Forging a SHA-1 MAC using a length-extension attack in Python

Mantej Singh Rajpal

SHA-1 (Secure Hash Algorithm 1) is broken. It has been since 2005. And yet, that hasn't stopped its continued use. For example, until early 2017 most internet browsers still supported SHA-1. As though to confirm that SHA-1 was really, truly dead, researchers from CWI Amsterdam and Google announced at the end of February 2017 they had performed a successful collision attack against SHA-1.

In this article, we'll learn how to break a SHA1-keyed message authentication code (MAC), part of the authentication message that confirms the message came from the stated sender. This is useful for impersonating someone else. But first, some background.

What is SHA-1?

SHA-1 is a cryptographic hashing algorithm designed by the United States National Security Agency back in the early 1990s. It is used to create a message digest, which in cryptography is a one-way mathematical algorithm that takes data of any size and maps it to a bit string of a fixed size (a hash function). So, feed SHA-1 an arbitrary amount of data and it will produce a 20-character message digest. Whether you input "Synopsys," "I love cryptography!", or Act I from Shakespeare's Hamlet, the result will always be exactly 20 bytes.

People have used (and still use) SHA-1 to produce MACs. We're going to break a SHA-1 MAC that has been prefixed with a key. Namely:

message digest = SHA1(key || message)

Given the message digest and message, it shouldn't be possible to modify the message and produce a new corresponding message digest. That is, unless there is complete knowledge of the key. This isn't actually the case.

In this example, we'll take the message "user=mantej;" and generate a SHA-1 message digest under an unknown key. We'll continue to reverse engineer the hash, and use the result to create a valid message digest for the message "user=mantej; ... ;admin=true" without any knowledge of the key.

But before we do that, let's find a SHA-1 implementation and make a couple modifications.

Modifying SHA-1

For our example we'll use a pure-Python implementation of SHA-1 from Stack Exchange, and compare its output to an online SHA-1 calculator to verify correctness.

```
def sha1(message):

    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476
    h4 = 0xC3D2E1F0

```

Figure 1. SHA-1 magic numbers

Now, a 160-bit SHA-1 hash is actually just made up of five internal 32-bit registers. These registers usually start at magic numbers (Figure 1).

We're going to modify the function definition so that we can pass in a SHA-1 message digest and 're-create' its state. We accomplish this by breaking our message digest into five 32-bit values (Figure 2), and passing in those values as function parameters in order to override the default magic numbers (Figure 3). By setting our own register values, we're able to 'pick up where we left off' and append an arbitrary amount of data to the original message.

```
# breaks SHA1 hash into 5 32-bit registers
def get_internal_state(sha1_decimal_digest):
    a = sha1_decimal_digest >> 128
    b = (sha1_decimal_digest >> 96) & 0xffffffff
    c = (sha1_decimal_digest >> 64) & 0xffffffff
    d = (sha1_decimal_digest >> 32) & 0xffffffff
    e = sha1_decimal_digest & 0xffffffff
    return [a, b, c, d, e]

```

Figure 2. Breaking SHA-1 digest into five 32-bit registers

```
def sha1(message, h0=0x67452301, h1=0xEFCDAB89, h2=0x98BADCFE,
         h3=0x10325476, h4=0xC3D2E1F0, length=None):
    # length can optionally be injected for length-extension attacks
    if length == None:
        length = len(message)*8

```

Figure 3. Overriding default magic numbers

Additionally, we need to append the bit-length of our forged message (Figure 4). Specifically, we need to compute length(key || original message with padding || new message). We can't

immediately compute this value since, if you recall, the key is completely unknown to us. Not to worry. So, let's just brute-force--that is, bombard it with possibilities--for key length (Figure 5).

```
26
27      # for length-extension attacks, append the injected length
28      # otherwise, just append the original length
29      pBits+='{0:064b}'.format(length)
30
```

Figure 4. Inject bit-length of our forged message for length-extension attacks

```
71
72  # taking stabs at the key length until we guess correctly & forge a MAC
73  for i in range(1, 33):
74      m, d = forge_message(message, message_digest, i, b';admin=true')
75      if validate(m, d):
76          print
77          print "[*] Secret-Prefix MAC Generated!"
78          print "[*] Assumed Key Length is %s" % (i)
79          print "[*] Forged Message: %s" % (m)
80          print "[*] Forged Digest under Secret Key: %s" % (d)
81
```

Figure 5. Brute-forcing for key length

Performing the attack

Let's use a validate function that takes the forged message and forged message digest as input, computes SHA-1(key || forged message), and compares it to the forged digest. If they are identical (i.e., mission success!), validate() returns true. It looks something like this:

```
10
11  # returns True if the digest corresponds to the message under the secret key
12  def validate(message, message_digest):
13      global key
14      return authSHA1(key, message) == message_digest
15
```

Figure 6. Validates our forged message digest under the secret key

For the sake of completeness, here's the forge_message function:

```
46
47  # forged_message = "A"*keylen || original message || glue padding || new message
48    # get_internal_state = breaks original message digest into [5] 32-bit registers
49    # forged_digest = SHA-1 digest under secret key for our forged message
50  def forge_message(message, message_digest, keylen, new_message):
51      forged_message = glue_padding("A"*keylen + message) + new_message
52      # remove key from our forged message (it's not the correct key anyways)
53      forged_message = forged_message[keylen:]
54
55      decimal_digest = int(message_digest, 16)
56      h = get_internal_state(decimal_digest)
57      # call SHA1 directly with fixated registers & additional data to forge
58      forged_digest = sha1(new_message, h[0], h[1], h[2], h[3], h[4], (keylen+len(forged_message)) * 8)
59
60      return (forged_message, forged_digest)
61
```

Figure 7. Forges a message utilizing technique discussed above

Lastly, we'll write a program that takes the message "user=mantej;" and generates a SHA-1 hash under a randomly generated secret key between 1 and 32 bytes. Utilizing everything described above, I'm going to brute-force for key length, append ";admin=true" to the original message, and generate a valid MAC under the secret key--without any knowledge of the key itself.

And. Here. We. Go.

```
$ python sha1-length-extension-attack.py

[*] Actual key length is 17

[*] Trying to forge message with key length 1 ...
[*] Trying to forge message with key length 2 ...
[*] Trying to forge message with key length 3 ...
[*] Trying to forge message with key length 4 ...
[*] Trying to forge message with key length 5 ...
[*] Trying to forge message with key length 6 ...
[*] Trying to forge message with key length 7 ...
[*] Trying to forge message with key length 8 ...
[*] Trying to forge message with key length 9 ...
[*] Trying to forge message with key length 10 ...
[*] Trying to forge message with key length 11 ...
[*] Trying to forge message with key length 12 ...
[*] Trying to forge message with key length 13 ...
[*] Trying to forge message with key length 14 ...
[*] Trying to forge message with key length 15 ...
[*] Trying to forge message with key length 16 ...
[*] Trying to forge message with key length 17 ...

[*] Secret-Prefix MAC Generated!
[*] My Guessed Key Length is 17
[*] Forged Message: user=mantej;??;admin=true
[*] Forged Digest under Secret Key: 778cc6cbb0552faa775d63b5b9fe3fad5f72385f
```

Figure 8. Performing the length extension attack

Moral of the story? Only use SHA1(key || message) as a MAC if security isn't important.