

Forum:[Hack Forums](#)

Thread: [How Dropbox stores its passwords; demo in Go](#)

EmptyRedData - December 21, 2016, 4:54 pm

Dropbox came out with an article recently on how it stores its passwords. You can find the short article here: <https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/>

The article itself explains everything about their choices in algorithms and why they layer the encryption like they do. What this post will do is translate their work into Go.

Firstly, we start by taking in a password and converting it into SHA512 hash. Afterwards we then hash it through bcrypt with a work factor of 10. Dropbox also mentions that they use a salt with bcrypt, but the Go team's implementation of bcrypt includes salt generation as viewed here: <https://github.com/golang/crypto/blob/master/bcrypt/bcrypt.go#L129>

This can all be done neatly in a single function with a []byte, error output.

The bcrypt default cost is 10, so this is nice and readable. Firstly we just take a byte slice as our only argument to keep it simple. Then we create our hash, check and make sure there are no errors, and finally bcrypt the Sum of that hash. Returning a nice byte slice we can plug straight into the next step.

For some brevity, we won't be writing any code that actually reads the secret key for AES from its file. We'll just be lazy and put that key as a constant in our code. If you are actually going to implement this in any code, set it up properly and have the secret key in a file outside of the code and the database.

Now to the actual part of encrypting the hash. When they say AES-256, we'll be using 256 bit AES-GCM. This also gives us the check that not only is our data encrypted, but that it has integrity checks as well. Although this doesn't guard against nonce misuse / collisions, we won't be too concerned with that. I suspect that with all the users Dropbox has, they might have to worry about it and have a different implementation or use AES-GCM-SIV. I don't know what they use since it isn't specified in the article.

Anyways, here is what we will use to encrypt the data via AES-GCM. AES automatically will select 256-bit by using the appropriate key size. We'll just bum the Encrypt function from here: <https://github.com/gtank/cryptopasta/blob/master/encrypt.go>

This repo specializes in having crypto code for these basic scenarios at the ready. That and the author has a lot of credentials creating and breaking secure crypto, so I trust it.

We're only bumming the Encrypt function, the rest we will make as we need it. All we need to do is put in our pepper (again, that shouldn't be in the code, this is just a demo) and have a way to pass that through our system. With all that in place, here is what the encryption code looks like in its entirety.

With that, we have what we will be putting in the database. All you need to do is now extract out the

key from the database, decode the AES outer shell, and then compare any incoming hashed password guess against what you have and that's it.

LastSession V - December 21, 2016, 4:58 pm

This is HQ as fuck for a regular. Nice OP.

SIM-1725686 - December 22, 2016, 8:28 am

Thanks for this article, it's well-written and clear to follow. I'm going to implement a similar strategy in Dart for keeping my applications secure.

Pseudonym - December 22, 2016, 2:24 pm

Eh, that makes me question competency of Dropbox developers. Quoting from the original article: First, the plaintext password is transformed into a hash value using SHA512. This addresses two particular issues with bcrypt. Some implementations of bcrypt truncate the input to 72 bytes, which reduces the entropy of the passwords. Other implementations don't truncate the input and are therefore vulnerable to DoS attacks because they allow the input of arbitrarily long passwords. By applying SHA, we can quickly convert really long passwords into a fixed length 512 bit value, solving both problems.

So to eliminate reduction of entropy to 576 bits they reduce it to 512? And instead of stopping DoS by checking input length they hash it? Doesn't seem too smart.

On the other hand using AES is quite sensible; even though one might think that public key cryptography would be safer, speed and additional storage requirement would make it infeasible.

(Dec 21, 2016 05:54 PM)EmptyRedData Wrote: [\[CITE\]](#) All you need to do is now extract out the key from the database, decode the AES outer shell, and then compare any incoming hashed password guess against what you have and that's it. Is there particular reason to do it this way? I don't know about AES-GCM, but with some ciphers it's faster to encrypt than decrypt.

EmptyRedData - December 22, 2016, 6:20 pm

(Dec 22, 2016 03:24 PM)Pseudonym Wrote: [\[CITE\]](#) Eh, that makes me question competency of Dropbox developers. Quoting from the original article:

First, the plaintext password is transformed into a hash value using SHA512. This addresses two particular issues with bcrypt. Some implementations of bcrypt truncate the input to 72 bytes, which reduces the entropy of the passwords. Other implementations don't truncate the input and are therefore vulnerable to DoS attacks because they allow the input of arbitrarily long passwords. By applying SHA, we can quickly convert really long passwords into a fixed length 512 bit value, solving both problems.

So to eliminate reduction of entropy to 576 bits they reduce it to 512? And instead of stopping DoS by checking input length they hash it? Doesn't seem too smart.

On the other hand using AES is quite sensible; even though one might think that public key cryptography would be safer, speed and additional storage requirement would make it infeasible.

Is there particular reason to do it this way? I don't know about AES-GCM, but with some ciphers it's faster to encrypt than decrypt.

As for Dropbox's approach on using SHA512, I can see them wanting to do that to mitigate the particular DoS attack they mentioned. At the least they are adding another set of cycles for a hash cracker. I don't see how it would increase or decrease the strength of the hash, but rather just mitigate that one particular attack.

As for the reason behind decrypting and then comparing hashes, I really didn't even think about speed requirements. I know AES-GCM is really quick in both encrypting in decrypting. In the case

that one is faster than the other, I would still go the route with decrypting. The reason I think this would be a good choice is because the fact that the hash is AES encrypted doesn't mean too much. I think bcrypt is the star here, the AES is just to prevent people from lifting hashes and going to town without breaking into their filesystem as well as the database. That being said, in the event someone was trying to login, I would decrypt it in case it took a couple of tries. That way you still have the hash, and could just compare multiple guesses without having to decrypt more than once. In the other instance you would have to encrypt each time they guess.

That's just my thoughts though, I am no crypto architect or performance guru, so I would take what I say with a bit of salt and do more research.