Exploiting SHA-1-signed messages

Context

A few weeks ago, I was asked to review a coworker's PHP code, which provides an API for a web game. The specifications for this API were sent to a mobile development company, in charge of making an iPhone app for the game. All requests from API clients had to be signed using the SHA-1 hash function, prefixing the message with a salt *for security reasons*. I pointed out that this was probably not safe, and here is a detailed explanation of why. Please note that I am not a cryptographer, and that you should take my advice with a grain of salt.

Signing with SHA-1

Let's consider the simple case of a user who wants to transfer money to a friend. The API will need 3 parameters:

```
from=123&to=456&amount=50
```

The salt was invariant, embedded as a constant string in the mobile app. That vulnerability is too easy to exploit, so let's pretend we only intercepted the packet on a public WiFi and focus on the problem with SHA-1. This is how the signature is checked on the server:

```
$salt = "s3cRe7-#!@~";
$signature = sha1($salt.$msg);
if($signature == $x_signature_header) { // sent as an HTTP header
        // correct
```

```
} else {
        // discard message
}
```

In the case of our transaction, the correct signature for the message is:
`sha1("s3cRe7-#!@~from=123&to=456&amount=50")` = `"18d2fccddeba1b08304fb53e849be5711d55b2e0"`. This is included in the packet we've intercepted.

How SHA-1 is computed

The SHA-1 algorithm is simple; it starts by padding the message with zeros up to a multiple of 64 bytes, and processes each 64-byte block of the message in sequence *using the hash of each chunk to compute the next*. It also includes the message's length as a 64-bit integer at the end of the last 64-byte chunk.

In our case, this is what the first and only chunk looks like:

```
73 33 63 52 65 37 2d 23 21 40 7e 66 72 6f 6d 3d    s3cRe7-
                      #!@~from=
31 32 33 26 74 6f 3d 34 35 36 26 61 6d 6f 75
            6e    123&to=456&amoun
74 3d 35 30 80 00 00 00 00 00 00 00 00 00 00
            00    t=50............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
            20    ...............
```

This chunk is 64 bytes long, it starts with the **salt**, followed by the **message**, followed by **a single "1" bit** (0x80 is 10000000 in binary), followed by **padding zeros** and the **length in bits of our input**: len(salt) + len(msg) = 11 + 25 = 36 bytes = 288 bits = **0x120**

bits.

This single chunk `c0` is hashed using SHA-1's default seed values: h0,h1,h2,h3,h4 = 0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0.

Injecting data into a new chunk

We can add data to this message by creating a second chunk. This means considering that first chunk as the beginning of a new message, and injecting data only after the first 64 padded bytes. Because SHA-1 reuses the output of the first chunk as the initial h0..h4 values to process the second chunk, we can compute the hash of (c0 + injected data) by reusing the initial hash and only hashing the second chunk. Our new message will contain the **paddings zeros** and the **first original size**, but that's not really a problem with a PHP target.

The data we want to inject redirects the transaction to another account, and adds a generous tip:

```
&to=666&amount=99999
```

The injected string is 20 bytes long, which means that the complete message is 64 + 20 = 84 bytes = **0x2a0 bits**.

With this information we create the second chunk:

```
26 74 6f 3d 36 36 36 26 61 6d 6f 75 6e 74 3d
            39    &to=666&amount=9
39 39 39 39 80 00 00 00 00 00 00 00 00 00 00
            00    9999............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
            00    ...............
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02
                a0    ................
```

As we saw, hashing c0 produces 18d2fccddeba1b08304fb53e849be5711d55b2e0, which is split into h0..h4: 0x18d2fccd, 0xdeba1b08, 0x304fb53e, 0x849be571, 0x1d55b2e0. Those values will be used as the input of the main SHA-1 loop, in order to process the second chunk as if it was the continuation of the first.

Processing that second chunk with the custom h0..h4 values generates the following hash: **5a77fe8376811dbf7533509cceb6c8c53f98e253**.
This result is the hash of (chunk 0 + injected data), which is (salt + message + padding + size). We want to recreate this new input in order to send a fake order without knowing the salt. As far as I know, we can only guess its size. If we guess that the salt is 11 bytes long, here is the message we'll send:

```
66 72 6f 6d 3d 31 32 33 26 74 6f 3d 34 35 36
            26    from=123&to=456&
61 6d 6f 75 6e 74 3d 35 30 80 00 00 00 00 00
            00    amount=50.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
            00    ...............
00 00 00 01 20 26 74 6f 3d 36 36 36 26 61 6d 6f    ....
                  &to=666&amo
            75 6e 74 3d 39 39 39 39 39
                        unt=99999
```

This message is made of the end of the first chunk with our guess of an 11-byte salt, to which we have added our new data.

Signature check on the server side

On the server side, the following code reads the input and prints out its signature:

```
/* PHP */
$salt = "s3cRe7-#!@~";
$msg = urldecode($_SERVER['QUERY_STRING']); // read input.

$signature = sha1($salt . $msg);        // compute signature.
```

The `sha1()` call will process the following chunks:

The first chunk is hashed with the default h0..h4 = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0):

```
73 33 63 52 65 37 2d 23 21 40 7e 66 72 6f 6d 3d   s3cRe7-#!@~from=
31 32 33 26 74 6f 3d 34 35 36 26 61 6d 6f 75 6e   123&to=456&amoun
           74 3d 35 30 80 00 00 00 00 00 00 00 00 00
                        00   t=50.............
           00 00 00 00 00 00 00 00 00 00 00 00 00 01
                        20   ................
```

Producing h0..h4 = (0x18d2fccd, 0xdeba1b08, 0x304fb53e, 0x849be571, 0x1d55b2e0), which are then used to hash the second chunk:

```
26 74 6f 3d 36 36 36 26 61 6d 6f 75 6e 74 3d 39   &to=666&amount=9
39 39 39 39 80 00 00 00 00 00 00 00 00 00 00 00   9999............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 a0   ................
```

Producing **5a77fe8376811dbf7533509cceb6c8c53f98e253**, which is what we had predicted. We have added data to our message, and used the original hash to sign the new chunk.

PHP is happy with the input, the later values affected to the `to` and `amount` variables replacing the earlier ones which contained binary data:

```
Input = array(3) {
        ["from"]=> string(3) "123"
        ["to"]=> string(3) "666"
        ["amount"]=> string(5) "99999"
}

Hash = string(40) "5a77fe8376811dbf7533509cceb6c8c53f98e253"
```

And the signature is correct, too! The transaction is executed.

Message signature

- Don't use a hash function as a symmetric signature.
- Don't roll your own.
- Use HMAC.

I highly recommend this 1-hour presentation on cryptography by Colin Percival, which includes specific dos and don'ts for different use cases: (PDF), (video), (Hacker News discussion).

Code

A demo code for this article is available on GitHub.

If you enjoyed reading this article, you should follow me on twitter.