Universiteit van Amsterdam

Master System & Network Engineering

Securing the last mile of DNS with CGA-TSIG

Research Project 2

Student:

Marc Buijsman

mbuijsman@os3.nl

Supervisors:

Matthijs Mekking

(NLnet Labs)

matthijs@nlnetlabs.nl

Jeroen van der Ham

(UvA)

vdham@uva.nl

7 January 2014

Abstract

This document describes the research performed during the course of Research Project 2 of the Master's in System and Network Engineering at the University of Amsterdam, under the supervision of NLnet Labs. Research has been done on the use of a proposed protocol called CGA-TSIG as a potential solution to the last mile problem of the Domain Name System (DNS). A proof of concept has been implemented, which resulted in the identification of several problems with the proposed protocol. Nevertheless, it was found that CGA-TSIG can work as believed to be intended, even though the protocol only works on IPv6. An automated method for authenticating the IP addresses of recursive name servers is additionally needed for CGA-TSIG to be a useful and fully secure solution to the last mile problem, but research to such methods was out of the scope of this project and would require future research.

Contents

1 Introduction

The Domain Name System (DNS) is a distributed database with the primary goal to provide domain name to IP address translation. It was designed without security in mind, allowing for vulnerabilities like IP address spoofing to be exploited. Several improvements have been introduced to make the DNS more secure, like DNSSEC [1][2][3], DNSCurve [7], and TSIG [19]. DNSSEC provides authentication of DNS data's authoritative source, data integrity, and authenticated denial of existence by using public-key cryptography. DNSCurve provides data integrity and also confidentiality by encrypting the data. TSIG is a protocol that can be used for assuring data authentication and data integrity of DNS transactions between two hosts that share a secret key.

Despite the existence of these security protocols, DNS security is still not watertight in most practical situations. The problem is that client computers in general are forwarding their queries to pre-configured DNS recursive resolvers, asking them to fetch the answers from authoritative name servers and return them, usually without any DNS security deployed between the client and the resolver (also known as the recursive name server). This so-called "last mile" of DNS, which is depicted in figure 1.1, is therefore vulnerable to man-in-the-middle attacks. The data retrieved from the authoritative name servers by the recursive name server can usually be secured with DNSSEC, however.

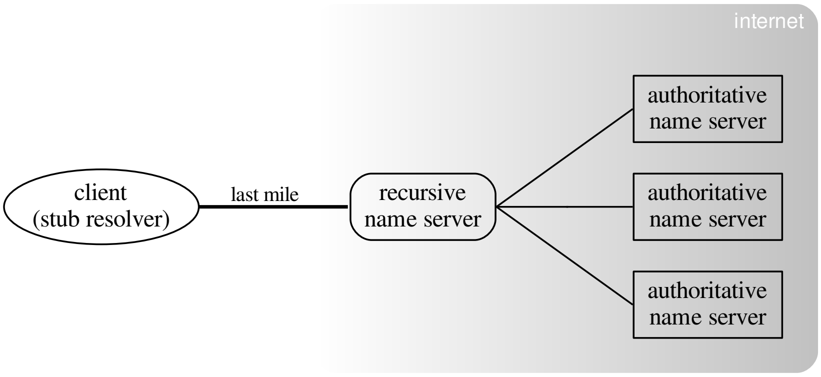
internet
authoritative name server
client (stub resolver)
last mile
recursive name server
authoritative name server
authoritative name server

Figure 1.1: The "last mile" is the link between the client and the recursive name server.

There are several reasons why the last mile is usually not secured with the existing security protocols. First of all, the last mile exists because most client computers come with only a stub resolver installed, which is a light-weight resolver that cannot do much more than forwarding and receiving DNS messages to and from recursive resolvers which will do the heavy work. This way, clients do not need to recursively query multiple name servers and validate the responses with DNSSEC themselves. However, any man in the middle residing on the last mile will be able to forge DNS responses and validation results if the last mile has not been secured.

The client may perform DNSSEC validation itself. In order to do so, it needs to build up its own chain of trust for every DNS response it requests. Alternatively, clients can install and run a recursive resolver locally to avoid the last mile problem. In both cases, it requires the user to be able to configure and manage the recursive server, which involves acquiring the DNSSEC root key in a secure way. Although the key might be included with the operating system, it needs to be retrieved manually if the client has missed a key roll-over period (which is a rare event, but the consequences can be devastating as portrayed by Michaelson et al. [15]). In some applications it might also be too expensive to perform the resolution, caching, and validation locally.

Alternatively, DNSCurve could be used to secure the last mile. Nevertheless, the standardisation process of the protocol does not seem to be progressing well and the protocol has not seen widespread adoption. Another possibility is to use TSIG, which adds a smaller worst case computational load for validation than DNSSEC. However, TSIG requires shared secrets to be exchanged manually and is not scalable. The protocol is most useful for performing dynamic DNS updates or zone transfers, which usually requires a relatively small number of hosts to configure shared secret keys that are being managed by a dedicated system administrator. For the average user, acquiring the secret key poses a problem, and due to the large number of secret keys that a recursive resolver would need to manage for all its clients it is not scalable. If a client is moving from network to network, being assigned to different resolvers, it would need to have a key for each resolver it encounters, impacting the scalability as well.

A solution to TSIG's key distribution problem has been proposed by Rafiee and Meinel [16]. The proposal combines TSIG with another existing method, Cryptographically Generated Addresses (CGA), into the CGA-TSIG protocol. It uses CGA as the algorithm type for TSIG. CGA [6] is an algorithm that was originally designed to be used with the Secure Neighbor Discovery protocol (SEND) [4] on IPv6 networks. Unlike standard TSIG, CGA-TSIG uses public-key cryptography to sign DNS messages and uses CGA to authenticate the public key. As a result, it makes TSIG scalable and therefore useful for the last mile. A draft for a proposed RFC on CGA-TSIG has been submitted to the IETF [17]. It can be investigated how well this protocol could be used as an alternative solution to the last mile problem.

1.1 Research question

The research done for this project was focused on the investigation of CGA-TSIG as a solution to the last mile problem. This was the only protocol that was in the scope of the project due to time constraints. The research question has been defined as follows:

Is CGA-TSIG an adequate solution to the last mile problem of DNS in IPv6 networks?

In order to be able to answer this question, the following sub-questions have been defined:

- *Does CGA-TSIG provide the necessary security?*
- *Is the CGA-TSIG specification correct?*

1.2 Approach

The CGA-TSIG protocol is a combination of CGA and TSIG, and relies on the security these two protocols provide. In order to be able to determine if CGA-TSIG provides the necessary security for the last mile, both CGA and TSIG need therefore be researched. This has been outlined in section 2, where TSIG is discussed in section 2.1 first, followed by CGA in section 2.2.

At the time of writing, the CGA-TSIG protocol is still in the process of being specified. The current specification has the status of an IETF individual draft [17] and is not yet complete. The specification can be improved, however, by verifying the current version of the draft. This could be done with formal verification methods like modelling, or by implementing a proof of concept. The last option has been chosen mainly due to the limited time that could be spent to perform a verification, but also because of the IETF community's belief in "rough consensus and *running code*" [11]. By making a proof of concept it can be shown that the protocol works as intended, but it can also be helpful in revealing any problems with the specification. Recommendations can be made on any unspecified details as well, for which some assumptions would need to be made in order to create a working implementation.

Any problems that were encountered in the process can be addressed and improvements can be made to the draft in order to solve these problems. The implementation can then be updated to reflect the new improvements and therefore be used to perform another verification, eventually leading to a complete and sound specification.

In order to be able to implement a proof of concept, the CGA-TSIG protocol has to be researched first and will be discussed in section 3. The implementation of the proof of concept will then be discussed in section 4, with the results described in section 4.2. A conclusion is given in section 5, followed by a short discussion on possible future work in section 6.

2 Background

The proposed CGA-TSIG protocol is a combination of the existing TSIG protocol and the CGA method, both of which will be explained first. The TSIG protocol will be discussed in section 2.1, followed by the CGA method in section 2.2.

2.1 Transaction Signature (TSIG)

The TSIG protocol [19] provides a means of authenticating DNS messages, like those sent during dynamic DNS updates and DNS zone transfers, as well as responses from a recursive name server, by using shared-key encryption. It uses one-way hashing together with a key to create a signature that is sent with a DNS message. The receiving host can verify the authenticity of the message by validating the signature with the key it shares with the sending host. The key needs to be distributed over a secure channel which might only be possible out-of-band. Table 2.1 shows the wire format of a TSIG resource record which holds the signature and which is added to the message. The meaning of the highlighted fields will be explained in section 2.1.1.

		Field Name	**Field Size**	**Notes**
DNS TSIG RR		Name	variable	used for the key identifier in domain name syntax
		Type	2 octets	must be `TSIG`
		Class	2 octets	must be `ANY`
		TTL	4 octets	must be `0`
		RdLen	2 octets	number of octets in RDATA
	RDATA	Algorithm Name	variable	algorithm name in domain name syntax
		Time Signed	6 octets	seconds since 1 January 1970 UTC
		Fudge	2 octets	seconds of error permitted in Time Signed
		MAC Size	2 octets	number of octets in MAC
		MAC	variable	signature defined by Algorithm Name
		Original ID	2 octets	original message ID
		Error	2 octets	expanded RCODE covering TSIG processing
		Other Len	2 octets	number of octets in Other Data
		Other Data	variable	application-specific other data (*optional*)

Table 2.1: TSIG resource record wire format, with highlighted TSIG variables

2.1.1 Signature creation

When a host signs a DNS message, it will put the signature in the MAC field which is encapsulated by the TSIG resource record. The length of the signature depends on the used signature algorithm. Any TSIG implementation must at least support the HMAC-MD5 algorithm [14], which produces a keyed digest over the message that is taken as the message signature. However, the input digest components consist of more than only the message. The use of these components depends on if the message is a request or response, as well as on TSIG errors.

If a host decides to send a signed request to another host, then the message it wants to send is concatenated with the so-called TSIG variables as a preparation for the digest operation. This is the second digest component. The resulting concatenated block is digested as a single input structure using a key it shares with the other host, and the output hash will be the message signature. The input structure looks as follows:

Component	Notes
DNS message (request)	without TSIG RR
TSIG variables (request)	most of the TSIG RR

The DNS message does not contain the TSIG resource record when fed to the digest algorithm. Nevertheless, the TSIG variables make up most of the TSIG resource record, where the variables are mostly values that are signed in order to increase security. The TSIG variables are highlighted in grey in table 2.1. They are concatenated in the order as shown to become the TSIG variables input component for the digest algorithm. Once the signature has been created, it is put inside the MAC field and the TSIG resource record is then appended to the additional section of the DNS message.

When the other host receives the request and wants to send a signed response, then it uses the same digest components and an additional third component. This is the MAC field from the request message (preceded by its MAC size field), which is concatenated to the front of the message that the host wants to return and the TSIG variables as follows:

Component	Notes
MAC (request)	the concatenated MAC size and MAC data fields from the request
DNS message (response)	without TSIG RR
TSIG variables (response)	most of the TSIG RR fields

The inclusion of the request MAC requires the request message to be signed, and so a response must not be signed if the request was not signed. By including the request MAC, the requesting host can verify that the response it received is the answer to that particular request.

2.1.2 Signature verification

A host receiving a request (referred to as the server) checks the request for the following TSIG errors in this order:

1. A **key error** can occur when the key identifier in the TSIG resource record's name field is not recognised. In this case the server returns an error message that is not signed and that thus contains a TSIG resource record with an empty MAC field. The corresponding TSIG error code is put in the resource record's error field. Two hosts can share multiple keys.
2. When the server's current time deviates too much from the time signed value, then a **time error** is returned in a message that is signed with the same key used by the client. The request MAC digest component is included as is the case with a normal signed response message, unless it does not validate.
3. If the signature in the MAC field cannot be verified, then this is a **signature error**. An unsigned error message (with an empty MAC field) is returned.

If no errors are found, then the server will send a regular signed response message as discussed in the previous section.

When the client receives the response from the server, it tries to validate the response signature. It does this by digesting the same components that the server used to create the signature. As a result, the client must have cached the MAC field before sending the request. If the resulting keyed digest does not match the signature, then the message is not accepted.

If the signature does validate, then the client will check for errors like the server has done when verifying the request. First, the key is checked to be the same as the one used for the request. Then the client checks the time at which the response was signed and generates an error if its own time is outside the valid range. The valid range is defined by the time signed value plus and minus the fudge value. The time check is part of the protocol to counter replay attacks. To prevent these values from being spoofed, they are signed as part of the TSIG variables component. If the server (which uses the same time check rule) responded with a time error, then it would have put the perceived client's time into the time signed field of the response TSIG resource record and its own timestamp in the other data field instead. This is done to make it possible for the client to verify a message containing a time error without failing due to another time error. Finally, the client checks if the server returned a signature error and may retry the request in that case, possibly with a different key if another one is available.

2.1.3 Security

TSIG provides message authentication and data integrity protection on the link between the host sending the signed message and the receiving host, depending on the security provided by the used signature algorithm. As such, it can be used to protect against man-in-the-middle attacks. The provided security relies on a trust relationship between each pair of hosts. If a validating host assumes that only the intended remote host (or group of hosts) knows the same key and if that host is trusted, then it can be assumed that any validated message originated at that host and that its integrity has been preserved during transit. Any message that does not validate due to a signature error or any other TSIG error should be regarded as not authentic and therefore be discarded.

TSIG cannot be used to verify the authenticity of data if it did not originate at the host that is being authenticated, as is the case with a recursive name server (which is not the authoritative source). Although it can be used to secure the last mile, the data that is returned has been fetched from elsewhere by the recursive name server. In order to verify the data's authenticity, the relevant authoritative name server needs to be authenticated which can be done by the recursive name server with methods like DNSSEC. If the client trusts the recursive name server to be faithful then it can be assumed that the data in any answer that has been validated with TSIG has been passed on unaltered and that the server did not falsify any validation results like those from DNSSEC.

Protection against replay attacks has been discussed in the previous section. The fudge value must not be too small in order to allow for clock skew between the two host, but also not too large to keep the window in which replay attacks are possible to a minimum length. A time error could therefore mean that a message was replayed, but it could also occur when the clocks of the two hosts are too much out of sync. Nevertheless, a message that does not validate due to a time error must never be accepted.

2.1.4 Drawbacks

The TSIG protocol is useful for the authentication of hosts during dynamic DNS updates and zone transfers. However, in case of protecting the last mile it becomes less attractive. Since a recursive name server usually has relatively many clients that make use of its DNS service, it would need to maintain a shared key for each of the clients. All these keys, which need to stay secret, somehow have to be delivered securely at the clients and be configured in their stub resolvers. As a result, the protocol scales poorly. When using TSIG for dynamic DNS updates or zone transfers, relatively few hosts will need to have a key configured and it is feasible for a dedicated system administrator to do so. However, securely retrieving a key and using it to configure a stub resolver could pose a problem to a regular user.

Even after a key has been distributed successfully, the problem resurfaces each time the key is replaced when it has been compromised or due to other security reasons. If a recursive name server's complete key database becomes compromised, then the distribution becomes even more cumbersome since the problem has to be overcome again for each of its clients.

Another problem is that mobile devices will be assigned to different recursive name servers regularly. Even though the number of name servers between which a mobile device switches may remain small, it still means that a new shared key needs to be generated and distributed each time such a device contacts a server it has not contacted before. It also does not make much sense for a recursive name server to authenticate its clients, since it generally accepts anonymous requests. However, a shared key is still needed for the client to authenticate the server and the client is also required to sign its queries according to the TSIG specification.

2.2 Cryptographically Generated Addresses (CGA)

The CGA method [6] is primarily used in the SEND protocol [4] as a means of verifying that a message came from a certain IPv6 address by binding a public key to the address. As the name suggests it does this by cryptographically generating the IPv6 address, which will be assigned to the host that holds the associated private key. In the first place, CGA makes it possible to verify that a public key belongs to a certain host, after which a signed message can be authenticated with that public key by using the key's cryptographic algorithm (like RSA [18]).

The second half of the IPv6 address, called the interface identifier, is generated by taking part of the resulting hash of a cryptographic digest operation on the public key and a few other parameters. One of these parameters is the first half of the IPv6 address, called the network prefix or subnet prefix, which is fixed depending on the host's subnet. The complete CGA parameters data structure that is used as input to the digest algorithm is shown in table 2.2.

	Field Name	**Field Size**	**Notes**
	Modifier	16 octets	random bit string
	Subnet Prefix	8 octets	the first 8 octets of the generated IPv6 address
	Collision Count	1 octet	number of duplicate addresses detected
	Public Key	variable	DER-encoded ASN.1 SubjectPublicKeyInfo structure
Extension Fields	Extension 1 Data Type	2 octets	*optional*
	Extension 1 Data Length	2 octets	
	Extension 1 Data Value	variable	
		[2, ..., n-1]	
	Extension n Data Type	2 octets	*optional*
	Extension n Data Length	2 octets	
	Extension n Data Value	variable	

Table 2.2: CGA parameters data structure

All the CGA parameters are publicly available. The modifier is a random value that is used to add randomness to the generated address. The rationale behind the use of a randomly generated modifier is that it would become impossible to link addresses that have been bound to the same public key, thus improving privacy (although this is not relevant for recursive name servers). Nevertheless, the host can still be recognised by the public key itself if it is not changed as well. The collision count is a value that is produced by the CGA generation process as explained in the next section. The public key is specific to the used public-key algorithm and its length is encoded in the ASN.1 structure itself. The CGA parameters data structure can be extended with optional extension fields.

2.2.1 CGA generation

The CGA generation algorithm produces an IPv6 address and has not been specified for IPv4. The generation process involves the creation of two hashes, one of which is used to form the 64-bit interface identifier of the IPv6 address. Apart from the subnet prefix, the public key, and any extension fields, the algorithm takes a fourth input value named *sec*. This is an unsigned three-bit integer with a value between and including 0 and 7, which can be used to exponentially increase the CGA generation cost in order to increase security (as will be discussed in section 2.2.4). After the *sec* value has been chosen, the following process is performed to generate the CGA:

1. A random 128-bit modifier is generated.
2. The CGA parameters data structure (a concatenation of its fields) is digested using the SHA-1 algorithm [10], with the subnet prefix and collision count fields set to zero. The 112 leftmost bits of the resulting hash value are taken as *hash2*.
3. If the 16×*sec* leftmost bits of *hash2* are not zero, then the modifier is incremented by one and the process returns to step 2. Otherwise, the process continues with the next step.
4. The 8-bit collision count is initialised to zero.
5. The CGA parameters data structure, with the subnet prefix and collision count values set indeed, is digested using the SHA-1 algorithm. The 64 leftmost bits of the resulting hash value are taken as *hash1*.
6. The interface identifier is formed from *hash1* by overwriting its three leftmost bits with the *sec* parameter and setting the seventh and eighth bit (denoted as the 'u' and 'g' bits) to zero.
7. The generated interface identifier is concatenated with the 64-bit subnet prefix to form an IPv6 address, with the subnet prefix to the left and the interface identifier to the right.
8. If required, duplicate address detection is performed to see if the generated address is already in use. If this is the case, then the collision count is incremented by one and the process returns to step 5. If a third collision occurs, then the generation process is ceased and an error is reported.
9. If no error occurred then the final CGA parameters data structure will consist of the input subnet prefix, public key, and extension fields, and the final values for the modifier and collision count.

This process is illustrated in figure 2.1. The resulting cryptographically generated IPv6 address is depicted in figure 2.2, with the generated interface identifier highlighted in grey. The host holding the private key that is paired with the digested public key can use that private key to prove that it is associated with the generated address. This is true even if it has not (yet) been assigned the address. The security implications thereof, as well as the reason for including the subnet prefix in the digest operation, will be discussed in section 2.2.4.

2.2.2 CGA verification

When a host receives a signed message with a source IP address that happens to be a cryptographically generated address (CGA), it can verify the message's authenticity by first verifying the address's associated public key. In SEND [4], the CGA signature and public key are included in its packets' option fields in order for the receiving host to be able to perform the verification. With CGA there are usually two verification stages:

- **CGA verification** in which the received public key is authenticated by verifying that is belongs to the host with the source CGA.
- **Signature verification** in which the received message is authenticated by verifying that it was signed with the private key that is associated with the received public key. The relevant public-key algorithm, like RSA, is used to do so.

If the CGA verification succeeds, then it can be assumed that any message for which the signature verification also succeeds is authentic in the sense that it was signed and sent by the host behind the CGA, and that the message's source address cannot have been spoofed. The signature verification process will be discussed more elaborately in section 2.2.3. For the CGA verification, the following steps are executed:

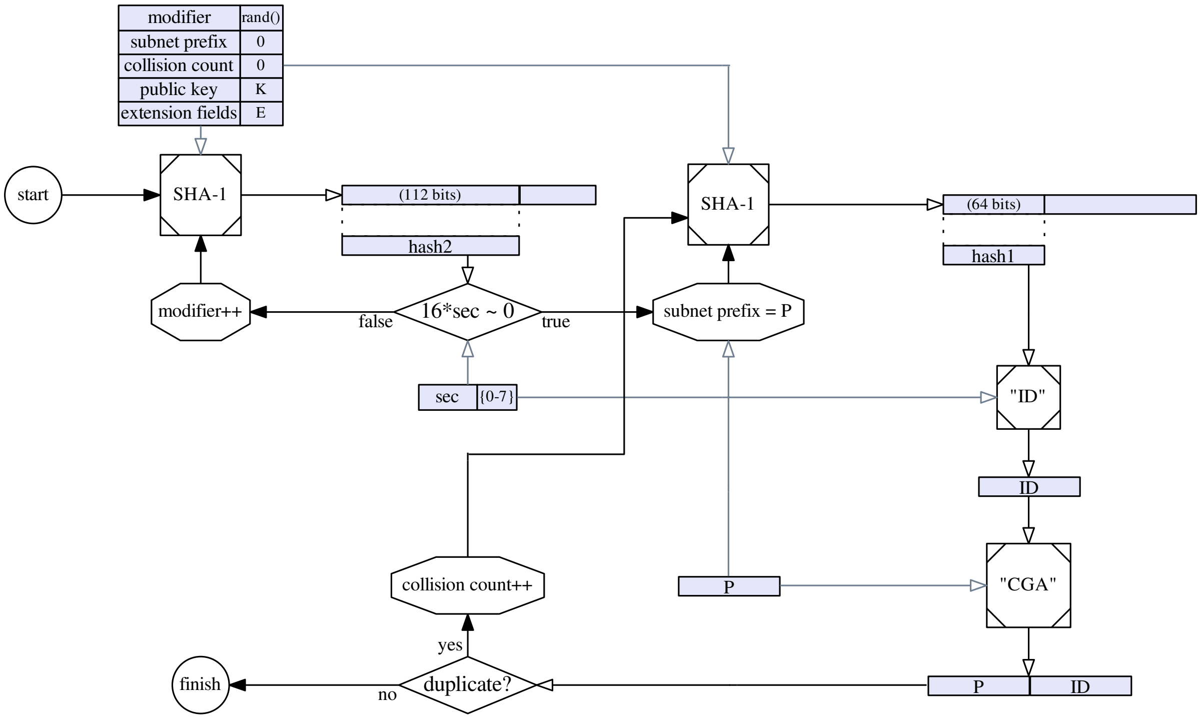

Figure 2.1: Diagram illustrating the CGA generation process

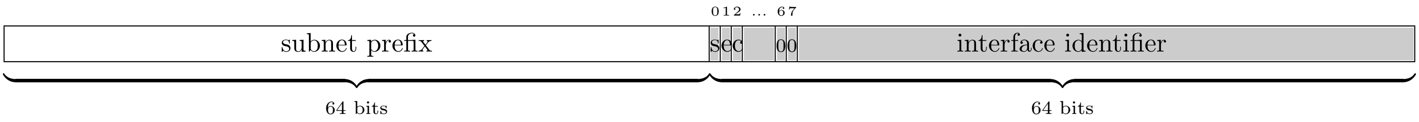

Figure 2.2: CGA format

1. The collision count is checked to be 0, 1, or 2. If it has a different value, then the verification fails.
2. The subnet prefix from the CGA parameters data structure is compared with the subnet prefix of the source address. If they are not equal, then the verification fails.
3. The CGA parameters data structure is digested using the SHA-1 algorithm. The 64 leftmost bits of the resulting hash value are taken as *hash1*.
4. The interface identifier of the source address is compared with *hash1*, ignoring the three leftmost bits (the *sec* parameter) and the seventh and eighth bit (the 'u' and 'g' bits). If any of the remaining 59 bits differ, then the verification fails.
5. The *sec* parameter is extracted from the interface identifier of the address by reading its three leftmost bits.
6. The CGA parameters data structure is digested using the SHA-1 algorithm, with the subnet prefix and collision count fields set to zero. The 112 leftmost bits of the resulting hash value are taken as *hash2*.
7. If any of the 16×*sec* leftmost bits of *hash2* are not zero, then the verification fails. Otherwise, the verification is successful.

If the CGA verification succeeds, then the public key has been authenticated as belonging to the host holding the cryptographically generated address. The next step would be to authenticate the message by verifying its signature with the public key, as will be discussed in the next section.

2.2.3 CGA signatures

According the CGA specification, any message that is to be signed with a CGA-bound key should be concatenated with a 128-bit type tag before it is digested. This is due to security considerations as will be explained in section 2.2.4. The type tag is specific to the protocol in which the CGA method is used. For SEND for example, the type tag is defined to be `0x086F CA5E 10B2 00C9 9C8C E001 6427 7C08`. The specification also prescribes the use of RSA only for the signing operation, specifically the RSASSA-PKCS1-v1_5 signature algorithm [13], together with the SHA-1 hashing algorithm. However, it would be possible to use a different public-key algorithm and hashing algorithm.

The CGA signing operation is no different from regular signing operations, except for the prior concatenation with the 128-bit type tag. To sign a message, a host should do the following:

1. The 128-bit type tag is concatenated with the message. The type tag is put to the left and the message to the right.
2. The resulting concatenation is signed with the RSASSA-PKCS1-v1_5 algorithm, using the private key whose associated public key has been bound to the host's CGA, and with the SHA-1 hashing algorithm.

The message is then sent together with the resulting signature and the CGA parameters data structure from table 2.2 to the intended recipient. The public key in the data structure is the one that was used to generate the CGA and whose associated private key was used to sign the message. The other parameters are the output values of the CGA generation procedure as described in section 2.2.1. The receiving host can verify the message's signature as follows:

1. The CGA verification is performed as described in section 2.2.2. It needs the sender's CGA and the received CGA parameters data structure as input.
2. The relevant protocol's 128-bit type tag and the received message are concatenated in the same way as in the signing operation.
3. The signature is verified with the RSASSA-PKCS1-v1_5 algorithm and the SHA-1 hashing algorithm. The concatenation resulting from the previous step together with the signature and the public key from the CGA parameters data structure are the input for this operation.

If both the CGA verification and the signature verification succeed, then the message can be assumed to be authentic.

2.2.4 Security

One property of the CGA method is that anyone is able to bind a self-generated public key to an IPv6 address. This means that anyone will also be able to positively authenticate the public key by verifying the address. In order for a recipient to know if it can trust a message, even if its signature can be verified with the authenticated public key, it will need to know if it can trust the host behind the source address. The recipient will therefore need to authenticate the source address by some means. If this is not done, then any attacker could spoof the address with another valid CGA and use its own private key to create valid signatures for forged messages. The CGA specification does not elaborate on how the initial authentication of a CGA could be done. This is a separate issue that will require additional research.

If a recipient has authenticated the address and thus cannot be fooled by a different address, then the only approach left for an attacker is to try to impersonate the CGA. Assuming that it is not possible for the attacker to find the original private key, a collision of the hash used for the interface identifier will therefore have to be found for a public key that is paired with the attacker's own private key. This requires the attacker to search for a combination of CGA parameters that yields an interface identifier equal to that of the target address.

However, the CGA specification makes it hard for an attacker to do so. For example, the duplicate address detection collision count has a very limited number of valid values. Therefore, it can also hardly be used in the search for a hash collision. This is important because once an attacker has found a suitable modifier value in order for *hash2* to have enough leading zeros, then the collision count could be used to exhaustively search for a *hash1* without the need to find a new modifier value because the collision count is not used in generating *hash2*. It is therefore important that an implementation does not skip step 1 of the verification process as described in section 2.2.2. The collision count is not included in the *hash2* generation during the CGA generation process for this reason; it would require a host to find a fitting *hash2* again each time an address collision occurs, which would significantly increase the generation cost for greater *sec* values. By limiting the collision count to a small set of values, a denial of service attack during the duplicate address detection process can also be prevented since the process will be stopped after three collisions. In general, the limit of three will be sufficient because it is unlikely that three collisions would occur due to the large address space.

The subnet prefix is included in the digest operation in order to bind it to the generated interface identifier as well. This way, a recipient can verify that the received public key belongs to that exact address and not possibly to an address with the same interface identifier, but with a different subnet prefix. As a side effect, an attacker will need to use a fixed subnet prefix and has less room to play with the parameters in order to find a hash collision. Since the CGA specification prescribes to use the subnet prefix from the CGA parameters data structure during CGA verification, it is important not to skip step 2 of the verification process as described in section 2.2.2. If that check is not done, then an attacker could try different subnet prefix parameter values indeed for finding a hash collision (similar to how an unlimited collision count could be exploited) and yet still use the original subnet prefix in the address. An improvement to the CGA specification could be to omit the subnet prefix from the CGA parameters data structure and let the verifying host simply extract the subnet prefix from the address itself (and then insert it into the data structure). This would rule out any implementation errors that could occur related to this verification step. Also, the subnet prefix is just like the collision count not included in the *hash2* generation and so a host does not need to find a fitting *hash2* again if it changes subnet but still uses the same public key. It can simply use the same modifier value and public key to generate a new interface identifier for the new subnet prefix.

The CGA specification defines the use of the SHA-1 algorithm to generate the hash that will be used to form the address's interface identifier. Of the algorithm's 160 output bits, only 59 are used (the 64 leftmost bits minus the three bits of the *sec* parameter and the two 'u' and 'g' bits). On first sight, this would make it significantly less costly to find a hash collision since the remaining 101 bits can be arbitrary, allowing the SHA-1 output to be one of $2^{101} \approx 2.5 \cdot 10^{30}$ hashes instead of exactly one with an average cost of $O(2^{59})$ [5]. However, the *sec* parameter is used to mitigate this weakness by increasing the number of hash bits that become relevant by generating a second hash, effectively increasing the hash length. It increases the cost of finding a collision of the 59 bits from *hash1* plus the $16 \times sec$ bits from *hash2* with a factor of $2^{16 \times sec}$ to a total average cost of $O(2^{59+16 \times sec})$. As discussed, an attacker cannot use arbitrary values for the subnet prefix and collision count to search for a hash collision. Therefore, the search space is limited to the 2^{128} possible values of the modifier for a particular public key (but multiple public keys can be tried as well). The cost of finding a hash collision with the largest possible *sec* value (which is 7) is on average $O(2^{171})$.

The *sec* value cannot be spoofed since it is part of the address as shown in figure 2.2. If the original *sec* value is greater than zero and the attacker would use a *sec* value equal to zero instead (in which case no bits would need to be zero), then the resulting address will not match the target address. Even if the attacker would set the *sec* value of the resulting address to the original *sec* value while still using a value of zero (or any other value smaller than the original value), then the CGA verification will fail because the verifying host will find too few leading zeros in *hash2*. Although it will take a relatively large amount of time for a legitimate host to generate an address with a greater *sec* value, it does improve security, and the generation process might be started relatively early before an old address is planned to be replaced by a newly generated address.

At the beginning of section 2.2, the randomness of the modifier has been discussed. As said, it does not improve the privacy of the host if it uses the same public key for the different addresses. However, this would only be the case if the recipient needs to verify the CGA. In other contexts, where no CGA verification is needed and the host's public key is not used, it does improve the privacy by making addresses bound to the same public key not linkable. It also allows a host to have multiple addresses that are bound to the same key pair. In case of the last mile, however, the privacy of recursive name servers is not relevant since their service is publicly available.

As mentioned in section 2.2.3, the message that is to be signed is first concatenated with a 128-bit type tag. Each type tag is a randomly generated value from the CGA message type name space and must be uniquely defined for the protocol in which the CGA method is used. The type tag is used to prevent related-protocol attacks, in which a signed message from one protocol is replayed as a message for another protocol which may trigger erroneous behaviour. By using a different type tag for each protocol, this kind of attack can be prevented when one of the two protocols uses its own type tag in the digest operation. If the message was signed with a different type tag, then the signature verification would fail.

Finally, the strength of the signatures created with the private key whose corresponding public key has been bound to a CGA depends on the strength of the used algorithms, like RSA and SHA-1, as well as on the key length. If RSA or SHA-1 are at some point in time considered to be not secure enough, then a different public-key algorithm or digest algorithm could in principle be used. The security that the CGA method provides relates to the authentication of the public key, and a public key of a different algorithm could be bound to an IP address as well.

3 CGA-TSIG

The proposed CGA-TSIG protocol by Rafiee et al. [17] makes use of the existing TSIG resource record, but changes the TSIG specification in that it uses public-key cryptography instead of shared-key cryptography, avoiding the key distribution problem and thus making the protocol scalable. Furthermore, it allows hosts to authenticate other hosts' public key by using CGA, which makes it unnecessary for a user to manually verify the key. Nevertheless, the user will still need to verify the source address as has been discussed in section 2.2.4. Specifically for the last mile problem, the CGA-TSIG proposal makes it possible to use TSIG but without the need for stub resolvers to sign requests (which normally is a requirement of TSIG as mentioned in section 2.1.4). As a result, only the recursive name servers need to have a CGA, while their clients do not.

CGA-TSIG uses the TSIG resource record's other data field to send the CGA parameters from table 2.2 along with the DNS message. The complete wire format of a TSIG resource record as specified in the CGA-TSIG draft is shown in table 3.1. The use of the original TSIG fields has been assumed to be as defined in the TSIG specification, except for the name field that is normally used to identify the shared key, and the MAC field that has become superfluous as the CGA-TSIG specification defines a new signature field. The way these two fields should be handled in case of CGA-TSIG has not been specified in the draft, nor has the size of the field that encodes the length of the CGA-TSIG data field. These specification flaws, as well as the meaning of the dotted and highlighted fields, will be discussed in section 4.2.

Of the newly defined fields in the CGA-TSIG data, some require a bit more explanation. Apart from the algorithm type field which identifies the public-key algorithm used for signing, the field merely denoted as "type" (not to be confused with TSIG's second field) identifies the algorithm that was used to generate the host's IP address. The only value currently defined is '1' for CGA, but other algorithms similar to CGA could be used instead. The IP tag, old public key, old signature, and associated length fields are used to authenticate changes to the host's IP address or public key. The CGA parameters data structure from table 2.2 as defined in the CGA specification is included as-is in the other data field, preceded by a length field indicating its size. The signature field with associated length field is used instead of the MAC field to hold the signature for the message.

3.1 Signature creation

If a host wants to sign a DNS message and use CGA-TSIG as the algorithm to do so, then it will need to digest the message (amongst others) with the SHA-1 hashing algorithm (see section 2.2.3) and use the resulting hash and its private key as input to the signing procedure. This is the private key whose corresponding public key has been bound to the host's CGA. Hence, the host will need to have a CGA generated and assigned, and it will need to cache the associated parameters for later use.

When a DNS message is being signed with CGA-TSIG, a TSIG resource record will be added to the message containing the signature and CGA parameters. These values will be put in the CGA-TSIG data field per table 3.1. The CGA-TSIG specification defines a number of steps that a host should follow to generate all the contents of the CGA-TSIG data field. Any ambiguities and inconsistencies with this procedure will be discussed in section 4.2. The steps are as follows:

1. The required parameters are obtained from cache. These include the CGA parameters from the generation process of the host's CGA as discussed in section 2.2.1. Furthermore, an "old IP address" is retrieved if available. This is a CGA that has been replaced by a newly generated CGA and is put in the IP tag field for verification purposes. If an old IP address is not available, then its value will be set to zero. In this step, the CGA-TSIG specification mentions that the old IP address is concatenated with the modifier, subnet prefix, public key, and collision count, referring to the CGA specification for the exact order of the CGA parameters.
2. The signature is generated. The specification of this step starts again with a concatenation pro-

					Field Name	Field Size	Notes
DNS TSIG RR					Name	variable	*use not specified*
					Type	2 octets	must be `TSIG`
					Class	2 octets	must be `ANY`
					TTL	4 octets	must be `0`
					RdLen	2 octets	number of octets in RDATA
	RDATA				Algorithm Name	variable	must be `CGA-TSIG.`
					Time Signed	6 octets	seconds since 1 January 1970 UTC
					Fudge	2 octets	seconds of error permitted in Time Signed
					MAC Size	2 octets	number of octets in MAC
					MAC	variable	*use not specified*
					Original ID	2 octets	original message ID
					Error	2 octets	expanded RCODE covering TSIG processing
					Other Len	2 octets	number of octets in Other Data
		Other Data			CGA-TSIG Len	*not specified*	number of octets in CGA-TSIG Data
			CGA-TSIG Data		Algorithm Type	2 octets	public-key cryptography algorithm identifier
					Type	2 octets	identifier for the algorithm used in SEND
					IP Tag	16 octets	old IPv6 address if a new one has been generated
					Parameters Len	1 octet	number of octets in Parameters
				Parameters	Modifier	16 octets	random bit string
					Subnet Prefix	8 octets	the first 8 octets of the generated IPv6 address
					Collision Count	1 octet	number of duplicate addresses detected
					Public Key	variable	DER-encoded ASN.1 SubjectPublicKeyInfo structure
					Extension Fields	variable	*optional*
					Signature Len	1 octet	number of octets in Signature
					Signature	variable	the CGA-TSIG signature
					Old Public Key Len	1 octet	number of octets in Old Public Key
					Old Public Key	variable	old public key if a new one has been created
					Old Signature Len	1 octet	number of octets in Old Signature
					Old Signature	variable	signature created with the Old Public Key

Table 3.1: CGA-TSIG resource record wire format as specified in the draft, with dotted fields that form the TSIG variables together with the highlighted fields, where the latter are the only fields among the digest components

cedure, this time with the components in a different order: modifier, public key, collision count, subnet prefix, the DNS message, IP tag (old IP address), and time signed field. These fields (that is, excluding the message itself) are highlighted in grey in table 3.1. The concatenation is signed with the private key that should have been obtained in the previous step. The signature is then added to the newly defined signature field.

3. If the host has changed its key pair, then the specification prescribes to "add the old public key and message, signed by the old private key, to CGA-TSIG data." In contrast, it additionally prescribes to sign only the timestamp from the time signed field with the old private key and to add the resulting signature to the old signature field.

3.2 Signature verification

When a host receives a DNS message signed with CGA-TSIG, which therefore contains a TSIG resource record with the required parameters per table 3.1, it follows a verification procedure to authenticate the message. The CGA-TSIG specification includes different verification procedures for different scenarios, such as when CGA-TSIG is used for zone transfers or for the last mile. Since the scope of this research is limited to the last mile, only the related verification process will be discussed. Any ambiguities and inconsistencies with this procedure will be discussed in section 4.2 as well.

According to the CGA-TSIG specification, the receiving host (or client) should store the acquired public key when it receives a response from that particular recursive name server for the first time. This would allow the client to authenticate the name server when it has changed its IP address. When a stub resolver wants to authenticate the received message, the following steps are executed:

1. The CGA verification procedure is performed as described in section 2.2.2 using the parameters from the parameters field inside the CGA-TSIG data field. If the verification fails, then the public key is considered to be invalid for the message's source IP address and the message is discarded.
2. It is checked if the timestamp from the time signed field adheres to the following equation:

   $$(\text{current\_system\_time} - x) \leq \text{time\_signed} \leq \text{current\_system\_time}$$

   How the value of x should be determined has not been specified, only that it is a number of minutes. If the time signed does not fall in the specified range, then the message is considered to have been replayed and is discarded.
3. The source IP address is verified. If it does not match the known address for the recursive name server, then the message is discarded.
4. The public key that was authenticated in step 1 is used to verify the message's signature (obtained from the signature field inside the CGA-TSIG data field). If the verification fails, then the message is discarded.
5. The public key is verified in the sense that it is checked if it matches any public key that the client saved previously. If a match is found, then the verification procedure is finished and the message is processed. Otherwise, the procedure continues with the next step.
6. The old public key is verified. If its field length is zero or if it does not match a public key in the client's storage, then the message is discarded. Otherwise, the next step is executed.
7. The old signature is verified with the old public key. If the verification succeeds, then "the new public key should be replaced with the old public key" for this name server in the client's storage and the message is processed. Otherwise, the message is discarded.

3.3 Advantage over TSIG

The advantage of CGA-TSIG compared to the original TSIG specification is that it uses public-key cryptography instead of shared-key cryptography. As a result, the key distribution problem as discussed in section 2.1.4 is avoided since the public keys can be sent out in the open, which makes the protocol scalable. It is therefore suitable for protecting the last mile, where there are many clients that contact a particular recursive name server. CGA makes it possible to automatically authenticate the received public key, where the authentication of a shared key as used in the original TSIG specification requires more effort.

Nevertheless, the problem shifts from exchanging and authenticating the key to exchanging and authenticating the IP address. As discussed in section 2.2.4, the CGA has to be authenticated by some means or it could happen that an attacker inserts its own address instead. The IP addresses of recursive name servers are usually automatically configured by methods like DHCP [8][9], but this always happens in the background as is not authenticated. A user is never prompted with the configured addresses and asked to verify them. Regular users who are not aware of what is happening in the background will probably also not try to verify that the addresses belong to their ISP, for example. In order to take away this problem and make the authentication of the IP addresses effortless, this authentication should somehow be automated. However, the problem of the initial IP address configuration is not in the scope of this research project.

4 Proof of concept

As a part of the research project, a proof of concept has been implemented in order to verify the CGA-TSIG specification with regard to the last mile scenario. As a result, only the signature verification procedure relevant to the last mile scenario (as described in section 3.2) has been implemented and tested. By implementing CGA-TSIG, an independent verification of the specification can be performed and most issues related to using CGA-TSIG for the last mile are likely to be found. No other independent implementation was known to be openly available at the time of writing.

4.1 Implementation

The proof of concept, written in C, is based on the `ldns` library[1] (unreleased version 1.6.17) from NLnet Labs. The library already contains TSIG support, which has been extended to support the CGA-TSIG algorithm limited to the last mile scenario. The verification steps related to the authentication of a new public key (steps 5, 6, and 7 as described in section 3.2) have been left out because it is not clear how a client should have obtained the server's new IP address (as will be discussed more elaborately in section 4.2.3) and due to time constraints. The code for the proof of concept can be found in the online repository[2] of NLnet Labs and the file differences can be found in appendix A.

In order to be able to test with cryptographically generated addresses, a small tool has also been created which uses the Scapy6 library (now merged with Scapy[3]) to generate a CGA for a given public key. This implementation, which has been written in Python, can also be found in the repository of NLnet Labs (in the "cga-gen" directory) and in appendix B.

In the following sections, the issues that have been found in the course of the research project (including during the implementation of the proof of concept) will be discussed. The general implementation results will be discussed afterwards.

4.2 Results

The CGA-TSIG proposal [17] as specified contains a number of ambiguities and inconsistencies. Some of these have been found during the implementation of the proof of concept. Others came to light when the protocol was researched while preparing for the implementation, several of which have already been mentioned in the previous sections. Each of the found ambiguities and inconsistencies will be discussed in the following sections, where they are grouped by issues related to the CGA-TIG resource record in section 4.2.1, the signature creation procedure in section 4.2.2, and the signature verification procedure in section 4.2.3. The general results of the implementation of the proof of concept will then be discussed in section 4.2.4.

4.2.1 CGA-TSIG resource record issues

As shown in table 3.1, several details are missing regarding the format of the TSIG resource record as used by CGA-TSIG. The author of the CGA-TSIG draft has been contacted to ask for her vision on filling in these details, which has been used as a guideline while implementing the proof of concept.

[1] `http://www.nlnetlabs.nl/projects/ldns/`
[2] `http://git.nlnetlabs.nl/ldns/?h=cga-tsig`
[3] `http://www.secdev.org/projects/scapy/`

Name field

One detail relates to the name field, which is normally used for identifying the symmetric key that should be used. In case of CGA-TSIG, there is no need to identify a key since a public key is being used instead which is even sent with the message. Nevertheless, it has not been specified how the name field should be handled. The author replied with the statement that it can be set to a random value. This does not, however, seem to have any advantage over simply using the shortest possible fully qualified domain name, which is a single dot (“.”). On the contrary, if the field would be set to a random value of any length larger than that of a single dot, then the packet would become unnecessarily large since the field is not relevant at all. By setting the field to a single dot, the largest possible amount of space is saved.

MAC field and signature field

The use of the MAC field has also not been specified, although it might seem logical to use it for the CGA-TSIG signature. According to the contacted author, however, it should not be used when the signature is created with a key that is not a shared secret as is the case with CGA-TSIG. For this reason, a separate signature field has been defined in the other data field to accommodate for the CGA-TSIG signature.

In her reply, the author states that the MAC field should be left empty, with the MAC size field set to zero. However, the TSIG specification defines the use of MACs based on a *secret key*, and a private key is also a secret key. If it should be interpreted as a *shared* secret key on the other hand, then the TSIG specification could easily be updated by additionally allowing the use of asymmetric keys. This would limit the number of exceptions required at occurrences of the MAC field in the TSIG specification to a minimum, preventing numerous redirections to the signature field where the MAC field would normally have been accessed. After all, both fields serve the same purpose of carrying a signature from one host to the other. Some exceptions are nevertheless inescapable for CGA-TSIG regarding the way in which the field holding the signature is handled. These will be further discussed in section 4.2.2.

TSIG’s other data field

CGA-TSIG uses the other data field of the TSIG resource record to send its data to the intended recipient, like the CGA parameters that the recipient will need in order to be able to do the CGA verification and message authentication. However, the original TSIG specification uses the other data field as a holder for a timestamp in case of a time error, which has been discussed in section 2.1.2. It does not seem to take the use of the other data field by new signature algorithms like CGA-TSIG into account, since it makes no notion of it and specifies that the other data length must be set to ‘6’ (which is the length of the timestamp).

It also does not specify how any extra fields inside the other data field should be formatted, nor does the CGA-TSIG specification. Since a message containing a time error should also be signed according to the TSIG specification, it could happen that the other data field is needed for both the timestamp and the CGA-TSIG data although it is not clear how the two should be combined. A solution would be to reserve the first 6 octets in the other data field for a timestamp in case of a time error, so that it would be clear for an implementation adhering to TSIG that it will find a timestamp at that location. The CGA-TSIG data can then be placed right after the timestamp. If the TSIG error field indicates no time error, then this would not be required and the CGA-TSIG data and its length field could occupy the first 6 octets as usual. If CGA-TSIG is used for the last mile, however, then the double use of the other data field would never occur. This will be further discussed in section 4.2.2.

Sizes of the length-encoding fields

CGA-TSIG defines the CGA-TSIG data field inside the other data field to hold its data. Since this field is of variable length, it is preceded by the CGA-TSIG length field which is used to encode the length of the CGA-TSIG data. However, the size of this length field itself has not been specified, although it should be fixed in order for an implementation to know how many octets belong to that field. According to the contacted author, it should be specified as a 1-octet sized field.

During the implementation of the proof of concept, however, some issues arose related to the size of the length fields defined by the CGA-TSIG specification. Apart from the CGA-TSIG length field, the other length fields inside the CGA-TSIG data field have also been defined as fields with a size of 1 octet. This allows for an encoding of up to 255 octets. However, a 2,048-bit public key would already require a field size of $\frac{2{,}048}{8} = 256$ octets, in which case the field that encodes the length of the parameters data structure comes short. Likewise, the old public key length field cannot encode a key of the same size or greater. As a result, the total length of the CGA-TSIG data cannot be encoded in the CGA-TSIG length field either. During tests of the implementation, no attention was paid to the length of the used public key. It turned out to be a 2,048-bit key, which made the length fields overflow and caused the signature verification to fail. It is strongly recommended to use 2 octets for all the defined length fields. This would allow an encoding of at most 65,535 octets (a factor of already $\frac{65{,}535}{255} = 257$ greater), which is more than sufficient for the current purpose. It would be no use increase it to 3 octets or more, since values greater than 65,535 cannot be encoded in the other data length field (which is also 2 octets long).

Since the CGA parameters data structure is added to the resource record as-is, there is only one length field (the parameters length field) indicating the size of the entire data structure and there are no separate length fields for the variably sized public key and extension fields. This makes it possible to pass the entire data structure to the CGA verification function with minimum effort. However, most of the parameters still need to be parsed separately. The public key, for example, is also needed for the signature verification. During the implementation of the proof of concept, it became clear that the length of the public key must be extracted from the ASN.1 encoding itself before being able to parse it. This makes the parser function unnecessarily complex in that it will need to be able to do more than only parsing simple length fields in order to parse the particular data field. Furthermore, the length of the remaining extension fields has to be deduced from the other parameters' lengths by subtracting them from the value in the parameters length field. It would be easier to add a preceding length field in front of both fields in favour of the parameters length field. Since the parameters need to be parsed separately anyway, it would be no problem to concatenate them before proceeding with the CGA verification operation. On the other hand, however, it might be best to maintain the parameters field with its length field to accommodate for the parameters of alternatives to the CGA method, but additional length fields for variably sized parameters would still come in handy.

Old public key field

The format of the public key inside the CGA parameters data structure has been defined to be a DER-encoded ASN.1 structure according to the CGA specification. However, the format of CGA-TSIG's old public key has not been defined. It might be assumed to be a DER-encoded ASN.1 structure analogous to the format of the public key in the parameters field, but this must be clearly specified in order for implementations to know how to parse it.

Algorithm type field

The data type of the algorithm type field conflicts with the related notes, which describes its contents as the "name of the algorithm" with RSA as the default. It refers to the CGA specification, where the numerical string (in domain name syntax) `1.2.840.113549.1.1.1` is the only identifier that is mentioned with respect to RSA. However, the data type is defined as a 16-bit unsigned integer, instead of a variable domain name type. According to the contacted author, the field should indeed contain an integer, where the value of '0' corresponds to RSA. This definition has only been mentioned in the draft's appendix and not in the main specification text. The identifier '0' would need to be registered with IANA. However, the numerical string mentioned earlier has already been registered and could be used instead. Moreover, the identifier `1.2.840.113549.1.1.5` might even be more suitable [12], since it combines RSA with the digest algorithm SHA-1 (which is used in standard CGA but for which no separate identifier field has been defined). This way, both the public-key algorithm and the digest algorithm can be specified in one field.

4.2.2 Signature creation issues

According to the original TSIG specification [19], a host that wants to sign a DNS message with TSIG has to concatenate the message with the TSIG variables prior to the signing operation. This has been discussed in section 2.1.1. However, the CGA-TSIG signature creation procedure as described in section 3.1 does not fully comply with the TSIG specification. It does also not completely adhere to the CGA specification as described in section 2.2.3.

TSIG variables inclusion

The TSIG variables are highlighted in grey in table 2.1. Since the other data field belongs to the TSIG variables, all the fields inside the other data field as defined by CGA-TSIG should be included in the digest operation as well. However, the CGA-TSIG specification only includes a subset of the TSIG variables in the digest operation. The fields that form this subset are highlighted in table 3.1, where the remaining fields that belong to the TSIG variables are marked with a dotted background. If CGA-TSIG would adhere to the TSIG specification by including the dotted fields as well, then the rise of compatibility issues will be less likely. Nevertheless, an exception has to be defined for the signature field and its length field, which cannot be included since their values will not be known before the digest operation. The same might apply to the old signature field, which will be discussed later.

Also, it does not suffice to digest the time signed field but not the fudge field with regard to preventing replay attacks. By spoofing the fudge field of an old intercepted message and setting it to a sufficiently large value, a signed but possibly outdated message can be replayed and be accepted. If the message contains an answer to a query, for example, then the returned IP address could in the meantime have been changed for the queried domain name. The client would then be directed to the old IP address, which at the time might be in use by a different host. This is the reason why it is part of the TSIG variables as defined by the TSIG specification and an example of why it would be wise to follow that specification.

TSIG variables order

The order in which the highlighted fields are to be concatenated is also not strictly defined. Both step 1 and step 2 of the signature creation process as described in section 3.1 make mention of a concatenation procedure. In step 1, the order in which the fields are mentioned for this procedure are:

- old IP address (the IP tag field)
- modifier
- subnet prefix
- public key
- collision count

Note that the time signed field is not included. It is however included in step 2, where the fields are mentioned in the following order:

- modifier
- public key

- collision count
- subnet prefix
- *DNS message*
- IP tag
- time signed

Here, even the message is put in between the TSIG variables, which is contrary to the TSIG specification. The CGA-TSIG specification also says that all the CGA parameters are to be included, but makes no mention of the extension fields in contrast to the other parameters. Additionally, the specification fails to mention that the collision count, the subnet prefix, and the extension fields should have been put in the cache that is being accessed in step 1. It does mention that (the location of) the private key should have been cached, but it is not among the parameters that are said to be obtained in step 1. Yet, the private key is obviously required for the signing procedure and is referred to in step 2.

To prevent any confusion and ambiguous interpretations, the concatenation procedure should be defined in only one of the steps and the order must be strictly specified. Since the TSIG specification concatenates the TSIG variables in the order in which they sit inside the resource record, CGA-TSIG should adhere to that order and include all the dotted fields in table 3.1 too (except for the signature fields). The TSIG variables should then be concatenated with the DSN message, where the message precedes the variables.

MAC field and signature field

The TSIG specification prescribes that a response must only be signed if the request was signed, and that the MAC field from the request must be concatenated with the DNS message and TSIG variables as described in section 2.1.1. However, the MAC field is not used in CGA-TSIG. By including the MAC field in the digest operation, a requesting host can verify that the received response is an answer to the particular request it sent. For CGA-TSIG, the signature field (together with its length field) could be used instead. It would then need to update the original TSIG specification by defining an additional check of the algorithm name in order to determine if the MAC field or the signature field should be used. However, it may be better to simply use the MAC field as has been discussed in section 4.2.1. The TSIG specification also defines that the responder should use the same symmetric key as the one used by the requester, but for CGA-TSIG this is irrelevant since asymmetric keys are used and each host signs with its own private key. This would also require an update to the TSIG specification.

Last mile signing procedure

In case of a stub resolver sending a query to a recursive name server, the query does not need to be signed since the recursive name server accepts anonymous requests. Therefore, the CGA-TSIG specification states that the CGA-TSIG data field does not need to be included in the query's TSIG resource record. This would however require an exception to be added the TSIG specification in that unsigned messages for which CGA-TSIG verification is requested should be accepted, provided that the receiver has been configured as a recursive name server that accepts anonymous requests. Additionally, it should be specified that no MAC or signature field must be added to the response's digest components. Adding either would not make sense since the MAC field is always empty and since no signature field is added to the query.

The CGA-TSIG specification does not clearly define what a stub resolver should do in order to tell the recursive name server to sign the response with CGA-TSIG. After inquiring the author, it became clear that the stub can do so by setting the algorithm name to "`CGA-TSIG.`" and the CGA-TSIG data length to zero. No CGA-TSIG data needs to be included because the client does not sign the query. The data is only relevant in case of a signed message, where the receiving host will need it in order to verify the public key and the message. In case of the last mile, it is sufficient for the client to merely set the algorithm name in order to signal the recursive name server to sign the response with CGA-TSIG.

It would also not make sense for a recursive name server to check the time signed value of an anonymous request, since it would not have been signed. This field (and the fudge field) can therefore be set to zero by the stub resolver. This has not been defined in the CGA-TSIG specification, although it should define an update to the TSIG specification regarding anonymous requests. As a result, a time error would also never occur when replying to anonymous queries and no timestamp would need to be put in the other data field together with the CGA-TSIG data (as discussed in section 4.2.1).

CGA type tag and the other digest components

To return to step 2 of the described signature creation procedure, the operation after digesting the DNS message and the TSIG variables is to sign the message using the resulting hash. In accordance to the CGA specification, this is done with the RSA algorithm by default. However, no mention is made of a 128-bit type tag which is a digest component required by the CGA specification. As described in section 2.2.3, the message that is to be signed should be concatenated with a type tag that is unique for the protocol that uses CGA verification. In this case, the protocol is CGA-TSIG and the "message" would be the concatenation of the DNS message with the TSIG variables, and possibly the signature field (or otherwise MAC field) from the request. The complete concatenated block would then look as follows:

Component	Notes
128-bit CGA type tag	should be randomly generated and assigned to CGA-TSIG
[signature (request)]	the concatenated signature length and signature data fields from the request [only included in a response to a signed request, not applicable to the last mile]
DNS message	without TSIG RR
TSIG variables	the highlighted and dotted fields from table 3.1 in order of appearance, except for the signature fields

This block should be digested using the SHA-1 algorithm in accordance to the CGA specification, and be used as input to the signing operation, using RSA by default. The resulting signature is then put in the CGA-TSIG signature field (or otherwise the MAC field).

Old public key and old signature

Like the first two steps of the signature creation process, step 3 is not clearly defined either. This step is done if the host created a new key pair and the output is a signature that is to be placed in the old signature field. According to this step the old public key and the message, signed with the old private key, are added to the CGA-TSIG data. This ambiguous definition should probably be interpreted in the sense that the old public key is added to the CGA-TSIG data, and that the signature (rather than the message itself) resulting from the signing procedure with the message and the old private key as input is added to the CGA-TSIG data too.

However, this is subsequently contradicted by stating that the time signed value is solely signed with the old private key and that the resulting signature is put in the old signature field. None of the two definitions specify that the new public key must be signed with the old private key. Nevertheless, it should be in order to be able to authenticate the new public key, which is why the old public key is included in the CGA-TSIG data in the first place.

Instead of signing only the message or time signed value with the old private key, it would be logical to use the same input data structure as is used for the main signing operation with the new private key. Since this data structure will need to be constructed anyway, it can easily be used for creating both the old and the new signature. The data structure also contains the new public key in the parameters field, which belongs to the TSIG variables and thus to the digest components. As a result, the old signature field must then also be excluded from the TSIG variables that are being digested since the signature cannot be known beforehand. By including the new public key in the digest operation, the old public key can be used to authenticate the new public key and therefore the new cryptographically generated address as well if the CGA verification succeeds (which is also newly generated because the new public key would need to have been bound to an address). The handling of the old public key and old signature fields will be further discussed in section 4.2.3.

4.2.3 Signature verification issues

As is the case with the signature creation procedure, the signature verification procedure of CGA-TSIG can be improved. This procedure has been described in section 3.2.

Time signed check

CGA-TSIG defines a time signed check as described in step 2 of the verification process that is different from the definition in the original TSIG specification. Since there seems to be no reason for CGA-TSIG to use a different procedure, there is also no reason for not using the original procedure and creating yet another exception for CGA-TSIG. For comparison, TSIG defines the time signed check as follows:

$$(\text{time\_signed} - \text{fudge}) \leq \text{current\_system\_time} \leq (\text{time\_signed} + \text{fudge})$$

If the current system time falls inside this range, then the time signed check succeeds.

The value of x in the definition described in step 2 could be interpreted to be similar to the fudge value. However, it is not clear why the fudge value itself is not being used since it has been specifically introduced in the TSIG specification for this purpose. It is also not clear if the sender or the recipient of the signed message should determine the value of x. With regard to the TSIG specification, it is clear that the fudge value is set by the sender, which signs it as a part of the TSIG variables to prevent it from being spoofed and ultimately to keep the risk of replay attacks to an acceptable minimum. Also, the fudge value is a number of seconds and x would be a number of minutes. The time signed value is also a number of seconds, so it would be logical for x to be a number of seconds as well. Nevertheless, the fudge value should be used instead.

The definition in step 2 takes into account that the recipient's current system time could be x minutes ahead of the server's time (including transmission time) as given by the time signed value. However, it does not take into account that it could lag behind, which is another possibility. Therefore, if the time signed value is greater than the recipient's current system time, even if it would not be greater than the current system time plus x minutes, then the time signed check would fail. In order to prevent this kind of issues, it is best to follow the original TSIG specification where possible (like with the time signed check definition) and to only update it where necessary.

Old public key and old signature

During the CGA-TSIG verification process, a stub resolver verifies the public key it received in step 5 as described in section 3.2. It was mentioned that the client should have saved the public key it acquired from the recursive name server when it received an answer for the first time. If the public key matches the key that was saved, then the message will be processed. Otherwise, it is checked if the old public key matches.

Unfortunately, this approach could be problematic. If the recursive name server would have generated a new key pair, then consequently it would also have generated a new CGA to which the new public key is bound. However, the CGA-TSIG specification does not elaborate on how the client should know that the server changed its keys and what its new IP address is. If the client does not know, then it will try to contact the server at its old address. If the server released its lease on that address right after generating the new CGA, then the client is dead in the water. It would therefore be inevitable for the server to hold on to the old address for a certain period of time to allow its clients to still contact the server, while asking them to update the address when they do so. The new address could be put in the IP tag field (contrary to the definition that it can only hold an old IP address), which is signed since it is part of the TSIG variables. This way, the client can authenticate the new address and use it the next time it contacts the server, at which point it will receive the new public key. If a client misses the transition period in which the server holds on to both addresses, however, then it would still be unable to retrieve the new address.

On the other hand, if the client is believed to have received an update of the server's IP address via a method like DHCP [8][9], then the server will not need to hold on to its old address and no transition period is required. It would then indeed be necessary to send the old public key along in order for the client to be able to authenticate the new public key and address. When the client contacts the server at the new address, it will receive the new public key and find that it does not match any key it saved earlier. It will then check if the old public key matches a saved key as described in step 6 of the verification process. If this is the case, then the signature in the old signature field will be verified in step 7 using the old public key and whatever fields were used in the digest operation. These input fields should have been defined in step 3 of the signature creation operation, as described in section 3.1. However, this definition is ambiguous as has been discussed in section 4.2.2.

Nevertheless, in applications other than the last mile (like zone transfers) it cannot be assumed that the hosts will receive IP address updates via methods similar to DHCP. Each host that might expect requests from another host will need to hold on to its old IP address if it generated a new one for a period of time, and use the IP tag field as described earlier. However, a host with a new CGA that wants to send a request can do so by simply using its new address as the source address and sending the old public key with an associated old signature along as described. Nonetheless, if it also expects requests then it should still hold on to its old address until all hosts it knows have been notified of the new address. It would clarify the rationale behind the way the new public key is authenticated using the old public key if the CGA-TSIG specification would elaborate on how a new IP address is assumed to be obtained in the different scenarios.

The verification process defined for the last mile scenario apparently assumes that the new IP address is propagated via an automated method like DHCP. If the new address has been bound to the same public key as the one that was also bound to the old address, then the key will be recognised and the verification process is finished after step 5. If a new public key was used, then the process continues with step 6. Assuming that the new public key replaces the old public key of a particular recursive name server and that the new CGA is thus assigned to that same server, then the old public key can be used to authenticate this change. However, it is possible that the new CGA is an address of a different server that does not have an old public key. The server would therefore leave the old public key field empty and step 6 would fail, in which case the message would be discarded. Nevertheless, this server could still be legitimate and it would be better to treat the new CGA as any other initially configured CGA by attempting to authenticate the CGA, instead of discarding the message right away. Also, step 7 defines that the new public key should be replaced with the old public key if the verification of the old signature succeeds, but this should be corrected by switching the words "new" and "old" in order for this definition to make sense.

Order of the verification steps

During the implementation of the proof of concept, the signature verification steps were put in a different order than defined. By switching step 1 and step 3 (as described in section 3.2), the relatively cheap operations of checking if the source IP address matches the address to which the stub resolver dispatched its query and the time signed check can be done before the relatively expensive CGA verification. This way, less effort is required by the recipient if the IP address check or the time signed check fails, reducing the impact of denial of service attacks.

4.2.4 General proof of concept results

The proof of concept has been based on the CGA-TSIG specification with the last mile scenario in mind, as well as on the replies from the author of the CGA-TSIG specification to the inquiries. As a result, a working implementation has been created even though many ambiguities and inconsistencies have been identified while implementing the proof of concept, as has been described in the previous sections. The implementation deviates from the specification with regard to the size of the length-encoding fields, for which 2 octets have been used instead of 1 as discussed in section 4.2.1. It does follow the CGA-TSIG specification regarding the TSIG variables and other digest components (as discussed in section 4.2.2), using only the highlighted fields from table 3.1 and not the dotted fields. Nevertheless, the code has been written in such a way that the remaining TSIG variables can easily be added and it makes no difference if these are included or not in order to demonstrate that the CGA-TSIG implementation works.

Additionally, a bug in the `ldns` library was found while implementing the proof of concept. This bug relates to the SHA-1 implementation of the library, which caused the input data structure to be modified. Since the same data structure was used to create both *hash1* and *hash2* subsequently, the first *sec* times two octets of *hash2* were therefore not equal to zero although they should have been because a *sec* value greater than 0 was used at the time. When the SHA-1 function of OpenSSL was used thereafter instead, the CGA verification did succeed. Evidently, the bug manifested itself as soon as the *sec* value was raised from 0 to 1, because only then any bits of *hash2* became relevant. In the meantime, a patch has been implemented by the `ldns` team.

The main additions to the existing TSIG code of the `ldns` library include the public-key signature generation and verification support (complementing the existing symmetric-key implementation), as well as the CGA verification function. All additions can be found in appendix A. In order to generate CGAs, a third-party implementation of the CGA generation procedure was used, which made testing the implemented CGA verification function more reliable. No errors were found with regard to this implementation. The code of the created tool that was used to call the third-party CGA generation function can be found in appendix B.

5 Conclusion

Since the CGA-TSIG specification still has the status of a draft, it is understandable that there are a number of issues that will need some attention. Nevertheless, the draft in its current form is far from close to a final specification. There is a large number of problems that were encountered while researching and implementing the protocol, which have been discussed in section 4.2 and for which solutions have been suggested.

The main issue that has been found is that the current CGA-TSIG specification does not adhere to the original TSIG specification very well. This makes the protocol look like a completely new specification, rather than an extension to TSIG. However, it was a logical choice to specify CGA-TSIG as an extension to TSIG, since it already provides an existing framework for signing DNS messages. Nevertheless, CGA-TSIG should follow the TSIG specification more strictly so that minimum effort is required to extend existing TSIG implementations. For example, all the TSIG variables should be included (but an exception has to be made for the signature fields), and there is no need to unnecessarily update the TSIG specification as is the case with the time signed check.

Since CGA-TSIG is an algorithm type for the TSIG protocol, it provides DNS message authentication and data integrity on the link between two hosts. The strength of this security depends on the security provided by the used algorithm type, which is CGA-TSIG in this case. By default, CGA-TSIG uses RSA to sign DNS messages. However, CGA can be seen as a framework similar to TSIG in the sense that a different signature algorithm could be chosen. As a result, the security of CGA-TSIG depends on the chosen public-key algorithm and RSA could be replaced as soon as it is deemed to be not secure enough. Since CGA-TSIG uses a public-key algorithm for signing, it can be made as secure as the chosen public-key algorithm. Additionally, the key length can be increased to improve security.

Nevertheless, CGA has another security aspect in the sense that it cryptographically binds a public key to an IP address. By binding it to an address, the public key can be authenticated as belonging to the host behind the IP address. The strength of this bond must be great enough to prevent attackers from impersonating a host behind a certain CGA. This can be done by using a high *sec* value, which can be increased depending on the required strength in order to provide the required security.

By using public-key cryptography, CGA-TSIG avoids the key distribution problem that exists with regular (shared-key) TSIG which makes it suitable for the last mile, where there are many clients that contact a recursive name server. The clients can simply request the server's public key and verify that messages signed with the corresponding private key come from the server and not from a different host. To be able to do so, the server will need to have a CGA to which the public key has been bound and which is used to receive requests from clients. Since the recursive name server accepts anonymous requests, the clients do not need to be authenticated and so they do not need to have a key pair and a CGA themselves. This keeps the burden of generating CGA's (with the associated *sec* values) at the side of the servers.

After implementing the proof of concept, it was found that CGA-TSIG can work as believed to be intended. However, CGA only works on IPv6 networks and it might take a while before IPv6 is in widespread use. As a result, it might also take a while before CGA-TSIG becomes useful to the general public as a solution to the last mile problem. Nevertheless, there is still the problem of the initial IP address authentication. In order for a client to be certain that it received the CGA of a trusted recursive name server, the address would somehow need to be verified. The initial CGA authentication would need to be automated since the average user is not likely to be aware of the importance of verifying the address and would therefore not do so manually. Solutions to this problem might exist, but research to these solutions was out of the scope of the project. Nevertheless, it would be useful for the CGA-TSIG specification to elaborate on how this initial CGA authentication can be done since it is an important link in the chain of security that CGA-TSIG provides. All in all, however, CGA-TSIG could prove to be a viable solution to the last mile problem of DNS once the use of IPv6 is common and an automated CGA authentication method has been outlined.

6 Future work

Automated solutions to the initial CGA authentication might exist. However, research to these solutions was out of the scope of this project. Future research could be done on possible solutions in order to be able to automatically authenticate a recursive name server's CGA, which is essential to the security that CGA-TSIG provides with regard to the last mile problem.

Future research could also be done regarding the impact on the performance of signing and verifying each DNS message with public-key cryptography, as used by CGA-TSIG. This could best be done when the CGA-TSIG specification is complete. It could also be investigated how the protocol performs compared to for example DNSCurve, which uses public-key cryptography to encrypt the complete message.

References

[1] R. Arends, R. Austein, R. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements, March 2005. URL http://tools.ietf.org/html/rfc4033.

[2] R. Arends, R. Austein, R. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions, March 2005. URL http://tools.ietf.org/html/rfc4034.

[3] R. Arends, R. Austein, R. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions, March 2005. URL http://tools.ietf.org/html/rfc4035.

[4] J. Arkko, J. Kempf, B. Zill, and P. Nikander. SEcure Neighbor Discovery (SEND), March 2005. URL http://tools.ietf.org/html/rfc3971.

[5] T. Aura. Cryptographically Generated Addresses (CGA). *Proceedings of the 6th Information Security Conference(ISC'03), Bristol, UK, LNCS*, 2851:29–43, 2003.

[6] T. Aura. Cryptographically Generated Addresses (CGA), March 2005. URL http://tools.ietf.org/html/rfc3972.

[7] M. Dempsky. DNSCurve: Link-Level Security for the Domain Name System, February 2010. URL http://tools.ietf.org/search/draft-dempsky-dnscurve-01.

[8] R. Droms. DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6), December 2003. URL http://tools.ietf.org/html/rfc3646.

[9] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6), July 2003. URL http://tools.ietf.org/html/rfc3315.

[10] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1), September 2001. URL http://tools.ietf.org/html/rfc3174.

[11] P. Hoffman. The Tao of IETF: A Novice's Guide to the Internet Engineering Task Force, November 2012. URL http://www.ietf.org/tao.html.

[12] The Internet Assigned Numbers Authority (IANA). Data Structure for the Security Suitability of Cryptographic Algorithms (DSSC), November 2009. URL http://www.iana.org/assignments/dssc/dssc.xhtml.

[13] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) 1: RSA Cryptography Specifications Version 2.1, February 2003. URL http://tools.ietf.org/html/rfc3447.

[14] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication, February 1997. URL http://tools.ietf.org/html/rfc2104.

[15] G. Michaelson, P. Wallström, R. Arends, and G. Huston. Rolling Over DNSSEC Keys. *The Internet Protocol Journal*, 13(1):2–16, 2010.

[16] H. Rafiee and C. Meinel. A Secure, Flexible Framework for DNS Authentication in IPv6 Autoconfiguration. *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 165–172, 2013.

[17] H. Rafiee, M. von Loewis, and C. Meinel. Transaction SIGnature (TSIG) using CGA Algorithm in IPv6, September 2013. URL http://tools.ietf.org/html/draft-rafiee-intarea-cga-tsig-06.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] P. Vixie, O. Gudmundsson, D. Eastlake, and B. Wellington. Secret Key Transaction Authentication for DNS (TSIG), May 2000. URL http://tools.ietf.org/html/rfc2845.

Acknowledgements

After a struggle of several years to complete my Master's, I am glad and relieved to have finally arrived at this point. I would therefore like to thank everyone who helped me to get this far very, very much, in particular my supervisors Matthijs Mekking from NLnet Labs and Jeroen van der Ham from the University of Amsterdam who both gave me the great support and feedback I needed. I would also want to thank the rest of the NLnet Labs team for helping me out with the problems I encountered and the friendly stay they provided to me at the company, as well as the System and Network Engineering team for their support and great lessons. Last, but certainly not least, I want to thank my family and friends for their support and sympathy, in particular my parents to whom the last few years have proven to be a stress test as well (and yes, they passed). Once again, thank you!

Marc Buijsman

Appendices

The code in appendix A shows the implementation of the proof of concept based on the `ldns` library.

The tool in appendix B was created in order to be able to generated CGA's by calling the Scapy6 library.

Appendix A Proof of concept implementation

tsig.c (additions and deletions)

```
5d4
+  * and CGA-TSIG [draft-rafiee-intarea-cga-tsig-06]
123d121
+       *result_len = 0; // =bugfix?
276,1050d273
+ /**
+  * returns a new initialized ldns_cga_rdfs structure.
+  * \param[out] rdfs the output structure (will be allocated)
+  * \return status (OK if success)
+  */
+ static ldns_status
+ ldns_cga_rdfs_new(ldns_cga_rdfs **rdfs)
+ {
+    if (!rdfs) {
+       return LDNS_STATUS_NULL;
+    }
+
+    *rdfs = LDNS_MALLOC(ldns_cga_rdfs);
+
+    if (!*rdfs) {
+       return LDNS_STATUS_MEM_ERR;
+    }
+
+    /* initialize */
+    (*rdfs)->algo_name = NULL;
+    (*rdfs)->type = NULL;
+    (*rdfs)->ip_tag = NULL;
+    (*rdfs)->modifier = NULL;
+    (*rdfs)->prefix = NULL;
+    (*rdfs)->coll_count = NULL;
+    (*rdfs)->pub_key = NULL;
+    (*rdfs)->ext_fields = NULL;
+    (*rdfs)->sig = NULL;
+    (*rdfs)->old_pub_key = NULL;
+    (*rdfs)->old_sig = NULL;
+
+    return LDNS_STATUS_OK;
+ }
+
+
+ /**
+  * frees the ldns_cga_rdfs structure and its components.
+  * \param[in] rdfs pointer to the ldns_cga_rdfs structure
+  */
+ static void
+ ldns_cga_rdfs_deep_free(ldns_cga_rdfs *rdfs)
+ {
+    if (!rdfs) {
+       return;
+    }
+
+    ldns_rdf_deep_free(rdfs->algo_name);
+    ldns_rdf_deep_free(rdfs->type);
+    ldns_rdf_deep_free(rdfs->ip_tag);
+    ldns_rdf_deep_free(rdfs->modifier);
+    ldns_rdf_deep_free(rdfs->prefix);
+    ldns_rdf_deep_free(rdfs->coll_count);
+    ldns_rdf_deep_free(rdfs->pub_key);
+    ldns_rdf_deep_free(rdfs->ext_fields);
+    ldns_rdf_deep_free(rdfs->sig);
+    ldns_rdf_deep_free(rdfs->old_pub_key);
+    ldns_rdf_deep_free(rdfs->old_sig);
+   LDNS_FREE(rdfs);
+ }
+
+ /**
+  * checks if a number of bytes are available.
+  * \param[in] pos the current position
+  * \param[in] count the number of bytes
+  * \param[in] size the size of the buffer
+  * \return boolean int indicating if it is available
+  */
+ static int
+ ldns_cga_available(size_t pos, int32_t count, uint16_t size)
+ {
+    if (count < 0) {
+       return 0;
+    }
+
+    return (pos + (size_t)count <= (size_t)size);
+ }
+
+ /**
+  * convert CGA-TSIG data to an RDF.
+  * \param[in] rdfs the ldns_cga_rdfs structure
+  * \param[in/out] rdf pointer to the relevant field in rdfs (RDF will be allocated)
+  * \param[in] data the data
+  * \param[in] len the length of data (must be negative if not variable)
+  * \param[in] h2n positive to do host to network conversion, if data is not in network order (0 otherwise)
+  * \return status (OK if success)
+  */
+ static ldns_status
+ ldns_cga_data2rdf(ldns_cga_rdfs *rdfs, ldns_rdf **rdf, const void *data, int len, uint8_t h2n)
+ {
+   ldns_rdf_type type;
+   size_t size;
+
+   if (rdf == &(rdfs->algo_name)) {
+     size = CT_ALGO_NAME_SIZE;
+   } else if (rdf == &(rdfs->type)) {
+     size = CT_TYPE_SIZE;
+   } else if (rdf == &(rdfs->ip_tag)) {
+     size = CT_IP_TAG_SIZE;
+   } else if (rdf == &(rdfs->modifier)) {
+     size = CT_MODIFIER_SIZE;
+   } else if (rdf == &(rdfs->prefix)) {
+     size = CT_PREFIX_SIZE;
+   } else if (rdf == &(rdfs->coll_count)) {
+     size = CT_COLL_COUNT_SIZE;
+   } else if (rdf == &(rdfs->pub_key)
+           || rdf == &(rdfs->ext_fields)
+           || rdf == &(rdfs->sig)
+           || rdf == &(rdfs->old_pub_key)
+           || rdf == &(rdfs->old_sig)) {
+     if (len < 0) {
+       return LDNS_STATUS_ERR;
+     }
+     size = (size_t)len;
+   } else {
+     return LDNS_STATUS_ERR;
+   }
+
+   if (len < 0 && size == 1) {
+     type = LDNS_RDF_TYPE_INT8;
+   } else if (len < 0 && size == 2) {
+     type = LDNS_RDF_TYPE_INT16;
+   } else {
+     type = LDNS_RDF_TYPE_UNKNOWN;
+   }
+
+   if (h2n && type == LDNS_RDF_TYPE_INT8) {
+     *rdf = ldns_native2rdf_int8(type, *(uint8_t*)data);
+   } else if (h2n && type == LDNS_RDF_TYPE_INT16) {
+     *rdf = ldns_native2rdf_int16(type, *(uint16_t*)data);
+   } else {
+     *rdf = ldns_rdf_new_frm_data(type, size, data);
+   }
+
+   if (!*rdf) {
+     return LDNS_STATUS_MEM_ERR;
+   }
+
+   return LDNS_STATUS_OK;
+ }
+
+ /**
+  * convert CGA-TSIG data to host representation.
+  * \param[in] data the data
+  * \param[in] len the length of data
+  * \return the integer representation
+  */
+ static uint16_t
+ ldns_cga_data2host(void *data, size_t len)
+ {
+   if (len == 1) {
+     return (uint16_t)*(uint8_t*)data;
+   } else if (len == 2) {
+     return ldns_read_uint16(data);
+   }
+   return 0;
+ }
+
+ /**
+  * copies the CGA-TSIG data fields to RDFs, assuming it is at front of Other Data; could perhaps be
     implemented like RDATA parsing.
+  * \param[in] other_data_rdf pointer to the Other Data RDF
+  * \param[out] rdfs the output ldns_cga_rdfs structure (will be allocated)
+  * \param[out] pubk the parsed RSA public key (will be allocated)
+  * \param[out] opubk the parsed old RSA public key (will be allocated, or NULL if none)
+  * \return status (OK if success)
+  */
+ static ldns_status
+ ldns_tsig_od2cga_rdfs(ldns_rdf *other_data_rdf, ldns_cga_rdfs **rdfs, RSA **pubk, RSA **opubk)
+ {
+   uint16_t other_len, cga_tsig_len, param_len, sig_len, pubk_len, old_pubk_len, old_sig_len;
+   int ext_len;
+   uint8_t *data, *pubkp;
+   uint32_t pos = 0;
+   ldns_status status;
+
+   if (!other_data_rdf) {
+     return LDNS_STATUS_NULL;
+   }
+
+   other_len = (uint16_t)ldns_rdf_size(other_data_rdf);
+
+   /* first 2 bytes encode other data's length */
+   if (other_len <= 2) {
+     return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+   }
+
+   /* point to the first byte of the real other data */
+   data = ldns_rdf_data(other_data_rdf) + 2;
+   other_len -= 2;
+
+   /* get cga-tsig len */
+   if (!ldns_cga_available(pos, CT_LEN_SIZE, other_len)) {
```

```
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    cga_tsig_len = ldns_cga_data2host(data + pos, CT_LEN_SIZE);
+
+    /* check size constraints */
+    if (cga_tsig_len + CT_LEN_SIZE > other_len) {
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    /* return if no CGA-TSIG data */
+    if (cga_tsig_len == 0) {
+      return LDNS_STATUS_NO_DATA;
+    }
+
+    /* point to the first byte of the real CGA-TSIG data */
+    data = data + CT_LEN_SIZE;
+    pos = 0;
+
+    /* allocate structure holding the RDFs */
+    status = ldns_cga_rdfs_new(rdfs);
+
+    if (status != LDNS_STATUS_OK) {
+      return status; // rdfs has not been allocated yet
+    }
+
+    /* get algorithm name */
+    if (!ldns_cga_available(pos, CT_ALGO_NAME_SIZE, cga_tsig_len)) {
+      LDNS_FREE(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->algo_name), data + pos, -1, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      LDNS_FREE(*rdfs);
+      return status;
+    }
+
+    // algo must be 0 (RSA) for now
+    if (ldns_cga_data2host(ldns_rdf_data((*rdfs)->algo_name), CT_ALGO_NAME_SIZE) != 0) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_UNKNOWN_ALGO;
+    }
+
+    pos += CT_ALGO_NAME_SIZE;
+
+    /* get type */
+    if (!ldns_cga_available(pos, CT_TYPE_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->type), data + pos, -1, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return status;
+    }
+
+    // type must be 1 (CGA) for now
+    if (ldns_cga_data2host(ldns_rdf_data((*rdfs)->type), CT_TYPE_SIZE) != 1) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_UNKNOWN_ALGO;
+    }
+
+    pos += CT_TYPE_SIZE;
+
+    /* get IP tag */
+    if (!ldns_cga_available(pos, CT_IP_TAG_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->ip_tag), data + pos, -1, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return status;
+    }
+
+    pos += CT_IP_TAG_SIZE;
+
+    /* get param len */
+    if (!ldns_cga_available(pos, CT_PARAM_LEN_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    param_len = ldns_cga_data2host(data + pos, CT_PARAM_LEN_SIZE);
+
+    pos += CT_PARAM_LEN_SIZE;
+
+    // expect parameters (i.e. param_len > 0)
+    if (param_len == 0 || !ldns_cga_available(pos, param_len, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    /* get modifier */
+    if (!ldns_cga_available(pos, CT_MODIFIER_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->modifier), data + pos, -1, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return status;
+    }
```

```
+
+    pos += CT_MODIFIER_SIZE;
+
+    /* get subnet prefix */
+    if (!ldns_cga_available(pos, CT_PREFIX_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->prefix), data + pos, -1, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return status;
+    }
+
+    pos += CT_PREFIX_SIZE;
+
+    /* get collision count */
+    if (!ldns_cga_available(pos, CT_COLL_COUNT_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->coll_count), data + pos, -1, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return status;
+    }
+
+    pos += CT_COLL_COUNT_SIZE;
+
+    /* get public key */
+    ext_len = param_len - CT_MODIFIER_SIZE - CT_PREFIX_SIZE - CT_COLL_COUNT_SIZE; // max length
+
+    // expect a public key by default (i.e. ext_len > 0)
+    if (ext_len <= 0 || !ldns_cga_available(pos, ext_len, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    // 1.2.840.113549.1.1.1 = 2A 86 48 86 F7 0D 01 01 01 (RSA ID)
+
+    pubkp = data + pos;
+
+    *pubk = d2i_RSA_PUBKEY(NULL, (const unsigned char**)&pubkp, ext_len);
+
+    if (!*pubk) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      return LDNS_STATUS_ERR;
+    }
+
+    pubk_len = (uint16_t)(pubkp - (data + pos));
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->pub_key), data + pos, (int)pubk_len, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      RSA_free(*pubk);
+      return status;
+    }
+
+    pos += pubk_len;
+
+    /* get extension fields (if any) */
+    ext_len -= pubk_len;
+
+    if (ext_len > 0) {
+      status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->ext_fields), data + pos, ext_len, 0);
+
+      if (status != LDNS_STATUS_OK) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        return status;
+      }
+
+      pos += ext_len;
+    }
+
+    /* get signature len */
+    if (!ldns_cga_available(pos, CT_SIG_LEN_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      RSA_free(*pubk);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    sig_len = ldns_cga_data2host(data + pos, CT_SIG_LEN_SIZE);
+
+    pos += CT_SIG_LEN_SIZE;
+
+    /* get signature */
+    // expect a signature (i.e. sig_len > 0)
+    if (sig_len == 0 || !ldns_cga_available(pos, sig_len, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      RSA_free(*pubk);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->sig), data + pos, (int)sig_len, 0);
+
+    if (status != LDNS_STATUS_OK) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      RSA_free(*pubk);
+      return status;
+    }
+
+    pos += sig_len;
+
+    /* get old public key len */
+    if (!ldns_cga_available(pos, CT_OLD_PK_LEN_SIZE, cga_tsig_len)) {
```

```
+      ldns_cga_rdfs_deep_free(*rdfs);
+      RSA_free(*pubk);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    old_pubk_len = ldns_cga_data2host(data + pos, CT_OLD_PK_LEN_SIZE);
+
+    pos += CT_OLD_PK_LEN_SIZE;
+
+    /* get old public key (if any) */
+    if (old_pubk_len > 0) {
+      if (!ldns_cga_available(pos, old_pubk_len, cga_tsig_len)) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+      }
+
+      pubkp = data + pos;
+
+      *opubk = d2i_RSA_PUBKEY(NULL, (const unsigned char**)&pubkp, old_pubk_len);
+
+      if (!*opubk) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        return LDNS_STATUS_ERR;
+      }
+
+      if ((uint16_t)(pubkp - (data + pos)) != (uint16_t)old_pubk_len) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        RSA_free(*opubk);
+        return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+      }
+
+      status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->old_pub_key), data + pos, (int)old_pubk_len, 0);
+
+      if (status != LDNS_STATUS_OK) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        RSA_free(*opubk);
+        return status;
+      }
+
+      pos += old_pubk_len;
+    } else {
+      *opubk = NULL;
+    }
+
+    /* get old signature len */
+    if (!ldns_cga_available(pos, CT_OLD_SIG_LEN_SIZE, cga_tsig_len)) {
+      ldns_cga_rdfs_deep_free(*rdfs);
+      RSA_free(*pubk);
+      RSA_free(*opubk);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    old_sig_len = ldns_cga_data2host(data + pos, CT_OLD_SIG_LEN_SIZE);
+
+    pos += CT_OLD_SIG_LEN_SIZE;
+
+    /* get old signature (if any) */
+    if (old_sig_len > 0) {
+      if (!ldns_cga_available(pos, old_sig_len, cga_tsig_len)) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        RSA_free(*opubk);
+        return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+      }
+
+      status = ldns_cga_data2rdf(*rdfs, &((*rdfs)->old_sig), data + pos, (int)old_sig_len, 0);
+
+      if (status != LDNS_STATUS_OK) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        RSA_free(*opubk);
+        return status;
+      }
+
+      pos += old_sig_len;
+    }
+
+    /* check size constraints */
+    if (ldns_cga_available(pos, 1, cga_tsig_len)) {
+        ldns_cga_rdfs_deep_free(*rdfs);
+        RSA_free(*pubk);
+        RSA_free(*opubk);
+        return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    return LDNS_STATUS_OK;
+  }
+
+  /**
+   * concatenates data fields.
+   * \param[in] rdfs an array of pointers to the to-be-concatenated RDFs in order
+   * \param[in] num the number of elements contained by rdfs
+   * \param[out] buffer the output buffer (will be allocated)
+   * \return status (OK if success)
+   */
+  static ldns_status
+  ldns_tsig_concat_data(ldns_rdf **rdfs, uint8_t num, ldns_buffer **buffer)
+  {
+    uint8_t i;
+    uint32_t size = 0;
+
+    if (!rdfs) {
+      return LDNS_STATUS_NULL;
+    }
+
+    for (i = 0; i < num; i++) {
+      if (rdfs[i]) {
```

```
+        size += ldns_rdf_size(rdfs[i]);
+      }
+    }
+
+    *buffer = ldns_buffer_new(size);
+
+    if (!*buffer) {
+      return LDNS_STATUS_MEM_ERR;
+    }
+
+    for (i = 0; i < num; i++) {
+      if (rdfs[i] && ldns_rdf_size(rdfs[i]) > 0) {
+        ldns_buffer_write(*buffer, ldns_rdf_data(rdfs[i]), ldns_rdf_size(rdfs[i]));
+      }
+    }
+
+    return LDNS_STATUS_OK;
+ }
+
+
+ /**
+  * concatenates the fields that are input for the CGA signature creation and
+  * verification operations (not to be confused with the CGA verification, see
+  * the ldns_cga_verify() function).
+  * \param[in] pkt_wire the wire without TSIG RR (message to be signed)
+  * \param[in] pkt_wire_size the wire size
+  * \param[in] time_signed_rdf the time signed field (all other fields defined
+  *            by TSIG should probably be included too)
+  * \param[in] rdfs the CGA-TSIG fields from Other Data that are to be included
+  * \param[out] buffer the output buffer (will be allocated)
+  * \return status (OK if success)
+  */
+ static ldns_status
+ ldns_cga_concat_msg(uint8_t *pkt_wire, size_t pkt_wire_size,
+     ldns_rdf *time_signed_rdf, ldns_cga_rdfs *rdfs, ldns_buffer **buffer)
+ {
+    ldns_status status;
+    ldns_rdf *wire_rdf;
+
+    if (!pkt_wire || !time_signed_rdf || !rdfs) {
+      return LDNS_STATUS_NULL;
+    }
+
+    /* temporarily encapsulate the wire in an RDF */
+    wire_rdf = ldns_rdf_new(LDNS_RDF_TYPE_UNKNOWN, pkt_wire_size, pkt_wire);
+
+    if (!wire_rdf) {
+      return LDNS_STATUS_MEM_ERR;
+    }
+
+    // NOTE: no 128-bit type tag defined for CGA-TSIG
+
+    // extension fields not mentioned in draft (but should probably be included,
+    // if the parameters should be included at all; it does not conform to CGA,
+    // but it propably does conform to TSIG since they are part of Other Data)
+    ldns_rdf* cmpts_rdfs[7] = {rdfs->modifier,
+                               rdfs->prefix,
+                               rdfs->coll_count,
+                               rdfs->pub_key,
+                               wire_rdf,
+                               rdfs->ip_tag,
+                               time_signed_rdf};
+
+    /* concatenate the input */
+    status = ldns_tsig_concat_data(cmpts_rdfs, 7, buffer);
+
+    if (status != LDNS_STATUS_OK) {
+      *buffer = NULL; // buffer has not been allocated yet
+    }
+
+    ldns_rdf_free(wire_rdf);
+
+    return status;
+ }
+
+
+ /**
+  * copies the CGA-TSIG data fields into the Other Data RDF.
+  * \param[in] rdfs the input ldns_cga_rdfs structure
+  * \param[out] other_data_rdf the resulting Other Data RDF (will be allocated)
+  * \return status (OK if success)
+  */
+ static ldns_status
+ ldns_cga_rdfs2tsig_od(ldns_cga_rdfs *rdfs, ldns_rdf **other_data_rdf)
+ {
+    uint16_t cga_tsig_len, param_len, sig_len;
+    uint16_t old_pubk_len = 0;
+    uint16_t old_sig_len = 0;
+    ldns_rdf *ctl_rdf, *pl_rdf, *sl_rdf, *opkl_rdf, *osl_rdf;
+    ldns_buffer *buffer = NULL;
+    ldns_status status = LDNS_STATUS_OK;
+
+    /* calculate lengths */
+    param_len = CT_MODIFIER_SIZE + CT_PREFIX_SIZE + CT_COLL_COUNT_SIZE
+                + ldns_rdf_size(rdfs->pub_key);
+
+    if (rdfs->ext_fields) {
+      param_len += ldns_rdf_size(rdfs->ext_fields);
+    }
+
+    sig_len = ldns_rdf_size(rdfs->sig);
+
+    cga_tsig_len = CT_ALGO_NAME_SIZE + CT_TYPE_SIZE + CT_IP_TAG_SIZE + CT_PARAM_LEN_SIZE
+                   + param_len + CT_SIG_LEN_SIZE + sig_len + CT_OLD_PK_LEN_SIZE
+                   + CT_OLD_SIG_LEN_SIZE;
+
+    if (rdfs->old_pub_key) {
+      old_pubk_len = ldns_rdf_size(rdfs->old_pub_key);
+      cga_tsig_len += old_pubk_len;
+    }
+
+    if (rdfs->old_sig) {
+      old_sig_len = ldns_rdf_size(rdfs->old_sig);
+      cga_tsig_len += old_sig_len;
+    }
+
+    /* put the length fields in RDFs for the concat operation*/
+    ctl_rdf = ldns_native2rdf_int16(LDNS_RDF_TYPE_INT16, cga_tsig_len);
+    pl_rdf = ldns_native2rdf_int16(LDNS_RDF_TYPE_INT16, param_len);
+    sl_rdf = ldns_native2rdf_int16(LDNS_RDF_TYPE_INT16, sig_len);
+    opkl_rdf = ldns_native2rdf_int16(LDNS_RDF_TYPE_INT16, old_pubk_len);
+    osl_rdf = ldns_native2rdf_int16(LDNS_RDF_TYPE_INT16, old_sig_len);
+
+    if (!ctl_rdf || !pl_rdf || !sl_rdf || !opkl_rdf || !osl_rdf) {
+      status = LDNS_STATUS_MEM_ERR;
+      goto clean;
+    }
+
+    ldns_rdf* cmpts_rdfs[16] = {ctl_rdf,
+                                rdfs->algo_name,
+                                rdfs->type,
+                                rdfs->ip_tag,
+                                pl_rdf,
+                                rdfs->modifier,
+                                rdfs->prefix,
+                                rdfs->coll_count,
+                                rdfs->pub_key,
+                                rdfs->ext_fields,
+                                sl_rdf,
+                                rdfs->sig,
+                                opkl_rdf,
+                                rdfs->old_pub_key,
+                                osl_rdf,
+                                rdfs->old_sig};
+
+    /* concatenate the fields */
+    status = ldns_tsig_concat_data(cmpts_rdfs, 16, &buffer);
+
+    if (status != LDNS_STATUS_OK) {
+      goto clean;
+    }
+
+    /* create the Other Data RDF */
+    *other_data_rdf = ldns_native2rdf_int16_data(ldns_buffer_capacity(buffer), ldns_buffer_begin(buffer));
+
+    ldns_buffer_free(buffer);
+
+    if (!*other_data_rdf) {
+      status = LDNS_STATUS_MEM_ERR;
+    }
+
+    clean:
+    ldns_rdf_free(ctl_rdf);
+    ldns_rdf_free(pl_rdf);
+    ldns_rdf_free(sl_rdf);
+    ldns_rdf_free(opkl_rdf);
+    ldns_rdf_free(osl_rdf);
+
+    return status;
+ }
+
+ /**
+  * performes CGA verification of an IPv6 address [RFC3972].
+  * \param[in] ns the sockaddr_in6 struct containing the IP address of the remote name server
+  * \param[in] param the cga parameters
+  * \return status (OK if success)
+  */
+ static ldns_status
+ ldns_cga_verify(struct sockaddr_in6 *ns, ldns_cga_rdfs *rdfs)
+ {
+    ldns_status status = LDNS_STATUS_OK;
+    ldns_buffer *concat = NULL;
+    unsigned char hash[LDNS_SHA1_DIGEST_LENGTH], id[8];
+    uint16_t i;
+    unsigned char sec;
+
+    if (!ns || !rdfs) {
+      return LDNS_STATUS_NULL;
+    }
+
+    ldns_rdf* param_rdfs[5] = {rdfs->modifier,
+                               rdfs->prefix,
+                               rdfs->coll_count,
+                               rdfs->pub_key,
+                               rdfs->ext_fields};
+
+    /* collision count must be 0, 1 or 2 */
+    if (ldns_cga_data2host(ldns_rdf_data(rdfs->coll_count), CT_COLL_COUNT_SIZE) > 2) {
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    /* subnet prefix must match */
+    if (memcmp(&ns->sin6_addr, ldns_rdf_data(rdfs->prefix), 8) != 0) {
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    }
+
+    /* generate hash1 */
+    status = ldns_tsig_concat_data(param_rdfs, 5, &concat);
+
+    if (status != LDNS_STATUS_OK) {
+      return status; // we can return safely, concat has not been allocated yet
+    }
+
+    (void)ldns_sha1(ldns_buffer_begin(concat), ldns_buffer_capacity(concat), hash);
+
+    memcpy(id, ns->sin6_addr.s6_addr + 8, 8);
+
+    /* extract the sec parameter */
+    sec = id[0] >> 5;
+
+    /* hash1 (first 8 octets) must match the interface ID of the address,
+     * ignoring bits 0, 1, 2, 6 and 7 of the first byte */
+    hash[0] &= 0x1c;
+    id[0] &= 0x1c;
+
+    if (memcmp(hash, id, 8) != 0) {
+      status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+      goto clean;
+    }
+
+    /* generate hash2 */
+    memset(ldns_buffer_at(concat, 16), 0, 9);
+
+    (void)ldns_sha1(ldns_buffer_begin(concat), ldns_buffer_capacity(concat), hash);
+
+    /* 2*sec leftmost bytes of hash2 must be zero */
+    sec *= 2;
+
+    for (i = 0; i < sec; i++) {
+      if (hash[i++] != 0 || hash[i] != 0) {
+        status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+        goto clean;
+      }
+    }
+
+    clean:
+    ldns_buffer_free(concat);
+    return status;
+ }
+
+
1056,1060c279
+    if (!ldns_pkt_tsig_verify_next_2(pkt, wire, wirelen, key_name, key_data, orig_mac_rdf, NULL, 0, 0)
+        != LDNS_STATUS_OK) {
+      return false;
+    }
+    return true;
---
-    return ldns_pkt_tsig_verify_next(pkt, wire, wirelen, key_name, key_data, orig_mac_rdf, 0);
1063,1070d281
+ ldns_status
+ ldns_pkt_tsig_verify_2(ldns_pkt *pkt, uint8_t *wire, size_t wirelen, const char *key_name,
+    const char *key_data, ldns_rdf *orig_mac_rdf, const struct sockaddr_storage *ns_out, size_t ns_out_len)
+ {
+    return ldns_pkt_tsig_verify_next_2(pkt, wire, wirelen, key_name, key_data, orig_mac_rdf, ns_out,
      ns_out_len, 0);
+ }
+
+
1075,1086d285
+    if (ldns_pkt_tsig_verify_next_2(pkt, wire, wirelen, key_name, key_data, orig_mac_rdf, NULL, 0,
      tsig_timers_only)
+        != LDNS_STATUS_OK) {
+      return false;
+    }
+    return true;
+ }
+
+ ldns_status
+ ldns_pkt_tsig_verify_next_2(ldns_pkt *pkt, uint8_t *wire, size_t wirelen, const char* key_name,
+    const char *key_data, ldns_rdf *orig_mac_rdf, const struct sockaddr_storage *ns_out, size_t ns_out_len,
+    int tsig_timers_only)
+ {
1098,1105d296
+    struct sockaddr_storage *ns_in;
+    size_t ns_in_len;
+    struct sockaddr_in6 *out_in6, *in_in6;
+    unsigned char hash[LDNS_SHA1_DIGEST_LENGTH];
+    ldns_buffer *concat = NULL;
+    ldns_cga_rdfs *cga_rdfs = NULL;
+   RSA *pubk = NULL;
+   RSA *opubk = NULL;
1109d299
+    char *algorithm_name = NULL;
1111d300
+    // save pointer to the packet's tsig rr
1114c303
+    if (!orig_tsig) {
---
-    if (!orig_tsig || ldns_rr_rd_count(orig_tsig) <= 6) {
1116c305
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
---
-      return false;
1118,1122d306
+
+    if (ldns_rr_rd_count(orig_tsig) <= 6) {
+      ldns_rdf_deep_free(key_name_rdf);
+      return LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+    } // get the contents of the rdata fields
1124c308
+    time_signed_rdf = ldns_rr_rdf(orig_tsig, 1); // NOTE: not being checked?
---
-    time_signed_rdf = ldns_rr_rdf(orig_tsig, 1);
1131,1136d314
+    algorithm_name = ldns_rdf2str(algorithm_rdf);
+    if(!algorithm_name) {
+      ldns_rdf_deep_free(key_name_rdf);
+      return LDNS_STATUS_MEM_ERR;
+    }
+
1144d321
+    // copy the wire, but with the tsig rr removed (NOTE: check if is NULL?)
1147,1201c324,326
+    if (strcasecmp(algorithm_name, "cga-tsig.") == 0) {
+      /* 1. IP check (3) */
+      ns_in = ldns_rdf2native_sockaddr_storage(ldns_pkt_answerfrom(pkt), 0, &ns_in_len);
+
+      if (!ns_out || !ns_in) {
+        status = LDNS_STATUS_NULL;
+        goto clean;
+      }
+
+     if (ns_out_len != ns_in_len) {
+       status = LDNS_STATUS_ERR;
+       goto clean;
+     }
+
+ #ifndef S_SPLINT_S
+     if ((ns_out->ss_family != AF_INET6) || (ns_in->ss_family != AF_INET6)) {
+       LDNS_FREE(ns_in);
+       status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+       goto clean;
+     }
+ #endif
+
+     out_in6 = (struct sockaddr_in6*)ns_out;
+     in_in6 = (struct sockaddr_in6*)ns_in;
+
+     if (memcmp(&out_in6->sin6_addr, &in_in6->sin6_addr, LDNS_IP6ADDRLEN) != 0) {
+       LDNS_FREE(ns_in);
+       status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+       goto clean;
+     }
+
+     LDNS_FREE(ns_in);
+
+     /* extract CGA-TSIG data fields */
+     status = ldns_tsig_od2cga_rdfs(other_data_rdf, &cga_rdfs, &pubk, &opubk);
+
+     if (status != LDNS_STATUS_OK) {
+       status = LDNS_STATUS_CRYPTO_TSIG_BOGUS; // better to check for server error
+       goto clean;
+     }
+
+     /* 2. CGA check (1) */
+     status = ldns_cga_verify(out_in6, cga_rdfs);
+
+     if (status != LDNS_STATUS_OK) {
+       goto clean;
+     }
+
+     /* 3. signature check (4) */
+     status = ldns_cga_concat_msg(prepared_wire, prepared_wire_size,
+         time_signed_rdf, cga_rdfs, &concat);
+
+     if (status != LDNS_STATUS_OK) {
+       goto clean;
+     }
---
-   status = ldns_tsig_mac_new(&my_mac_rdf, prepared_wire, prepared_wire_size,
-       key_data, key_name_rdf, fudge_rdf, algorithm_rdf,
-       time_signed_rdf, error_rdf, other_data_rdf, orig_mac_rdf, tsig_timers_only);
1203,1236d327
+     (void)ldns_sha1(ldns_buffer_begin(concat), ldns_buffer_capacity(concat), hash);
+
+     if (!RSA_verify(NID_sha1, hash, LDNS_SHA1_DIGEST_LENGTH,
+         ldns_rdf_data(cga_rdfs->sig), ldns_rdf_size(cga_rdfs->sig), pubk)) {
+       status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+       goto clean;
+     } else {
+       status = LDNS_STATUS_OK;
+     }
+
+     /* 4. old public key/signature steps go here */
+
+   } else {
+     // calculate the mac
+     status = ldns_tsig_mac_new(&my_mac_rdf, prepared_wire, prepared_wire_size,
+         key_data, key_name_rdf, fudge_rdf, algorithm_rdf,
+         time_signed_rdf, error_rdf, other_data_rdf, orig_mac_rdf, tsig_timers_only);
+
+     if (status != LDNS_STATUS_OK) {
+       goto clean;
+     }
+
+     // compare the macs
+     if (ldns_rdf_compare(pkt_mac_rdf, my_mac_rdf) == 0) {
+       ldns_rdf_deep_free(my_mac_rdf);
+       status = LDNS_STATUS_OK;
+     } else {
+       ldns_rdf_deep_free(my_mac_rdf);
+       status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
+       goto clean;
+     }
+   }
+
+   clean:
1238d328
+   ldns_rdf_deep_free(key_name_rdf);
1240,1244c330,334
+   /* Put back the values
+    * NOTE: pkt has not been used for generating anything in the meantime,
+    *       ldns_tsig_prepare_pkt_wire() removes the TSIG without touching pkt.
+    *       Remove this?
+    */
---
-   if (status != LDNS_STATUS_OK) {
-     ldns_rdf_deep_free(key_name_rdf);
-     return false;
-   }
-   /* Put back the values */
1248,1253c338
+   ldns_cga_rdfs_deep_free(cga_rdfs);
+
+   if (pubk) {
+     RSA_free(pubk);
+     pubk = NULL;
+   }
---
-   ldns_rdf_deep_free(key_name_rdf);
1255,1257c340,345
+   if (opubk) {
+     RSA_free(opubk);
+     opubk = NULL;
---
-   if (ldns_rdf_compare(pkt_mac_rdf, my_mac_rdf) == 0) {
-     ldns_rdf_deep_free(my_mac_rdf);
-     return true;
-   } else {
-     ldns_rdf_deep_free(my_mac_rdf);
-     return false;
1259,1262d346
+
+   ldns_buffer_free(concat);
+
+   return status;
1271,1282c355
+   return ldns_pkt_tsig_sign_next_2(pkt, key_name, key_data, NULL, NULL, NULL, NULL,
+       fudge, algorithm_name, query_mac, NULL, NULL, NULL, 0, 0, 0);
+ }
+
+ ldns_status
+ ldns_pkt_tsig_sign_2(ldns_pkt *pkt, const char *key_name, const char *key_data,
+   RSA *pvt_key, RSA *pub_key, RSA *old_pvt_key, RSA *old_pub_key,
+   uint16_t fudge, const char *algorithm_name, ldns_rdf *query_mac, uint8_t *ip_tag,
+   uint8_t *modifier, uint8_t *prefix, size_t coll_count, int request_only)
+ {
+   return ldns_pkt_tsig_sign_next_2(pkt, key_name, key_data, pvt_key, pub_key, old_pvt_key, old_pub_key,
+       fudge, algorithm_name, query_mac, ip_tag, modifier, prefix, coll_count, request_only, 0);
---
-   return ldns_pkt_tsig_sign_next(pkt, key_name, key_data, fudge, algorithm_name, query_mac, 0);
1289,1298d361
+   return ldns_pkt_tsig_sign_next_2(pkt, key_name, key_data, NULL, NULL, NULL, NULL,
+       fudge, algorithm_name, query_mac, NULL, NULL, NULL, 0, 0, tsig_timers_only);
+ }
+
+ ldns_status
+ ldns_pkt_tsig_sign_next_2(ldns_pkt *pkt, const char *key_name, const char *key_data,
+   RSA *pvt_key, RSA *pub_key, RSA *old_pvt_key, RSA *old_pub_key,
+   uint16_t fudge, const char *algorithm_name, ldns_rdf *query_mac, uint8_t *ip_tag,
+   uint8_t *modifier, uint8_t *prefix, uint8_t coll_count, int request_only, int tsig_timers_only)
+ {
1317,1332d379
+   unsigned char hash[LDNS_SHA1_DIGEST_LENGTH];
+   ldns_buffer *concat = NULL;
+   ldns_cga_rdfs *cga_rdfs = NULL;
+   unsigned char *pubk_buf = NULL;
+   unsigned char *sig_buf = NULL;
+   unsigned int sig_len = 0;
+   int pubk_len = 0;
+   uint16_t val = 0;
+
+   // suppress compile warning
+   if (old_pvt_key) {
+     RSA_free(old_pvt_key);
+     old_pvt_key = NULL;
+   }
+   //
+
1367c414,416
+   if(!fudge_rdf || !orig_id_rdf || !error_rdf) {
---
-   other_data_rdf = ldns_native2rdf_int16_data(0, NULL);
-
-   if(!fudge_rdf || !orig_id_rdf || !error_rdf || !other_data_rdf) {
1372,1376c421,424
+   if (!request_only) {
+     if (ldns_pkt2wire(&pkt_wire, pkt, &pkt_wire_len) != LDNS_STATUS_OK) {
+       status = LDNS_STATUS_ERR;
+       goto clean;
+     }
---
-   if (ldns_pkt2wire(&pkt_wire, pkt, &pkt_wire_len) != LDNS_STATUS_OK) {
-     status = LDNS_STATUS_ERR;
-     goto clean;
-   }
1378,1547c426,428
+     if (strcasecmp(algorithm_name, "cga-tsig.") == 0) {
+       if (!pvt_key || !pub_key || !ip_tag || !modifier || !prefix) {
+         status = LDNS_STATUS_NULL;
+         goto clean;
+       }
+
+       /* allocate structure holding the RDFs */
+       status = ldns_cga_rdfs_new(&cga_rdfs);
+
+       if (status != LDNS_STATUS_OK) {
+         goto clean; // cga_rdfs has not been allocated yet
+       }
+
+       /* set encryption algorithm */
+       val = 0;
+
+       status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->algo_name),
+           &val, -1, 1);
+
+       if (status != LDNS_STATUS_OK) {
+         goto clean;
+       }
+
+       /* set validation algorithm */
+       val = 1;
+
+       status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->type),
+           &val, -1, 1);
+
+       if (status != LDNS_STATUS_OK) {
+         goto clean;
+       }
+
+       /* set IP tag */
+       status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->ip_tag),
+           ip_tag, -1, 0);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* set modifier */
+        status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->modifier),
+            modifier, -1, 0);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* set prefix */
+        status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->prefix),
+            prefix, -1, 0);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* set collision count */
+        status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->coll_count),
+            &coll_count, -1, 0);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* convert public key */
+        pubk_len = i2d_RSA_PUBKEY(pub_key, &pubk_buf);
+
+        if (pubk_len <= 0) {
+          status = LDNS_STATUS_CRYPTO_TSIG_ERR;
+          goto clean;
+        }
+
+        /* set public key */
+        status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->pub_key),
+            pubk_buf, pubk_len, 0);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        if (old_pub_key) {
+          free(pubk_buf);
+          pubk_buf = NULL;
+
+          /* convert old public key (assuming it is also a SubjectPublicKeyInfo structure) */
+          pubk_len = i2d_RSA_PUBKEY(old_pub_key, &pubk_buf);
+
+          if (pubk_len <= 0) {
+            status = LDNS_STATUS_CRYPTO_TSIG_ERR;
+            goto clean;
+          }
+
+          /* set old public key */
+          status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->old_pub_key),
+              pubk_buf, pubk_len, 0);
+
+          if (status != LDNS_STATUS_OK) {
+            goto clean;
+          }
+        }
+
+        /* concatenate the message and the fields */
+        status = ldns_cga_concat_msg(pkt_wire, pkt_wire_len,
+            time_signed_rdf, cga_rdfs, &concat);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* digest the concatenation */
+        (void)ldns_sha1(ldns_buffer_begin(concat), ldns_buffer_capacity(concat), hash);
+
+        /* sign */
+        sig_buf = LDNS_XMALLOC(unsigned char, RSA_size(pvt_key));
+
+        if (!sig_buf) {
+          status = LDNS_STATUS_MEM_ERR;
+          goto clean;
+        }
+
+        if (!RSA_sign(NID_sha1, hash, LDNS_SHA1_DIGEST_LENGTH,
+            sig_buf, &sig_len, pvt_key)) {
+          LDNS_FREE(sig_buf);
+          status = LDNS_STATUS_CRYPTO_TSIG_ERR;
+          goto clean;
+        } else {
+          status = LDNS_STATUS_OK;
+        }
+
+        /* set signature */
+        status = ldns_cga_data2rdf(cga_rdfs, &(cga_rdfs->sig),
+            sig_buf, (int)sig_len, 0);
+
+        LDNS_FREE(sig_buf);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* create Other Data RDF */
+        status = ldns_cga_rdfs2tsig_od(cga_rdfs, &other_data_rdf);
+
+        if (status != LDNS_STATUS_OK) {
+          goto clean;
+        }
+
+        /* create empty mac RDF */
+       mac_rdf = ldns_native2rdf_int16_data(0, NULL);
+
+       if (!mac_rdf) {
+         status = LDNS_STATUS_MEM_ERR;
+         goto clean;
+       }
+     } else {
+       other_data_rdf = ldns_native2rdf_int16_data(0, NULL);
+
+       if(!other_data_rdf) {
+         status = LDNS_STATUS_MEM_ERR;
+         goto clean;
+       }
+
+       status = ldns_tsig_mac_new(&mac_rdf, pkt_wire, pkt_wire_len,
+           key_data, key_name_rdf, fudge_rdf, algorithm_rdf,
+           time_signed_rdf, error_rdf, other_data_rdf, query_mac, tsig_timers_only);
+
+       if (!mac_rdf) {
+         goto clean;
+       }
+     }
---
-   status = ldns_tsig_mac_new(&mac_rdf, pkt_wire, pkt_wire_len,
-       key_data, key_name_rdf, fudge_rdf, algorithm_rdf,
-       time_signed_rdf, error_rdf, other_data_rdf, query_mac, tsig_timers_only);
1549,1559c430
+     LDNS_FREE(pkt_wire);
+   } else if (strcasecmp(algorithm_name, "cga-tsig.") == 0) {
+       mac_rdf = ldns_native2rdf_int16_data(0, NULL);
+       other_data_rdf = ldns_native2rdf_int16_data(0, NULL);
+
+       if(!other_data_rdf || !mac_rdf) {
+         status = LDNS_STATUS_MEM_ERR;
+         goto clean;
+       }
+   } else {
+     status = LDNS_STATUS_CRYPTO_TSIG_ERR;
---
-   if (!mac_rdf) {
1562a434,435
-   LDNS_FREE(pkt_wire);
-
1584c457
+   goto end;
---
-   return status;
1590c463
+   ldns_rdf_free(time_signed_rdf); // should be deep_free?
---
-   ldns_rdf_free(time_signed_rdf);
1595,1598d467
+
+   end:
+   ldns_cga_rdfs_deep_free(cga_rdfs);
+   free(pubk_buf);
```

tsig.h (additions and deletions)

```
23,35d22
+ #define CT_LEN_SIZE           2
+ #define CT_ALGO_NAME_SIZE     2
+ #define CT_TYPE_SIZE          2
+ #define CT_IP_TAG_SIZE        16
+ #define CT_PARAM_LEN_SIZE     2
+ #define CT_MODIFIER_SIZE      CT_IP_TAG_SIZE
+ #define CT_PREFIX_SIZE        8
+ #define CT_COLL_COUNT_SIZE    1
+ #define CT_SIG_LEN_SIZE       2
+ #define CT_OLD_PK_LEN_SIZE    2
+ #define CT_OLD_SIG_LEN_SIZE   2
+
+
47,66d33
+
+ /**
+  * Contains RDFs for fields in CGA-TSIG other data
+ */
+ typedef struct ldns_cga_rdfs_struct
+ {
+     ldns_rdf *algo_name;
+     ldns_rdf *type;
+     ldns_rdf *ip_tag;
+     ldns_rdf *modifier;
+     ldns_rdf *prefix;
+     ldns_rdf *coll_count;
+     ldns_rdf *pub_key;     // the complete encoded SubjectPublicKeyInfo block
+     ldns_rdf *ext_fields;
+     ldns_rdf *sig;
+     ldns_rdf *old_pub_key;
+     ldns_rdf *old_sig;
+ } ldns_cga_rdfs;
+
+
95,110d61
+  * \param[in] ns structure with the IP of the queried remote resolver (for CGA-TSIG)
+  * \param[in] ns_len size of ns (for CGA-TSIG)
+  * \return LDNS_STATUS_OK if tsig is correct, error status otherwise
+  */
+ ldns_status ldns_pkt_tsig_verify_2(ldns_pkt *pkt, uint8_t *wire, size_t wire_size, const char *key_name,
     const char *key_data, ldns_rdf *mac,
+     const struct sockaddr_storage *ns, size_t ns_len);
+
+ /**
+  * verifies the tsig rr for the given packet and key.
+  * The wire must be given too because tsig does not sign normalized packets.
+  * \param[in] pkt the packet to verify
+  * \param[in] wire needed to verify the mac
+  * \param[in] wire_size size of wire
+  * \param[in] key_name the name of the shared key
+  * \param[in] key_data the key in base 64 format
+  * \param[in] mac original mac
119,136d69
+  * verifies the tsig rr for the given packet and key.
+  * The wire must be given too because tsig does not sign normalized packets.
+  * \param[in] pkt the packet to verify
+  * \param[in] wire needed to verify the mac
+  * \param[in] wire_size size of wire
+  * \param[in] key_name the name of the shared key
+  * \param[in] key_data the key in base 64 format
+  * \param[in] mac original mac
+  * \param[in] ns structure with the IP of the queried remote resolver (for CGA-TSIG)
+  * \param[in] ns_len size of ns (for CGA-TSIG)
+  * \param[in] tsig_timers_only must be zero for the first packet and positive for subsequent packets. If
     zero, all digest
+    components are used to verify the _mac. If non-zero, only the TSIG timers are used to verify the mac.
+  * \return LDNS_STATUS_OK if tsig is correct, error status otherwise
+  */
+ ldns_status ldns_pkt_tsig_verify_next_2(ldns_pkt *pkt, uint8_t *wire, size_t wire_size, const char
     *key_name, const char *key_data, ldns_rdf *mac,
+     const struct sockaddr_storage *ns, size_t ns_len, int tsig_timers_only);
+
+ /**
154,177d86
+  * \param[in] pvt_key the private key (for CGA-TSIG)
+  * \param[in] pub_key the associated public key (for CGA-TSIG)
+  * \param[in] old_pvt_key the old private key (NULL if not applicable; for CGA-TSIG)
+  * \param[in] old_pub_key the associated old public key (NULL if not applicable; for CGA-TSIG)
+  * \param[in] fudge seconds of error permitted in time signed
+  * \param[in] algorithm_name the name of the algorithm used
+  * \param[in] query_mac is added to the digest if not NULL (so NULL is for signing queries, not NULL is for
     signing answers)
+  * \param[in] ip_tag the IP tag (NULL if not applicable; for CGA-TSIG)
+  * \param[in] modifier the CGA modifier (for CGA-TSIG)
+  * \param[in] prefix the network prefix of the host's IPv6 address (for CGA-TSIG)
+  * \param[in] coll_count the CGA collision count (for CGA-TSIG)
+  * \param[in] request_only do not sign but only request the resolver to sign (for CGA-TSIG)
+  * \return status (OK if success)
+  */
+ ldns_status ldns_pkt_tsig_sign_2(ldns_pkt *pkt, const char *key_name, const char *key_data,
+     RSA *pvt_key, RSA *pub_key, RSA *old_pvt_key, RSA *old_pub_key,
+     uint16_t fudge, const char *algorithm_name, ldns_rdf *query_mac, uint8_t *ip_tag,
+     uint8_t *modifier, uint8_t *prefix, size_t coll_count, int request_only);
+
+ /**
+  * creates a tsig rr for the given packet and key.
+  * \param[in] pkt the packet to sign
+  * \param[in] key_name the name of the shared key
+  * \param[in] key_data the key in base 64 format (in PEM format for CGA-TSIG), must be null-terminated
187,212d95
+
+ /**
+  * creates a tsig rr for the given packet and key.
+  * \param[in] pkt the packet to sign
+  * \param[in] key_name the name of the shared key
+  * \param[in] key_data the key in base 64 format
+  * \param[in] pvt_key the private key (for CGA-TSIG)
+  * \param[in] pub_key the associated public key (for CGA-TSIG)
+  * \param[in] old_pvt_key the old private key (NULL if not applicable; for CGA-TSIG)
+  * \param[in] old_pub_key the associated old public key (NULL if not applicable; for CGA-TSIG)
+  * \param[in] fudge seconds of error permitted in time signed
+  * \param[in] algorithm_name the name of the algorithm used
+  * \param[in] query_mac is added to the digest if not NULL (so NULL is for signing queries, not NULL is for
     signing answers)
+  * \param[in] ip_tag the IP tag (NULL if not applicable; for CGA-TSIG)
+  * \param[in] modifier the CGA modifier (for CGA-TSIG)
+  * \param[in] prefix the network prefix of the host's IPv6 address (for CGA-TSIG)
+  * \param[in] coll_count the CGA collision count (for CGA-TSIG)
+  * \param[in] request_only do not sign but only request the resolver to sign (for CGA-TSIG)
+  * \param[in] tsig_timers_only must be zero for the first packet and positive for subsequent packets. If
     zero, all digest
+    components are used to create the query_mac. If non-zero, only the TSIG timers are used to create the
     query_mac.
+  * \return status (OK if success)
+  */
+ ldns_status ldns_pkt_tsig_sign_next_2(ldns_pkt *pkt, const char *key_name, const char *key_data,
+     RSA *pvt_key, RSA *pub_key, RSA *old_pvt_key, RSA *old_pub_key,
+     uint16_t fudge, const char *algorithm_name, ldns_rdf *query_mac, uint8_t *ip_tag,
+     uint8_t *modifier, uint8_t *prefix, uint8_t coll_count, int request_only, int tsig_timers_only);
```

resolver.c (additions and deletions)

```
1060,1072c1060,1063
+   ldns_pkt *pkt = NULL;
+   (void)ldns_resolver_query_ws(&pkt, r, name, t, c, flags);
+   return pkt;
+ }
+
+ ldns_status
+ ldns_resolver_query_ws(ldns_pkt **answer, const ldns_resolver *r,
+   const ldns_rdf *name, ldns_rr_type t, ldns_rr_class c, uint16_t flags)
+ {
+   ldns_status s = ldns_resolver_query_status(answer, (ldns_resolver *)r,
+         name, t, c, flags);
+   if (s != LDNS_STATUS_OK && s != LDNS_STATUS_CRYPTO_TSIG_BOGUS) {
+     ldns_pkt_free(*answer);
---
-   ldns_pkt* pkt = NULL;
-   if (ldns_resolver_query_status(&pkt, (ldns_resolver *)r,
-         name, t, c, flags) != LDNS_STATUS_OK) {
-     ldns_pkt_free(pkt);
1074c1065
+   return s;
---
```

```
-    return pkt;
1114c1105
+    if (stat != LDNS_STATUS_OK && stat != LDNS_STATUS_CRYPTO_TSIG_BOGUS) {
---
-    if (stat != LDNS_STATUS_OK) {
1256c1247
+    if (ldns_resolver_tsig_keyname(r)) {
---
-    if (ldns_resolver_tsig_keyname(r) && ldns_resolver_tsig_keydata(r)) {
1258,1282c1249,1252
+      if (ldns_resolver_tsig_keydata(r)) {
+        status = ldns_pkt_tsig_sign(query_pkt,
+                                    ldns_resolver_tsig_keyname(r),
+                                    ldns_resolver_tsig_keydata(r),
+                                    300, ldns_resolver_tsig_algorithm(r), NULL);
+      } else {
+        /*
+         * if keyname but no keydata, assume CGA-TSIG request;
+         * function will return error if algorithm != "cga-tsig.",
+         * so no explicit need to check here
+         */
+        status = ldns_pkt_tsig_sign_2(query_pkt,
+                                    ldns_resolver_tsig_keyname(r),
+                                    NULL,
+                                    NULL,
+                                    NULL,
+                                    NULL,
+                                    NULL,
+                                    300, ldns_resolver_tsig_algorithm(r),
+                                    NULL,
+                                    NULL,
+                                    NULL,
+                                    NULL,
+                                    0, 1);
+      }
---
-      status = ldns_pkt_tsig_sign(query_pkt,
-                                  ldns_resolver_tsig_keyname(r),
-                                  ldns_resolver_tsig_keydata(r),
-                                  300, ldns_resolver_tsig_algorithm(r), NULL);
1285c1255
+        return status;
---
-        return LDNS_STATUS_CRYPTO_TSIG_ERR;
1289c1259
+      return LDNS_STATUS_CRYPTO_TSIG_ERR;
---
-            return LDNS_STATUS_CRYPTO_TSIG_ERR;
```

resolver.h (additions and deletions)

```
686,702d685
+  * Send a query to a nameserver
+  * \param[out] **answer a pointer to a ldns_pkt pointer (initialized by this function)
+  * \param[in] *r operate using this resolver
+  *                (despite the const in the declaration,
+  *                 the struct is altered as a side-effect)
+  * \param[in] *name query for this name
+  * \param[in] *t query for this type (may be 0, defaults to A)
+  * \param[in] *c query for this class (may be 0, default to IN)
+  * \param[in] flags the query flags
+  *
+  * \return ldns_status LDNS_STATUS_OK on success
+  * if _defnames is true the default domain will be added
+  */
+ ldns_status ldns_resolver_query_ws(ldns_pkt **answer, const ldns_resolver *r, const ldns_rdf *name,
     ldns_rr_type t, ldns_rr_class c, uint16_t flags);
+
+
+ /**
```

net.c (additions and deletions)

```
519c519
+      // shouldn't ns be checked if NULL?
---
-
593a594
-      LDNS_FREE(ns);
613,614d613
+      LDNS_FREE(ns);
+
628,630c627,633
+      status = ldns_pkt_tsig_verify_2(reply, reply_bytes, reply_size,
+        ldns_resolver_tsig_keyname(r),
+        ldns_resolver_tsig_keydata(r), tsig_mac, ns, ns_len);
---
-      if (!ldns_pkt_tsig_verify(reply,
-                                reply_bytes,
-               reply_size,
-                                ldns_resolver_tsig_keyname(r),
-                                ldns_resolver_tsig_keydata(r), tsig_mac)) {
-        status = LDNS_STATUS_CRYPTO_TSIG_BOGUS;
-      }
635,636d637
+
+    LDNS_FREE(ns);
```

Appendix B CGA generation tool

cga-gen.py

```
#!/usr/bin/env python
##########################################################################
##                                                                      ##
## cga-gen.py - Generate a CGA and associated parameters using Scapy6.  ##
##                                                                      ##
## Copyright (C) 2013  Marc Buijsman                                    ##
##                                                                      ##
## This program is free software: you can redistribute it and/or modify ##
## it under the terms of the GNU General Public License as published by ##
## the Free Software Foundation, either version 3 of the License, or    ##
## (at your option) any later version.                                  ##
##                                                                      ##
## This program is distributed in the hope that it will be useful,      ##
## but WITHOUT ANY WARRANTY; without even the implied warranty of       ##
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        ##
## GNU General Public License for more details.                         ##
##                                                                      ##
##########################################################################

from netaddr.ip import IPNetwork, IPAddress
import netifaces as ni
import binascii as ba
import errno
import argparse
from scapy6send.scapy6 import CGAgen


def main(pfx, pub_key, sec, ext, dad, mf, col):
  try:
    pk = open(pub_key, 'rb').read()
  except IOError:
    print("Could not open file '" + pub_key + "'.")
    exit()

  mod = None

  if mf != None:
    try:
      mod = ba.a2b_base64(open(mf, 'rb').read())

      if len(mod) != 16:
        print("Modifier length is not equal to 16 octets.")
        exit()
    except IOError:
      mf = None
      mod = None
      print("Could not open file '" + mf + "', generating new modifier instead.")
    except ba.Error:
      print("Invalid modifier encoding.")
      exit()

  if pfx == None:
    try:
      a = ni.ifaddresses('eth0')[10][0]['addr']
    except KeyError:
      print("Could not get prefix: no IPv6 address found at 'eth0'; alternatively pass a prefix in command line argument.")
      exit()
    try:
      m = ni.ifaddresses('eth0')[10][0]['netmask']
    except KeyError:
      print("Could not get prefix: no subnet mask found at 'eth0'; alternatively pass a prefix in command line argument.")
      exit()

    pfx = str(IPAddress(int(IPNetwork(a).network) & int(IPNetwork(m).network)))
  else:
    if pfx[-1] != ':':
      pfx = pfx + ':'
    if len(pfx) < 2 or pfx[-2] != ':':
      pfx = pfx + ':'

  try:
    pk = PubKey(pk)
  except:
    print("Could not import public key. Wrong format?")
    exit()

  # generate CGA
  try:
    (addr, params) = CGAgen(pfx, pk, sec, ext, dad, mod, col)
  except socket.error, v:
    if v[0] == errno.EPERM:
      print("Need to be root to perform duplicate address detection.")
      exit()
    else:
      print("Invalid prefix.")
      exit()

  if addr == None or params == None:
    print("Unexpected error.")
    exit()

  mod = ba.b2a_base64(params.modifier)

  print("           CGA: " + addr)
  sys.stdout.write("      modifier: " + mod.rstrip())

  if mf == None:
    try:
      md = open('mod.out', 'w')
      md.write(mod)
      print(" (written to file 'mod.out')")
    except IOError:
      print(" (could not write to file 'mod.out')")
  else:
    print("")

  sys.stdout.write("collision count: " + str(params.ccount))

  if not dad:
    print(" (did NOT perform duplicate address detection)")
  else:
    print("")


if __name__ == "__main__":
  parser = argparse.ArgumentParser(description='Generate a CGA and associated parameters using Scapy6.')
  parser.add_argument('pk', metavar='K', help='file containing the public key in PEM PKCS8 format')
  parser.add_argument('-s', dest='sec', type=int, choices=range(8), default=0, help='the sec parameter (defaults to 0)')
  parser.add_argument('-d', dest='dad', default=False, action='store_true', help='perform duplicate address detection if set (disabled by default)')
  parser.add_argument('-m', dest='mod', default=None, help='file containing a 16-byte modifier in base64 format (generated by default)')
  parser.add_argument('-p', dest='pfx', default=None, help='the IPv6 prefix to concatenate the generated IPv6 identifier with (extracts from eth0 by default)')
  parser.add_argument('-c', dest='col', type=int, choices=range(3), default=None, help='collision count (generated by default)')
  parser.add_argument('-e', dest='ext', default=[], nargs='+', help='optional extension fields (none by default)')
  args = parser.parse_args()

  main(args.pfx, args.pk, args.sec, args.ext, args.dad, args.mod, args.col)
```