Security advisory: Timing attack in HMAC signature verification of Python OAuth [1]

Sébastien Martini[*]

Description

A vulnerability affects the server implementation [2] of the Python OAuth 1.0 library [1] when the signature algorithm HMAC-SHA1 [3] is used to authenticate requests to the server. This is a side-channel vulnerability in the signature verification algorithm [4] made possible by the presence of two additional flaws affecting the nonce and timestamp [5] mechanisms. These flaws may lead an attacker to forge a valid signature for a malicious request without having any knowledge of the client (consumer) secret keys. This attack only applies to requests made over a channel not providing confidentiality of communications.

Timing attack

To generate a valid signature the attacker would either have to recover all the secret keys or to generate a collision between his HMAC and the HMAC calculated by the server. Both are in practice expected to be computationally impracticable [6]. However it may be possible to reduce this complexity by exploiting a flaw at signature verification step and iteratively guess all the 27 symbols of the base64 signature. This flaw lies in the algorithm used to compare the signatures and is especially valuable when one of the strings is a secret value and the other one is controlled by the attacker. Here is the incriminated code [7]:

```
def check(self, request, consumer, token, signature):
    built = self.sign(request, consumer, token)
    return built == signature
```

`built` is the reference signature and `signature` is the string provided by the attacker. This test of equality is internally handled by this source file [8] in function `string_richcompare`:

```
 if (op == Py_EQ) {
     /* Supporting Py_NE here as well does not save
        much time, since Py_NE is rarely used.  */
     if (Py_SIZE(a) == Py_SIZE(b)
         && (a->ob_sval[0] == b->ob_sval[0]
         && memcmp(a->ob_sval, b->ob_sval, Py_SIZE(a)) == 0)) {
         result = Py_True;
     } else {
         result = Py_False;
     }
     goto out;
}
```

[*]seb@dbzteam.org – July 14, 2010

It eventually calls `memcmp()` which is known for leaking timings informations in some of its implementations. In these implementations strings are iterated from left to right, byte after byte, returning as soon as two bytes mismatches.

By carefully analyzing timings differences between these comparisons one attacker may recover the value of `signature`. He would do so by submitting a lot of different chosen signatures analyzing their timings, refining its predictions, guessing byte after byte until finally finding the right signature. This attack has been introduced last year by Nate Lawson [9, 10, 11], it is mostly the same context here.

Therefore, the time this comparison takes should be independent from its result and its arguments. Below is a proposed fix similar to [12]:

```
def check(self, request, consumer, token, signature):
    built = self.sign(request, consumer, token)
    sig_len = len(built)
    if sig_len != len(signature):
        return False
    diff = 0
    for pos in range(sig_len):
        diff |= ord(built[pos]) ^ ord(signature[pos])
    return diff == 0
```

Remark: the attacker needs to know a valid consumer key as well as the token key of its victim (only the non-secret parts for both keys). He might obtain them by eavesdropping the communications of his victim [13].

Bypassing the nonce and timestamp mechanisms

From previous section it becomes clear one attacker needs to submit the same request multiple times to successfully conduct his statistical analysis. The OAuth protocol uses two mechanims to prevent it. A nonce, unique, accepted a single time by a server for a given timestamp and a timestamp limiting the time period during which requests are accepted and rejecting old requests. These parameters being signed it should be impossible to perform this attack even with a non-constant time signature verification. However in this particular implementation these two mechanisms can be bypassed:

- The nonces are never verified, there was a delegation mechanism [14] with no concrete implementation but has been removed. Beside preventing this attack, nonces more broadly prevent replay attacks. Attacks where an attacker sends several times a single well-formed request. Providing a generic DataStore would thwart these attacks.

- The timestamp mechanism is currently implemented [15] with a small window of 5 minutes but there is a bug hindering its effectiveness:

```
def _check_timestamp(self, timestamp):
    timestamp = int(timestamp)
    now = int(time.time())
    lapsed = now - timestamp
    if lapsed > self.timestamp_threshold:
        raise Error('Expired timestamp: given %d and now %s has a '
            'greater difference than threshold %d' % (timestamp, now,
                self.timestamp_threshold))
```

In this method if `timestamp` is a value in the future (greater than `time.time()`) the difference `now - timestamp` will be negative and the following conditional test will never succeed. So an attacker only has to choose a timestamp with a big value to bypass this test. This test should be replaced by `if lapsed < 0 or lapsed > self.timestamp_threshold`.

With these two mechanisms proven ineffective an attacker may submit the same request continuously and perform the attack reported in the first sections.

Conclusion

Fixing the nonce would prevent replay attacks, fixing the timestamp would prevent an attacker from performing this HMAC attack and finally making the signature verification constant-time would add even more security with negligible consequences on performances.

References

[1] python-oauth2 – source tree. http://github.com/simplegeo/python-oauth2.

[2] python-oauth2 – server. http://github.com/simplegeo/python-oauth2/blob/763b4329cb8a0f22e7b4f574a28a6504a7e32e74/oauth2/__init__.py#L578.

[3] OAuth 1.0 – HMAC-SHA1 signature algorithm. http://tools.ietf.org/html/rfc5849#section-3.4.2.

[4] OAuth 1.0 – Requests verification. http://tools.ietf.org/html/rfc5849#section-3.2.

[5] OAuth 1.0 – Nonce and timestamp. http://tools.ietf.org/html/rfc5849#section-3.3.

[6] OAuth 1.0 – Entropy of secrets. http://tools.ietf.org/html/rfc5849#section-4.9.

[7] python-oauth2 – HMAC verification. http://github.com/simplegeo/python-oauth2/blob/763b4329cb8a0f22e7b4f574a28a6504a7e32e74/oauth2/__init__.py#L702.

[8] Python trunk – stringobject.c. http://svn.python.org/projects/python/trunk/Objects/stringobject.c.

[9] Keyczar security advisory. http://groups.google.com/group/keyczar-discuss/browse_thread/thread/5571eca0948b2a13.

[10] Nate Lawson ”Timing attack in google keyczar library”. http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/.

[11] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security and Privacy*, 7:65–68, 2009. ISSN 1540-7993.

[12] Keyczar fix. http://code.google.com/p/keyczar/source/diff?spec=svn414&r=414&format=side&path=/trunk/python/src/keyczar/keys.py.

[13] OAuth 1.0 – Confidentiality of requests. http://tools.ietf.org/html/rfc5849#section-4.2.

[14] python-oauth2 – dataStore. http://github.com/simplegeo/python-oauth2/commit/143fb346521f07e20c0cb93073dbb6cfeab6fe43.

[15] python-oauth2 – timestamp verification. http://github.com/simplegeo/python-oauth2/blob/763b4329cb8a0f22e7b4f574a28a6504a7e32e74/oauth2/__init__.py#L662.