D2.7

Relevant DNSSEC Concepts and Basic Building Blocks

Document Identification	
Date	31.08.2017
Status	Final
Version	Version 1.0

Related WP	WP 2	**Related Deliverable(s)**	none
Lead Authors	Martin Hoffmann	**Dissemination Level**	PU
Lead Participants	NLNET	**Contributors**	FHG, ATOS, TUG
Reviewers	G+D, DTU		

This document is issued within the frame and for the purpose of the LIGHT*est* project. LIGHT*est* has received funding from the European Union's Horizon 2020 research and innovation programme under G.A. No 700321.

This document and its content are the property of the *Lightest* Consortium. All rights relevant to this document are determined by the applicable laws. Access to this document does not grant any right or license on the document or its contents. This document or its contents are not to be used or treated in any manner inconsistent with the rights or interests of the *Lightest* Consortium or the Partners detriment and are not to be disclosed externally without prior written consent from the *Lightest* Partners.

Each *Lightest* Partner may use this document in conformity with the *Lightest* Consortium Grant Agreement provisions.

NOT TO BE DISTRIBUTED OUTSIDE THE LIGHTEST CONSORTIUM

1. Executive Summary

The trust infrastructure developed by the LIGHTest project makes use of the existing global Domain Name Service (DNS) for discovering information relevant for validation of trust. As a distributed database both in terms of organization of data as well as responsibility for operation and management, the DNS is very suitable for an infrastructure that aims to support integration and interoperation of various trust schemes

The original design of the DNS did not consider a number of attacks allowing miscreants to alter information retrieved via the DNS. The Domain Name Service Security Extensions (DNSSEC) have been developed to mitigate this problem. They allow users of the DNS to verify that the data they received is indeed the data intended. This ability for verification is vital for use of DNS in the context of a trust infrastructure.

Within the infrastructure, the DNS can be used for two specific tasks: verification of identity and retrieval of trust related data.

When communicating with a network resource or retrieving remote documents, certificates are used to prove the identity of the resource or authenticity of the document. The DANE protocol associates these identities with domain names and stores information under this domain name that can be used to limit the certificates allowed to be used with the identity. The concepts from DANE can be used in the LIGHTest architecture to limit the certificates in use by a trust scheme both as issuer certificates as well as for signing trust related information such as trust lists. Some new procedures need to be developed to apply the concepts to these specific use cases.

In order to verify trust, however, additional information needs to be queried, such as rules describing trust association, trust translation, and trust delegation. As such rule sets can become rather large, the DNS isn't suited for storing them. Instead, pointers should be stored that direct interested parties to retrieve this information using other, more appropriate protocols such as HTTP. These protocols will also allow limiting access based on authentication, whereas DNS's data is available publicly.

Such pointers are best provided in the form of URIs. Two DNS extensions are currently defined for storing URIs for a given domain name: the Dynamic Delegation Discovery System (DDDS), an extensive system for translating application-defined strings into URIs using the DNS as a database, and a much simpler concept that simply stores URIs for a given domain name. Use of either would require the definition of some usage rules as part of the LIGHTest project.

When implementing the resulting architecture as part of the project's pilots, various software may be necessary. There are three categories: DNS server software that operates as part of the global DNS system, DNS provisioning libraries and tools that can be used to manage the data to be stored as part of the LIGHTest project, and DNS libraries for querying the DNS as part of trust verification.

2. Document Information

2.1 Contributors

Name	Partner
Jaap Akkerhuis	NLNET
Rasmus Birkedal	DTU
Martin Hoffmann	NLNET
Frank-Michael Kamm	G&D
Stefan More	TUG
Olamide Omolola	TUG
Javier Presa Cordero	ATOS
George Thessalonikefs	NLNET
Miryam Villegas Jimenez	ATOS
Georg Wagner	TUG
Sven Wagner	USTUTT
Heiko Roßnagel	FHG
Wouter Wijngaards	NLNET

2.2 History

Version	Date	Author	Changes
0.1	13.07.2017	NLNET	Initial draft.
0.9	31.07.2017	NLNET	Final draft.
1.0	26.08.2017	NLNET	Final version.

3. Table of Contents

3.1 Table of Figures

3.2 Table of Tables

3.3 Table of Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Exchange
AXFR	Authoritative Zone Transfer
CA	Certificate Authority
ccTLD	Country-code Top Level Domain
DANE	DNS-based Authentication of Name Entities
DDDS	Dynamic Delegation Discovery System
DDoS	Distributed Denial of Service
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
DoS	Denial of Service
DS	Delegation Signer
DTLS	Datagram Transport Layer Security
ENUM	E.164 Number Mapping
gTLD	Generic Top Level Domain
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
ICANN	Internet Corporation for Assigned Names and Numbers

IDN	Internationalized Domain Names
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
IXFR	Incremental Zone Transfer
KSK	Key Signing Key
PKIX	Public Key Infrastructure (X.509)
RFC	Request for Comments
RR	Resource Record
RRset	Resource Record Set
SOA	Start of Authority
SRV	Service Resource Record
TCP	Transmission Control Protocol
TLD	Top Level Domain
TLS	Transport Layer Security
TTL	Time to Live
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WG	Working Group
ZSK	Zone Signing Key

4. Scope

This deliverable D2.7 provides an extensive look into the Domain Name System itself, extensions necessary for solving the goals of LIGHTest, as well as the available DNS software. As such, it provides necessary input for the design work carried out in the tasks 3.2, 4.2, and 5.2 that design DNS-based publication of trust schemes, trust translation, and trust delegation, respectively.

The deliverable starts with an introduction to the Domain Name System in chapter 5, introducing its history and goals, the structure of the data stored, how the DNS is operated, and how the data is arranged and managed. The chapter introduces the concept of the hierarchical domain names and zones for managing operationally connected data. It explains how the DNS is queried both in principle and how this translates to the actual wire protocol.

DNSSEC, the security extensions to DNS, are the matter of chapter 6. After discussing the threats that DNSSEC is attempting to solve, the chapter dives into the concepts and protocols underlying DNSSEC: digitally signing the data with keys associated with zones and the use of the hierarchical structure of the zones for key verification. A discussion follows of the changes necessary to both operating the components of the DNS infrastructure as well as how queries are being made to verify authenticity of DNS data. As this verification is important if a trust infrastructure is to be based on DNS, a look at the limitations of DNSSEC concludes the chapter.

The following two chapters look at the specific use cases that LIGHTest has for DNS: verification of identity and discovery of resource locations. Each chapter discusses the problem at hand, existing options, and makes suggestions for the LIGHTest architecture.

Finally, chapter 9 looks at the software related to DNS. It presents a market overview for software of three categories: DNS server software that can be used as part of the global DNS infrastructure, client libraries and tools that can be used for implementing components that update and manage DNS data, and client libraries that perform DNS queries as part of a trust verifier. For each of these categories, the chapter presents possible choices for use in the LIGHTest reference implementation and pilots.

5. Introduction to the DNS

In its early days, Arpanet, the research network that would eventually evolve into today's Internet, was small enough that each node could maintain a database giving human-readable names to all the nodes it would need to communicate with. Over time, this database, a simple text file named HOSTS.TXT, became centrally maintained. Each node would retrieve updated versions as they became available. With the network growing quickly, however, the file became large, making updates expensive and slow. On the other hand, dealing with the constant flow of requests for new names and updates developed into an administrative nightmare.

As a response, Paul Mockapetris devised the Domain Name System, or DNS for short. Its initial specification was published via the Internet Engineering Task Force as a pair of documents, RFC 882 [1] and RFC 883 [2], in November 1983. The system provides a network service that deals with the administrative issues of a central registry by eliminating it. Instead, the system mirrors the distributed nature of the Internet as a network of interconnected networks. It allows each participating network to set up, configure, and operate their own name resolution service and provides means for discovering and query these independent services.

Implementation and operation experience led to an updated specification in November 1987 as RFC 1034 [3] and RFC 1035 [4]. While there have been many updates and extensions, these two documents still provide the core specification for the DNS thirty years later.

This chapter introduces this core specification, how the Domain Name System works, and achieves its goal.

5.1 Questions and Answers

The initial intention of the name system was to give human-readable (and memorable) names to network hosts that are internally identified using numeric addresses. However, it quickly became apparent that users often aren't interested in communicating with specific hosts but rather like to peruse certain services. For instance, one of the most important applications of the early Internet was electronic mail. When sending an e-mail message, a user doesn't really care to which physical host the message is ultimately delivered so long as it eventually reaches the intended person. Here, the name doesn't necessarily identify a specific host but rather the recipient's mail system. It isn't a host name anymore but becomes a more abstract identifier. It becomes a *domain name.*

Whether the domain name identifies a host or a mail system depends on the context it is used in. When sending data packets out into the network, the name represents a physical host. When sending an e-mail message, the same name suddenly represents a mail system. Each of these cases requires different information to actually use the name – there are different questions to be answered.

As an example, let's assume we want to send an e-mail message to ‹alice@example.com›. The mail system responsible for messages sent to this address is named by the portion to

the right of the at-sign. As a first step, we need to ask the DNS for the physical hosts serving this mail system. The domain name we ask for is ‹example.com›. The kind of information asked for is stated through the *question type* (often abbreviated to *qtype).* Internally, this is a 16-bit integer value with each kind of information having been assigned its own value. Each of these well-known values has a short mnemonic assigned. The type for the information ‘Which hosts are responsible for an e-mail domain?’ is 15 with the mnemonic MX for ‘mail exchange.’

In the commonly used formal notation, the question is written like this:

```
example.com.    IN    MX
```

The first item is the domain name to query for, the question type is last. What is the IN in the middle, though? At the time when DNS was designed, other networks were being developed. It seemed prudent to let DNS being used for these networks as well. However, other networks have different address semantics and provide services in a different way making it necessary to allow for different semantics when translating names. An additional field called the *class* was included in questions and answers to allow addressing the type of network. As it happened, no other network picked up DNS. As a result, the only value for the class field today is the one reserved for the Internet with value 1 and mnemonic IN.[1] This mnemonic will appear in many of the following examples and, for the most part, can be safely ignored.

The resource records that answer this question have a type, too, now called the *record type.* Despite the different name, it is the exact same value. Using the same formal notation, the answer might look like this:

```
example.com.    86400   IN    MX   10 alcor.example.com.
example.com.    86400   IN    MX   15 capella.example.com.
example.com.    86400   IN    MX   15 deneb.example.com.
```

There are three resource records in the answer. The domain name, class, and record type are easily identified. The domain name is called the *owner* of the record since it specifies which name the record belongs to. The name is followed by a number called the *time to live* or TTL. It states how many seconds the record should be considered valid after being received. Thus, all three records in this answer can be reused for a day – if another e-mail message were to be sent to someone at ‹example.com› within a day, this answer can be used directly without asking the DNS again.

The remainder is the record data. The format and meaning of the record data depend on the record type. Internally, they follow a binary encoding; the text given above is merely a textual representation of the binary data for easier use.

[1] Some DNS server software uses the value originally registered for Chaosnet, a network research project at MIT, for querying internal status of the server. This, however, has never been standardized.

For the MX record type, the data of each record consists of a priority value and the domain name of a host providing mail services for the owner. The priority value allows giving preferred hosts – hosts with a smaller priority are preferred. In the example, all mail should first be delivered to a host named ‹alcor.example.com› with the two other hosts serving as fallbacks for when Alcor fails.

Connecting to ‹alcor.example.com› for delivering the e-mail message requires its address. This requires a different question type, an address or A question:

```
alcor.example.com.          IN   A
```

The answer to this question should be an IPv4 address:

```
alcor.example.com.   86400  IN   A   192.0.2.81
```

Being good citizens, we want to use IPv6, though. This is a yet another question, AAAA – pronounced ‘quad A’ and chosen because an IPv6 address is exactly four times the size of an IPv4 address.[2]

```
alcor.example.com.          IN   AAAA
```

The DNS has an answer for this question, too:

```
alcor.example.com.   86400  IN   AAAA  2001:0DB8::81
```

With this information, we can finally start delivering the message to Alice.

5.2 Domain Names

As the Internet started to grow from a few timeshared computers into a complex interconnected network of workstations, the idea of a hierarchical naming scheme for hosts emerged. Instead of communally agreeing on names for each and every host, each site would name its own hosts as it saw fit and the community at large only needed to agree on names for the sites. To identify a host globally, the host’s name needed to be qualified with the name of the site, more abstractly called the *domain* of the host. This system was further refined as the network kept growing. The sites in turn became too large and were split into a number of sub-sites each controlling its own names. This split was reflected in the name of the domain: it became a composite of the name of the sub-site and the site. Additional splits would increase the components in this composite.

As a result, the name space created by these names has a hierarchical structure. More specifically, the structure is a tree: the sites are child nodes of the tree’s root, the sub-sites are child nodes of their parent (sub-) sites, and the hosts are child nodes of their (sub-) sites.

Each node in this tree carries a designator called a label. The domain name of the node is this label prepended to the domain name of its parent node, separated with a dot. Since the

[2] A bit of network engineer humor, there.

domain name of the parent node in turn is a combination of that node's label and parent node's domain name, the domain name results in the sequence of the labels along the way from the node to the tree root.

To pick up the example from above, the domain name of the host ‹alcor.example.com› designates the node labeled ‹alcor› under ‹example.com› which in turn is a node labeled ‹example› under a node labeled ‹com› which, finally, is a direct descendant of the root node. Figure 1 shows how the name traces through the tree.

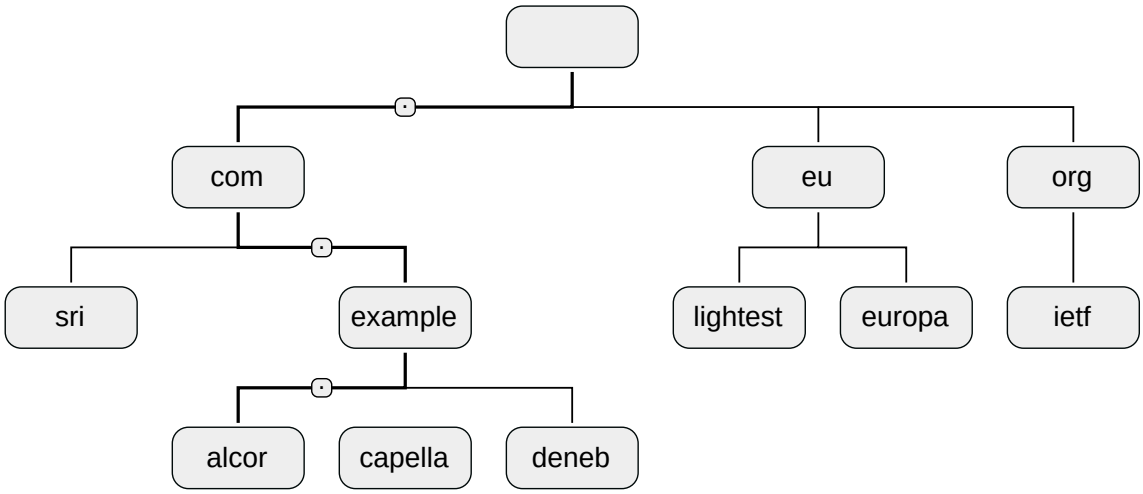

Figure 1. An excerpt from the domain name space. For each node, its label is shown. The domain name ‹alcor.example.com› is formed by following the marked path from the bottom up, concatenating the labels of each node with a dot.

Apart from looking familiar to anyone accustomed to how people are named in Western culture, this bottom-up scheme has the advantage that the root node is the last node in each and every (complete) domain name. If it is given a unique label, this label becomes a natural end-of-name marker making it unnecessary to invent an explicit marker or store the length of the name. And indeed, in DNS the root node is the only one that has an empty label.

All other labels are a strings of up to 63 bytes. The ASCII character set is used to interpret the individual bytes. This is important, since labels are case-insensitive. That is, ‹alcor›, ‹Alcor›, and ‹ALCOR› are all the same label. However, since ASCII only defines byte values up to 127, all values greater than that are quietly left without an interpretation.

Atop this very generous specification exist a convention that limits the characters allowed in the labels that are part of domain names that designate hosts. Here, only letters, digits, and hyphens are allowed with the further restriction that the label must start with a letter and not end with a hyphen. As the vast majority of labels are indeed used for form host names, this rule has become the de-facto standard.

In fact, when it was decided that DNS should be able to express names in more than just languages using the basic Latin alphabet encoded by ASCII, a method was devised to

encode the full range of Unicode using only the byte values allowed for host names. This method, called 'punycode,' forms the basis of Internationalized Domain Names or IDNs.

5.3 Distributed Authority

As mentioned, one important goal for the design of the DNS was to avoid a central registry in control of all names and resource records and allow participants in the network to independently administer their data. This meant that there needed to be a way to discover who a certain name belonged to and, DNS being a network service, where that operator would serve the data from.

Given the potentially vast size of the domain name space, it wasn't practical to store ownership with each domain name. Instead, DNS provides the means to cut the name space into contiguous regions of nodes that are all under the same ownership. These regions are called *zones.* The resource records owned by all the names in a zone are controlled by the zone and are served by the same set of servers. The zone is said to be *authoritative* for the domain names that are part of the zone. The servers that can be queried to deliver the resource records owned by these names are *authoritative name servers* (often shortened to just *name server* when there is no ambiguity).

Because in a tree all contiguous regions are themselves trees, the zones are trees under a top-most node called the zone's *apex.* This node is reachable from all other nodes of the zone by simply walking upwards. Additionally, it is the first node of a zone encountered when traversing from the DNS root. It therefore makes sense to store all necessary administrative information with this apex.

The node just above the apex is part of a different zone. Control needs to be transferred from one zone to another between those two nodes. This is called a *zone cut* and happens by mirroring some of the information of the apex node of the descendant zone with the parent zone. Thus, the servers responsible for the parent zone have knowledge that the apex node of the child zone exists and can provide all those records for this apex node that are required to discover the child zone properly.

These records are called *delegation records* since they delegate control. Classic DNS requires only one type of delegation records: NS (or Name Server) records. Each of these records gives the domain name for one host operating a name server authoritative for the zone.

For ‹example.com›, the apex of the zone from the example above, the NS records could look like this:[3]

```
example.com.    86400    IN    NS   nsa.example.com.
example.com.    86400    IN    NS   nsb.example.com.
```

[3] Since ‹example.com› is a real zone, the actual records are different, reflecting the real operator of the zone. It has been reserved for use in examples and documentation and is operated by IANA. Liberty has been taken to use different records here that better serve the narrative.

Here, the hosts named ‹nsa.example.com› and ‹nsb.example.com› are the two authoritative name servers for the zone.

Which creates a conundrum. The name servers to be used for discovering all domain names within the zone ‹example.com› are accessible using a domain name that is part of that very zone. That can't work and yet it is a very common scenario. After all, the zone is supposed to represent whoever operates that zone.

This is why in addition to delegation records, the parent zone can also store a number of *glue records,* resource records for different domain names returned included in the response to cut the Gordian knot.

To complete the example, the servers for ‹com› (which is the parent zone of ‹example.com›), need to also deliver the address records for the two name servers when asked for the zone's NS records:

```
nsa.example.com.    86400    IN    A      192.0.2.88
nsa.example.com.    86400    IN    AAAA   2001:0DB8::12
nsb.example.com.    86400    IN    AAAA   2001:0DB8::13
```

Figure 2 traces delegation from the root zone to ‹example.com› by showing all the relevant resource records and how they tie together.

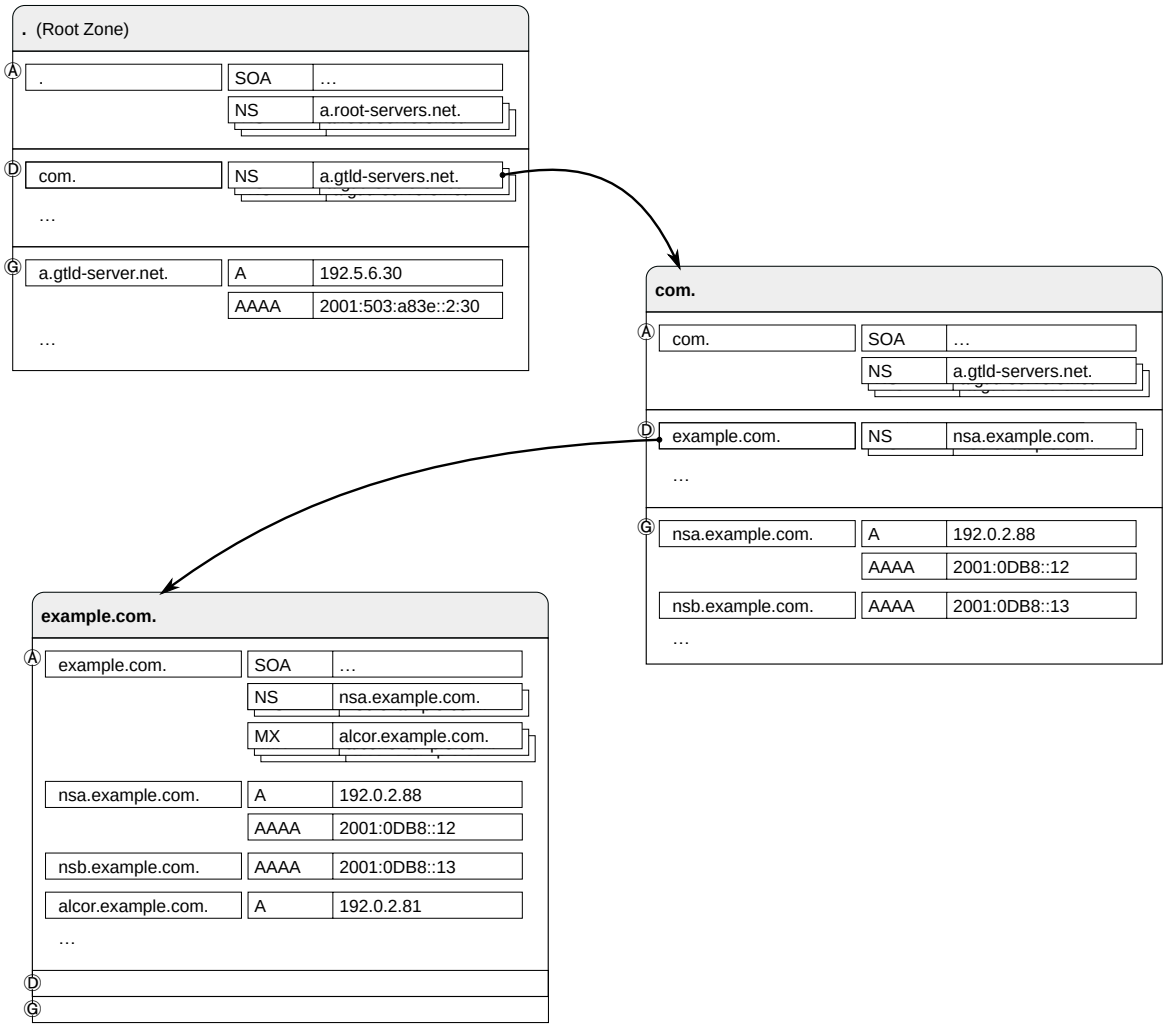

Figure 2. Delegation from the root zone to ‹com› and ‹example.com› showing the relevant authoritative, delegation and glue records for each of the three zones.

5.4 The Root Zone, TLDs, Registries and Registrars

Using the process of delegation, each zone effectively gains its legitimacy through an action of its parent zone, which authorizes the sub-zone by publishing the NS records pointing to it. This recursive process ends at the top with a zone that has the root node as its apex – the *root zone.* Which means that the policies governing the root zone influence the form of all names used on the Internet.

Because of this, the authoritative part of the root zone is extremely small: it consists of the root node only. It delegates to a set of child nodes called the *top-level domains* (or TLDs). Initially, this set was relatively small: six general purpose names for different kinds of organizations – ‹gov› for governmental, ‹edu› for educational, ‹com› for commercial, ‹mil› for military, ‹net› for network-related, and ‹org› for other organizations – plus one TLD for each of the two-letter country codes defined by ISO. In addition, the ‹arpa› domain was initially introduced as a temporary TLD into which all the existing host names from the flat era were

placed. It is also used for names that have special meaning within DNS. For instance, when looking up the host name assigned to an IP address, a domain name under ‹arpa› is generated for the IP address. In 1988, ‹int› was added for international organizations. [5] The explosive growth of the Internet during the late 1990s resulted in several attempts to add additional, more topical TLDs. As a first result, seven such TLDs were added between 2001 and 2004. Finally, in 2008 the New Top-level Domains Program opened the name space up by provided a process for requesting and operating additional TLDs.

Together with IP address allocation and allocation of names and numbers for Internet protocols, control of the root zone and consequently the TLDs comprises something called the *IANA function*, where IANA stands for Internet Assigned Numbers Authority. IANA formed more or less spontaneously when the need for such a function arose. It was initially performed by Jon Postel and Joyce K. Reynolds and, once it became a little more formalized, funded by the US government. It was transferred to a new, private organization, the Internet Corporation for Assigned Names and Numbers, ICANN, in late 1998 but remained under US oversight. After much discussion, the function was finally transferred into the control of an international community in 2016.

Most top-level domains are operated by their own organizations upon an agreement with ICANN. These organizations are called *registries.* They can have their own policies restricting who can apply for a *second-level* domain with them. For the two-letter country code top-level domains (ccTLDs), control is transferred to an entity of that country, typically an organization firmed for that purpose by the country's ISPs or a government agency. While some of these TLDs limit applications for a domain name to persons and organizations residing in the country, others are more open.

The other, non-country domains, often called generic top-level domains or gTLDs, are mostly operated by commercial entities for profit, with some exceptions for those original TLDs reserved for the government. Access to second-level domains under a gTLDs can be restricted or open, depending on policies and intentions for these domains.

While applicants would initially request names directly from the registry, over time many registries adopted a process whereby applicants would instead use the services of an intermediary called *registrars.* These, often commercial entities, would be accredited with several TLDs, making it easier to register a variety of domain names through a single entity. Often, these registrars also offer to operate the zones for these domain names on behalf of their customers, freeing them from the necessity to create and operate the authoritative name servers for their zones.

5.5 Operation of the DNS

While leaving operation of the name servers to the registrars or some other dedicated DNS operator is a good solution for relatively simple zones, complex and ever-changing networks often require to maintain control over the zone's content. Yet providing the infrastructure for reliable name resolution – an absolute necessity for any kind of network service – isn't easily done.

Much like in the example shown above, zones almost always provide more than one name server. This will keep the name service available even if one of the servers fails. If the servers reside in the same location, however, both will become unreachable in case of network failures either in the location or for part of the network when there is a segmentation. Thus, the servers should be in different locations, ideally in far away parts of the world. Another option is to use technologies such as anycast to make multiple servers in different parts of the world share the same address and have DNS queries routed to the nearest server.

For anyone not in command of vast resources, operating such a topology all by themselves is nigh impossible. DNS recognizes this and provides a means to outsource part of the operation of a zone: the different name servers authoritative for a zone can be operated by different parties. Means are provided to synchronize the zone data between these servers.

The scenario typically used is to have a single name server which holds the one true copy of the zone data, called the *primary name server* or *master name server.* The maintainer of the zone updates the data on this server only. All other name servers mentioned in the NS records function as *secondary name servers.* They receive their zone data from the primary via a mechanism called *zone transfer.*

Whenever zone data changes on the primary server, it will send notifications to all the secondary servers by way of a special DNS message called a NOTIFY. In response to this message, the secondary servers fetch the updated zone data from the primary server using yet another special DNS interaction termed AXFR, a somewhat creative abbreviation for ‘authoritative transfer.’

However, the secondary server shouldn’t just start such a zone transfer on any NOTIFY they receive. Zones can be big and zone transfer can take a long time. Instead, the secondary server should be able to check if it perhaps already has the current version of the zone data. This is one of the reasons why each zone carries a sort of bookkeeping record with its apex node. This record, called the SOA record for ‘start of authority,’ contains, among other things, a version number for the zone data in the form of an ever-increasing serial number.

Before a secondary server initiates the AXFR, it first fetches the SOA record and compares it with the copy it already has. Only if the serial number has increased will it start with the actual zone transfer.

But as zones become bigger and update more often, full zone transfers become a very ineffective means of synchronization, even if guarded by the serial number. A companion mechanism called IXFR for ‘incremental zone transfer’ allows the secondary servers to request information on changes to the zone data. With this mechanism, the secondary server tells the primary the serial number of the zone data it has and will receive only those records that have been added, changed, or deleted.

For this zone synchronization mechanism to work, the primary server needs to know all the secondary servers so it can send a NOTIFY to them and the secondary servers need to know which primary server to send their zone transfer requests to. But other than that information, which needs to be configured into the name servers, no restrictions apply.

Pending an agreement, secondary name servers can be operated by anyone – which makes their task ideal to be outsourced to dedicated service providers.

But even operating the primary name server isn't something everyone is comfortable being responsible for. DNS is often used in distributed denial of service attacks, something that can overwhelm and cripple a small IT department. This is where a topology called *hidden primary* (or *hidden master)* comes into play. Here, the primary name server doesn't actually appear in the NS records of the zone, only the secondary servers are listed. Thus, the primary server will never receive normal DNS queries from the wider Internet. It only needs to be able to communicate with the secondary servers and can be secured accordingly.

5.6 Management of Zone Data

The primary server, regardless of whether it is hidden or public, needs to have access to the data of the zone: all the resource records for nodes that are part of the zone as well as delegation and glue records. The original specification suggested to use a simple text file for this format representing each of the records in a standardized format. Since these files contain the one true version of the zone data, they are called *master files* by the DNS specification and *zone files* in practice.

Their format is essentially the same as in the examples given above. Each line contains one resource record. It starts with the domain name of the record's owner. Since all such names have to be within the zone they end with the domain name of the zone's apex, called the *origin.* To save people a lot of time, the names can be given relative to this origin. If they end properly in a dot they are complete domain names and if they are not, the origin is appended to complete them. For instance, to include records for ‹alcor.example.com› as part of the zone ‹example.com› the domain name stated for these records could either be ‹alcor.example.com.› (note the trailing dot) or just ‹alcor›. If the previous record was for Alcor, too, the name can also be left out and the line be indented with a least one white-space character instead.

The domain name is followed by the TTL, the class, and the resource record type. The former is a number while the latter two are the mnemonics used for class and type. Since these mnemonics are taken from a single shared name space of only letters, the order of the three elements doesn't really matter. However, because TTL and class are optional, the type always needs to be last. If left out, both are replaced by the last explicitly stated value.

The record type is followed by a textual representation of the record data. This representation is specified for each record type. Thus, whoever reads the zone file needs to know of all the record types used in the file and their textual representation.[4]

As with all formats, there is a number of additional rules for escaping special characters and violating the One Record Per Line rule for very long records.

[4] RFC 3597 [6] extends the master file format to also allow representation of unknown class and record type values as well as the record data for unknown types.

To illustrate how such a zone file looks in practice, here is a zone file for ‹example.com› as we have defined it so far:

```
$ORIGIN example.com.

example.com. 86400 IN SOA ( nsa.example.com.
                            hostmaster.example.com.
                            2017062101
                            86400 3600 604800 300 )

             86400    NS nsa.example.com.
                      NS nsb.example.com.

             86400    MX 10 alcor.example.com.
                      MX 15 capella.example.com.
                      MX 15 deneb.example.com.

nsa          86400    A    192.0.2.12
                      AAAA 2001:0DB8::12
nsb          86400    AAAA 2001:0DB8::13
alcor        86400    A    192.0.2.81
                      AAAA 2001:0DB8::81
capella      86400    A    192.0.2.42
                      AAAA 2001:0DB8::42
deneb         3600    A    192.0.2.43
                      AAAA 2001:0DB8::43
```

The first line is a control directive and spell's out the zone's origin. All the other lines contain resource records: first, the SOA record and the two NS records necessary for bookkeeping, followed by three MX records that direct incoming e-mail for ‹example.com›. Finally, there are address records for five hosts within the domain.

The greatest advantage of using zone files is that virtually all name servers understand and process it. As simple text files, they can be placed in a version control system or can be generated programmatically from other data.

However, as zones become more dynamic with content changing more often, managing these files can become difficult. If zones are managed through APIs – an example would be dynamic DNS services where customers can update the host address for their machines in automatically response to it changing – or if they are controlled via configuration management systems, creating the intermediary zone file format from other data can be burdensome.

This is why some name servers provide alternative approaches to storing zone data. This could be a relational database or a directory service. For instance, the DNS name server part of the Microsoft Windows operating system can use Active Directory for storing zone data. Since this is used throughout the system for management purposes already, DNS zone management is thus kept in line with how other parts of the system are controlled.

As an alternative to manipulating the zone files or whatever other database the server uses directly, a procedure exists to update zone data through means of DNS. This is called DNS UPDATE after the special type of DNS messages employed for it. [9] In essence, the changes to be made to the data for a zone – records to be added, removed, or updated – as well as conditions that must be met in order for the update to be applied are collected into a DNS message and sent to the primary server. The conditions can be added to make sure that the manipulation isn't going to move the zone data into an unexpected state.

On many Unix systems, a command line tool named *nsupdate* exists that can be used to construct and send these messages. However, not all authoritative name servers support the update mechanism, so this isn't necessarily a portable way of modifying zone data.

5.7 Querying the DNS

The concept of delegation whereby the name servers maintaining the data for a zone can be discovered only from the parent zones means that someone querying for data potentially needs to query many name servers to eventually discover those authoritative for the zone that can deliver an answer. Many of these intermediary queries happen over and over. For instance, the root zone's servers need to be asked for the name servers for the ‹com› zone every time someone browses to a domain in this extremely popular TLD. Since these records change very rarely, they can be cached instead of being requested time and again. Moreover, as many systems in a local network will ask for records in the same zone – for instance, everyone is using the most popular search engine all the time –, it makes sense to provide this cache for the entire local network.

Such a cache is best provided in a transparent manner: Instead of being explicitly queried and filled by the local users, a better procedure is to proxy all DNS queries through this cache. If it already knows the answer, it can return it immediately. If it doesn't, it can retrieve the records from the wider Internet and both return and store them in the cache for later reuse.

This has an additional advantage: the users only need to ask one instance for answers instead of having to perform the rather complex search for a responsible name server themselves. The cache takes over the responsibility to recursively discover the records: it is a *recursive resolver* or simply a *recursor.*

All the other systems in the local network as well as all the applications running on these systems now only need to query the recursive resolver which means their logic can be a whole lot more simple. Since they are only formulating queries and processing the answers but don't do all the heavy lifting, they are called *stub resolvers.* Indeed each networking application comes with a stub resolver. It is typically part of the system library used by the application.

The recursive resolvers to be used need to be configured in the system. Since this is such a fundamental part of the network configuration – without DNS the system effectively wouldn't be able to do anything useful with its Internet connection –, discovery of the local DNS resolvers was added to the DHCP protocol for auto-discovering the network configuration,

which is how most systems get their configuration and update it when switching to different networks.

In practice, there exists a hybrid form between a stub and recursive resolver. While being configured as a network's resolver receiving the queries from all local stub resolvers, it in turn redirects all of them to a fixed set of upstream resolvers while caching the answers. These hybrids, called *forwarders* or *caches,* often appear in home routers to improve response times for often asked queries by omitting the round trip to the ISP's recursors.

In complex network topologies, elaborate combinations of recursors and forwarders are used to provide an efficient and reliable DNS service to all systems.

5.8 Down to the Wire

Communication between all the components, resolvers, both stub and recursive, and name servers and even the special procedures like zone transfer or update, use the same wire protocol. The protocol follows a request/response pattern where someone creates and sends out a request that eventually will be processed by someone else who returns a response with either the results if processing succeeded or an error message. Furthermore, both requests and responses use the exact same message format.

Messages consist of a fixed-length header with bookkeeping information, such as whether the message is a request or response, what type of operation was requested, etc. The header is followed by the question section which describes the information requested. It is a list of questions, each consisting of a domain name, class, and record type. Finally, there are three sections each containing a (potentially empty) list of resource records. These sections are the *answer section* for records that answer the question asked, the *authority section* for records describing who is authoritative for the answer, and the *additional section* for records that help with or are necessary for making sense of the answer.

In a regular query the client wants to get all the resource records identified by a triple of a domain name, class, and record type. Not incidentally, that is exactly what a question is. Thus, the request for a regular query contains exactly one question in the question section and leaves all the record sections empty.

In the normal topology, this request message is sent by the stub resolver to its upstream resolvers. Upstream will check whether it can answer the request from its cache. For this to be possible, it needs to hold the answer in its cache and that answer mustn't have expired yet. To check this, less time must have passed since it inserted the answer than the smallest TTL of any of the records that are part of the answer. This is because an answer has to be complete – the resolver can't just drop expired records and return the rest.

If the cache entry is valid, the resolver can simply take it and construct a response message to be returned to the stub resolver. The TTL of all the records included in that message needs to reflect the time that has passed since the records were added to the cache. As a result, the TTL in a response is often lower than the one given in the original zone file: it shows the time left until an upstream resolver will have to fetch the response anew. If records have been modified upstream, it will also take this time until both the upstream resolver and,

transitively, the stub resolver will realize the fact. This explains the delay experienced between updating records in the zone and clients actually picking up the changes. The delay depends on the TTL of the records in the zone. If it is set to 86400 seconds, as is often the case, it can take up to a day for clients to catch up, depending on when exactly they asked for a record last before it was changed.

Choosing a good value for the TTL is therefore an important decision when crafting zone content. Cached responses arrive more quickly at a client's improving overall response time, for instance when opening a web page. A large TTL makes it more likely that a client will receive such a cached response. On the other hand, large TTLs make it more difficult to change values, which make operational changes more difficult as long transitional phases become necessary and disaster recovery more complicated.

If a response isn't available from its cache, the upstream resolver will have to forward the request somewhere. A simple forward can just send it to one of its upstream resolvers, pushing away the work.

Eventually, a recursive resolver will have to determine the name servers authoritative for the zone the requested records are part of. It needs to start somewhere, so it needs to know of at least one name server that can be used to bootstrap the process. Because of the tree structure of the domain name space and delegation from there into sub-trees, name servers for the root zone can be used to discover any zone.

Thus, the resolver forwards the request to one of the root servers. Apart from its exposed location, this server is no different than any other name server. When it processes the request, it looks at the domain name in the question first. There is one of four things that can happen. If it has at least one record (of whatever record type) for the name, the name exists and is part of a zone the server is authoritative for. This means that it can assemble the response and return it. If there are records of the type asked for, it will add these to the answer. Otherwise, it will return an empty answer as part of the otherwise successful response.

If the server doesn't have records for the domain name, it might at least have delegation records for a suffix of the name. This would mean that some other name server is responsible for the sub-tree the domain name lies in and the server can direct the requestor to a name server that brings it closer to the requested domain name.

If there are no records for the name and no delegation towards the domain name but the server is authoritative for a zone whose apex is a suffix of the requested domain name, the server is authoritative for the requested domain name but knows of no records owned by it. In this case the server can authoritatively say that the name does not exist, an error message known as NXDOMAIN.

Finally, if the requested domain name isn't below the apex of any of the zones known to the server, it can't really make any statement and will refuse to process the request.

For the request towards the root server, the delegation case will happen if the requested name is an existing TLD. If, as in the initial example, the request is for the MX records for

‹example.com›, the root server should produce a response pointing the resolver to the name servers that are authoritative for the ‹com› zone. Using the format used by the Unix tool DIG for displaying DNS messages, its request would look something like this:[5]

```
;; QUESTION SECTION:
;example.com.                    IN    MX

;; ANSWER SECTION:

;; AUTHORITY SECTION:
com.                  172800     IN    NS    b.gtld-servers.net.
com.                  172800     IN    NS    a.gtld-servers.net.
com.                  172800     IN    NS    c.gtld-servers.net.

;; ADDITIONAL SECTION:
b.gtld-servers.net.   172800     IN    A     192.33.14.30
b.gtld-servers.net.   172800     IN    AAAA  2001:503:231d::2:30
a.gtld-servers.net.   172800     IN    A     192.5.6.30
```

The answer section is empty because the name server doesn't have any records that answer the request. The authority section contains the delegation records of the zone the server thinks is responsible for the requested domain name, that is, its NS records. The additional section contains the glue records: all address records for the name servers of the delegated zone that the server knows of. In this example, the root server only has been told address records for two of the three servers returned.

When the recursor receives this response, it can try its request again at a name server closer to the requested name. It will pick one of the name servers, possibly start another query for its address records if they aren't included in the additional section, and repeat the request there. For the example, this will, once again result in a response with an empty answer, this time pointing to the name servers for ‹example.com›. Another step is necessary – the request needs to be sent to one of these servers.

Here, finally, records are available. The server might respond like so:

```
;; QUESTION SECTION:
; example.com.                IN    MX

;; ANSWER SECTION:
example.com.          86400    IN    MX    10 alcor.example.com.
example.com.          86400    IN    MX    15 capella.example.com.
example.com.          86400    IN    MX    15 deneb.example.com.

;; AUTHORITY SECTION:
example.com.          86400    IN    NS    nsa.example.com.
```

[5] The real response will contain more records. The ‹com› zone actually has thirteen name server entries.

```
example.com.          86400   IN   NS    nsb.example.com.

;; ADDITIONAL SECTION:
alcor.example.com.    86400   IN   A     192.0.2.81
alcor.example.com.    86400   IN   AAAA  2001:0DB8::81
capella.example.com.  86400   IN   A     192.0.2.42
capella.example.com.  86400   IN   AAAA  2001:0DB8::42
deneb.example.com.     3600   IN   A     192.0.2.43
debeb.example.com.     3600   IN   AAAA  2001:0DB8::43
nsa.example.com.      86400   IN   A     192.0.2.12
nsa.example.com.      86400   IN   AAAA  2001:0DB8:12
nsb.example.com.      86400   IN   AAAA  2001:0DB8:13
```

There now are records in the answer section! Much like in the delegation case, the authority section lists the name servers for the zone. The additional section now is larger. It lists all the address records, both IPv4 and IPv6, for the mail servers mentioned in the MX records and the name servers from the NS records.

5.9 Message Size Limits

Recursive discovery of name servers requires a recursor to rapidly fire off requests to a number of different servers. This process needs to be fast if it shouldn't incur a large delay in an application. A connection-oriented transport protocol such as TCP requires a number of round-trips just to establish a connection before any payload data can be exchanged. On the other hand, most connection-less protocol such as UDP don't guarantee packet delivery. The only way to deal with packet loss is to stop waiting for a response after a certain time and declare a packet lost. Favoring the faster response time for the much more common 'happy path,' DNS uses UDP.

This, however, has one additional drawback: When using UDP, DNS messages need to fit into a single IP packet.[6] The initial DNS specification therefore limited the size of DNS messages when using UDP to a conservative 512 bytes. An extension mechanism has since allowed to increase the size where knowledge of the underlying network topology allows. In practice, the limit is 1280 bytes.

This still isn't an awful lot and needs to be considered when designing applications and extensions for DNS. Data sets that can become large cannot be stored in DNS directly. Instead, a better strategy is to only store pointers in DNS describing how data sets can be retrieved using other protocols. Uniform Resource Identifiers (URIs), a standard mechanism describing both the protocol to be used and the location of resources, are a prime candidate for such a pointer. The data LIGHTest is making available via DNS – trust lists, trust translation and trust delegation declarations – is data that can become quite large. Because of that, chapter 8 will explore this option in detail.

[6] Technically, IP fragmentation would allow to split a UDP packet into several IP packets, however, this often breaks in the presence of firewalls or NAT devices.

Another case benefitting from this strategy is often changing data. As described above, the caches used by the various resolvers all introduce a potential delay for propagation of updated DNS data. Data retrieved from, say, an HTTP server, however, can be updated as often as needed or even generated dynamically for each request.

If data doesn't change very often and DNS is merely used to verify that information retrieved in some other way is indeed correct, another strategy is to store a hash over the data in DNS. Chapter 7 will show how this is used by DANE to store the fingerprint of a certificate in DNS instead of the actual certificate.

Yet however small the records are, there will always be cases where a response won't fit into a single packet. One example are zone transfers where all the records of an entire zone are to be included in the response. Only the smallest of zones will fit into one UDP packet. This is why a server can indicate that it had to truncate the message when constructing the response and why DNS can also use TCP as its transport protocol. Originally, TCP support was optional but the added data from DNSSEC as explained in the next chapter has made it more common that messages grow too big and TCP is now mandatory. [8]

But even with TCP there still is a size limit. When using this transport protocol, all DNS messages are prefaced with their length expressed as a sixteen bit value, meaning that a message can be at most 65536 bytes long. For transferring large zones, even this may not be enough. The mechanism therefore allows splitting the data over several DNS messages.

5.10 Extending DNS

Like many Internet protocols, the ability to be extended to new use cases nobody could even think of during design is part of DNS's DNA. Indeed, as the curious case of the class attribute shows, the specification provides more options for extension than is actually needed.

Of course, being extendable doesn't necessarily mean that a protocol is the best tool for a given problem. The original design choices inform a set of limitations. If these collide with the problem, one can try to work around them or look elsewhere.

For DNS one such limitation is that it has been conceived as a public database. All the data stored is always available to everyone who asks. There is no method of authorization for regular queries.[7] While it is possible to work around this and make zones only available based on the source address of the requestor, this isn't intended and has a tendency to break in unexpected ways. Use cases that require authorization for accessing information are better off using a different protocol or employ a hybrid strategy where DNS is only used to discover the location of the data and transport protocol for access.

Another point to consider is that the only input when searching for data is the domain name. This is often quite useful, since a domain name is guaranteed to be under the control of a

[7] For operational queries, such as zone transfers or UPDATEs, a mechanism called *transaction signatures* or TSIG allows the receiving party to verify authorization of the sender and correctness of the data. However, this mechanism requires bilateral, administrative agreements.

single party limiting the chance of an accidental name collision. That is, domain names are well suited as unique identifiers. If searches are based on such identifiers, then DNS is a good choice for a database, particularly, if the identifiers can be chosen freely to conform with the domain structure.

Existing unique identifiers can be converted into domain names. For instance, for reverse pointer lookups – queries to identify the host name associated with a given IP address –, IP addresses are converted into domain names by using the decimal representation of the address components, bytes for IPv4 addresses or segments for IPv6, as labels. The resulting sequence of labels is then anchored somewhere well-defined in the DNS name space. For IPv4 reverse lookups, this is ‹in-addr.arpa›, so that the address 192.0.2.88 turns into the domain name ‹88.2.0.192.in-addr.arpa›. If the sequence of labels cannot be modeled in accordance with the ownership of individual identifiers, zone delegation cannot follow this ownership requiring administrative overhead for coordinating updates of zone data between owners of identifiers and owners of the respective zones.

If search input isn't based on identifiers at all but rather on a set of specific values, DNS is probably not a good choice. A borderline case with a feasible workaround is when a unique identifier is combined with a small set of specific values. For instance, the SRV mechanism allows to discover the hosts and port numbers where a certain network service is provided for a domain. Here, the domain is the unique identifier whereas the service is additional input. This input is provided as a series of prefix labels to the domain name, each with a concrete meaning. In case of SRV, the prefixes are first the name of the service, followed by the transport protocol since the protocols for some service can used both atop TCP and UDP. To avoid collisions with actual domain names, these prefix labels are formed by including characters that are illegal for host names, specifically, they start with an underscore. So, when searching for the host names that provide the HTTP service for ‹www.example.com›, one would query for the SRV records for ‹_http._tcp.www.example.com›.

The alternative option would have been to include this information – transport protocol and service – in the record data of the individual SRV records. But since a query can only be for all the records of a given type under a given domain name, each query would return the records for all the services provided for this domain. This is both a waste of bandwidth and a potential security issue as it would allow to enumerate the services provided. The 'underscore prefix labels,' as cumbersome as they appear at first, neatly sidestep these issues.

6. DNSSEC

6.1 Threats to DNS

To have a better understanding on how DNSSEC provides solutions, it is necessary to know the existing problems. Even while there are several different classes of threats to the DNS, few of them are specific to singularities of the DNS protocol itself.

6.1.1 Man or Monkey in the Middle (MITM) Attacks

The well-named man (or monkey) in the middle attack is one of the principal techniques most often employed in computer-based hacking. Basically, it consists of being in the middle of the conversation secretly, with the possibility to modify the communication between the two hosts who believe they are communicating with each other directly.

Regarding the DNS protocol, the hacker can simply tell either party, commonly the resolver, whatever it wants that party to receive. The receiver of data from a DNS name server cannot know the authenticity of its origin or verify its integrity. This is because DNS does not detail a procedure for servers to give any authentication detail for the data they will push down to clients. In case of a resolver, there is no mechanism to check the integrity and authenticity of the data sent by name servers. In addition, the resolver can only verify the authenticity of the origin of a DNS response data packet using the source IP address of the DNS server, destination and source port numbers and DNS transaction ID. The attacker can fabricate in an easy way a DNS server's response packet to answer properly using the same parameters. The receiver of this new answer has to trust as reliable the data provided by the attacker. The attacker might even choose to return an unaltered answer of a reply message while using other parts of the message to modify it.

Even more, with the ceaselessly growing usage of wireless networks, the access to non-secure networks is more common allowing new types of man-in-the-middle attacks

6.1.2 DNS Spoofing

In the situation where a DNS server cannot resolve a given query, it forwards the query to a second DNS server higher up in the tree of server. Here ends the communication between DNS client and first contacted DNS server for that particular request. This mechanism is called DNS forwarding.

One kind of man-in-the-middle attack is trying to replace the mentioned higher DNS server during DNS forwarding. The DNS client is fooled into thinking that it is receiving a response from a trusted DNS server when, in fact, it is being *spoofed*. Each DNS packet has an associated Transaction ID, a16-bit field that DNS servers use to determine what the original query was. This attack can be performed by means of guessing what the next DNS response transaction ID will be and sending a reply with the guessed sequence number to a DNS client. Due to its small amount of bits and the server UDP port associated with DNS is a well-known value, there are only 2^{32} possible combinations of ID and client UDP port for a given client and server. It implies a security breach against brute force attacks to know this

combination. Because of the mechanism used by this technique, this type of attack is known as 'ID guessing.'

6.1.3 Cache poisoning

A primary component of the DNS architecture is the ability to cache responses to queries in order to reduce the access times associated with the DNS service. DNS servers cache all information for all zones the DNS server is authoritative for and the results of all recursive queries performed since their last start up to save time in case they receive a similar query again. As it could be imagined, the DNS cache poisoning involves replacing the information stored in the cache records. The next time the DNS server is queried, it will reply with the incorrect information.

There are some variations on this kind of attack, but what they all have in common is the records infected, the resource records.

6.1.4 Name Chaining

This variation of cache poisoning is achieved by means of resource records whose record data includes a domain name which can be used as a place where an attacker puts wrong data into a target's cache. The most affected in this class of records are CNAME, NS, and DNAME. False information, associated with these names, can be injected into the victim's cache using the additional section of the response. An attacker can introduce arbitrary DNS names of the attacker's choosing, and provide further information that is claimed to be associated with those names. [9]

6.1.5 Hijacking

This kind of attacks tries to take advantage of a weakness in the administrative side of domain name services rather than technically attack the infrastructures or DNS servers.

Also known as name-jacking, this attack consists of appropriating the domain name or taking control by technical means to divert traffic to a rogue domain under the control of the attacker, such as by modifying the name servers hosting the site, or modifying the behavior of a trusted DNS server. In the case the integrity of DNS has been compromised, anyone who attempts to reach a website could connect to the attacker's websites without knowing it. Once there, they may be tricked into exposing any kind of sensitive information.

The common use of the term encompasses a number of attacks and incidents including [10]:

- impersonation of a domain name registrant in correspondence with a domain name registrar,
- forgery of a registrants account information maintained by a registrar,
- forgery of a transfer authorization communication from a registrant to a registrar,
- impersonation or another fraudulent act that leads to the unauthorized transfer of a domain from a rightful name holder to another party,
- unauthorized DNS configuration changes that disrupt or damage services operated under a domain name, including web site defacement, mail service disruption, pharming and phishing attacks.

6.1.6 Denial of Service (DoS)

An understandable definition could be that an attacker attempts to prevent users from accessing all or part of the information or services hosted in a system by targeting the host and its network connection. The most common type of DoS attack occurs when an attacker overflows a network with useless information, also known as flooding the network. These attacks could be targeted at a specific service like DNS, usually to the specific root servers DNS relies on.

It is important to highlight the risk that DNS servers that are not broken down by the DoS attack but manage to process the incoming flood of request can then be used as a springboard for an amplification attack against other target servers. Since DNS responses are significantly larger than the requests, a DoS attack with much higher bandwidth can be produced by spoofing the source addresses of the request as those of the target servers.

As mentioned before, these attacks could be aimed to the DNS servers or the network infrastructure. Using the reference of OSI levels, it is possible to find DoS attacks on the network infrastructure between the clients and the servers in all layers of the network infrastructure [11]. It could mean that if the whole network gets down that it cannot be expected that DNS continues to work but possible DNS local services can be run as a backup.

This category includes the reflection attack which occurs when thousands of requests are sent using the name of the victim as the source address. When recipients answer, it means all responses will converge on the official sender, whose infrastructure is then affected.

DoS attacks are hard to prevent because it is very difficult to distinguish a DoS attack from a normal peak in the visits to a large website. Using authentication could allow to make the distinction by identifying the single origin of a DoS attack by looking at the distribution of packets over IP addresses.

6.1.7 Distributed Denial of Service (DDoS)

Distributed attacks are a more elaborate way of a DoS that involves a huge amount of attackers, generally, launched simultaneously by a large number of systems. It is almost impossible to detect such an attack if multiple hosts run the attack in a coordinated way against their target. This type often involves so called *botnets*, which consist of thousand machines controlled using various tools by the attacker so that the combined bandwidth exhausts the available bandwidth of most victim's systems.

Commonly used forms of DDoS attacks, both past and present:

- Transport layer SYN flood: the requester first sends a SYN message to initiate a TCP conversation with a host, that responds with a SYN-ACK message and its corresponding receipt confirmation. In case of the attack, the requester sends multiple SYN messages to the targeted server, but does not transmit any confirmation ACK messages. The requester can also dispatch spoofed SYN messages, causing the server to send SYN-ACK responses to a falsified IP address.

- Ping of Death: where the attacker manipulates IP protocol by sending packets larger than the maximum byte allowance divided in fragments. Once reassembled the fragments it creates a packet larger than allowed, causing servers to reboot or crash.

- ICMP Flood (also known as *Nuke)*: This attack occurs as a result of sending corrupt and fragmented ICMP packets that overload the targeted network's bandwidth and impose extra load on the firewall. This attack relies on compromising user networks and is an old distributed denial of service attack.

6.1.8 Denial of Existence

Denial of existence is a mechanism that informs a resolver that a certain domain name does not exist. It is also used to signal that a domain name exists but does not have records of a specific type.[12] The threat appears when the resolver is unable to detect whether an attacker removes resource records from a response. Depending on the nature of record types the non-existence may cause an immediate failure. Authenticated denial of existence uses cryptography to sign the negative response.

6.2 Digital Signatures for DNS Records

In order to be able to respond to these threats, a resolver needs to be provided with a way to verify the authenticity of the resource records it receives as answers to queries. A common strategy for verifying the authenticity of data is to use public-key cryptography to provide signatures of the data.

Instead of using a single key for both encrypting and decrypting data, public-key algorithms employ separate keys for the two steps. This pair of keys is chosen in such a way that whatever is encrypted using one key only be decrypted with the other. If the owner of the key pair openly publishes one of the keys, the *public key,* and keeps the other one, the *private key,* thoroughly under lock, everyone can use the public key to encrypt a message that only the owner can decrypt again.

But the keys could also be used the other way around: If the encryption key is kept private and the decryption key made public, the key pair's owner can encrypt a message using the private key which everyone can decrypt with the public key. While this seems a bit pointless at first, it is worth noting that the original data and the decrypted data will only match if the matching pair of keys was used – i.e., if data indeed came from the owner – and if the encrypted data hasn't been tampered with. If the original data itself is included in the message, it can be compared with the decrypted data. If they match, this application proves that the original data is indeed from the owner of the private key and that neither the original data nor the encrypted data have been changed. In turn, if they don't match, one of these wasn't true. It isn't possible to say which, but at least it is clear that the data shouldn't be used.

Sending both the original and encrypted data side-by-side isn't very efficient. The encrypted data doesn't really need to contain all the information to recreate the original data. Instead, a more compact representation of the original data could be encrypted and sent along, provided that it is not possible (or at least very expensive) to create alternative original data

that would result in the same compact representation – resulting in verification incorrectly succeeding. Cryptography provides so called *hash functions* that provide exactly this property. They create a short *message digest* from input data of any length that fulfills these conditions. If this digest is encrypted using a private key, it becomes a *digital signature.* The process of creating a digital signature and adding it to the data before sending it out is called *signing a message.*

When verifying some data accompanied by a digital signature, a receiver first decrypts the digital signature into the message digest using the public key published by the sender. It then calculates the digest of the received data and compares it to the decrypted signature. The data verifies as authentic if and only if the two match.

In the case of the DNS, a network user wishes to verify that the data received in response to some query is authentic. The data to be signed is the DNS message answering the query. It needs to be signed by whoever assembled it – and the user needs access to the public key of that party.

In the case of a stub resolver this is relatively straightforward. The upstream resolvers have to be configured somehow, anyway, so the public keys used by them can be installed as part of that configuration. A recursive resolver, however, has to talk to a wide variety of name servers, potentially each and every name server on the whole Internet. Manually installing and maintaining all the public keys used by all of these servers is impossible.

But what is the answer, exactly? It consists of all the resource records of a given type for a given domain name and class. No matter who requests this answer, it will always be the same so long as this set of records doesn't change at the primary name server authoritative for the zone covering the domain name. If the digital signature were created by that primary name server and passed along when transferred to the secondary servers of the zone and when answering queries, then it could just be forwarded and cached by resolvers.

This is exactly what DNSSEC does: when updating a zone's data, all the resource records for the same domain name, class, and record type are collected into a *resource record set* – written and even pronounced as *RRset.* A digital signature is then created for this RRset. Before that happens the records are sorted into a well-defined order, though, so that a client can do the same when later re-creating the digest for verification.

This signature needs to be published. Yet the existing DNS protocol shouldn't be changed in any way disturbing operation of clients and resolvers that aren't aware of DNSSEC. Which means that the message format needs to remain the same. Luckily, the DNS provides means for extension, one of which is additional resource record types. Enter the RRSIG record: one such record will be added for each RRset of the zone. It will be stored under the same domain name and class as those of the RRset and contains the record type and signature as well as some additional bookkeeping information as the record data.

6.3 A Chain of Keys

Which leaves the question of the keys. Which key pair should be used when creating these signatures and how will the clients make sure they have the correct public key when trying to

verify the signatures. The latter point is quite important. Imagine an adversary is in the position to intercept and manipulate communication from and to a resolver. It can now generate a key pair and use the private key to create signatures for forged upstream answers. The resolver will verify these answers as authentic if the adversary somehow manages to convince it to accept the public key of the adversary's key pair. This is the classic case of a man-in-the-middle attack. All public-key cryptography is susceptible to this attack and requires measures to allow verification of public keys.

At this point the scheme to allow verification of the authenticity of data relies on the verification of the authenticity of the public keys. The latter can be achieved by using the scheme itself and signing the key; adding a signature to it using some other key pair. Granted, this only delays the inevitable: at some point there needs to be a key that is known to be correct. This is called the *trust anchor.* It needs to be made known to the system somehow. If a chain of digital signatures leads from the public key used to sign the data to this trust anchor then a key is legitimate.

Or, almost: a key needs not only to be legitimate in general but it needs to be authorized to sign the particular data, i.e., the particular RRset. It makes sense to tie that authorization to the authorization to manipulate the RRset in the first place: the zone. If the entire zone uses the same key, it can be published as part of the zone data much in the same way as the authoritative name servers are published. What's more, in the same way that those name servers are legitimized by being made part of the parent zone as delegation records, the key can be legitimized, too.

This is exactly what happens: the key, or rather keys – a zone is allowed to use more than one key at any time –, are published in the form of DNSKEY records under the domain name of the apex of the zone. A digest of these keys, called (in a case of mixed analogies) a *fingerprint,* is published in DS records as part of the delegation records in the parent zone. The parent zone uses its own keys to sign these DS records, i.e., to create an RRSIG record for it. By doing so, the parent zone signs the keys transitively by signing their fingerprints and confirms that these keys are legitimately used to sign the child zone.

Figure 3 shows the relationships between the resource records and the corresponding RRSIG records, their signing keys referenced in the the DNSKEY record and the secured delegation through the DS records signed in the parent zone.

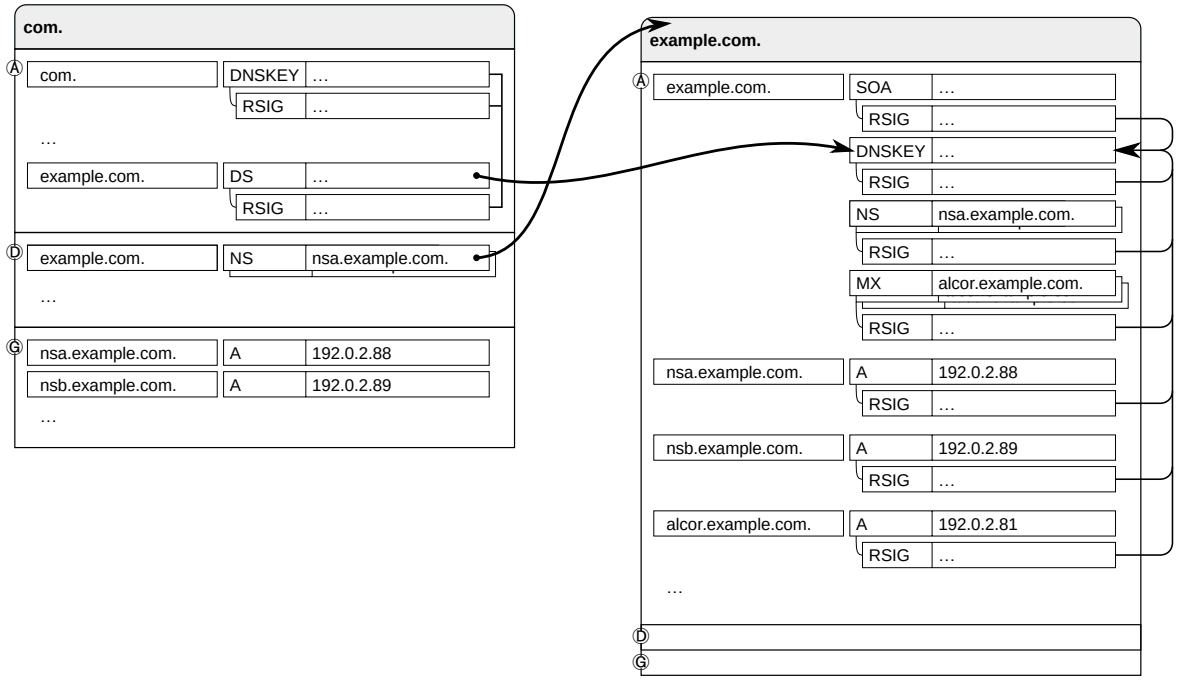

Figure 3. Delegation from ‹com› to ‹example.com› with DNSSEC. The figure shows how the DS record in the parent zone relates to the DNSKEY records in the child zone. It also shows the RRSIG records in both zones.

The same is done for the keys of the parent zone by its parent zone and onwards all the way to the root zone. Finally, the keys of the root zone serve as the trust anchor and need to be known to a resolver that wishes to verify DNS data. Using these configured root keys, the resolver can follow the chain all the way down to the individual RRset.

Relying on the chain of keys for verification has a few important consequences. For one, just providing the DNSSEC-related resource records for an individual zone is not enough to allow verification for it. In order to be able to use these records, the zone's parents all the way up to the root need to be signed, too. If only one zone isn't signed in this way, the chain is broken and verification is impossible.

While the root zone itself has been signed since 2010, still not all TLD zones are signed or allow their delegate zones to provide them with DS records to be installed as part of delegation. When relying on DNSSEC, it is thus important to choose a TLD that does provide DNSSEC and make sure that all zones between that TLD and the zone holding the intended records are signed and properly delegate DNSSEC information, too.

What's more, one has to trust that all the parent zones remain true and honest. Nothing really stops the parent domain from changing delegation records or entirely taking over a child zone. It is therefore somewhat important to keep the number of zones towards the root not under one's own control small. In addition, care should be taken when deciding on a TLD to place the zone under, choosing a TLD run by a trustworthy organization instead of, for instance, picking a TLD solely on the sound of the resulting domain name.

6.4 Operating DNSSEC-aware Zones

Manually providing all the additional resource records necessary to allow a zone to be verified via DNSSEC is not very practical at best and, for larger zones, nigh impossible. This task is therefore given to software. Either the primary name server of the zone is aware of DNSSEC and can take over the task of re-signing the zone every time the zone data is updated or special dedicated software is being deployed which signs the zone before passing it on to the primary name server. The latter case is similar to the concept of a hidden primary introduced in chapter 5, an arrangement that should also be considered if the former case is chosen.

One of the reasons for this is the matter of key management. The integrity of DNSSEC relies entirely on the integrity of the private keys used for zone signing: securing the key against loss or misuse is an important operational consideration. Since the system performing the key signing needs to have access to the private key, securing this system is vital.

This is all the more important because delegation makes it more difficult to exchange a compromised key. For this to happen, the DS records of the parent zone need to be changed, which may require manual intervention and incur delays.

To mitigate this problem, DNSSEC allows for a kind of layered key management. Instead of using the keys registered with the parent zone directly, they are only used to sign the DNSKEY RRset. This RRset contains a number of additional keys that are actually used for signing all the other records of the zone. According to their use, the keys found in the DS records is called *key signing keys* (KSK), the additional ones are *zone signing keys* (ZSK). Because only the zone signing keys are needed when zone data is updated, only they need to be kept online. The key signing keys are only required when the DNSKEY RRset is updated. Since that happens relatively rarely, they can be stored away safely offline.

In this scenario, if a zone signing key is compromised and needs to be removed, only its DNSKEY records need to be removed and the DNSKEY RRset resigned with the key signing keys. The operator of the parent zone does not need to be involved at all.

6.5 Verifying DNS Records

Once the DNSSEC resource records are part of the zone data, users can start validating the answers they receive. This consists of two steps: verifying that any RRset provided in the answer matches the signature provided in the corresponding RRSIG record and verify that for the key used to generate the signature exists an unbroken chain of signatures tracing all the way to the trust anchors.

Especially the second part might require a lot of records to be gathered from various name servers and then their signatures verified. Many of these records will be used again and again for subsequent queries. For instance, each query for any record in a .com domain will require the DS and DNSKEY record sets for ‹com›. The caching provided of the existing DNS model already helps with retrieval of all these records. It makes sense to extend caching to also include the verification result for the records instead of having every stub resolver – which is part of each individual application process – redo all the checks. However, such an

extended cache needs to be trustworthy. A typical way to achieve this is to operate a verifying forwarding or recursive resolver as part of the local system and have stub resolvers use this resolver instead of the one provided as part of the local network.

The result of this process can be one of three things. In the best case, everything pans out: the record set is signed with an unbroken chain of keys. If the record set isn't signed at all or if the chain of keys is broken at some point, no assertion can be made regarding the authenticity of the records; they are said to be *insecure.* This isn't necessarily bad, the fact just needs to be considered.

If, however, verification of any of the signatures involved fails, verification fails. It doesn't matter whether this happens for the signature of the RRset itself or of any of the DNSKEY or DS sets along the chain. The records are said to be *bogus* and must not be used. While the most likely reason is that a misconfiguration happened on a system along the way, it is also possible that someone has tampered with the records – the very threat DNSSEC has been designed to deal with.

The difference between insecure and bogus is subtle but important. If an answer is insecure, verification couldn't be completed because of lack of DNSSEC support in some zone along the way to the root. The response is as good as if there hadn't been any DNSSEC at all. Bogus, on the other hand, means that something is wrong with the records and they mustn't be used.[8]

A centralized verifying resolver will therefore report a server failure instead of relaying bogus records. A user can circumvent this through a special flag in the query, called the CD bit for 'checking disabled.' It instructs the resolver to just do that: not to verify any records and just pass them along.

Another flag, the AD bit for 'authentic data,' is used by a verifying resolver to signal whether the resource records contained in the response could be verified. If the resolver determines one or all of the records to be insecure, it will still return the records but signal their status by not setting the AD bit in the response.

6.6 Limits of DNSSEC

As a result of using DNSSEC, a client can only say whether the records received as an answer to a query are authentic: whether they have been made available with the content delivered by whoever de facto controls the DNS domain they are part of. While that is exactly what DNSSEC set out to provide, it might be worthwhile to keep in mind what DNSSEC does not provide.

Perhaps most importantly, it does not provide for confidentiality. Queries and responses are still sent in plain text over the network, entirely open to any eavesdropper. It also doesn't change the original model of directing queries to intermediary resolvers. These resolvers will

[8] The choice of terms may be a bit unfortunate. Bogus sounds less dangerous than insecure when in fact, the meaning is exactly the other way around.

still be able to gather data about queries made by their respective users and, based on that data, profile their behavior.

Recent work in IETF has tried to deal with these issues. The DPRIVE working group has developed standards for sending DNS communication through encrypted channels using TLS and DTLS. [13] [14] As relatively recent developments, practical deployment experience is currently gathered.

Even with transport encryption and the options of authentication TLS provides, DNS remains a public database.

7. Verifying Identity with DNS

A decision about whether to trust someone for a specific transaction has to be founded on reliably establishing the identity of that someone – before being able to trust someone, we need to be sure to know who they are. While often difficult in the world of personal interactions, this becomes more challenging still with telecommunication. Here, an entity can only be represented by data, a unique token standing in for the entity – not entirely unlike a person's name and date of birth together represent the person in a legal document.

7.1 The System of Certificate Authorities

Verifying identity involves making certain that the entity is indeed authorized to use the token. The method for doing this has already been introduced in the last chapter. In DNSSEC, the authorization of someone to update DNS records needs to be verified. This was achieved by employing digital signatures as a proof that the entity updating the records was in possession of a secret key representing their authorization. Similarly, a digital signature over the token can be used to prove the possession of the secret key authorizing the use of the token.

As in DNSSEC this authorization is normally confirmed transitively starting at some trusted anchor point. While in DNSSEC this chain could follow zone delegation, no such natural relationship exists in the more general case. Instead, the chain is established through so-called *certificates.* Each certificate states that a certain key pair is authorized to be used for a certain identifying token, that is, for a certain entity. Crucially, it also carries a digital signature attached to it by some entity called the *certificate issuer.* As a result, the certificate states that the issuer confirms the key pair's authorization to represent the entity. An entity issuing certificates for other entities is called a *certificate authority* or CA.

Each CA in turn has at least one certificate authorizing their key pair to be used to issue certificates for this CA. Such a certificate could have been issued by some other CA. It could also be issued by the CA itself. This is called a *self-signed certificate*. Because it is self-affirming, no authority can be derived from it; it has to be verified by other means. This happens in the same way as with the root keys in DNSSEC in that they are installed as trust-worthy on the system doing the verification. In fact these certificates – which can be normal certificates signed by other entities – have a similar name: *root certificates.*

The root certificates act as an anchor for certificate verification. When trying to verify an as yet unknown certificate, a chain needs to be built from the certificate via the CA certificate of its issuer and, recursively, their issuers to at least one of the root certificates of the verifier. Only if such a chain exists can the certificate be accepted as verified.

This system of transitive verification based on a number of well-known and trusted CAs forms the basis how identities are verified on the Internet currently. Most systems include a regularly updated list of root certificates. One commonly used list is the Mozilla CA Certificate Store [15]. At the time of writing, this list included 182 entries, each of which is typically used to issue certificates for a CA that then provides certificates used by Internet services.

This results in a rather large number of CAs. Each of these CAs is allowed to issue certificates for any service. In DNSSEC, because the keys are tied to the hierarchical structure of the domain name tree, a key can only be used for domain names part of the sub-tree under its apex. No such limitation exists for the CA system.

An adversary only needs to gain control over one of these CAs to be able to issue to certificates for any service – or their false impersonation of it. This isn't a very good system to build an infrastructure for trust verification on.

7.2 Storing Certificates in DNS

A solution of the issue of any CA being able to issue a valid certificate for a service could be to store the real certificate in DNS. A user of the service could perform a DNS query and compare the certificate stored there with the one offered by the service for its communication. With DNSSEC, this solution becomes realistic, since records stored in DNS can now be verified for authenticity, too.

Over the years, a number of protocols have been proposed to do just that. RFC 2538 provides a resource record type for storing certificates or OpenPGP keys[9], the CERT record. Additionally, it defines the domain names for looking up these records for a small number of services, such as for e-mail addresses or the host names for Internet services using the TLS protocol for providing encryption for their protocols.

A similar record type has been defined for SSH, a protocol for terminal communication, which uses public keys for identifying both users and servers. It normally presents a fingerprint to users allowing them perform a manual check. The SSHFP resource record defined by RFC 4255 [17] allows the key associated with a server to be published in DNS, allowing automated checks, for instance if SSH is used in machine-to-machine communication.

IPSEC, the security extensions for the low-level IP protocol – used for instance in VPNs –, similarly uses server keys that can be published with the IPSECKEY record type defined in RFC 4025. [18]

Ultimately, none of these methods are used widely. Learning from the experience gained with them, however, has led to a new attempt by IETF to provide a framework for storing certificate information in DNS.

7.3 DANE

"DANE is a set of mechanisms and techniques that allow Internet applications to establish cryptographically secured communications by using information made available in DNS. By binding the key information to a domain name and protecting that binding with DNSSEC, applications can easily discover authenticated keys for services." — WG charter

[9] OpenPGP is a system for public key cryptography providing encryption and digital signatures. It does not use certificates but rather raw public keys that are verified via the "web of trust," i.e., by mutually verifying the validity of a key – which really is just a sort of ad-hoc certificate.

DANE stands for 'DNS-based Authentication of Named Entities.' It is used to authenticate TLS client and server entities while taking out the intermediary, i.e., Certification Authorities (CA). Initially, TLS used CAs to provide assurance that the client is speaking to the right server. However, a major challenge is that CAs introduce a single point of failure. If an attacker compromises a CA, the attacker can generate fake certificates for popular domains. One notable such case occurred in 2011, when DigiNotar, among other things responsible for the certificates of the Dutch government, issued fraudulent certificates after a security break. [31] This is the reason why it is better to rely on the network itself using the DNSSEC features rather than relying on CAs.

DANE is a standard that uses DNS and DNSSEC to provide secure authentication for any internet host and it provides a mechanism to specify that a host supports, and actually requests, TLS-encrypted communication. DANE allows a domain holder to determine which CA is allowed to issue certificates for the domain and this solves the problem that any CA could issue certificate for the domain.

7.3.1 TLSA

In order for clients to query the DNS-server for DANE entries, a new DNS resource record was introduced, which is called TLSA.[10] It was first defined in RFC6698 [19] and extended in RFC7218 [20] with acronyms. This new resource record limits the trust anchors used to verify the domain.

A TLSA resource record contains four fields named, respectively, *Certificate Usage, Selector, Matching Type,* and *Certificate Association Data.*

Within a TLSA record, the administrator of the DNS zone can specify one of four different ways to verify a certificate using the Certificate Usage field. The four possibilities are:

- A *CA Constraint* is indicated in the Certificate Usage field when the field is set to 0. It specifies a CA certificate or public key of a certificate which the client must find in the validation path of the certificate given by the server in TLS. Since the particular certificate must be found in the path of the certificate presented by the server, only certificates from the indicated CA are accepted. This effectively limits the CAs that can issue certificates for the particular service on a host.
- A *Service Certificate Constraint* is indicated in the Certificate Usage field when the field is set to 1. This specifies the certificate or the certificate's public key that must match the certificate given by a particular service on a host. It effectively limits which certificate can be used by a particular service.
- A *Trust Anchor Assertion* is indicated in the Certificate Usage field when the field is set to 2. This specifies the certificate or the certificate's public key that must be used as a trust anchor when validating the certificate presented by the server in TLS. This is especially useful for a domain issuing certificates with its own CA that might not be in the end-user's list of trust anchors.

[10] TLSA a merely the record name. It doesn't stand for anything.

- A *Domain-issued Certificate* is indicated in the Certificate Usage field when the field is set to 3. This specifies the certificate or the certificate's public key that must match the certificate given by a particular service on a host. It effectively limits which certificate can be used by a particular service. Note that the difference between this and Service Certificate Constraint is that Service Certificate Constraint must pass the PKIX validation while the PKIX validation is not tested for Domain-issued Certificate.
  (PKIX validation is the process of verifying that a given certificate path is valid under a public key infrastructure i.e., it is traceable to a trust anchor)

The Selector field within a TLSA record specifies which part of the TLS certificate given by the server will be matched against the associated data. The option 0 in this field determines that the full certificate is matched while option 1 specifies that only the certificate's public key is matched.

The Matching Type field within the TLSA record determines the format of the certificate association data. Option 0 in this field specifies that there should be an exact match of the content chosen via the Selector field, option 1 specifies that a SHA-256 hash of the selected content should be used, and option 2 specifies that a SHA-512 hash of the selected content should be used.

Finally, the Certificate Association Data field specifies the data to be matched in the TLSA record according to the other fields.

A TLSA record binds a certificate to specific ports and protocols in order to distinguish between the domain and the port and protocol. The ports and protocols have an underscore added in front of their definition, making them uniquely definable.

The following listing gives a short example of a TLSA resource record. It binds a certificate to a TCP connection using port 443, typically used by a web server, to the domain ‹www.example.com›. The record specifies that the certificate usage uses a CA constraint, the selector is a full certificate, and the certificate must fully match. Thus, a client connecting to the service expects to be presented with a certificate where the chain of issuers contains the specified certificate.

```
_443._tcp.www.example.com.  86400  IN  TLSA (
          0 0 1
          D2ABDE240D7CD3EE6B4B28C54DF034B97983A1D16E8A410E4561CB106618E971
)
```

As can be seen, reading such a record is not trivial. In order to make the TLSA record more readable, mnemonics for the first three data fields have been defined in RFC7218 [20]. Using these, the same resource record will look like in the following example. No parameters have been exchanged.

```
_443._tcp.www.example.com.  86400  IN  TLSA (
          PKIX-TA CERT FULL
          D2ABDE240D7CD3EE6B4B28C54DF034B97983A1D16E8A410E4561CB106618E971
)
```

In conclusion, TLSA resource records bind certificates to protocols and ports and give the maintainer the chance to select the certificate verification path to verify the pinned certificate.

7.3.2 SMIMEA

The TLSA mechanism is limited to certificates used as part of secured machine-to-machine communication. Another protocol that uses certificates is secure e-mail via the S/MIME framework. [21] As part of this framework parts of an e-mail message can be encrypted and signed. Certificates are used as proof of identity much in the same way that TLS uses them to prove the identity of a server. Thus, the same issues exist: any CA related to an installed root certificate can issue certificates for any e-mail address.

Much like TLSA, the SMIMEA mechanism [22] provides a number of ways to limit the certificates that are acceptable for a certain e-mail address. For this, the e-mail address is translated into a domain name. It uses a one-way hash function to encode the local part of the address[11] so that the user name cannot be recovered from the domain name, thus improving privacy. For instance, the address ‹alice@example.com› would be translated into

```
2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db._smimecert.exam
ple.com
```

The SMIMEA records stored under the resulting domain name have the exact same resource record format as TLSA and provide the exact same information. A certificate used when sending secure e-mail message from that address must verify with using the information given in the records.

7.4 Options for LIGHTest

In LIGHTest, certificates will appear in three different places:

- as part of an electronic transaction whose trustworthiness needs to be verified,
- as part of secure network communication, and
- as part of signatures for trust-related information.

In each of these cases, the certificates are used for verifying data and LIGHTest needs to provide a way to verify in turn whether the certificates are indeed authorized to be used for this data.

In principle, DANE provides a solution for exactly this problem using DNS. The TLSA mechanism has been designed specifically for the second appearance if TLS is used as the transport protocol for secure network communication. LIGHTest only needs to specify that such records must be present and all certificates must validate according to the rules prescribed by the TLSA records.

For the first appearance as part of an electronic transaction, there is mechanism yet. The record data of either TLSA or SMIME records can be used to deliver the information necessary

[11] That is, everything before the @ sign.

for verification – as they are identical, either can be chosen purely on taste. If they are to be used, a domain name for where these records will be placed needs to be specified and standardized as part of the LIGHTest project. Similarly, information for verification of certificates used with trust-related information can be stored in DANE resource records under a yet to be specified domain name.

8. Indicating Resource Locations

One of the conclusions in chapter 5 was that the DNS isn't suited for data sets that consist of large individual entries. The data published as part of the LIGHTest infrastructure for trust discovery and verification, trust translation, and trust delegation is exactly such data. This chapter will explore the suggestion given in chapter 5 to employ other protocols for access to the data and only store pointers in DNS. It will start by introducing URIs as the preferred format for these pointers and will then outline two existing solutions for placing such URIs in DNS, concluding with a recommendation for the LIGHTest architecture.

8.1 Uniform Resource Identifiers

Pointers to other resources are an intrinsic part of hypertext systems where references to other resources are included in the text and can be followed interactively. The format of these pointers used in what came to be known as the World Wide Web eventually developed into the standard format used by many Internet protocols. It has been given the name *Uniform Resource Identifiers* or URIs. [23]

As the name suggests, a URI is first an identifier for a resource, a term used very generically. This may be something that can be retrieved and displayed by a hypertext system, it may be a mailbox for electronic communication, or even a real-world object that cannot be accessed via any electronic means at all. In order to categorize resources, a URI always starts with a *scheme*. The remainder of the URI can only be interpreted in the context of this scheme. It also defines which operations are meaningful for this particular resource.

As is appropriate for a hypertext system, a large number of schemes allow the location and retrieval of a document over the network. Often, the scheme defines the transport protocol to use for retrieval and the remainder of the URI contains parameters used by that protocol. For instance, in the URI

```
https://www.example.com/index.html
```

the scheme 'https' states that the transport protocol HTTP with TLS encryption is to be used, that the domain name of the server is ‹www.example.com›, and that this server should be asked for a document '/index.html'.

This kind of URI is ideal as a pointer to a resource to be stored in a resource record in DNS. Once a client has retrieved the record, it has all the information necessary in order to proceed with retrieving the information. A great advantage of using URIs is that the architecture is future proof: If a new transport protocol comes along and proves more adequate, it merely needs to define an associated URI scheme in order to be used, requiring no changes to the discovery architecture at all.

A drawback is that a URI of a scheme that does not allow direct retrieval could be stored in such a record, too. For instance, it is possible to encode International Standard Book Numbers (ISBNs) as URIs. Naturally, such a URI is of limited use to an automatic trust validator, which will have to treat a record with such a URI as inexistent.

8.2 NAPTR and DDDS

Even though a URI containing an ISBN number doesn't (necessarily) represent a network accessible object, it would still be useful for a hypertext system to be able to retrieve and show some information for the book represented by that number. RFC 2168 [24] explores how to make this possible using the DNS. It does so by defining a new resource record type called NAPTR, short for Naming Authority Pointer, that is repeatedly applied to an identifier until eventually enough information has been gathered to retrieve a resource.

Subsequent work [25] has generalized the idea into a more general framework called the Dynamic Delegation Discovery System, DDDS. Instead of only rewriting URIs with rules stored in DNS, it allows transforming application-specific string by recursively applying rewrite rules stored in some database. DNS is one of these databases – it has remained the only one defined so far – and URI resolution is but one application.

Another application is ENUM [26], a system for translating telephone numbers into various kinds of URIs, such as URIs for the Session Initiation Protocol (SIP) that can be used to route a phone call via the Internet. Since the purpose of ENUM is somewhat similar to what will likely be required for LIGHTest – simple translation of a string into a URI –, the following introduces the function of DDDS using it as an example.

Each application starts out with a string it wishes to transform with DDDS. In ENUM, this string is a telephone number in international format starting with the international country code. These numbers are also known as E.164 numbers as they are agreed upon based in ITU-T regulation E.164. [27] An example for such a number is +353209108462[12]. This string needs to be converted into a key that can be used to query the database for an initial set of rewrite rules. This is called the *first well known rule* since it provides the start of the process.

Since ENUM only uses the DNS database and the key for DNS is a domain name, the telephone number needs to be converted into a domain name. This happens by starting with the domain name ‹e164.arpa› and then taking each of the digits of the number (disregarding the leading plus) and prefixing it as a new label to the domain name. This will make the phone number appear backwards and with dots between each digit: ‹2.6.4.8.0.1.9.0.2.3.5.3.e164.arpa›. Because telephone numbers are hierarchical identifiers like domain names, this approach allows to arrange DNS zone splits along the lines of ownership over telephone number blocks.

The resulting domain name is now queried for NAPTR records. In the example, the answer could contain three such records and their data portion could be this:

```
; order pref flags services           substitute
  10    10   "u"   "E2U+sip"          "!^\\+(.*)$!sip:\\1@example.com!"
  10    11   "u"   "E2U+email:mailto" "!^.*$!mailto:info@example.com!"
```

[12] When printing phone numbers for human consumption, spaces, hyphens, or other symbols are often added to make reading easier. The canonical form of an E.164 number removes all these symbols but retains the leading plus.

```
10    12    "u"    "E2U+ifax:mailto"   "!^.*$!mailto:fax@example.com!"
```

The record data is rather complex, consisting of five fields. For orientation, the first line adds a comment showing the names of each of these fields.

The output of the database query is an ordered list of rules. Since DNS doesn't have an implied order of the resource records, the first field, *order,* is used to allow this ordering. All records with the same order value are part of the same rule, smaller order values come first. Rules are visited one after another in order, the first applicable rule is used. In the example, the order is 10 for each record, so there is only one rule with three entries.

Each entry describes one possible substitution of the original string. The *services* field supplies a description of the result of the rewrite. It is an application specific string. The prefix 'E2U+' indicates rules for ENUM (short for 'E.164 to URI'), the remainder describes the service within ENUM. In the first element 'sip' indicates voice calls using the SIP protocol. Thus, this entry is intended to translate the telephone number into a SIP URI. In the second entry, 'email:mailto' declares it to be for electronic mail with the target URI using the 'mailto' scheme used for regular Internet electronic mail. The last entry also uses a 'mailto' target URI but data sent should conform to the 'facsimile using internet mail' (IFAX, [28]) service which provides a way to send a fax via e-mail.

The three entries are ordered within the rule via the *preference* value (abbreviated as 'pref' above). Through this ordering, the operator can express which services they would prefer to be used. Here, they would prefer someone to call, if that isn't an option, sent an email, and if that isn't an option either, sent a fax via e-mail. If none of these are an option, the rule cannot be used and processing happens with the next rule, if there is any left, or fails.

If one of the services is acceptable, the original string is transformed into an output value using the last field, the *substituting expression.* This kind of expression is inspired by how the Unix tool 'sed' performs text substitution. It consists of two parts: a regular expression that is applied to the original string and resulting in a number of matches, and a replacement expression that constructs the output from these matches.

An introduction to regular expressions and replacement expression is well beyond the scope of this document. Books have been written on the matter, to which the interested reader is referred [29]. In the first entry above, the regular expression produces a match for the telephone number without the leading plus and the replacement expression takes this match as the local part of a SIP URI in the ‹example.com› domain. The other two entries simply match the whole telephone number but produce fixed e-mail addresses independent from the input.

There's but one field left: *flags.* Its purpose is to dictate what to do with the output string. The values, too, are application defined, but there are two general possibilities. Either the process is finished and the output of the replacement expression is considered the final, or *terminal,* output, or the output is used as the database key (i.e., domain name for the DNS database) for yet another round of queries and rule application. This allows a multi-step approach for translating strings where the database entries for each step are owned by different entities.

In ENUM, there is no need for this recursive processing. Instead the flag will always be 'u' declaring that the output is a URI that is the final output of the ENUM lookup.

There are a number of additional details in the specification making DDDS a rather complex system. Which may be the reason why it hasn't found widespread use in other systems. Even ENUM hasn't found the hoped for adoption, although it can be debated whether technical or political complexities are the main culprit. On the technical side, an issue that has often been mentioned is the use of regular expression and the difficulty to both craft them and understand existing ones.

8.3 The URI Resource Record

A look through the list of defined resource record types will reveal an entry for a URI resource record type defined in RFC 7553 [30]. The record data of this type is much simpler than that of the NAPTR record. It only contains a priority, a weight, and a target URI. The priority is similar to NAPTR's order field: if there is more than one record, the one with the smallest priority is to be considered first with other records only there as a backup if the operation with the result from the first one fails. The weight fields allows load balancing. If multiple records with the same priority exist, the records are weighted by this field and then one is chosen randomly.

The actual purpose of the URI record is similar to that of the SRV record introduced briefly in section 5.10 in that allows to discover who will provide a certain network service for a domain. Like it, underscore prefix labels are used to state the service and transport protocol in question. The difference is that where the SRV record shows the host names of the servers, the URI record allows a redirection to a different URI. For instance, when asking for the HTTP service for ‹example.com›, this could be the result:

```
;; ANSWER SECTION:
_http._tcp.example.com.  86400  IN  URI  (
    10 10 https://www.example.com/about/
)
```

Since there is only record, priority and weight, both 10 here, don't really matter. The URI given is to be used as the base URI for any request for HTTP service in ‹example.com›.

This isn't really what the LIGHTest project needs. However, this particular use is tied to those specific prefix labels. The record type itself, which is quite convenient, can easily be used with a different prefix label as part of a new specification for a revised usage.

8.4 Options for LIGHTest

The two options, NAPTR and URI, both provide a way to convert a domain name into a URI. While the URI method has a significantly simpler format and procedure, NAPTR has the advantage of providing additional input besides the domain name. This could for instance be important to fulfill the privacy requirements of the project. As an example, in the case of trust publications, the extra input can be the trust issuer to be checked for membership in a trust scheme. Using this input, a substitution expression can include the issuer name into the produced URI which can, in turn, be used by the HTTP server of the trust publication

authority to generate a trust list that only contains that issuer and does not reveal the other members.

Both methods require a certain amount of standardization work. For the NAPTR route, a new DDDS application would need to be defined. For the URI option, a specification would need to be created detailing the new use of the resource record and registering the new prefix labels. Both would require liaising with IETF as the accepted standards body for DNS.

9. DNS Software

This chapter gives a market overview of existing DNS software. Thereby, it is distinguished between DNS server software (see section 9.1), DNS update libraries (see section 9.2), and verifying resolver libraries (see section 9.3). Due to the fact that there are numerous software products available, certain criteria were defined which are relevant for the LIGHTest infrastructure. These are

- Supported Platforms
- Supported Languages
- License
- Security Extensions

These criteria were applied for the market study. In section 9.4, DNS software and library options for the LIGHTest infrastructure are presented.

9.1 Server Software

The results of the market study for DNS server software are summarized in alphabetic order in Table 1. In addition, a short description of the software products is given in alphabetic order in the following. The information are taken from the corresponding websites listed in Table 1 and the links provided in these websites.

BIND (Berkeley Internet Name Domain) is the most widely used Name Server and is de facto the standard DNS server. It enables publishing DNS information on the Internet, as well as resolving DNS queries. It is a free software product, supports DNSSEC, and runs on most Unix and Linux and some Windows platforms.

Dnsmasq is a lightweight product, which provides DNS services for small networks. It includes a local DNS server for the network, with forwarding of all query types to upstream recursive DNS servers and cacheing of resource records. Modern internet standards such as IPv6, DNSKEY and DNSSEC are supported.

DNRD (Domain Name Relay Daemon) is a caching, forwarding DNS proxy server. It is most useful on vpn or dialup firewalls, but it can also be used as a DNS cache for minor networks.

gdnsd is an authoritative-only DNS server. It does geographic balancing, redirection, weighting, and service-state-conscious failover at the DNS layer. Its focus is on high performance and low latency service.

Knot DNS is a high-performance authoritative-only DNS server. It scales well on symmetric multiprocessing systems and enables nonstop operations. DNSSEC with NSCEC and NSEC2 is supported.

MaraDNS is a small, open-source DNS server. It is lightweight and easy to set up and cross-platform applicable.

The **Microsoft Windows Server 2012 R2** provides several enhancements in the DNS Server functionality and the software supports authoritative, recursive and hybrid mode. This includes enhanced DNS logging and diagnostics, full DNSSEC support, and dynamic DNS forwarders.

Nominum provides a commercial authoritative (**Vantio AuthServe**) and a cache server (**Vantio CacheServe**). Vantio AuthServe uses dual master servers, and an automated DNSSEC lifecycle management. Vantio CacheServe provides effective and efficient in-browser communications using their N2 DNS-based platform and purpose-built application suite.

NSD (Name Server Daemon) from NLNET is an authoritative name server. It is a high performance, RCF compliant, simple and open source name server. The latest current stable release is NSD 4.1.16. DNSSEC is supported.

NxFilter is basically a forwarding DNS server with filtering and caching ability. It can be also used as an authoritative DNS server. NxFilter supports dynamic DNS service.

OpenDNSSEC is a policy-based zone signer to automate keeping track of DNSSEC keys and the signing of zones. OpenDNSSEC takes in unsigned zones, adds digital signatures and other records for DNSSEC. It secures zone data just before it is published in an authoritative name server. OpenDNSSEC is maintained by NLNET.

pdnsd is a caching DNS server with permanent caching to hard disk for long term retention. It is designed to cope with unreachable or down DNS servers. DNSSEC and EDNS are supported.

Posadis is a powerful authoritative and caching DNS server which runs on many operating systems. In addition, a graphical editor for DNS master files, a graphical DNS query tool as well as library for developing client and server applications are provided.

PowerDNS is an open source authoritative and recursive DNS server, which supports DNSSEC. In addition, debugging tools and an API to provision zones and recursive server are provided.

Secure64 provides a commercial authoritative (**Secure64 DNS Authority**) and a cache server (**Secure64 DNS Cache**). Secure64 DNS Authority is a DNS authoritative server that has been designed from the ground up with a secure architecture. No BIND code is shared. Secure64 DNS Cache is a scalable, secure caching DNS server, which is also not based on BIND and it can be configured to validate DNSSEC signed answers.

Simple DNS Plus is a commercial authoritative and recursive DNS server software product for Windows.

Unbound from NLNET is a recursive, and caching DNS resolver. It is designed in a modular approach which easily enables DNSSEC validation and stub-resolvers.

YADIFA (Yet Another DNS Implementation For All) is an open-source authoritative Name Server with DNSSEC capabilities. It runs on multiple platforms. It has a simple configuration syntax and is very efficient in terms of memory and speed.

Table 1: Market overview DNS Server Software

Name	Server Type	Platforms	Language	License	Security Extensions	Link
BIND	Authoritative, Recursive	Linux, MacOS, Windows	C, C++, Python	MPL	DNSSEC	https://www.isc.org/downloads/bind/
Dnsmasq	Authoritative	Linux, MacOS	C	GPL	DNSSEC, DNSKEY	http://www.thekelleys.org.uk/dnsmasq/doc.html
DNRD	Recursive	Linux	C	GPL		http://dnrd.sourceforge.net/
gdnsd	Authoritative	Linux, MacOS	C	GPL		http://gdnsd.org/
Knot DNS	Authoritative	Linux, MacOS	C	GPL	DNSSEC, NSEC3	https://www.knot-dns.cz/
MaraDNS	Authoritative, Recursive	Linux, MacOS	C	BSD		http://maradns.samiam.org/
Microsoft DNS	Authoritative, Recursive	Windows		Commercial	DNSSEC, NSEC3	https://technet.microsoft.com/de-de/library/cc730921(v=ws.11).aspx
Nominum Vantio CacheServe	Recursive	Linux		Commercial	DNSSEC	http://www.nominum.com/product/caching-dns/
Nominum Vantio AuthServe	Authoritative	Linux		Commercial	DNSSEC	http://www.nominum.com/product/vantio-authserve/

NSD	Authoritative	Linux, MacOS	C	BSD	DNSSEC	https://www.nlnetlabs.nl/projects/nsd/
NxFilter	Recursive	Windows		Individual		http://nxfilter.org/p3/
OpenDNSSEC	Zone Signer	Linux, MacOS, Windows	C,C++	BSD	DNSSEC	https://www.opendnssec.org/
pdnsd	Recursive	Linux, MacOS	C	GPL	DNSSEC,	http://members.home.nl/p.a.rombouts/pdnsd
Posadis	Authoritative, Recursive	Linux, MacOS, Windows	C++	GPL		http://posadis.sourceforge.net/
PowerDNS	Authoritative, Recursive	Linux		GPL	DNSSEC	https://www.powerdns.com/index.html
Secure64 DNS Authority	Authoritative	Linux		Commercial	DNSSEC	https://secure64.com/dns-products/secure64-authoritative-dns/
Secure64 DNS Cache	Recursive	Linux		Commercial	DNSSEC	https://secure64.com/dns-products/dns_caching_server/
Simple DNS Plus	Authoritative, Recursive	Windows		Commercial	DNSSEC	http://simpledns.com/
Unbound	Recursive	Linux, MacOS, Windows	C	BSD	DNSSEC	https://unbound.net/
YADIFA	Authoritative	Linux, MacOS	C	BSD	DNSSEC, NSEC3	http://www.yadifa.eu/

9.2 DNS update libraries

The results of the market study for DNS update libraries are summarized in alphabetic order in Table 2. In addition, a short description of the products is given in alphabetic order in the following. The information are taken from the corresponding websites listed in Table 2 and the links provided in these websites.

DDClient is Perl client which is used to update dynamic DNS entries. The client runs on Unix, Linux and Max OS X systems.

dDNS Broker (formerly IP Monitor) is a dynamic DNS update client for Mac OS X. It works with more than 40 dynamic DNS service providers.

DirectUpdate is a client updater that runs automatically and transparently in the background. It is designed for Windows systems and supports many different dynamic DNS providers.

DynSite is an automatic IP updater developed for Windows systems. Dynamic DNS services are supported.

ez-ipupdate is a dynamic DNS client, which supports several dynamic DNS protocols and which includes daemon support.

FreeDNS Update from Afraid.org is a free client software for dynamic DNS, static DNS subdomain and domain hosting. It is developed for Windows systems using the Microsoft.NET Framework 4 technology.

inadyn is a dynamic DNS clients from inatech**.** There are several forks from the original inadyn, e.g **inadyn-mt**, which run on multiple platforms.

Java Dynamic DNS Client is a dynamic DNS client, which runs as application or daemon. It is platform independent and requires only that java is enabled.

NC DNS Updater for Mac OS X automatically updates IPs for namecheap's dynamic DNS Service. NC DNS Updater runs as a daemon and does not require a logged in user to function.

Table 2: Market overview DNS update libraries

Name	Platforms (Linux, MacOS, Windows)	Language	PublicSource Code	Link
DDClient	Unix-like	Perl	GPL	https://sourceforge.net/p/ddclient/wiki/Home/
dDNS Broker	Mac OS X		Commercial	https://itunes.apple.com/de/app/ddns-broker-formerly-ip-monitor/id1050307950?mt=12&ign-mpt=uo%3D4
Direct Update	Windows		Commercial	http://www.directupdate.net/
DynSite for Windows	Windows	multilinguale	Commercial	http://noeld.com/dynsite.asp
ez-ipupdate	Linux	C	GPL	https://sourceforge.net/projects/ez-ipupdate/
FreeDNS Update	Windows	C#	GPL	*http://www.nesociety.org/community-services/afraid-updater-service-4/*
Inadyn	Linux, Windows, Mac OS X, OpenBSD	C	GPL	https://sourceforge.net/projects/inadyn-mt/files/
IvmaiDNS	Linux, Windows, Solaris	Java	GPL	http://ivmaidns.sourceforge.net/
Java Dynamic DNS Client	Cross-platform	Java	LGPL	http://rzodyndns.sourceforge.net/

NC DNS Updater	Mac OS X		Freeware	https://github.com/aidanamavi/nc-dns-updater

9.3 Verifying resolver libraries

The results of the market study for verifying resolver libraries are summarized in alphabetic order in Table 3. In addition, a short description of the products is given in alphabetic order in the following. The information are taken from the corresponding websites listed in Table 2Table 3 and the links provided in these websites.

dnsjava is an implementation of DNS in Java for DNS queries but also for zone transfers and dynamic updates. All record types are supported, including DNSSEC.

getdns is an asynchronous DNS API written in C. It makes all types of DNS (inclusive DNSSEC) information easily available and enables end-to-end trust in the DNS architecture.

IvmaiDNS is a java DNS client implementation, which consists of a java library and utilities for looking up internet domain names.

Table 3: Market overview DNS resolver libraries

Name	Platforms (Linux, MacOS, Windows)	Language	PublicSource Code	Link
dnsjava	Cross-Platform	Java	BSD	http://www.xbill.org/dnsjava/
getDNS	Linux, Windows, Mac OS X	C	BSD-new	https://getdnsapi.net/
IvmaiDNS	Linux, Windows, Solaris	Java	GPL	http://ivmaidns.sourceforge.net/

9.4 Options for LIGHTest

The software overview enables a selection of suitable products for building the LIGHTest reference infrastructure and pilots. According to the objectives of LIGHTest, main criteria for the selection are the following:

- Multi-Platform use,
- BSD or Apache license,
- support of DNSEC extensions.

As both the provisioning software and the trust verifier are intended to be written in Java, the client libraries need to be for that language.
According to the criteria, the following products are suited to be implemented in the LIGHTest infrastructure by the time of writing. The final selection of the implemented components will be made in the corresponding tasks of the work packages 3, 4, 5, and 6.

For the server software used in the reference infrastructure, the combination of NSD, OpenDNSSEC, and Unbound is well suited. As all of these are products maintained by NLNET, experience with deployment and configuration is readily available as well as any additional support should it become necessary.

For the update libraries either lvmaiDNS or the Java Dynamic DNS Client can be used, depending on the eventual design of the provisioning software.

For the verifying resolver libraries, dnsjava is well suited.

10. Conclusion and Outlook

During its look into the Domain Name System, the deliverable has shown that the system is well suited as the foundation for a trust infrastructure for the Internet, provided the design of the infrastructure takes into account some limitations deriving from the design of the DNS itself.

In particular, the deliverable has shown why the DNS should not be used to publish trust-related information itself but rather use the system only to publish the location of such information. The public nature of the DNS and its lack of any mechanism to limit access to information to authenticated users needs to be considered where sensitive information is involved.

The mechanism to verify the DNS data provided by DNSSEC introduce a high amount of confidence into the authenticity of that data as needed when using this information to verify whether an electronic transaction can or should be trusted.

The deliverable looked at a number of concrete options to include the data to be stored in the DNS as part of LIGHTest. While it found that likely none of the existing extensions can be used as they are, it is likely that they only need small procedural modifications.

The task 3.2, 4.2, and 5.2 will more closely look into how these options apply to the LIGHTest architecture and which concrete modifications will be necessary.

11. References

[1] P.V. Mockapetris. *Domain names: Concepts and facilities.* RFC 882, Internet Engineering Task Force, November 1983.

[2] P.V. Mockapetris. *Domain names: Implementation specification.* RFC 883, Internet Engineering Task Force, November 1983.

[3] P.V. Mockapetris. *Domain names – concepts and facilities.* RFC 1034, Internet Engineering Task Force, November 1987.

[4] P.V. Mockapetris. *Domain names – implementation and specification.* RFC 1035, Internet Engineering Task Force, November 1987.

[5] J. Postel. *Domain Name System Structure and Delegation.* RFC 1591, Internet Engineering Task Force, March 1994.

[6] A. Gustafssonn. *Handling of Unknown Resource Record (RR) Types.* RFC 3597, Internet Engineering Task Force, September 2003.

[7] P. Vixie, S. Thomson, Y. Rekhter, J. Bound. *Dynamic Updates in the Domain Name System (DNS UPDATE).* RFC 2136, Internet Engineering Task Force, April 1997.

[8] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, D. Wessels. *DNS Transport over TCP – Implementation Requirements.* RFC 7766, Internet Engineering Task Force, March 2016.

[9] D. Atkins and R. Austein. *Threat Analysis of the Domain Name System (DNS)*. RFC 3833, Internet Engineering Task Force, August 2004.

[10] ICANN Security and Stability Advisory Committee (SSAC). *Domain Name Hijacking: Incidents, Threats, Risks, and Remedial Actions.* ICANN, July 2005. https://archive.icann.org/en/announcements/hijacking-report-12jul05.pdf

[11] National Cybersecurity and Communications Integrity Center. *DDoS Quick Guide.* January 2014. https://www.us-cert.gov/sites/default/files/publications/DDoS%20Quick%20Guide.pdf

[12] R. Gieben, W. Mekking. *Authenticated Denial of Existence in the DNS*. RFC 7129, Internet Engineering Task Force, February 2014.

[13] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, P. Hoffman. *Specification for DNS over Transport Layer Security (TLS).* RFC 8757, Internet Engineering Task Force, May 2016.

[14] T. Reddy, D. Wing. *DNS over Datagram Transport Layer Security (DTLS).* RFC 8094, Internet Engineering Task Force, February 2017.

[15] *Mozilla CA Certificate Store.* https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/.

[16] D. Eastlake, O. Gudmundsson. *Storing Certificates in the Domain Name System (DNS).* RFC 2538, Internet Engineering Task Force, March 1999. Obsoleted and replaced by: S. Josefsson. *Storing Certificates in the Domain Name System (DNS)*. RFC 4398, Internet Engineering Task Force, March 2006.

[17] J. Schlyter, W. Griffin. *Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints.* RFC 4255, Internet Engineering Task Force, January 2006.

[18] M. Richardson. *A Method for Storing IPsec Keying Material in DNS.* RFC 4025, Internet Engineering Task Force, February 2005.

[19] P. Hoffman, J. Schlyter. *The DNS-Based Authentication of Name Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA.* RFC 6698, Internet Engineering Task Force, August 2012.

[20] O. Gudmundsson. *Adding Acronyms to Simplify Conversations about DNS-based Authentication of Named Entities (DANE).* RFC 7218, Internet Engineering Task Force, April 2014.

[21] B. Ramsdell, S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification.* RFC 5751, Internet Engineering Task Force, January 2010.

[22] P. Hoffman, J. Schlyter. *Using Secure DNS to Associate Certificates with Domain Names for S/MIME.* RFC 8162, Internet Engineering Task Force, May 2017.

[23] T. Berners-Lee, R. Fielding, L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986, Internet Engineering Task Force, January 2005.

[24] R. Daniel, M. Mealling. *Resolution of Uniform Resource Identifiers using the Domain Name System.* RFC 2168, Internet Engineering Task Force, June 1997.

[25] M. Mealling. *Dynamic Delegation Discovery System (DDDS) Part One: The Comprehensive DDDS.* RFC 3401, Internet Engineering Task Force, October 2002.

[26] S. Bradner, L. Conroy, K. Fujiwara. *The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM).* RFC 6116, Internet Engineering Task Force, March 2011.

[27] International Telecommunication Union. *The International Public Telecommunication Numbering Plan.* Recommendation ITU-T E.164, Edition 6.0, November 2010.

[28] R. Buckley, D. Venable, L. McIntyre, G. Parsons, J. Rafferty. *File Format for Internet Fax.* RFC 3949, Internet Engineering Task Force, February 2005.

[29] Jeffrey Friedl. *Mastering Regular Expressions.* 3rd edition, O'Reilly Media, August 2006.

[30] P. Falstrom, O. Kolkman. *The Uniform Resource Identifier (URI) DNS Resource Record.* RFC 7553, Internet Engineering Task Force, June 2015.

[31] Robert Charetter. *DigiNotar Certificate Authority Breach Crashes e-Government in the Netherlands.* IEEE Spectrum Online, September 2011. http://spectrum.ieee.org/riskfactor/telecom/security/diginotar-certificate-authority-breach-crashes-egovernment-in-the-netherlands

12. Project Description

LIGHTest project to build a global trust infrastructure that enables electronic transactions in a wide variety of applications

An ever increasing number of transactions are conducted virtually over the Internet. How can you be sure that the person making the transaction is who they say they are? The EU-funded project LIGHTest addresses this issue by creating a global trust infrastructure. It will provide a solution that allows one to distinguish legitimate identities from frauds. This is key in being able to bring an efficiency of electronic transactions to a wide application field ranging from simple verification of electronic signatures, over eProcurement, eJustice, eHealth, and law enforcement, up to the verification of trust in sensors and devices in the Internet of Things.

Traditionally, we often knew our business partners personally, which meant that impersonation and fraud were uncommon. Whether regarding the single European market place or on a Global scale, there is an increasing amount of electronic transactions that are becoming a part of peoples everyday lives, where decisions on establishing who is on the other end of the transaction is important. Clearly, it is necessary to have assistance from authorities to certify trustworthy electronic identities. This has already been done. For example, the EC and Member States have legally binding electronic signatures. But how can we query such authorities in a secure manner? With the current lack of a worldwide standard for publishing and querying trust information, this would be a prohibitively complex leading to verifiers having to deal with a high number of formats and protocols.

The EU-funded LIGHTest project attempts to solve this problem by building a global trust infrastructure where arbitrary authorities can publish their trust information. Setting up a global infrastructure is an ambitious objective; however, given the already existing infrastructure, organization, governance and security standards of the Internet Domain Name System, it is with confidence that this is possible. The EC and Member States can use this to publish lists of qualified trust services, as business registrars and authorities can in health, law enforcement and justice. In the private sector, this can be used to establish trust in inter-banking, international trade, shipping, business reputation and credit rating. Companies, administrations, and citizens can then use LIGHTest open source software to easily query this trust information to verify trust in simple signed documents or multi-faceted complex transactions.

The three-year LIGHTest project starts on September 1st and has an estimated cost of almost 9 Million Euros. It is partially funded by the European Union's Horizon 2020 research and innovation programme under G.A. No. 700321. The LIGHTest consortium consists of 14 partners from 9 European countries and is coordinated by Fraunhofer-Gesellschaft. To reach out beyond Europe, LIGHTest attempts to build up a global community based on international standards and open source software.

The partners are ATOS (ES), Time Lex (BE), Technische Universität Graz (AT),EEMA (BE), G&D (DE), Danmarks tekniske Universitet (DK), TUBITAK (TR), Universität Stuttgart (DE), Open
Identity Exchange (GB), NLNet Labs (NL), CORREOS (ES), IBM Danmark (DK) and Globalsign (FI). The Fraunhofer IAO provides the vision and architecture for the project and is responsible for both, its management and the technical coordination.
The Fraunhofer IAO provides the vision and architecture for the project and is responsible for both, its management and the technical coordination.