly this purpose (verifying messages), and which demonstrates once again that **you should not ever write crypto code**, even if it's just assembling some stuff to pass to SHA1.

It's kind of funny that the code I found came from another sec challenge, though.

Level 8: PasswordDB

The final boss. I was the *third* player to get this far.

A Python app, again. But not Flask. No, not Flask, because it doesn't even have a real Web interface. It speaks HTTP, but it only takes a JSON blob and spits one back out at you.

No, no, not Flask. This is [Twisted](#).

Awesome.

Here's the deal: they wrote a service that acts as a little password vault for the final password. It listens on HTTP for a JSON blob containing a possible password, and either confirms or denies that the password is correct. As a helpful secondary feature, I can also provide a list of host/port pairs for it to ping ("webhooks") with the same response it gives to me, so this service can be used for remote authentication or something.

There is no database. There is no JavaScript, no HTML. The password isn't even stored in the primary service: it's broken into four pieces and given to four other processes, then forgotten. When the master service gets a request, it breaks the given password into four chunks, connects to the other processes one at a time via TCP, and checks that each chunk is valid. As soon as a chunk is reported invalid, the master service stops trying and reports a failure to the client.

As a clue, the description emphasizes that level 2 is not correctly firewalled, and lets slip that it even has `sshd` running. Which is nice, because the master service once again only connects to other machines in the Stripe network.

And that's all I get.

Hmmmmmm.

I freely admit I spent the rest of the afternoon and evening puzzling over this. I downloaded it, I ran it locally, I found nothing. I consulted blackhat friends and Twisted expert friends. Nothing. Even after they started sprinkling hints (explicitly mentioning the version of Twisted, emphasizing it was *not* a timing attack), the best I could find was... er... a timing attack. Which didn't work. I read over Twisted's changelog half a dozen times looking for something, anything, that could make a crack of a difference.

The webhooks were useless; they just received the same information I did. The `sshd` was useless, because it was only useful for using the webhooks. The JSON encoding and decoding was solid. I found a way to discover what ports the four side processes were bound to, but they were all bound to localhost, so that wasn't helpful. All I could think of was getting onto the actual machine and looking at the `argv` for the children, because that's the only place the password still existed. And that wasn't really keeping with the spirit of this contest.

It wasn't until late at night that the first player beat level 8, knocking me down to fourth on the [leaderboard](#). Stripe released yet another hint: they'd changed the app slightly, so logged output would include the host and port instead of just an incrementing request id. Still no idea. It was clearly reasonable to brute-force one chunk at a time—the password was a 12-digit number, so this required only 4000 guesses—but there was no way to target only a single chunk.

I slept on it.

I woke up from dreams of networking. Clearly it had to be the webhooks, but those didn't send any useful information. I joined #level8 at the suggestion of a coworker, but only a few people had figured out the trick, and they weren't saying anything helpful.

If the webhooks weren't revealing anything *directly*, it had to be a side channel attack. It absolutely, definitely wasn't a timing attack. So what else would a TCP connection reveal?

As a last resort, talking out loud helps. I was convinced the chunk servers' bound ports were still important, somehow.

```
11:22 < subleq> i am completely stuck
11:22 < eevee> i am also lacking inspiration
11:23 < hm_> does chunk_Server ports matter ?
11:23 < eevee> i have ssh and nc and webhooks and my ports but these things do not go together usefully. i am missing something
11:24 < hm_> i have chunks ports.. but i dont see how it matters in the method i m trying now
11:24 < trevis_> i would venture to say that ports dont really matter, at least with my approach
11:24 < eevee> i assume getting the ports was where the twisted version hint came from
11:24 < eevee> oh really
11:24 < trevis_> i have a working local, but remote fails pretty hard
11:25 < hm_> me too
11:25 < hm_> local find the chunks in minutes.. remote has lot of noise
11:25 < subleq> what noise?
11:25 < trevis_> remote, im barely even able to get requests out now it seems
11:25 < subleq> it can't be time, i can't measure a difference in time even locally
```

Ports, ports, ports. I better hurry if the server is already overloaded. Ports, ports, ports.

Ports...

```
11:26 < eevee> wait
11:26 < eevee> oh no
11:26 < eevee> oh no i think i get it
11:26 < eevee> oh fuck
```

I've never had IRC logs of a flash of inspiration before.

Ports.

It is common knowledge that services listen on a port. Web servers, for example, tend to listen on port 80. Most people who would describe themselves as computer-literate are, at least, dimly aware of this.

Slightly less common knowledge is that when you connect to a server, *you* use a port as well. But it's not a fixed, known port like 80; it's just some random-ass big number. When the server responds to you, it sends the response to this port, called an *ephemeral port* (because it's released as soon as the connection ends). I imagine most people who've done much networking know this, but nobody really thinks about it because it's almost always handled automatically by very low-level networking code.

I performed a quick comparison.

With the latest Twisted, 12.something, the *client* ports used to *connect* to the chunk servers are effectively random. Not useful at all. With the Twisted version Stripe specified in their very first hint, the client ports are *consecutive*.

And that, my friends, is the exploit.

Allow me to illustrate.

- I ask if the password is 111222333444.
- The master server breaks it up into four chunks: 111, 222, 333, 444.
- The master server asks the first chunk server if its chunk is 111. This requires an ephemeral port, say, 50000.
- The chunk server responds with *yes*.
- The master server asks the second chunk server if its chunk is 222. Now the ephemeral port is 50001.
- This chunk server responds with *no*.
- The master server **stops trying** and dutifully tells me *no*.

The master server tested *two chunks* in the process of checking my password, and it used up *two ports*. If I turn around and immediately check a password again, it'll use ports 50002 and 50003.

Okay, that's neat, but doesn't tell me anything, because I can't see those ports.

Oh wait I totally can! I SSHed into the level 2 server, left running a crappy little Python script that listened on a socket and printed out the *client port used to connect to it*, and tried the above again with a webhook pointed at my script.

Now the sequence of events is:

- I try 111222333444.
- The master server asks the first chunk server if its chunk is 111. This uses port 50002. Chunk server says *yes*.
- The master server asks the second chunk server if its chunk is 222. This uses port 50003. Chunk server says *no*.
- The master server stops trying and decides *no*. It contacts my webhook to say *no*. This uses port 50004, *which I can see*. Then it tells me *no* as well and stops.

Now, if I immediately try the *same password* a second time, I know two ports that were used: 50004 and 50007. That's three used for the request, which means two chunk servers were contacted, which means **the first chunk must be correct**.

Thus it's possible to bruteforce, one chunk at a time. I wrote a script to do this, sending off a request to the server, ignoring the response, and immediately listening for a webhook.

There's a slight wrinkle here, because other people are also using the same machine, and they're chewing up client ports too. So getting a delta of 3 doesn't mean the first chunk *must* be right; only that it *could* be right. It could also be wrong, but something else could have used a port in the interim. On the other hand, if the delta is 2, then the master server can *only* have contacted one chunk server and immediately given up, so the first chunk must be incorrect.

To be reasonably confident, I decided that several deltas of 3 with no deltas of 2 for the same chunk means it's probably correct.

I let it run, and gradually cracked some chunks. Only a few people had finished level 8 when I started, but an hour or so later I was barely a third done,

and the leaderboard was rapidly filling up. I ended up racing someone else on IRC for the very last slot on the first page (which was the only page at the time): I'd been sloppy and missed the correct second chunk, he'd been sloppy and done similar, and he was using Go versus my Python. For the last chunk I bruteforced against the app from my own machine, not bothering to check for ports, and I ended up running `curl` in a loop in eight terminals, each one trying a different first digit, and crossed my fingers as he rapidly caught up. The `curls` started finishing, still with no answer, and I was starting to freak out when I finally got my last chunk of `882`. (Ugh.) I captured the flag and, as tradition requires, performed the [engineer's victory dance](#).

You can see [my terrible script](#) if you so desire. (There's also a [list of solvers](#).) I am so sorry for the tabs. The level 2 machine didn't have my `.vimrc` and I was in a hurry. Please don't think less of me.

I don't know if level 8 even has a real moral. This is so obscure I can't even find where Twisted mentions changing it, and it was a very specific set of circumstances that let me crack the password.

End

This was a ton of fun, and mad props to Stripe for setting it up (again!). I kinda regret not even trying the first one, and I certainly look forward to the third. :)

I'm pretty sure we should get special t-shirts for finding bonus solutions to some of the puzzles, though. And maybe literal gray hats.

Posted by Eevee Aug 29th, 2012 [essay](#)




[« Quick doesn't have to mean dirty](#)

Comments



Leave a message...

 Andy · 9 days ago

"I don't know if level 8 even has a real moral."

It was deliberately esoteric, but that's level 8 for you. It's intended to show that you can leak information in strange ways when you least expect it, and no list of classic vulnerabilities can protect you from that.

2 ^ | v · Reply · Share ›

 Dr. Dos · 9 days ago

I expecting the "engineer victory dance" to be the tf2 hoedown taunt.

1 ^ | v · Reply · Share ›

 Dr. Dos · 9 days ago · parent


expected. Wonderful typing on my part.

0 ^ | v · Reply · Share ›

 Jérôme Mahuet · 8 days ago


That was a really fun contest actually! I loved the last level.

0 ^ | v · Reply · Share ›

 Meg L. Streetman · 9 days ago

While I can only understand the basics of this, it seems a lot more interesting and fun than lawyering.


0 ^ | v · Reply · Share ›

 Ryan Cavicchioni · 9 days ago

I also solved level 5 with the leaked session key and I thought that this was the intended solution until I joined the IRC channel. I was stumped and then accidentally triggered that 500 error and had a feeling that there was something useful in the debug output.

I was having difficulty with level 8 because the server was so busy. I decided to reset the level and hope that I was bounced to a server that was less crowded which worked.

0 ^ | v · Reply · Share ›

 kshade · 9 days ago

The morale was "don't do obscure stuff with your passwords to make them more secure", wasn't it?


0 ^ | v · Reply · Share ›

ALSO ON FUZZY NOTEPAD

What's this? X


Quick doesn't have to mean dirty

11 · 77 comments · a month ago

 stuartpowers — This was just a joy to read, I found it /fun/ in a way many blog posts are not. Things like "I'm ev...

Heteroglot: #15 in COBOL - fuzzy notepad

0 · 12 comments · 4 hours ago

 JD — ...if you're still taking suggestions, I heartily recommend Piet.

Recent Posts

- [Stripe CTF 2.0](#)
- [Quick doesn't have to mean dirty](#)
- [Flora](#)
- [Python FAQ: Passing](#)
- [Python FAQ: Descriptors](#)
- [Python FAQ: Webdev](#)
- [PHP: a fractal of bad design](#)
- [Python FAQ: Equality](#)
- [On principle](#)
- [tmux is sweet as heck](#)

GitHub Repos

- Status updating...

[@eevee](#) on GitHub

Latest Tweets

- [2h](#) gotta say: it's pretty cool to have written some cobol, and i totally understand the decisions they made, and i will never do this again
- [2helliotjaystocks.com/blog/good-ridd...](#) someone else notices that paypal are unaccountable fuckbags. stop using them please
- [3hme.veekun.com/blog/2012/09/0...](#) it's friday night so let's write some fuckin cobol [#blog](#)
- [5h](#) another's hypothesis: tumblr is popular *because* the lack of commenting makes it difficult for anyone to point out what a dipwad you are

Follow [@eevee](#)

Copyright © 2012 - Eevee - Powered by [Octopress](#)