An RC4 Password Hashing Function

July 23, 2014
nullprogram.com/blog/2014/07/23/

There was an interesting /r/DailyProgrammer challenge[1] this week to write a program that properly hashes passwords for storage in an account database. It's a necessary part of any system that needs to securely authenticate users. Storing user passwords in plain text is not just bad practice, it's irresponsible. If the database is compromised, the attacker learns every user's password. Since people are likely to re-use passwords on different websites, the database could be used to infiltrate accounts on other sites.

The solution to this problem is to run the password through a one-way cryptographic hash function before storing it in the database. When the database is compromised, it's more difficult to work backwards to recover the passwords. Examples of one-way hash functions are MD5, SHA-1, and the SHA-2 family. Block ciphers can also be converted into hash functions (e.g. bcrypt from Blowfish), though it must be done carefully since block ciphers were generally designed with different goals in mind.

```
md5("foobar");
// => "3858f62230ac3c915f300c664312c63f"

sha1("foobar");
// => "8843d7f92416211de9ebb963ff4ce28125932878"
```

However, these particular functions (SHA-1 and SHA-2) alone are poor password hashing functions: they're much too fast! An offline attacker can mount a rapid brute-force attack on these kinds of hashes. They also don't include a *salt*, a unique, non-secret, per-hash value used as additional hash function input. Without this, an attacker could prepare the entire attack ahead of time — a *rainbow table*. Once the hashes are obtained, reversing them is just a matter of looking them up in the table.

Good password hashing needs to be slow and it needs support for salt. Examples of algorithms well-suited for this purpose are PBKDF2, bcrypt, and scrypt. These are the functions you'd want to use in a real application today. Each of these are also more generally *key derivation functions*. They can strengthen a relatively short human-memorable passphrase by running it through a long, slow procedure before making use of it as an encryption key. An brute-force attacker would need to perform this slow procedure for each individual guess.

Alternatively, if you're stuck using a fast hash function anyway, it could be slowed down by applying the function thousands or even millions of times recursively to its own output. This is what I did in order to strengthen my GPG passphrase[2]. However, you're still left with the problem of applying the salt. The naive approach would be a plain string concatenation with the password, but this likely to be vulnerable to a length extension attack[3]. The proper approach would be to use HMAC[4].

RC4 Solution

For my solution to the challenge, I wasn't looking for something strong enough to do key derivation. I just need a slow hash function that properly handles a salt. Another important goal was to keep the solution small enough to post as a reddit comment, and I wanted to do it without using any external crypto libraries. If I'm using a library, I might as well just

include/import/require PBKDF2 and make it a 2-liner, but that would be boring. I wanted it to be a reasonably short C program with no external dependencies other than standard libraries. Not counting the porcelain, the final result weighs in at 115 lines of C, so I think I achieved all my goals.

So what's the smallest, modern cryptographic algorithm I'm aware of? That would be RC4[5], my favorite random generator! Unlike virtually every other cryptographic algorithm, it's easy to commit to memory and to implement from scratch without any reference documentation. Similarly, this password hashing function can be implemented entirely from memory (if you can imagine yourself in some outlandish sitation where that would be needed).

Unfortunately, RC4 has had a lot of holes punched in it over the years. The initial output has been proven to be biased, leaking key material, and there's even good reason to believe it may already be broken by nation state actors. Despite this, RC4 remains the most widely used stream cipher today due to its inclusion in TLS. Most importantly here, almost none of RC4's weaknesses apply to this situation — we're only using a few bytes of output — so it's still a very strong algorithm. Besides, what I'm developing is a proof of concept, not something to be used in a real application. It would be interesting to see how long it takes for someone to break this (maybe even decades).

Before I dive into the details, I'll link to the source repository. As of this writing there are C and Elisp implementations of the algorithm, and they will properly validate each other's hashes. I call it **RC4HASH**.

- https://github.com/skeeto/rc4hash

Here are some example hashes for the password "foobar". It's different each time because each has a unique salt. Notice the repeated byte `12` in the 5th byte position of the hash.

```
$ ./rc4hash -p foobar
c56cdbe512c922a2f9682cc0dfa21259e4924304e9e9b486c49d
$ ./rc4hash -p foobar
a1ea954b1296052a7cd766eb989bfd52915ab267733503ef3e8d
$ ./rc4hash -p foobar
5603de351288547e12b89585171f40cf480001b21dcfbd25f3f4
```

Each also validates as correct.

```
$ ./rc4hash -p foobar -v c56cdbe5...b486c49d
valid
$ ./rc4hash -p foobar -v a1ea954b...03ef3e8d
valid
$ ./rc4hash -p foobar -v 5603de35...bd25f3f4
valid
```

The Algorithm

RC4 is a stream cipher, which really just means it's a fancy random number generator. How can we turn this into a hash function? The content to be hashed can be fed to the key schedule algorithm in 256-byte chunks, as if it were a key. The key schedule is a cipher initialization stage that shuffles up the cipher state without generating output. To put all this in terms of C, here's

what the RC4 struct and initialization looks like: a 256-element byte array initialized with 0-255, and two array indexes.

```
struct rc4 {
    uint8_t S[256];
    uint8_t i, j;
};

void rc4_init(struct rc4 *k) {
    k->i = k->j = 0;
    for (int i = 0; i < 256; i++) {
        k->S[i] = i;
    }
}
```

The key schedule shuffles this state according to a given key.

```
#define SWAP(a, b) if (a ^ b) {a ^= b; b ^= a; a ^= b;}

void rc4_schedule(struct rc4 *k, const uint8_t *key, size_t length) {
    int j = 0;
    for (int i = 0; i < 256; i++) {
        j = (j + k->S[i] + key[i % length]) % 256;
        SWAP(k->S[i], k->S[j]);
    }
}
```

Notice it doesn't touch the array indexes. It can be called over and over with different key material to keep shuffling the state. This is how I'm going to mix the salt into the password. The key schedule will first be run on the salt, then again on the password.

To produce the hash output, emit the desired number of bytes from the cipher. Ta da! It's now an RC4-based salted hash function.

```
void rc4_emit(struct rc4 *k, uint8_t *buffer, size_t count) {
    for (size_t b = 0; b < count; b++) {
        k->j += k->S[++k->i];
        SWAP(k->S[k->i], k->S[k->j]);
        buffer[b] = k->S[(k->S[k->i] + k->S[k->j]) & 0xFF];
    }
}

/* Throwaway 64-bit hash example. Assumes strlen(passwd) <= 256. */
uint64_t hash(const char *passwd, const char *salt, size_t salt_len) {
    struct rc4 rc4;
    rc4_init(&rc4);
    rc4_schedule(&rc4, salt, salt_len);
    rc4_schedule(&rc4, passwd, strlen(passwd));
    uint64_t hash;
    rc4_emit(&rc4, &hash, sizeof(hash));
    return hash;
}
```

Both password and salt are the inputs to hash function. In order to validate a password against a hash, we need to keep track of the salt. The easiest way to do this is to concatenate it to the hash itself, making it part of the hash. Remember, it's not a secret value, so this is safe. For my solution, I chose to use a 32-bit salt and prepend it to 20 bytes of generator output, just like an initialization vector (IV). To validate a user, all we need is a hash and a password provided by the user attempting to authenticate.

Fixing a Flaw

Right now there's a serious flaw. If you want to find it for yourself, stop reading here. It will need to get fixed before this hash function is any good.

It's trivial to find a collision, with is the death knell for any cryptographic hash function. Certain kinds of passwords will collapse down to the simplest case.

```
hash("x", "salt", 4);
// => 8622913094354299445
hash("xx", "salt", 4);
// => 8622913094354299445
hash("xxx", "salt", 4);
// => 8622913094354299445
hash("xxxx", "salt", 4);
// => 8622913094354299445

hash("abc", "salt", 4);
// => 8860606953758435703
hash("abcabc", "salt", 4);
// => 8860606953758435703
```

Notice a pattern? Take a look at the RC4 key schedule function. Using modular arithmetic, the password wraps around repeating itself over 256 bytes. This means passwords with repeating

patterns will mutate the cipher identically regardless of the number of repeats, so they result in the same hash. A password “abcabcabc” will be accepted as “abc”.

The fix is to avoid wrapping the password. Instead, the RC4 generator, seeded only by the salt, is used to pad the password out to 256 bytes without repeating.

```
uint64_t hash(const char *passwd, const char *salt, size_t salt_len) {
    struct rc4 rc4;
    rc4_init(&rc4);
    rc4_schedule(&rc4, salt, salt_len);
    uint8_t padded[256];
    memcpy(padded, passwd, strlen(passwd));
    rc4_emit(&rc4, padded + strlen(passwd), 256 - strlen(passwd));
    rc4_schedule(&rc4, padded, sizeof(padded));
    uint64_t hash;
    rc4_emit(&rc4, &hash, sizeof(hash));
    return hash;
}
```

This should also help mix the RC4 state up a bit more before generating the output. I’m no cryptanalyst, though, so I don’t know if it’s worth much.

Slowing It Down

The next problem is that this is way too fast! It shuffles bytes around for a few microseconds and it’s done. So far it also doesn’t address the problems of biases in RC4’s initial output. We’ll kill two birds with one stone for this one.

To fix this we’ll add an adaptive difficulty factor. It will be a value that determines how much work will be done to compute the hash. It’s adaptive because the system administrator can adjust it at any time without affecting previous hashes. To accomplish this, like the salt, the difficulty factor will be appended to the hash output. All the required information will come packaged together in the hash.

The difficulty factor comes into play in two areas. First, it determines how many times the key schedule is run. This is the same modification CipherSaber-2 uses[6] in order to strengthen RC4’s weak key schedule. However, rather than run it on the order of 20 times, our hash function will be running it hundreds of thousands of times. Second, the difficulty will also determine how many initial bytes of output are discarded before we start generating the hash.

I decided on an unsigned 8-bit value for the difficulty. The number of key schedules will be 1 shifted left by this number of bits (i.e. `pow(2, difficulty)`). This makes the minimum number of key schedules 1, since any less doesn’t make sense. The number of bytes skipped is the same bitshift, minus 1, times 64 (`(pow(2, difficulty) - 1) * 64`), the muliplication is so that it can skip large swaths output. Therefore the implementation so far has a difficulty of zero: one key schedule round and zero bytes of output skipped.

The dynamic range of the difficulty factor (0-255) puts the the time needed on a modern computer to compute an RC4HASH between a few microseconds (0) to the billions of years (255). That should be a more than sufficient amount of future proofing, especially considering that we’re using RC4, which will likely be broken before the difficulty factor ever tops out.

I won't show the code to do this since that's how it's implemented in the final version, so go look at the repository instead. The final hash is 26 bytes long: a 208-bit hash. The first 4 bytes are the salt (grabbed from /dev/urandom in my implementations), the byte is the difficulty factor, and the final 21 bytes are RC4 output.

In the example hashes above, the `12` constant byte is the difficulty factor. The default difficulty factor is 18 (`0x12`). I've considered XORing this with some salt-seeded RC4 output just to make the hash look nice, but that just seems like arbitrary complexity for no real gains. With the default difficulty, it takes almost a second for my computers to compute the hash.

I *believe* RC4HASH should be quite resistant to GPGPU attacks. RC4 is software oriented, involving many random array reads and writes rather than SIMD-style operations. GPUs are really poor at this sort of thing, so they should take a significant performance hit when running RC4HASH.

Break My Hash!

For those interested in breaking RC4HASH, here are a couple of hashes of English language passphrases. Each is about one short sentence in length (`[a-zA-Z !.,]+`). I'm not keeping track of the sentences I used, so the only way to get them will be to break the hash, even for me.

```
f0622dde127f9ab5aaee710aa4bfb17a224f7e6e93745f7ae948
8ee9cdec12feabed5c2fde0a51a2381b522f5d2bd483717d4a96
```

If you can find a string that validates with these hashes, especially if it's not the original passphrase, you win! I don't have any prizes in mind right now, but perhaps some Bitcoin would be in order if your attack is interesting.

tags [c crypto]

1. http://redd.it/2ba46z
2. https://nullprogram.com/blog/2012/06/24/
3. http://en.wikipedia.org/wiki/Length_extension_attack
4. http://en.wikipedia.org/wiki/Hash-based_message_authentication_code
5. https://nullprogram.com/blog/2008/08/09/
6. https://nullprogram.com/blog/2009/04/24
7. https://lists.sr.ht/~skeeto/public-inbox
8. mailto:~skeeto/public-inbox@lists.sr.ht?Subject=Re%3A%20An%20RC4%20Password%20Hashing%20Function
9. https://lists.sr.ht/~skeeto/public-inbox?search=An+RC4+Password+Hashing+Function
10. https://nullprogram.com/blog/comments/#2014-07-23