

Bachelor's Thesis (UAS)

Information Technology

System Engineering

2012

Ekaterina Andreeva

DNSSEC Key Management and Exchange



TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | System Engineering

07.12.2012 | 96 pages

Instructors: Tuomas Merilä, Ossi Väänänen

Ekaterina Andreeva

DNSSEC Key Management and Exchange

The Domain Name System (DNS) is a system that translates human-readable domain names into numeric IP addresses and vice versa. Public DNS server infrastructure forms a distributed database that contains all the domain names and corresponding IP addresses.

Legitimacy of DNS has been essentially based on the trust. When a user accesses a network service e.g. a website using domain name, he or she trusts that the name is correctly translated in to IP address of the service he/she intended to access. Hackers exploiting the vulnerabilities of DNS protocol can gain the access to confidential information, steal passwords and private data.

In order to authenticate DNS responses and make it more secure, a new extension to DNS architecture was introduced called DNSSEC (Domain Name Security Extensions).

DNSSEC is a set of extensions for DNS, which includes functionality for the following steps:

- Authentication of DNS data (check the reliability of the site)
- Checking data integrity
- Checking denial of existence (e.g. dummy DNS server can respond to user that the domain does not exist, when in fact it does).

By analyzing the available technical documentation and RFC (Request for Comments) documents, this thesis aims to outline best practices and guidelines for DNSSEC key management and exchange theory. The results of this work can be used as a baseline for example when implementing the DNSSEC key management functionality to DNS management software.

KEYWORDS:

DNS, DNSSEC, KSK, ZSK, key rollover, management systems.

Table of Contents

1. Introduction	1
2. Domain Name System (DNS)	3
2.1 <i>DNS process</i>	6
2.1.1 DNS zones and records.....	6
2.1.2 DNS name resolution process.	10
2.2 <i>DNSSEC</i>	13
2.2.1 The DNSSEC Resource Records and zone file format.	16
2.2.2 DNSSEC process	20
2.3 <i>DNSSEC key management</i>	23
2.3.1 Algorithm choice.....	23
2.3.2 Key length and lifetime	24
2.3.3 Key rollover	26
2.3.4 ZSK (Zone Signing Key) Rollover.....	27
2.3.5 KSK (Key Signing Key) Rollover	29
2.3.6 Emergency rollover	30
2.3.7 Key Storage and Security.....	32
2.4 <i>DNSSEC Key Exchange</i>	33
2.4.1 Publishing the Public Key.....	34
2.5 <i>Management systems and tools</i>	35
2.6 <i>Theory Summary</i>	37
2.7 <i>Requirements of KSK / DS exchange mechanism</i>	38
3. Case study of DNSSEC in Finland	40
3.1 <i>The environment description</i>	40
3.2 <i>Nixu NameSurfer Suite and its current DNSSEC functionality</i>	41
3.3 <i>Proposed functionality of the Key Exchange Mechanism</i>	43
3.4 <i>Development environment</i>	45
3.5 <i>Design of the Python script</i>	47
3.6 <i>Implementation of the script</i>	48
3.6.1 Class “WSClient”	48
3.6.2 Class “DSChecker”	52

3.6.3 Class “NSClient”	53
3.6.4 “Main” function	55
3.7 Testing	59
3.7.1 DNS queries	59
3.7.2 NameSurfer API	63
3.7.3 Web Service client	66
3.7.5 Testing the final script	77
4. Conclusion	80
REFERENCES	82
Appendix 1 “FicoraWS.py” code.....	84
Appendix 2. “NSAPI.py” code.	87
Appendix 3. “DSUploader.py” code.....	88

Figures

FIGURE 1. DNS HIERARCHICAL STRUCTURE	5
FIGURE 2. DNS NAME RESOLUTION PROCESS	11
FIGURE 3. "DNS SPOOFING" / "DNS CACHE POISONING" ATTACK TECHNIQUE.....	12
FIGURE 4. PUBLICLY AVAILABLE DNS ROOT PUBLIC KEY (TRUST ANCHOR)	15
FIGURE 5. DNS NAME RESOLUTION PROCESS WITH DNSSEC.....	21
FIGURE 6. CREATING DNSSEC KEY IN NAMESURFER SUITE.....	42
FIGURE 7. SELECTING AUTOMATIC KEY ROLLOVER METHOD AND INTERVAL IN NAMESURFER SUITE.....	43
FIGURE 8. ZONE "EXAMPLE.COM" IN NAMESURFER SUITE	64
FIGURE 9. EXPORTED ZONE'S MASTER FILE WITH KEY SIGNING KEY.....	64

1. Introduction

One of the key pillars of the Internet infrastructure is Domain Name System, the translation of numerical IP addresses to human-readable and more easily memorable domain names. Most of the time when resources are accessed on the Internet, or any other TCP/IP based network, domain names are used such as “www.google.com” or “www.turkuamk.fi”. Even when sending an email to ammattikorkeakoulu@turkuamk.fi, email clients are dealing with DNS.

To provide the translation to actual numerical IP address, which is used when network traffic is inspected at network package level, DNS servers are consulted. These servers host a figurative “phonebook” containing name-address pairs, providing the information, for example, a web browser would need to finally display Google’s homepage for the user who has entered the URL “google.com”.

However, for years DNS system has been vulnerable to address spoofing. A “man-in-the-middle” could intercept the DNS answer and redirect the communication, originally meant to be directed to online bank, to the culprit’s own servers. To bring verification and validation to DNS communication, a technology called *DNSSEC* (short from *Domain Name System Security Extension*) was introduced.

DNSSEC validation is based on a chain of trust where the DNS server higher in hierarchy is able to validate answers for domain below it. Each DNS record is associated with signature, and keys used to sign can be always validated starting from the DNS root. This system requires, though, that each domain administrative authority needs to provide the information of keys to the parent domain. This is called DNSSEC key exchange. For example, Finnish domains such as “turkuamk.fi”, “turku.fi” and “yle.fi” would need to provide details of their key signing keys to the “.fi” domain, which is Finland’s country code top-level

domain administered by Ficora (Finnish Communications Regulatory Authority). Ficora is the domain name registry for “.fi” which means it has full authority of domains under “.fi”.

This thesis is researching and implementing DNSSEC key exchange, based on best practices and using existing interfaces, to Nixu Software’s NameSurfer Suite DNS management software appliance. The planned key exchange mechanism will automate the key uploading for domains under the “.fi” country code top-level domain. Nixu Software’s DNS management products are used in many levels of DNS hierarchy: by registrars, ISPs, and individual organizations managing one or more of their own sub domains.

2. Domain Name System (DNS)

When the Internet was originally founded in 1969 in university environment, it had been mainly used for educational and research needs. Because the networks in university campuses were relatively small, users had no problems with remembering numbers as addresses of the devices and destinations. However, during the next decade with the help of new networking technologies such as TCP/IP protocol, ARPANET (name of the Internet's predecessor) started to expand rapidly.

"TCP/IP made possible to connect, or inter-network, ARPA-like networks" (<http://sixrevisions.com/resources/the-history-of-the-internet-in-a-nutshell/>) together which caused a quick expansion of the ARPANET. Along the TCP/IP protocol came the Internet Protocol (IP) addresses. Using dot-decimal notion, an IP address consists of four decimal numbers from 0 to 255, 32 bits in length, resulting to over four billion unique addresses. Now each host in this quickly expanding network could have referable unique identifier. IP addresses and IP subnetting allowed building more advanced networks and network equipment, and whilst the IP addressing scheme was completely perfect for machines, it soon became quite a burden for people. Eventually it would become nearly impossible to remember or identify the numeric address as the number of hosts in the network increased. To mitigate this, the Domain Name System was introduced.

The Domain Name System (DNS) is a hierarchical, distributed database. It stores information for mapping Internet host names to IP addresses and vice versa, mail routing information, and other data used by Internet applications. [9]

The Domain Name System is like a phonebook of the Internet. IP addresses are phone numbers used by the network infrastructure itself, while the names associated to a number are for us humans. For example, the domain name "www.turku.fi" is a reference to the actual IP address "213.138.149.250" of the server where the city of Turku's website is hosted. An average Internet user

does not ever need to know these cryptic numerical addresses in the same sense as most people do not need to remember each telephone number in their phonebook.

It is quite clear at this point that validity of this *phonebook* is essential as the services that users access using domain names are increasingly critical, such as online banking services. If a con-man puts a “Bank” sign on his front door, some people might believe it, but most likely they will notice that this is not true; this bank office is fake. But online similar spoofing is much easier to disguise and much harder to identify as it is more common for a website to change its look than for a hometown’s bank building.

To better understand how the domain name system can be spoofed a better understanding of the DNS working mechanism is required. In the following sections the DNS basics will be explained.

As it was mentioned earlier, DNS can be represented using phonebook analogy - list of the names and corresponding numbers called IP addresses. But nowadays there are hundreds of thousands of domain names and the number keeps growing rapidly which makes managing the domain names a challenge. But hierarchical structure of the DNS makes this process simple.

The domain names database is distributed because otherwise the file that contains all the names and corresponding addresses, “*phonebook*”, will be too large. The name contains few fields separated by dots “.” and read from the right. The top level domain is the right most field, next one to the left - second level domain, next one - subdomain, additional name(s) that were created inside the second level domain and used, for example, for organization units, and the leftmost field is the host name, name of the particular computer in the organization.

The hierarchical structure of the DNS is shown in the figure below.

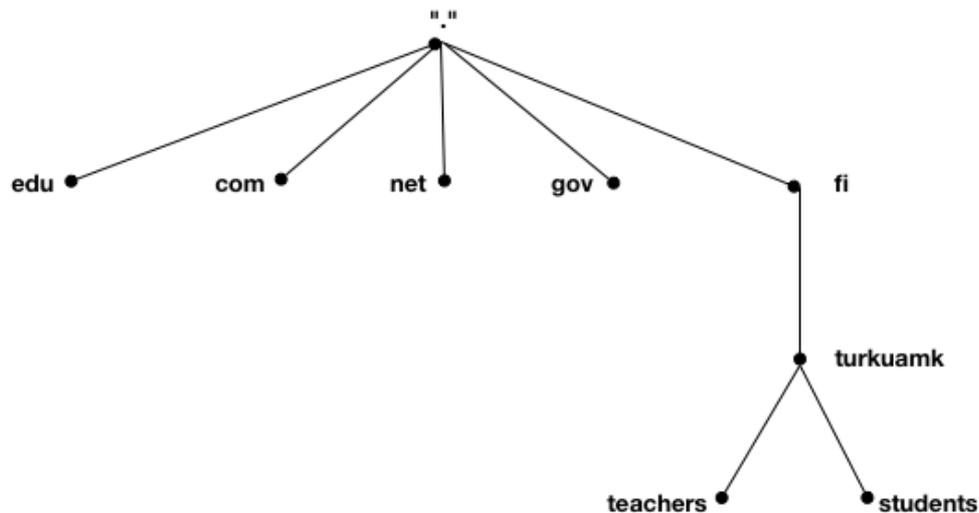


Figure 1. DNS hierarchical structure

On the top of the tree, there is the root domain, also known as “.” (“dot”). Domains one level below are called Top Level Domains (TLD). There is a number of more or less generic internet TLDs such as “.edu” for education, “.com” for commercial use, “.net” for Internet service providers “.gov” for government organizations. However, as there are not any restrictions on who can register a subdomain under these TLDs, many of the TLDs have lost their original meaning. Finally, the largest group of TLDs are formed by the country-code domains using ISO-3166-1 standardized two-letter country codes such as “.fi” for Finland, “.se” for Sweden etc.

On the next level down, there are usually placed domains reserved for private organizations, for instance, “turkuamk” as described in Figure 1. And finally in this example, the domain “turkuamk” includes two subdomains called “teachers” and “students”. Each subdomain is delegated from the parent domain above, and managed independently. By using this distributed structure, administrators do not need to take care of millions of records in a one enormous domain name database, but rather they can each just manage their own organization’s domains.

The above described DNS space is also known as “forward lookup zone” and the process of looking for an IP address associated with a name is known as

“forward resolution”. However, there is also “reverse lookup zone” and “reverse resolution” process, which is needed for searching for a name corresponding to an IP address. Reverse zone records contain numeric address on the first place and then the name of the host. Since the IP-addresses are registered together with the DNS domain names, search by IP-address can help to determine from which domain this IP-address comes. If there is no match found, then there is a chance that it could be a "fake."

2.1 DNS process

2.1.1 DNS zones and records

The basic principle of the DNS is the correspondence between the numerical IP addresses and human-readable names. However, as stated before, DNS is a distributed system and, for example, for a *com*-domain there is not one central database where all these names to IP mappings are stored. There are tens of millions of subdomains under *.com* alone and administrating all the different organizations that have subdomains under *.com* domain would be extremely inefficient. Instead, the DNS namespace is divided for administrative purposes into *zones*. For example, IP mapping of *www.nixussoftware.com* name is not defined in the zone of *.com*, but there exists an explicit authority delegation from *.com* for subdomain *nixussoftware.com*. The *Nixussoftware.com* zone is hosted on Nixu Software’s own name servers, and the organization can independently administrate the *nixussoftware.com* namespace. *Nixussoftware.com* could be even further divided into separate zones, so that each branch office could independently manage their hosts, e.g., *emea.nixussoftware.com*, *americas.nixussoftware.com* and so on.

Name server is the server that stores full information about the zone, precisely the file that contains that data. The server can be used for hosting several zones. For instance, two zones different from each other given in the earlier example, *emea.nixussoftware.com* and *americas.nixussoftware.com*, could be hosted on the one name server e.g., *ns.nixussoftware.com*. Nixu Software

happens to have third domain *howismydns.com*, completely unrelated to previous zones, but even this one could be hosted on the same name server. As long as the parent domain, in this case top-level domain *.com*, knows which name servers are authoritative for which domain, organizations can design their name server architecture quite freely.

Zone files are created according to the specific rules and guidelines; their format is described in the RFC 1035 [14] (*RFC* - Request For Comments, a document, describing methods, research or innovations applicable in the Internet-connected systems). A simple file could look like the one below:

```
$ORIGIN nixusoftware.com.
@           IN      SOA  ns.nixusoftware.com.
hostmaster.nixusoftware.com. (
                2012020219 28800 7200 604800 86400 )
                NS   ns.nixusoftware.com.
                NS   ns2.nixusoftware.com.
www          A     200.30.20.1
            MX    10 mail.nixusoftware.com.
intra       A     200.30.15.3
            MX    10 mail.nixusoftware.com.
            TXT   "Intranet web server"
emea        NS    ns1.emea.nixusoftware.com.
            NS    ns2.emea.nixusoftware.com.
americas    NS    ns1.americas.nixusoftware.com.
            NS    ns2.americas.nixusoftware.com.
ns1.emea    A     200.50.20.40
ns2.emea    A     200.50.20.41
ns1.americas A    200.50.20.15
ns2.americas A    200.50.20.16
```

This zone file is used to define the “*nixusoftware.com*” zone on the name server.

Zone files contain *directives* and *resource records*. *Directives* are needed to specify tasks for the name server and to take in use zone settings. *Resource records* define the actual parameters of the zone and assign names to the hosts inside the zone.

A zone file starts with the directive `$ORIGIN`. This directive adds a domain name to non-qualified records, like the ones that have only hostname. In the given example `$ORIGIN` is defined to be `nixusoftware.com`, which effects that the domain part can be omitted from the resource records following it. For instance, instead of defining the fully qualified domain name (FQDN) `www.nixusoftware.com.`, just the host name part `www` is sufficient for the address record.

Before the actual resource record (RR) in the zone file, there is the *class of the record*, which is in this case "IN" which stands for "Internet protocol. The *class* defines the protocol family or an instance of the protocol. Today in practice, a protocol is always a type of "IN" (other values are HS and CH both historic MIT protocols) and the whole class definition can be in most cases omitted from the RR description.

The next type of the record, *SOA (Source Of Authority)*, defines the DNS zone. `Ns.nixusoftware.com` is the name of the primary name server, which is the server storing the master zone file. `hosmaster.nixusoftware.com` is the domain administrator's email address, where the "@" sign of the email address is replaced with a dot.

The SOA record section in round braces contains the parameters which are needed for correct communication with other name servers. The first number, "2012020219" in this case, is called *Serial* and represents the version of the zone. Every time when any modification has been made in the zone file, this number is increased. When the secondary name server refreshes the data and notices that the number in its cache is smaller, it requests a new copy of the zone with recent changes. The next number, "28800" represents *Refresh* interval in seconds and tells to secondary servers how often the primary server should be checked for updates. If the primary server does not reply, *Retry* interval ("7200" seconds) specifies the time between communication attempts. The parameter *Expire* ("604800") tells to secondary servers for how long they

should serve the domain if the primary server does not respond to queries. Finally, the last one, “*Minimum*” sets the default *time-to-live (TTL)* for the records in the zone. *TTL* defines the maximum time the secondary server should keep the individual record in cache without refreshing it from the primary server.

NS records after parameters describe name servers serving the zone (NS stands for Name Server). In the given example, the names of the servers are *ns.nixusoftware.com* and *ns2.nixusoftware.com*. It is a good practice to have at least two name servers for the zone to ensure high availability of the zone data. Servers can be even split to different geographical locations to ensure low latencies globally.

The next section in the zone file is filled with the *A records*. Here comes again the “phonebook” analog, because as the record in the phonebook, *address record (A)* consists of a computer’s name and its IP address. “A” records provide human-readable names to numeric IP address translation. In the given example, the string “*www A 200.30.20.1*” means that the machine with the name “*www*” has IP address 200.30.20.1 in the network.

In the example zone, *www*, *intra*, *emea*, *americas* are all called nodes. Each node can have one or several resource records. For example, node “*intra*” has three:

```
intra          A      200.30.15.3
               MX    10 mail.nixusoftware.com.
               TXT   "Intranet web server"
```

The first is the A record, that specifies IP address of the computer; the second is an *MX (mail exchange)* record that specifies the mail server responsible for emails in the given zone; finally, there is a *TXT (text)* record that can contain comments about the node.

In the last section of the example there are again two NS records. But these records are used for the *delegation*. Like in the context of the SOA record where there were NS records defining which servers were authoritative for this particular zone (*ns.nixusoftware.com* and *ns2.nixusoftware.com*), also here it means that sub-domain “*emea.nixusoftware.com*” is managed and stored on name servers *ns1.emea.nixusoftware.com* and *ns2.emea.nixusoftware.com*. So whatever belongs to subdomains *emea* and *americas* are no longer this zone file’s or even this name server’s concern. The only exceptions are the last four lines of the zone, *A records* specifying the IP addresses for these external name servers. These records are often called *glue records*. These records exist to prevent an endless loop situation where name server replies that *ns1.emea.nixusoftware.com* is the authoritative server for zone *emea.nixusoftware.com*, and once a DNS client asks for *ns1*’s IP address, again it receives the same response from the server that *ns1.emea.nixusoftware.com* is authoritative for zone *emea.nixusoftware.com*.

There are over 20 RRs specified by the RFC and the ones mentioned in the example are some of the most common ones. A few more DNSSEC specific RRs will be introduced in the upcoming sections.

2.1.2 DNS name resolution process.

The hierarchical structure of DNS allows that any given DNS record can be resolved starting from one point, the root node “.”. The chart below describes the exact steps in the DNS name resolution process.

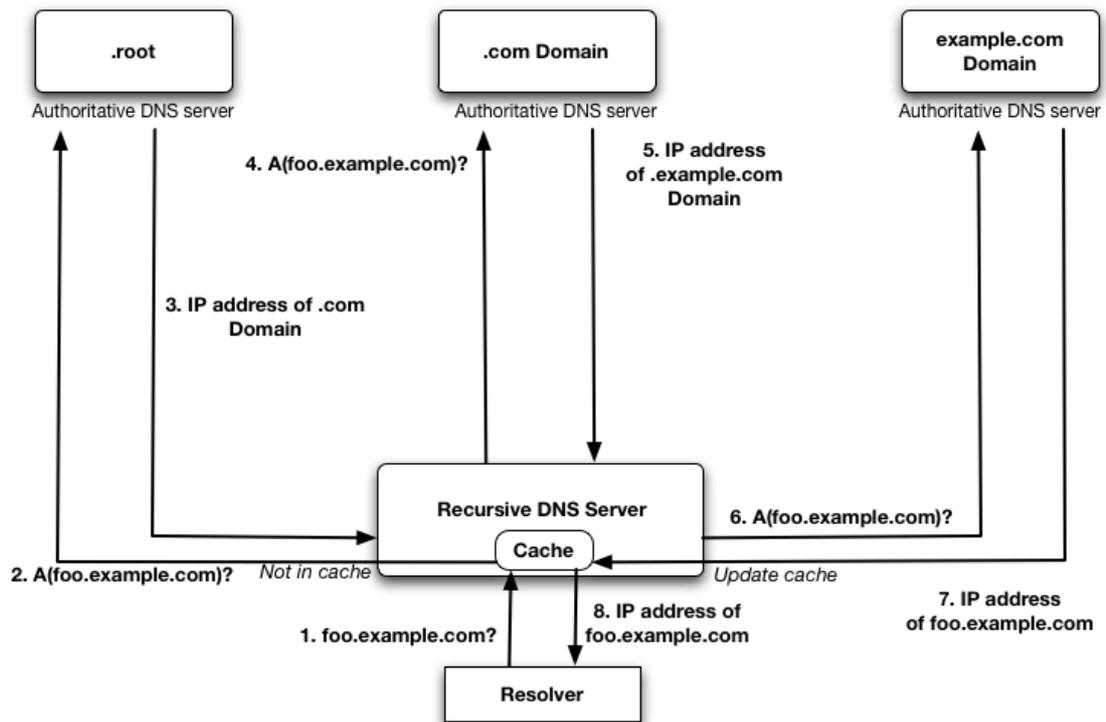


Figure 2. DNS name resolution process

In the given example, the local recursive DNS server responds to a query for “foo.example.com” domain. First, the DNS server checks its local cache and tries to find any matches stored based on previous similar queries. If there are no such entries and thus server cannot resolve the name, it will look up for another DNS server, starting from the root’s authoritative DNS server. The root server’s response contains information about the server, which is authoritative for “.com” zone. After the next iteration, the IP address of “example.com” domain’s authoritative DNS server is obtained. Finally, the “example.com” authoritative DNS server replies with the IP address of “foo.example.com” entry and the recursive server delivers the result to resolver. In addition, the recursive DNS server updates its cache with the new entry and stores it for a predefined amount of time. In the future if a client requests the same data, it will be provided with a reply from the server’s cache thus speeding up the resolving process considerably.

As a result of the described process, the user receives the web page he/she was interested in. However, there are some weak points.

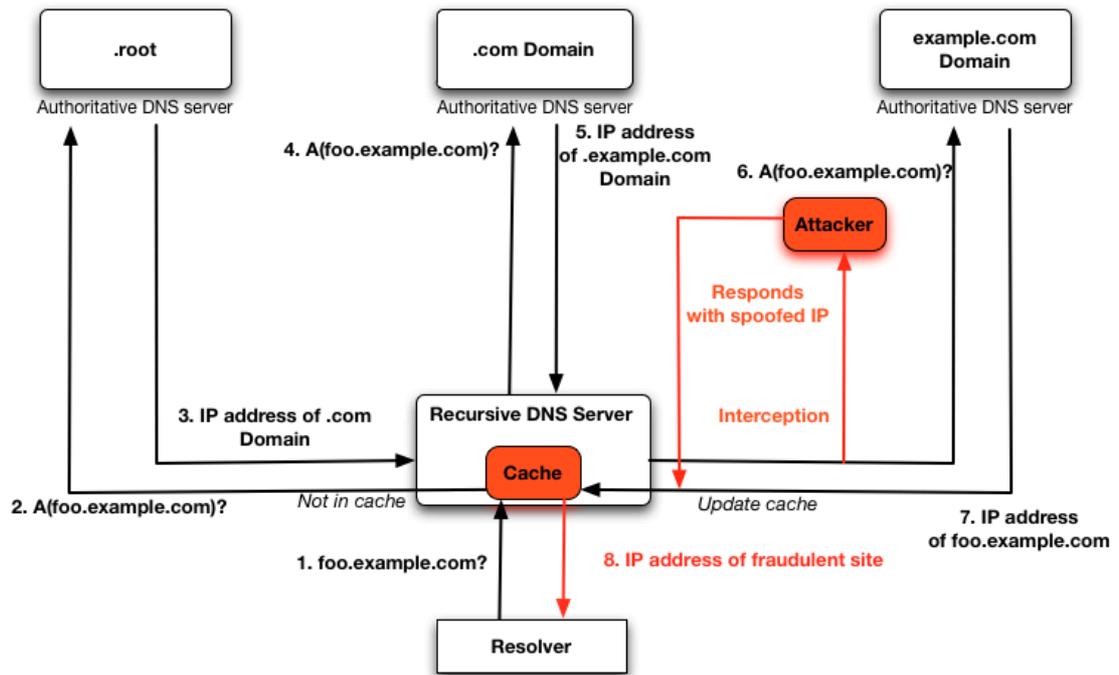


Figure 3. "DNS spoofing" / "DNS cache poisoning" attack technique

Figure above displays one of the most common hacker techniques, called “DNS Spoofing / DNS cache poisoning”. When a client is trying to obtain the address of the requested site, the attacker replaces the real one with the counterfeit. As the result, the user will be redirected to the bogus web site without noticing the difference and might leave there important information that can be stolen by malefactors. Moreover, wrong data will be stored in the recursive DNS server’s cache and each user who requested the same site would be redirected to the false one. Usually this type of attack cannot be detected immediately and it might take some time to liquidate it.

Imagine that a customer would like to visit a bank office and check his/her bank account. In real life, it is not that easy to imitate a branch office. If someone puts a “Bank” sign on the barn door, it is most likely that bank customers will notice that something is wrong. Today there are fewer reasons to go to physical bank office; every day finances can be controlled via online banking just by typing “www.bank.com” in the web browser. Next the operating system’s resolver asks from the one of the known DNS servers about the IP address of the requested

web site. But neither user nor web browser can verify that received data points to the real web site of the bank, which is located in “bank.com” domain. There is a chance that DNS-answer was falsified on one of the steps of the resolving process. The main threat in this situation is that the received IP address could point to the phisher’s server where the fake bank web site is located. This site can look exactly as the real one and even address in the browser’s address line could be the same. As a result, an innocent user without any uncertainty will provide all the needed bank login authentication information that will be used by criminals.

Even though DNS was invented in 1983, only in the 1990s Internet providers started to talk about ways to be able to validate the DNS information and describe the first edition of DNSSEC - Domain Name Security Extensions.

2.2 DNSSEC

Initially DNS had no data securing mechanism against falsified information. In order to protect the information from cache poisoning and “man-in-the-middle” attacks, DNSSEC (Domain Name System Security Extensions) was designed. The main purpose of DNSSEC is to validate the authenticity and integrity of the DNS data.

The idea of DNSSEC is that every DNS response can be verified by the client with the digital signature. DNS zone data consists of resource records (RR) and when a client requests DNS data, it means that the client is looking for appropriate RR. The purpose of DNSSEC is to assure that the requested RR really exist or not and they have not been modified during the DNS resolution process. This is achieved by using digital signing - when DNSSEC is enabled, every Resource Record is signed with the unique secret key and carries a digital signature.

DNSSEC DNS data signing relies on *asymmetric cryptographic algorithm*. This means that for encrypting and decrypting data two keys are required – a private to encrypt the message to be sent and a public to decrypt the received message. These keys form a *key pair* (two keys generated at the same time using same algorithm); data encrypted with the private key can be decrypted only with the corresponding public one. By using this authentication mechanism, a user in possession of the public key can be sure that the data he/she received is originated from the trusted source (or at least from the same source as the public key) and has not been changed during the transmission process.

A DNSSEC key can be either a *Zone Signing Key(ZSK)* used to sign all resource record sets in a zone, or a *Key Signing Key(KSK)* used only to sign the set of keys that is used to sign the zone. The reason why there are two different keys is to define a separate set of functions due to minimize the complexity of the tasks associated with the key update processes and zone re-signing.

KSK is used to sign a set of keys (i.e., a set of DNSKEY resource records) and this is the key that is passed to the parent zone for authenticated delegation. It can also be used as a trusted anchor for establishing a trusted chain for signature verification.

The *ZSK* is used to sign the all the sets of resource records in the zone. The public zone signing key (*ZSK-public*) is stored in the zone and so is publicly available.

From the DNS perspective, the process looks like this: administrator of the zone, for example “nixussoftware.com”, by using the secret key signs the zone data so that each RR will have its own digital signature. When a DNSSEC-enabled client requests for an RR from the authoritative DNS server, this signature will be received together with the RR in server’s reply. The zone’s

corresponding public key for encryption can be stored in some publicly available place, so every user can utilize it to validate the received data from the “nixusoftware.com” name server.

However, so far the above example has not solved the “trust” problem. If hackers can counterfeit DNS responses, they can also create fake key pairs that are required for data authentication and replace real ones with them. There needs to be a way to validate also the public key itself. To resolve this problem DNSSEC includes additional security mechanisms - digital signatures give a possibility to build “*chain of trust*” so that the public key used to authenticate signed data, can be verified by someone else higher up in the DNS chain. In practice, this means that, for example, information about Key Signing Keys of zones nixu.com, nixusoftware.com and example.com could be all stored in .com nameservers, therefore, becoming a “*trust anchor*” for domains below “.com”. Nameserver administrators would only then need to configure one “trust anchor” instead of one for each somethingsomething.com domain.

Using this technique, the most optimal trust anchor would be, of course, the DNS root zone “.” where all the delegations begin. “July 15, 2010: ICANN publishes the root zone trust anchor and root operators begin to serve the signed root zone with actual keys – The signed root zone is available.” (<http://www.root-dnssec.org/>) Figure 4 below displays the public key of DNS root that can be used for such purposes.

```
On 16 June 2010 around 21:20 UTC I witnessed a key generation procedure by which
a DNSSEC Key Signing Key for signing the DNS root has been created.

The representation of this key in the DS RR format is as follows:
. IN DS 19036 8 2
49AAC11D7B6F6446702E54A1607371607A1A41855200FD2CE1CDDE32F24E8FB5
```

Figure 4. Publicly available DNS root public key (trust anchor)

(<https://dnssec.surfnet.nl/wp-content/uploads/2010/07/signed-statement-rori.txt>)

2.2.1 The DNSSEC Resource Records and zone file format.

In the DNSSEC each signed zone has a key pair associated with it - private for encrypting and public for decrypting. As the zone is signed, a new set of RRs are added in the zone file - DNS Public Key (DNSKEY), Resource Record Signature (RRSIG), Next Secure (NSEC) and Delegation Signer (DS) [16].

DNSKEY Resource record

The public keys are stored in DNSKEY resource records and are used in the DNSSEC authentication process: a zone signs its authoritative RRsets by using a private key and stores the corresponding public key in a DNSKEY RR. A resolver can then use the public key to validate signatures covering the RRsets in the zone, and thus to authenticate them. [16]

The DNSKEY record looks like the example below:

```
example.com. 86400 IN DNSKEY      256 3 10
Av855BgBuhYcK+BF1l6lUPnppX8fupRPjiJA17kB5j0fXbirPqEuuP
wXRYJCpZu2YHRqt3GXLvmurDfoWgIjsdCTjhduTxXJMQGUS7nOAoQR
QsibykoVLCLE23oA9Oj5Pt2gda4a6KZOBYqk4VVBH725t2PfKX2lQx
xWtCJhRW4P9QM=
```

The first field, domain's name ("example.com" in given example), specifies the owner of the public key. Next three fields specify TTL (Time-to -live), class and type of the Resource Record.

The following fields provide more information about the key itself. "Value 256 indicates that the Zone Key bit (bit 7) in the Flags field has value 1 " [16], which means that record contains zone signing key. "If bit 7 has value 0, then the DNSKEY record holds some other type of DNS public key and MUST NOT be used to verify RRSIGs that cover RRsets." [16].

Value "3" represents the Protocol and this value is fixed.

The Protocol Field MUST have value 3, and the DNSKEY RR MUST be treated as invalid during signature verification if it is found to be some value other than 3. [16].

Value “10” represents the algorithm type that was used for generating the key. In the given example, it is RSA/SHA-512. A list of possible algorithms can be found in RFC 4034, Appendix A.1 “DNSSEC Algorithm Types”.

Finally, there is a Public Key field which stores the Public Key itself. This key can be used to verify rest of the zone’s resource record signatures.

RRSIG Resource Record

RRSIG (Resource Record Signature) is a digital signature and contains related information - time interval of validity of the signature, algorithm used, the name of the signer and the key tag to identify the DNSKEY RR. A correctly signed zone has a RRSIG record for each resource record in the zone.

Assuming that in the zone “example.com” there is a host with following name and IP address:

```
host1                A                200.30.15.3
```

Then, corresponding RRSIG record wil look like this:

```
RRSIG      A 10 3 86400 20120316000000 20120214120107
24515 example.com
017saw1QPJhGpSxTo8I4NbM495xv/zrEDcOus2tmsLcSoObMmNM5/s
c+Av5enSMnJNLVMyDHJKBs/kPwiqqxUeV+AuEdxCvMGvOUvWXawfFK
YF7c3eZGTxAglAe1MpuJAmKptHHqv7ck1R2hLYRR5EW0uCiTnBScoe
osXbXqJpY=
```

Letter “A” represent the type of RR that is signed with this particular RRSIG. Value “10” represents the algorithm type. Value “3” is the number of Labels (number of RRs that is needed for checking owner’s name. Since Root has Label = 0, 1st Level TLD =1 and so on). For example, “www.nixusoftware.com”

has 3 labels, meaning that value in the Label field will be “3” and for “nixussoftware.com” the value will be “2”. Number “86400” is TTL (Time-to live) value of the original record (RR that was signed).

Values “20120316000000” and “20120214120107” represent Signature Expiration and Signature Inception (when the signature was published in the zone), which define the period of validity of this particular RRSIG record. The format of these values is “YearMonthDayHoursMinutesSeconds”.

The next field with value “24515” is the key tag; “example.com” is the name of the signer (name of the zone).

The last field is the signature itself which can be validated using the corresponding zone signing key.

NSEC Resource Record

The two previously described DNSSEC Records are used to verify the received data. But in case the user requests some record from the DNS server and such record does not exist, server will return an empty answer. And this answer could not be signed because it does not contain any Resource Record.

The *NSEC (Next Secured)* resource record is required to verify denial of existence. In the previously given example zone “example.com” there are two records - one for “host.example.com” and another one for “host2.example.com”. If someone is requesting the record “host1.example.com”, the server will return a reply with the NSEC record, saying that the next secured record after “host.example.com” is “host2.example.com” and record “host1.example.com” does not exist.

NSEC record for the “host.example.com”:

```
host          A      200.30.20.1
```

```
NSEC host2.example.com A MX RRSIG NSEC
```

NSEC record for the “host2.example.com”:

```
host2                A      200.30.15.3
```

```
NSEC example.com A MX TXT RRSIG NSEC
```

The first field specifies the RR type(NSEC). The next entry represents the next authoritative name. The following fields (A, MX, RRSIG, NSEC for first example) indicate the RRs associated with the name, meaning that for name “host.example.com” only those records exist.

DS Resource Record

To be able to build the “chain of trust”, or, in other words, to interconnect signed domains, the “*Delegation Signer*” resource record is used.

The DS record stores the hash of the Key Signing Key public part and by this refers to a DNSKEY resource record. This record is usually placed in the delegation section in the parent’s zone file together with corresponding NS record defining which servers are authoritative for the child zone. In addition, the DS record indicates that the zone is signed.

After signing the zone with ZSK and KSK, the zone administrator needs to generate the DS record and send it to the parent zone administrator. Usually at the receiving end, there is a domain registrar, and most of the registrars provide a special secured web interface the DS record uploading, e.g., in case of GoDaddy, one of the largest .com registrars.

The DS Resource Record refers to a DNSKEY RR and is used in the DNS DNSKEY authentication process. A DS RR refers to a DNSKEY RR by storing the key tag, algorithm number, and a digest of the DNSKEY RR. Note that while the digest should be sufficient to identify the public key, storing the key tag and key algorithm helps make the identification process more efficient. By authenticating the DS record, a

resolver can authenticate the DNSKEY RR to which the DS record points.

The DS Resource Record and its corresponding DNSKEY Resource Record have the same owner name, but they are stored in different locations. The DS RR appears only on the upper (parental) side of a delegation, and is authoritative data in the parent zone. For example, the DS RR for "example.com" is stored in the "com" zone (the parent zone) rather than in the "example.com" zone (the child zone). The corresponding DNSKEY RR is stored in the "example.com" zone (the child zone). This simplifies DNS zone management and zone signing but introduces special response processing requirements for the DS RR. [16]

2.2.2 DNSSEC process

The previous examples in the “DNS” chapter described the DNS name resolution process and possible vulnerabilities in the DNS system. But how would the system work with DNSSEC enabled? Figure 5 will help to understand this process.

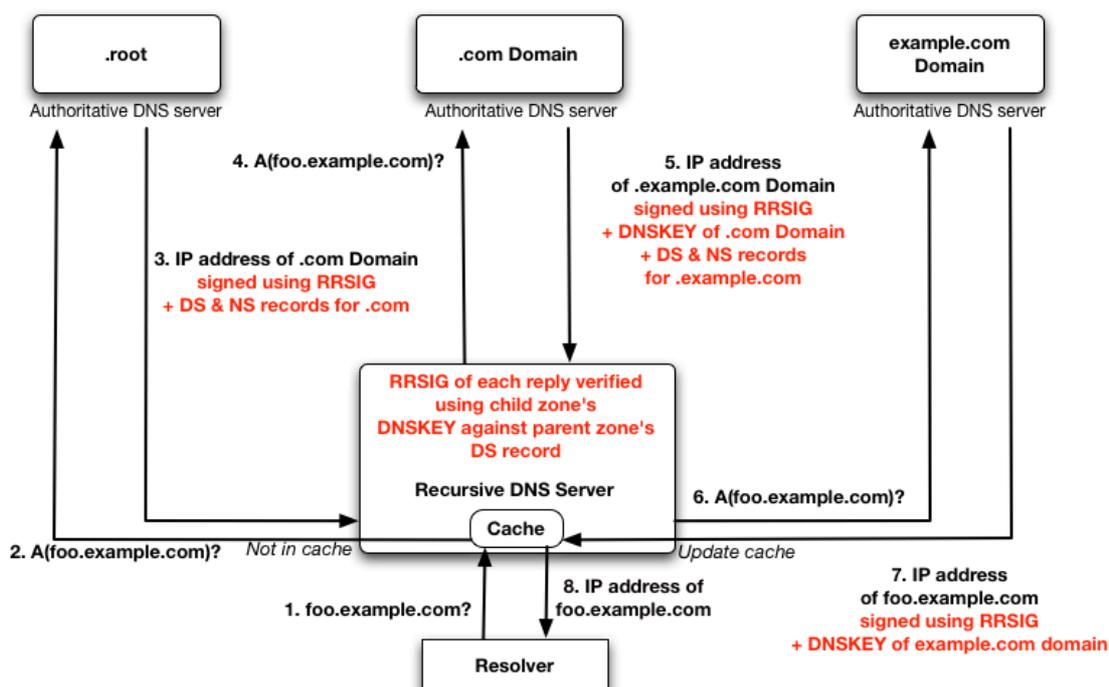


Figure 5. DNS name resolution process with DNSSEC

Assume that the user is requesting for the same “foo.example.com” domain as in the previously given examples.

1. Requesting domain name from recursive DNS server.
2. The recursive DNS server adds the bit “DO” (“DNSSEC OK”) to the request indicating that the resolver is able to accept DNSSEC security Resource Records and requests A record for “foo.example.com” from the root server.
3. The recursive DNS server knows that the domain root is signed by having a pre-configured “trust-anchor”, which could be either the domain root’s public key or the key’s corresponding DS resource record. Using the trust-anchor, the recursive server requests DNSKEY records for the root zone and compares the answer with the stored records.
4. Root server does not know neither the requested A record or the domain name, but knows where zone "com" is hosted. The server provides the

recursive server with “com” authoritative name server’s addresses together with the signed “Delegation Signer” record for the “com” zone.

The DS record is inserted at a zone cut (i.e., a delegation point) to indicate that the delegated zone is digitally signed and that the delegated zone recognizes the indicated key as a valid zone key for the delegated zone. [15]

5. The recursive DNS server validates the “DS” record with the trusted root zone’s ZSK (Zone Signing Key).
6. At this point, the recursive DNS server knows that the zone is signed and requests DNSKEY from the zone’s DNS server. After key validation, the DNS server asks “com” server about “foo.example.com”. However, “com” zone’s authoritative server does not know about it, but knows that zone “example.com” hosted on name servers “ns.example.com” and “ns1.example.com”. So the DNS server replies to the recursive DNS server with this data and again with the corresponding DS record.
7. Now the recursive DNS server has built a chain of trust to “example.com”, where it will obtain addresses of name servers of “foo.example.com” and its “DS” records.
8. Finally, the recursive DNS server iteratively gets addresses of DNS servers responsible for “foo.example.com”. It sends a request to those servers, receives the answer, validates information and provides an address record with “AD” (“Authenticated Data” [17]) bit in the response.
9. If there is no possibility to validate data, for instance, the DNS domain has DNSSEC broken, or the DS records do not match the corresponding keys, the resolver will return the error message “*servfail*”.

The process of the name resolution with DNSSEC enabled does not have any big difference from the process without it. Nowadays there are various utilities which can be used to automatically create digital signatures and sign the zones. However, it is not enough to sign the zone just once - for instance, as it was shown on the example because the RRSIG record has its period of validity. This means that the zone's administrator should periodically re-sign the zone with newly created signatures to prevent brute-force attacks, which is a type of attack where attacker is trying to find out the original private key used to sign the zone, by checking all the possible key combinations until the correct one is found.

For various reasons, keys in DNSSEC need to be changed once in a while. The longer a key is in use, the greater the probability that it will have been compromised through carelessness, accident, espionage, or cryptanalysis. Furthermore, when key rollovers are too rare an event, they will not become part of the operational habit and there is risk that nobody on-site will remember the procedure for rollover when the need is there. [6]

DNSSEC Key Management mechanisms will be explained in details in the next sections.

2.3 DNSSEC key management.

In the previous sections, the theory of DNSSEC was studied, including general principles and working mechanisms. In this section actual processes involved into DNS zone signing steps and DNSSEC data management will be discussed.

2.3.1 Algorithm choice

The very first step in the zone signing process is to choose the algorithm for generating the keys. The algorithm choice is based on the recommendations given in RFC 4641:

There are currently three different types of algorithms that can be used in DNSSEC: RSA, DSA, and elliptic curve cryptography. The latter is fairly new and has yet to be standardized for usage in DNSSEC.

RSA has been developed in an open and transparent manner. As the patent on RSA expired in 2000, its use is now also free.

DSA has been developed by the National Institute of Standards and Technology (NIST). The creation of signatures takes roughly the same time as with RSA, but is 10 to 40 times slower for verification [17].

We suggest the use of RSA/SHA-1 as the preferred algorithm for the key. The current known attacks on RSA can be defeated by making the key longer. As the MD5 hashing algorithm is showing cracks, we recommend the usage of SHA-1. [6].

Almost every name server that supports DNSSEC includes the tool for key generating. The most common DNS software is the BIND server whose latest versions includes command-line utility called “*dnssec-keygen*” that can be used to generate the DNSSEC key pair (private key and public) as defined in RFC 2535 and RFC 4034. This tool allows the administrator to choose the algorithm.

2.3.2 Key length and lifetime

Since there are two key types used in DNSSEC, Zone Signing Key and Key Signing Key, different parameters should be considered when creating the key pair. According to RFC 4641:

zone administrators will need to take into account how long a key will be used, how much data will be signed during the key publication period...and, optionally, how large the key size of the parent is [6].

Selected key length is determined by the ratio between the risk of key compromise and performance. Performance is determined by the time of

signature creation and signature verification. The packet length of DNS-response must also be considered, because DNSKEY resource records are sent in an additional section of the DNS-response.

Since the KSK is only used for signing sets of keys (a set of DNSKEY resource records) the performance is not a deciding factor. However, a compromised KSK can have a greater negative impact, because the key is actually a master key for the zone. The key compromise higher in the DNS-hierarchy can put most of the DNS-sub tree under spoofing attacks. Thus, KSK recommended that the key length is large: it has little impact on performance, but is susceptible to compromise.

When the administrator selects a key length for the ZSK, performance is a major factor, because the key used to sign all the sets of resource records in the zone. In terms of the impact of compromise, the compromise is limited only to the zone, because ZSK is not used to provide authenticated delegation to the parent zone. Consequently, the ZSK may be shorter than the KSK.

The selected usage period (rollover interval) is determined by the risk of a key. In the case of KSK, the amount of information being signed is not very large because the KSK signs only the DNSKEY resource records. This in combination with a large key length leads to the fact that the rollover interval of KSK can be long e.g., up to a year.

In the case of ZSK key, the risk is higher. ZSK is used to sign rest of the zones RR sets and it usually has a relatively small key length. These lead to the fact that the period of validity for ZSK keys must be less than the period of validity of KSK, which is usually one or two months.

It is very important to remember that the keys were created with a certain lifetime. Once signatures expire, data cannot be validated anymore and will be marked as “invalid” so clients cannot use it. That is why it would be useful to

have this process automated - administrators can receive alerts when the key's expiration date is coming and re-signing is needed.

2.3.3 Key rollover

DNSSEC public keys should be changed periodically, and the more frequently the keys are regenerated, the more difficult it is to reverse-engineer the matching private key. That is why it is very important to have a plan not only for DNSSEC key generation, but also for key rollover. As written in RFC 4641, the document describing DNSSEC operational practices:

The longer a key is in use, the greater the probability that it will have been compromised through carelessness, accident, espionage, or cryptanalysis. [6]

In RFCs, recommendations for key effectiveness period are given based on the key type.

... a reasonable key effectiveness period for Key Signing Keys is 13 months, with the intent to replace them after 12 months. An intended key effectiveness period of a month is reasonable for Zone Signing Keys. [6]

However, it is not recommended to strictly follow this advice and have a predictable schedule for keys rollover due to security reason. Another document called "Five Strategies for Flawless DNSSEC Key Management and Rollover" suggests that

instead of rolling the ZSK every 90 days like clockwork, you could pick a time within a 10-day window on either side of the otherwise predictable quarterly rollover date. [13]

The standard practice to rollover the keys are in staged manner, which means "to generate a new key pair, and then have the two public keys coexist at the publication point for a period of time." [2] Using this practice will give the clients and relying parties a time interval to obtain the upcoming key. During this

period, both keys, the old and the new one, will be used for validation and after the old key has expired it can be removed and only the new one will stay in use.

Resolvers that are using DNSSEC should refresh their local cache of zone keys in synchronization with a published schedule of key rollover, and ensure that they load a copy of the new key within the period when the two keys coexist. In this way when the old key is deprecated, responses from the zone servers can be locally validated using the new key. [2]

2.3.4 ZSK (Zone Signing Key) Rollover.

As stated earlier, zone signing key (ZSK) should be rolled over more often than key signing key (KSK). The most important reasons for that are the fact that zone signing key is usually much shorter than key signing key and the roll-over of zone signing key has smaller impact on DNS hierarchy:

During a ZSK rollover, all changes are local to the zone that renews its key: there is no need to contact other zones administrators to propagate the performed changes because a ZSK has no associated DS record in the parent zone. [5]

ZSK is used to sign zone's resource records and the signatures made with it are included in every DNS response. Because of that, ZSK needs to be short; otherwise networks and DNS servers will need to deal with a large amount of data. However, it can be compromised more easily, that is why it is recommended to change this key often, once in 90 days as it was mentioned in previous section.

RFC 4641 proposes two methods for the zone signing key (ZSK) rollover: *pre-publish method* and the *double signature method*.

When the *pre-publish method* is used, new zone keys are created and published in the zone in advance but these are not used for signing yet. The new key is added to the zone's key set and the key set's signatures are regenerated with key signing key. This "advance" period of time is

the time that it takes for zone data from the previous version of the zone to expire from old caches, i.e., the time it takes for this zone to propagate to all authoritative servers plus the Maximum Zone TTL value of any of the data in the previous version of the zone. [6]

This means that the administrator should consider the "Refresh Time" and "Minimum TTL" values defined in the zone's SOA record. If, for example, the common default value of 3600 seconds is used, the keys should be published in the zone in 2 hours before the actual key rollover.

The second stage in this method is to remove old signatures and generate new with new keys. At this point, the old key still remains in the zone file's key set in case that some name servers are still caching the old signatures and have not yet fetched the new zone data. This ensures that even those servers are able to validate the signatures. The duration of this stage could be

i.e., the time it takes for this zone to propagate to all authoritative servers plus the Maximum Zone TTL value of any of the data in the previous version of the zone. [6].

In the final stage, the old key can be removed for the key set completely. At this point, the zone file contains only the new key and new RRSIG records. In addition, usually at the final stage, the upcoming zone signing key could be introduced to the key set. Finally, the key set is re-signed with the key signing key.

If the *double signature* method is used, new signatures are generated together with the introduction of the new key, and will happen before the old signatures are removed from the zone data. So there is a period of time when the

signatures of current key and also the signatures of the upcoming key are coexist. This period lasts usually two times the current maximum zone TTL (the “Expire” value in zone SOA record) before the end of validity of the old keys. Both the old signatures and the old key will be removed one zone TTL prior to the expiry. Thus, there will be only one set of signatures created by a key in a single rollover chain at a time when using the pre-publish mode because the signatures that are about to expire will be removed simultaneously as the new signatures are added.

2.3.5 KSK (Key Signing Key) Rollover

Key Signing Key (KSK) does not need to be rolled over as often as the ZSK. This type of key is used for signing relatively small amount of data and the key can be longer than ZSK. Longer keys are not that easy to compromise, so KSK can be normally rolled over once in a year.

Note that KSK rollovers and ZSK rollovers are different in the sense that a KSK rollover requires interaction with the parent (and possibly replacing of trust anchors) and the ensuing delay while waiting for it. [6]

For the KSK rollover, the same principles can be applied as with ZSK rollover. The recommended rollover method is the double signature method. In KSK’s case, the double amount of signatures is not much of an issue as the amount of data needed to be signed is small (only the key set of the zone).

When the new key signing key is generated it is introduced to the zone’s key set. The key set is signed with the current KSK and with the new KSK. Thus, in the zone file, two signatures coexist so the data can be validated using both the old and the new KSKs.

At the same point, the new KSK is sent to the parent zone and a corresponding new DS record is created. The parent zone administrator replaces the old DS

with the new one. Once the DS record is present on the parent's all authoritative name servers, at least one TTL period of DS record needs to be waited before the old KSK can be removed from the child zone.

2.3.6 Emergency rollover

The previous section describes normal key rollover routine procedures for the zone's administrator. The case, where keys are compromised or lost and they need to be replaced, is called *emergency key rollover*.

Technically emergency key rollover is the same procedure as the scheduled one - old, or in this case compromised, keys need to be removed as soon as possible and replaced with the new ones. But here come to play some non-technical constraints.

If the keys are compromised and removed from the zone file, the name servers will know about that only after TTL is expired and they will ask for updates from the authoritative name server. If new keys are introduced during this time, the name server might not be able to validate the data and the site that was requested will be unavailable.

If the administrator decides not to remove compromised keys and proceed with the normal rollover procedure, the zone's data can be spoofed. In practice, a user with a validated DNS response could still get a computer virus or be redirected to a fake web page and leave their sensitive information.

Having said that, it is up to administrator to decide with what option he/she will go after compromising the keys.

Zone operators have to make a trade-off if the abuse of the compromised key is worse than having data in caches that cannot be validated. If the zone operator chooses to break the trust chain to the

compromised key, data in caches signed with this key cannot be validated. However, if the zone administrator chooses to take the path of a regular rollover, the malicious key holder can spoof data so that it appears to be valid. [6]

When zone signing key is compromised it is enough that zone administrator adds the new key as soon as possible. Zone data will be signed with new signatures, old key and RRSIGs will be removed and zone will be secured and operational.

Suggested steps in case of compromised zone signing key

1. Add new ZSK to key set
2. Sign the zone with new ZSK
 - a. Drop the insecure signatures immediately, which causes that resolvers with old compromised ZSK in cache will not be able to verify signatures until the old key's RRSIG expires.
 - b. Leave old signatures in the zone until the old key's RRSIG has expired, to ensure caching resolvers are still able to validate using old key.

Suggested steps in case of compromised key signing key

With KSK emergency rollover, the zone administrator is facing a choice between two options - break the chain of trust or not. The following steps from RFC 4641 describe steps required for keeping the chain of trust intact during the emergency rollover.

1. Introduce a new KSK into the key set and keep the compromised KSK in the key set.
2. Sign the key set, with a short validity period. The validity period should expire shortly after the DS is expected to appear in the parent and the old DSes have expired from caches.

3. Upload the DS for this new key to the parent.
4. Follow the procedure of the regular KSK rollover: Wait for the DS to appear in the authoritative servers and then wait as long as the TTL of the old DS RRs. If necessary, re-sign the DNSKEY RRSet and modify/extend the expiration time
5. Remove the compromised DNSKEY RR from the zone and re-sign the key set using your "normal" validity interval. [6]

However, the administrator might choose another option – the chain of trust will be temporarily broken and the zone will appear insecure.

Method A

1. Replace the current (compromised) KSK with the new one.
2. Send the new KSK or corresponding DS to the parent.
3. Until the parent replaces the DS, the zone will appear as 'bogus', hence the chain of trust is broken.

Method B

1. Immediately have the compromised key's DS removed from parent.
2. Zone becomes 'insecure'.

2.3.7 Key Storage and Security

DNSSEC keys created with BIND's default tools are stored locally on the BIND server as regular files. This means that potentially anyone with read access to server's file system is able to see the contents of the keys. Thus, the best way to keep keys is to store them on the non-network connected machines, or even on removable drives that can be placed in some safe location. Only an

authorized person should be granted with access to that data and use the keys only when zone signing is needed.

Depending on the level in the DNS hierarchy, security methods should be applied accordingly. The higher in hierarchy key is used, the more wide spread the impact is if the key is compromised. The “root” keys are the most sensitive, owning those means “owning the Internet” and the security measures to protect the private part of the key signing key are impressive. The root key is generated in a cryptographic black box which is basically a hardware appliance dedicated only to one purpose – to generate highly secure keys for the root zone. The appliance itself cannot be operated without proper authorization. There is a number of physical “smart cards” – electronic keys size of a credit card – divided to group of trusted Internet community representatives. From 14 smart card holders, at least five need to be present to unlock the cryptographic device. There is another group of trusted representatives called crypto officers who do not hold smart cards but instead a physical key to the safe located at ICANN (Internet Corporation for Assigned Names and Numbers) which holds a smart cards and the cryptographic device itself. This is in case of catastrophic situation where multiple smart card carriers are not able to get to the facility where the safe is located. In addition, the whole process of key generation occurs completely *off-line*. Many organizations do not necessarily have similar facilities or resources to secure their keys, but there are many details to learn from ICANNs key securing practice.

2.4 DNSSEC Key Exchange

When the key signing key is generated and the key chain is signed, the zone administrator should send the public part of the key to the parent. Then the parent zone administrators create the hash of that public key and insert it into Delegation Signer record in the parent zone and sign it with their own private key. This is how the *chain of trust* is built.

2.4.1 Publishing the Public Key

Since there is not a chain of trust yet between the child and the parent, delivering the key cannot be authenticated with DNSSEC. Child zone administrators should prove their identity and ownership of the zone to parent before key can be added. However, in case where same organization is administrating the parent domain and subdomains, this process is quite straightforward. When a subdomain is created and the administrators decide to sign the zone, they generate the zone signing key and the key signing key, then after the zone and the key set is signed, they create a DS record pointing to the public key signing key and insert it to the delegation section of the parent zone. Usually, these steps are manual and could be time-consuming if the organization has many subdomains. A good example could be an Internet Service Provider (ISP).

If the parent zone is external, proving the child domain's ownership is required. If the parent zone is TLD (Top Level Domain), usually authentication and key transfer are done via the registrar where the domain was purchased. This can be done, for example, by using the domain registrar's domain management portals. Usually, this is the same place where the domain owner can register new domains, see billing information, renew service contracts and so forth. The access to the management portal is usually secured with *https* (Hypertext Transfer Protocol Secure) which provides encrypted communication and secured authentication); hence, it does not rely on DNSSEC.

Registrars are the "consumer facing interface" of the registries. They handle domain name allocations and money transactions when an individual or an organization wants to acquire a domain name. Registrars communicate with registries which are the actual authorities managing TLDs.

A registry is typically responsible for the publication and distribution of zone files used by the Domain Name System. [8].

Any changes affecting the parent TLD zone the domain administrator does via that registrar are communicated to the registry. There exists a standard protocol for communication between registrars and registries called Extensible Provisioning Protocol (EPP).

EPP is a connection-oriented protocol that can be layered over multiple transport protocols. Clients establish a secure connection with a server, exchange identification, authentication, and option information, and then engage in a series of client-initiated command-response exchanges. All EPP commands are atomic and idempotent. [7]

By using this protocol, registrars submit to the registries information about created domain(s), changes made to the zone and receive confirmation response about performed actions from the registry. In 2005, Scott Hollenbeck in RFC 4310 introduced DNS Security Extension for EPP that allows registrars to submit DS records to registries to insert those into the parent zone.

2.5 Management systems and tools

Currently there are many tools for DNSSEC management - starting from simple command line utilities to software with GUI where routines are automated.

Modern DNS servers include such tools as a standard option.

The BIND server, one of the most used DNS servers, is shipped with “dnssec-keygen” and “dnssec-signzone” command line tools that allow zone administrators to start working with DNSSEC right after DNS server installing and configuring. By using “dnssec-keygen”, it is possible to generate signing keys with selected parameters - cryptographic algorithm, key size and so on. With “dnssec-signzone”, the actual signing is done - administrator specifies name of the zone to be signed and keys generated. In addition, this tool can generate DS record and roll over keys using pre-publish scheme. Quite good understanding of DNSSEC is required as very little of the DNSSEC management is automated when using these tools. For example, key rollover

needs to be manually initiated and the zone administrator must remember to create new zone keys according to the expiry schedule.

NSD, Network Service Daemon, is another popular open source DNS server. Many people prefer it instead of BIND because of security - NSD is less used than BIND and thus less attacked. The DNSSEC tool for NSD called Ldns and is almost identical to the “dnssec-keygen” and “dnssec-signzone” tools included into BIND.

Another more advanced and easier to use toolset for DNSSEC management is **OpenDNSSEC**. OpenDNSSEC is open source project done in collaboration with several organizations such as .SE (Swedish ccTLD authority), Nominet (UK ccTLD authority) and NLnet Labs. With OpenDNSSEC, it is possible to generate keys and sign the zone, perform automatic key rollover according to the policy, generate DS record and upload it to the parent zone. As the code is open source with BSD license (Berkeley Source Distribution license, very liberal type of software license), OpenDNSSEC software can be included also in commercial DNS management solutions.

OpenDNSSEC is a relatively easy to use command-line application for Unix-based systems. During the setup process, the administrator needs to import zones and policies into the database and once the system is set and started, the zones become signed. This is done with few simple commands typed on the command line interface. The administrator can set up a schedule for automatic rollover. Only manual procedure is generating DS records and uploading those to the parent zones.

Many commercial DNS management systems adopted DNSSEC. Software giant Microsoft introduced DNSSEC support in Microsoft Server 2008 R2 release. It is possible to generate keys and sign zones, perform key rollover (using pre-published and double signatures methods), create DS record and upload it to parent zones. Processes are not fully automated and there is no graphical user interface for these tasks but Microsoft administrators can deploy DNSSEC in the familiar environment without switching vendors.

Lastly, there are commercial DNS and IP management solutions, such as **NameSurfer Suite** by Nixu Software. This software appliance not only supports DNSSEC functions and provides automated key rollover, but also offers a graphical user interface for management. A similar functionality can be found in products of Infoblox, the maker of specialized hardware appliances with integrated DNS, DHCP and IP address management.

2.6 Theory Summary

Implementing DNSSEC is straightforward – the zone administrator needs to generate keys, sign the zone and include the DS record to the delegation section of the parent zone to indicate that the zone is signed. It is also important to perform periodical key rollovers to avoid key compromising. There are many tools that help to cope with these tasks. However, even advanced ones leave one step to be performed manually - uploading DS record to the parent zone.

As it was mentioned in the Section “Publishing Public Key”, the administrator needs to copy the generated DS and paste it into the registrar’s or the registry’s online portal. After that, the DS record will be included into the parent zone file and the zone will become secure. This needs to be repeated every time Key Signing Key is renewed.

First of all, a web interface is not often the most convenient method as servers used to store the keys are often equipped only with command line interface and might lack a web browser all together. When a key is manually moved to a host capable of accessing registrar’s online portal, there is a risk that part of the very long key will not be copied because of copy/paste error, resulting incorrect data in the parent zone file.

The most efficient and safest way would be to automate the DS transfer and eliminate the need for admin input altogether. This can be, however, very tricky

if not impossible when the interface is designed to be accessed by a user with a web browser. The first part of the problem is that there is no direct communication channel between the customer managing his own domain and the registry authority for the parent zone. All the communication between the customer and registry should be done using the registrar as a middleman. This brings up the second part of the problem which is that there is no standard way of communication between the registrar and the customer. Unlike registries and registrars that very often use EPP for intercommunication, each registrar has a basically different interface from the customer.

2.7 Requirements of KSK / DS exchange mechanism

Depending on registry or registrar, they might expect that the zone administrator will send either the DS record or the key signing key. Most of them though prefer DS.

However, there should be the way to authenticate the client who sends such sensitive data and the client must be sure that data was sent into the correct place.

Once the new key has been generated either manually by the administrator (for example, in case of emergency roll over) or scheduled by the management system and introduced to the key set, it should be automatically uploaded to the parent via API (Application Programming Interface).

According to the double signature rollover method, at this stage there are two keys coexisting in the zone file. After new DS has been uploaded to the parent zone, the DNS management system can poll it with DNS queries to check when the DS record has actually appeared. Once a new key is found, the system should wait one TTL, then remove the old key from the key set and resign it with the new KSK. The progress of this should be indicated on the user interface.

The zone administrator should have the possibility to set the key rollover interval frequency. There should also be a way to replace the old key immediately in case of an emergency rollover.

3. Case study of DNSSEC in Finland

In this chapter the actual experiment on implementing the Key Exchange Mechanism will be explained and the used software will be introduced. This part also contains the overview of Nixu Software products including the DNS Management System “NameSurfer Suite”. The existing DNSSEC functionality of the NameSurfer Suite is extended with the proposed Proof of Concept for the Key Exchange Mechanism.

The Key Exchange Mechanism will be implemented as a Python program in the Nixu Software NameSurfer Suite DNS Management System. This program will automate uploading of the DS record by fetching generated record from Nixu NameSurfer Suite and sending it to the “.fi” zone Authority’s DNS management system using Ficora’s web service interface for registrars.

3.1 The environment description

For the implementation this Proof of Concept, Nixu Software’s Centralized Management System was selected. Nixu NameSurfer Suite provides almost all needed automation for DNSSEC routines. The zone administrator can select the algorithm, generate the keys and sign the zone using a graphical interface. It is possible to perform scheduled Zone Signing Key Rollover automatically. But for the initial key exchange and for Key Signing Key rollover there is only one option - log in to the registrar's or registry’s portal and copy/paste the DS record. Including the key Exchange Mechanism script to the NameSurfer Suite will make current DNSSEC functionality in this product even more automated. This means that the zone administrator will use a single graphical interface to go through all the steps that are needed to secure the zone. However, for the proof of concept, the script is ran manually after creating a new key signing key, and the UI additions will be implemented at a later phase.

Another part of the POC environment is the Finnish Communications Regulation Authority (Ficora), “.fi” domain registry. As it was explained earlier,

usually customers do not contact the registry directly, but perform all the needed changes to their zones via registrar. The reason for this process is to avoid a huge workload and possible disorder if, for example, all the clients who registered their zones within “.com” domain would like to submit new data. However, in this particular case it was decided to make an exception. Ficora holds a relatively small domain space, so it is possible for a customer to contact the registry directly, i.e., automatically send the DS record created by DNSSEC Management System.

3.2 Nixu NameSurfer Suite and its current DNSSEC functionality

Nixu NameSurfer Suite is Nixu Software’s flagship product. It is a virtual software appliance for centralized DNS, DHCP and IP Address management.

It consists of four modules - the first one, DNS, allows administrators to create, delete and modify zones, manage them; create and delegate subdomains; import zones from another name servers. The user interface provides a variety of automations, such as using templates when creating new nodes, automatic creation of reverse records for corresponding forward records. When creating a new zone, the system checks if there is a parent zone already existing and if it does, then it automatically adds the child zone to the delegation part of the parent.

The second module, IPAM, is used for IP address management. It is an IP address database where administrators can manage actual addresses on the computers inside the zone.

The third module is used for managing remote servers. Nixu NameSurfer Suite is the management overlay, thus, instead of accessing each one of DNS and DHCP servers in the network, it can be done in one place. Administrators can add delegated zones to remote servers, manage server’s configuration, and see statistics such as server workload, queries per second and so on.

The last one, User Management, controls access to the modules listed. It is possible to create users in the system, add them to the groups and grant them specific permissions and access levels.

With the DNSSEC functionality of the Nixu NameSurfer Suite zone, the administrator can select the algorithm and generate the keys, sign zone and set up automatic Zone Signing Key rollover. This is done via a graphical user interface. The system can generate the DS record and the administrator needs to copy it and send to the parent. Once the initial keys are generated and key rollover is set up, the administrator does not have to resign the zone manually again anymore. New records that are added to the zone will be automatically signed.

The graphical user interface for generating keys and setting rollover is shown in Figures 7 and 8 below.

The screenshot shows the 'NameSurfer' web interface. The top navigation bar includes 'NAME SURFER' and 'Add DNSSEC key for zone example.com'. The left sidebar contains a menu with categories: GLOBAL TOOLS, DNS (with sub-items Zone, Settings, Add DNSSEC key), REMOTE SERVERS, IP ADDRESSES, CONFIGURATION, and MAIN MENU (with sub-items Help, Logged In as: admin, Log out). The main content area is titled 'Add DNSSEC key for zone example.com' and features the NIXU DDI logo. The form includes:

- Key inception on: 25.4.2012 (Beginning of the key validity period)
- Expires at: 26.5.2012 (End of key validity period)
- Key role: Zone signing key, Key signing key
- Generate a new private/public key pair
- Key algorithm: RSA/SHA-512
- Key size (bits): 1024
- Upload key file: Choose File (no file selected)
- Warning: RSA-SHA1 keys are not compatible with the use of NSEC3 authenticated denial of existence. If NSEC3 is enabled in the zone, any RSA-SHA1 keys will not be used for signing in the zone, even if they are enabled and not past their validity period. Please use RSASHA1-NSEC3-SHA1 key type instead of RSA-SHA1 on NSEC3 zones.
- Footer: (Private key file v1.2 format. Use dnssec-keygen utility to generate new keys.)

 An 'OK' button is located at the bottom left of the form area.

Figure 6. Creating DNSSEC key in NameSurfer Suite

Delegation Signer (DS) record data for current key		Algorithm
24515	10 1 b382980a73f6d1629dfa2a273fa5c36322d4dca3	SHA-1
24515	10 2 623b7b9dfd0bd68ada3cc6ddb3072425700c5b819ca5e16521198e6351689c0b	SHA-256

Figure 7. Selecting automatic key rollover method and interval in NameSurfer Suite

However, there is still manual work required - at the moment there is no possibility for Key Signing Key automated rollover, only for the Zone Signing Key. Since the DS record uploading is still a manual process, KSK rollover is required to be done by the zone administrator. This means that the administrator needs to create a new Key Signing Key and enable it so that it will automatically be added to the key set. However, the old key removal can be done only when the new DS record is added to the parent zone.

3.3 Proposed functionality of the Key Exchange Mechanism

The Key Exchange Mechanism will have the following functionality:

1. As an input, the script will need the following parameters: zone name, DS record, Ficora's Web Service connection details such as URL, username and authentication key.
2. The script initiates connection to the Ficora's Web Service. Once the connection established, the script sends the DS record to Ficora.
3. The script starts checking when the new DS record appears in the parent zone. To do that, the script should make DNS queries to the parent's authoritative name server(s).

4. Once the DS record is published under the delegation section in the parent zone, the script should wait at least double the Time-To-Live period before the old key signing key can be removed from the zone file.
5. As the last step, the old key is removed from the zone in NameSurfer Suite and the script stops.

Used tools and methods

Nixu NameSurfer Suite API (NSAPI) will be used for communication between the script and the NameSurfer Suite.

Nixu NameSurfer Suite Application Programming Interface (NSAPI) provides an interface for accessing much of NameSurfer Suite functionality such as managing users, groups, keys and views. For instance, it could be used for creating a separate front-end for Nixu NameSurfer built using the NSAPI.

The transport mechanism is built on top of XML-RPC, which is a lightweight protocol for performing Remote Procedure Calls.

Authentication and message integrity are achieved by using a shared secret- key mechanism. For each procedure call, a separate message signature is calculated from the procedure's name, the current time along with all of the RPC's argument values. This ensures that an intercepted message cannot be modified without access to the secret key. The timestamp prevents usage of the same call after a predefined time has elapsed. Messages are not encrypted so it is advisable to set up a secure connection between the client and server if any sensitive information is to be sent over the NSAPI. [12].

Ficora (Finnish Communications Regulatory Authority) offers a special Web Service interface for the clients. This service provides functionality such as registration and renewal of domain names, making changes in the domain names without a browser-based interface. This service is available only for

registrars, i.e., for the customers of Ficora. Ficora's Web Service uses standardized Web Services Description Language (WSDL).

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information [4].

Documentation for the Web Service can be found in Appendix 4. At the time of the thesis writing, it was publicly available at Ficora's web site https://domain.fi/info/attachments/61k1kXiEc/WS_toiminnallisuuspalvelukuvaus_EN.pdf

3.4 Development environment

The script is written in then Python language and the development was implemented in Mac OS X. Since OS X 10.8 comes with an older version of Python, 2.7.2 and the latest production version is 2.7.3, the newer version needed to be installed separately. Instead of compiling and manually installing our own Python, the latest production version of Python was installed using a third party package manager called "Homebrew" <http://mxcl.github.com/homebrew/>. "Homebrew" automates compilation and installation of the open source software on Mac. The functionality and usage are similar to "yum" manager for Linux Red Hat distributives. For example, in case of Python, it leaves the operating system's own installation of Python intact and installs the newer version in a different location.

First, "Homebrew" is installed by running a remote ruby script:

```
$ /usr/bin/ruby -e "$(/usr/bin/curl -fsSL  
https://raw.githubusercontent.com/mxcl/homebrew/master/Library/Contributions/install_homebrew.rb)"
```

Then Python 2.7.3 is installed from “Homebrew” using standard installation command (“*brew install [packagename]*”). The recommendation is to install the “framework” version so it is selected by giving the parameter “*—framework*”:

```
$ brew install python --framework
```

To ensure that the latest version of Python is used, and not the operating system default on the e, following lines were added to shell user’s bash profile file. This ensures that the first location where “python” binary is searched from is “*/usr/local/share/python*”:

```
export PATH=/usr/local/share/python:$PATH
```

After that, the Python version was verified with the command:

```
$ python -V  
Python 2.7.3
```

To easily install Python libraries during the development work, package manager PyPI (Python Package Index) was installed. Unlike the default one called “easy-install”, PyPI makes installing and uninstalling packages very simple.

Installing PyPI using “easy_install” manager:

```
$ easy_install pip
```

To further avoid the situation where third party libraries cause issues in the development environment, all the development was carried out in the isolated virtual environment using “virtualenv” - Virtual Python Environment builder.

To install “virtualenv”:

```
$ pip install virtualenv
```

To create the actual virtual environment inside the project folder:

```
$ virtualenv --distribute venv
```

With the following commands, the environment can be started (inside the project folder):

```
source venv/bin/activate
```

 - the command prompt will change to show the active environment.

And stopped:

```
deactivate
```

 - this will restore settings to normal.

3.5 Design of the Python script

Because this script is the Proof of Concept, at this stage it is not implemented in the NameSurfer Suite UI. Instead the script is run manually from command line and the parameters such as zone name and DS record are given as command line arguments. Once the script is moved to the production phase, it will be run from NameSurfer's UI and NameSurfer will automatically pass the required parameters to the script.

To make the script reusable once it is integrated as part of the NameSurfer Suite, it is divided into *classes* based on the functionality. Below is the overview of the classes.

The whole script with the comments can be found in Appendixes 1, 2 and 3

Class “WSClient”

This class is used for communication with Ficora's Web Service. It provides the functionality to create the web service request for adding the DS record and sending the request to the web service.

Class "DSChecker"

This class builds the DNS query to check if the DS record is published in the parent zone.

Class "NSClient"

This class is needed for the last step in the script's functionality as it provides communication to the NameSurfer's API. It includes the possibility to modify (add or remove) resource records in the zone. This class will be used to remove the old DS. Once the PoC is implemented as the part of the NameSurfer Suite, this class might become irrelevant because after receiving a positive reply about uploading the DS record, NameSurfer can perform the necessary action by itself. More information about API can be found in Nixu Software NSAPI RPC Reference manual [13]

3.6 Implementation of the script

The previous section gave a brief overview of the classes' functionality. This section will provide a more detailed description of the implementation of each of them.

3.6.1 Class "WSClient"

As it was mentioned earlier, this class is needed for communication between the script and the Web Service of the service provider. For testing purposes, Ficora created a special account and the parameters username and authentication key were used for authenticating the client when sending the request. As described in Section 3.2, Web Service uses WSDL language and to communicate with the service this class uses the "suds" Python module. Suds is a SOAP Python client which can be used to communicate with Web Services in

Python applications. Using the examples from suds documentation placed in <https://fedorahosted.org/suds/wiki/Documentation>. WSDL client was implemented with following piece of code:

```
from suds.client import Client
self.client = Client(url)
```

Class WSCClient takes the following information as parameters during initialization of the class:

zone - name of the zone for which the new key signing key was created
 username - user name for the Ficora's Web Service user
 authorizationkey - authentication key provided by Ficora for Web Service user authentication
 url - address of the Ficora's Web Service

Class has five functions: createAddDSRequest, sendAddDSRequest, __createMac, __getTimeStamp, and listDomains. Each function is going to be introduced below.

“createAddDSRequest” function

According to Ficora's Web Service documentation [14], the request for adding the DS record is called “AddDSRecord” and the XML format of the request is the following:

```
<ns2:AddDsRecord>
  <ns2:request>
    <ns1:Context>
      <ns1:user_name>iam</ns1:user_name>
      <ns1:mac>68ab2d905d85d9ed14fd3bb4a3c77b2bfe0fade0</ns1:mac>
      <ns1:timestamp>2011-03-03T13:12:00.5005630Z</ns1:timestamp>
    </ns1:Context>
    <ns1:DomainName> </ns1:DomainName> <ns1:AuthorizationKey>
    </ns1:AuthorizationKey> <ns1:DsRecords>
      <ns1:DsRecord>
        <ns1:KeyTag>9866</ns1:KeyTag>
        <ns1:Algorithm>7</ns1:Algorithm> <ns1:DigestType>1</ns1:DigestType>
        <ns1:Digest>d7c9cb1a153681a3d442b4257c2e16aca6f7f11d</ns1:Digest>
      </ns1:DsRecord>
    </ns1:DsRecords>
```

```
</ns2:request>
</ns2:AddDsRecord>
```

To build the described request, fields need to be filled with the required parameters. Detailed explanations can be found in Ficora’s Web Service documentation Chapter 3.17.3 “Request message” [14].

To create the request with suds client:

```
request = self.client.factory.create('ns1:AddDsRecordRequest')
```

And this is how the *AddDsRecordRequest* instance looks like in Python:

```
print request
(AddDsRecordRequest){
  Context =
    (context){
      user_name = None
      mac = None
      timestamp = None
    }
  DomainName = None
  AuthorizationKey = None
  DsRecords =
    (ArrayOfDsRecord){
      DsRecord[] = <empty>
    }
}
```

Suds has created a corresponding Python object as described in Ficora’s WSDL description and now it is just a matter of filling the fields with values. Values such as “username”, “AuthorizationKey” and “DomainName” were given as parameters during class initialization. The “Timestamp” value was created using the function `__getTimeStamp` - it takes system time, converts it to ISO8601 format to match Ficora’s requirement [14].

The “DsRecords” data field requires an array of DsRecord-objects for input. To get an instance of DsRecord object from the Web Service client:

```
dsrec = self.client.factory.create('ns1:DsRecord')
```

The *DsRecord* instance contains the following entries which define the DS record:

```

print dsrecord
(DsRecord){
    KeyTag = None
    Algorithm = None
    DigestType = None
    Digest = None
}

```

KeyTag indicates the key type; here it will be the key signing key tag. *Algorithm* shows which cryptographic algorithm was used for the key generation. The DS record to be uploaded is given as a parameter to the function “*createAddDsRequest*” and it is parsed into the *DsRecord* instance’s data fields.

Finally, the “MAC” value is generated from a string where all data fields are placed in exactly the same order as they are listed in the message. The separate function `__createMac` was written; this function takes these values, adds an authorization key and based on the result string, it generates a hash value. This hash value is then added to the message sent to the Web Service.

“**sendAddDSRequest**” function

After the request is built, it needs to be sent to the Web Service - this is the purpose of the “*sendAddDSRequest*” function. As a parameter, it takes the DS record, and passes it to the *createAddDSRequest* function to get the *AddDsRecordRequest* instance as a return. Then the function sends the *AddDsRecord* request to the web service with:

```
client.service.AddDsRecord(r)
```

and finally returns *WSAddDsRecordResponse*:

```

(WSAddDsRecordResponse){
    Code = None
    TimeStamp = None
    WebdomainValidationErrors =

```

```

(ArrayOferror_message){
    error_message[] = <empty>
}
}

```

This is an equivalent of “Success” or “Fail” message. The important part is the “code” parameter that returns boolean “true” or “false” indicating the result of the request sent. “WebdomainValidationErrors” will help with troubleshooting in case the request fails.

Functions used for the Web Service testing

There are some functions in the class WSClient that are not essential for the script but can be used for testing or other purposes - listDomains, ping and sendDomainRegisterRequest. With listDomains, the user can see the list of domains registered for him/her in Ficora’s system; ping sends an echo request to Web Service to verify that there is connectivity. By using sendDomainRegisterRequest, the user can send a request to apply for registering a new domain.

3.6.2 Class “DSChecker”

This class includes functions to verify when DS is visible in root name servers. When DS record is uploaded, publishing time needs to be verified by periodically querying the root servers using *checkDS*. Finally, after the double “Time-to-Live” value of the old DS has expired, it should be removed from the parent zone. For polling this information, the function *getRR TTL* can be applied.

Class takes as initial parameters the zone name, the root (parent) domain and optionally the list of the root servers. If *rootservers* is *None*, the private function *__getRootServers* (function called *private* if used only inside the class) will be called which resolves the root servers automatically. As a parameter, *__getRootServers* takes the name of the root domain.

“checkDS” function

After the request with the DS record was built and sent to the Ficora’s Web Service, its appearance on the root servers should be verified. The “*checkDS*” function uses *DNSPython* module functions to send DNS queries until it gets a reply containing the DS record. This function reads class variable *rootservers* - the list of the root server addresses from where the DS is queried. *CheckDS* returns boolean which is either “True” or “False depending whether the DS record is found from the root servers or not.

“getRR TTL” function

A waiting interval equal to two TTL values is needed to make sure that all the clients received a new DS record and the chain of trust was not broken. Here are used *DNSPython*’s functions *dns.message.make_query* to make a query and *dns.query.udp* to send it. As a parameter, *getRR TTL* takes the list of the root servers. The function returns the TTL value as an integer.

3.6.3 Class “NSClient”

This class provides connection to NameSurfer Suite API and has the ability to modify resource records in the given zone. Since the API is XML-RPC-based, the Python *xmlrpclib* module is used:

```
from xmlrpclib import Server, Fault
```

As an initial parameter, class takes the NameSurfer’s API key and value and the address of the server:

```
def __init__(self, keyname, transactionkey, serveraddr) :
```

```
    self.keyname = keyname
```

```
    self.transactionkey = transactionkey
```

```
    self.serveraddr = serveraddr
```

“__modifyRR” function

The Required parameters for this private function are the following: the operation (“Add” or “Delete”), zone name, node, resource record type, data to be modified and optionally RR’s TTL and view where the zone is. All the values are hashed with the SHA-256 algorithm and the values and the hash are passed to the xmlrpc module’s *Server* object’s *ns.dns.update* function:

```
def __modifyRR(self, operation, zone, node, type, data, ttl="", view="") :
    procedurename = "ns.dns.update"
    timestamp = str(int(time.time()))

    hash = hashlib.sha256()
    hash.update( procedurename \
        + "&" + self.keyname \
        + "&" + timestamp \
        + "&" + operation \
        + "&" + zone \
        + "&" + view \
        + "&" + node \
        + "&" + ttl \
        + "&" + type \
        + "&" + data \
        + "&" + self.transactionkey );

    try:
        myServer = Server(self.serveraddr)
        return myServer.ns.dns.update( self.keyname, timestamp, operation,
zone, \ view, node, ttl, type, data, hash.hexdigest() )

    except Fault as err:
        logging.error("Error occured while processing XML-RPC method call:
%d" % err.faultCode)
        logging.error("Fault string: %s" % err.faultString)

    return -1
```

The function returns the server's response or in case of exception logs the error.

“addRR” and “delRR” functions

These functions provide a simple wrapper for the private function `__modifyRR`. They can be called and used outside the class *NSClient*.

3.6.4 “Main” function

The main function initializes the execution of the script. It starts with reading two types of the parameters - saved in the configuration file and provided using the command line interface. The data stored in the configuration file *default.cfg* is more like application settings and these values are not likely to change between the script executions:

```
[webservice]
user = Andreeva
key = xxxxxxxxxxxxxxxx
url =
https://domainws.ficora.fi/fiDomainTest/DomainNameWS_FicoraDomainNameWS.svc?
wsdl

[domain]
parentdomain = fi.
parentservers =

[namesurfer]
keyname =
transactionkey =
serveraddr =
```

The first section defines the parameters required for user authentication against Ficora's Web Interface: user name, authentication key and the address of the

service. In the section *domain* are stored the parent domain name “.fi” and parent name server addresses. Finally, in the section *namesurfer* are placed the parameters needed for communicating with the Nixu NameSurfer Suite: API key name and value and URL of the NameSurfer’s API.

Values are read by *ConfigParser* object and stored in the variables:

```
config = ConfigParser.RawConfigParser()
        config.read("default.cfg")

username = config.get("webservice", "user")
authorizationkey = config.get("webservice", "key")
url = config.get("webservice", "url")
parentdomain = config.get("domain", "parentdomain")
parentservers = config.get("domain", "parentservers")
if len(parentservers) < 1:
    parentservers = None
nsaddress = config.get("namesurfer", "serveraddr")
nskeyname = config.get("namesurfer", "keyname")
nskey = config.get("namesurfer", "transactionkey")
```

The user provides another set of parameters taken by the “main” function as command line arguments:

```
python DSUploader.py -h
usage: DSUploader.py [-h] -z ZONE -k DSKEY [-p POLLFREQ] [-d DELETEKEY]
```

optional arguments:

```
-h, --help  show this help message and exit
-z ZONE     Zone name
-k DSKEY    DS key as space delimited text fields 'KEYTAG ALGORITHM
            DIGESTTYPE DIGEST'
-p POLLFREQ Polling frequency for checking the root for new DS. Default
            1800 seconds.
```

-d DELETEKEY Old key to be deleted from NameSurfer (OPTIONAL).

The Python module *argparse* brings advanced support of CLI arguments. The following values are taken as an input from command line: zone name for which the new KSK is introduced, corresponding DS record, polling frequency for checking if a new DS appeared on root servers and the old key signing key which needs to be removed from the zone after DS was published. The last parameter is optional.

First, a new instance of class *ArgumentParser()* called *parser* is created. Then this instance is configured to parse the required arguments.

```
parser = argparse.ArgumentParser()
parser.add_argument("-z", dest="zone", required=True, help = "Zone name")
parser.add_argument("-k", dest="dskey", required=True, help = "DS key as \
space delimited text fields \KEYTAG ALGORITHM DIGESTTYPE DIGEST\")
parser.add_argument("-p", dest="pollfreq", default=1800, type=int, help = \
"Polling frequency for checking the root for new DS. Default 1800 seconds.")
parser.add_argument("-d", dest="deletekey", help = "Old key to be deleted from \
NameSurfer (OPTIONAL).")
```

Here the *parser* instance of *argparse.ArgumentParser()* reads the values inputted from the command line.

```
cli = parser.parse_args()
zone = cli.zone
ds = cli.dskey
pf = cli.pollfreq
oldkey = cli.deletekey
```

The next new instance of the class *DNSChecker* is created and saved with the name *checker*. This instance calls function *getRR TTL* of the parent class, takes the resource record type as a parameter and saves the received TTL value in the variable *"ttl"*. This will be needed for the later steps.

```
checker = DNSChecker(zone, parentdomain, parentservers)
```

```
ttl = checker.getRR TTL("DS")
```

After that the “main” function creates the instance *ws* of the class *WSClient* to connect to Ficora’s Web Service:

```
ws = WSClient(zone, username, authorizationkey, url)
```

Then the new DS record is sent to the service:

```
resp = ws.addDS(ds)
```

The last step is to verify that the DS record was successfully received by Ficora and published in the parent zone. For this, the script is checking the response received from the Web Service. In case of boolean “false”, the function will log an error message. Otherwise, a while-loop is started that keeps sending DNS queries to the root servers to check the DS record until it is found in the response. The *checkDS* function of the class *DNSChecker* will return boolean “true” which is the condition of breaking the loop.

```
if resp.Code == False:
    logging.error("Error Response: %s" \
        %resp.WebdomainValidationErrors.error_message[0].description)
else :
    logging.info("DS request sent, polling parent for new DS")
    while True:
        if checker.checkDS(ds) is True: break
        logging.info("DS not found. Checking again after %s seconds" %pf)
        time.sleep(pf)
    logging.info("DS found from parent servers.")
```

If the old key signing key was given as parameter to the script (“-d”, “deletekey”), the *delRR* function of class *NSClient* will be used to remove the key from the zone in NameSurfer Suite. Here is going to be used variable “*ttl*” that stores old DS’ Time-to-Live value. To make sure that all caching clients

have refreshed the key from the root servers, the zone administrator should wait double TTL time and then remove the old key from the zone. This function makes this process automated:

```
if oldkey is not None and len(oldkey)>0:
    logging.info("Waiting for %s seconds before removing DS from NameSurfer"
%(ttl*2))
    time.sleep(ttl*2)
    nsc = NSClient(nskeyname, nskey, nsaddress)
    if nsc.delRR(zone, zone, "DNSKEY", oldkey) is not -1:
        logging.info("Key \'%s\' removed from NameSurfer" %oldkey)
```

3.7 Testing

Python, being a scripting-like language, is easy to run and prototype parts of the functionality while writing the code. Different command-line interpreters for Python such as *ipython* used in this project can make executing and testing parts of the program as easy as copy pasting the code from the editor to the shell:

```
n [22]: string1 = "Hello"
```

```
In [23]: string2 = "World!"
```

```
In [24]: def printer(s1, s2) :
.....:     print s1, s2
.....:
```

```
In [25]: printer(string1,string2)
Hello World!
```

```
In [26]:
```

This led to a development/testing method where it was possible to execute very specific functions, e.g., sending message to Ficora's Web Service and immediately seeing which kind of response is received. It was the way to start working on the parts of the script that were the most challenging without a need to even begin designing the classes, methods and so on.

3.7.1 DNS queries

Testing the DNS functionalities of the script provided by the class *DNSChecker* was relatively strait forward. It was possible to verify the correct DNS response by comparing it with the output of the DNS diagnostic tools such as “dig” or “nslookup”. All the DNS testing was made against the public DNS services. Functions were run from the iPython shell after importing the *DNSChecker* class there (for example, by copying and pasting the whole class into the iPython’s shell window).

Fetching TTL of old DS

This test case is needed to make sure that the function *getTTL* returns some result and that the result is the TTL value of the record given as a parameter. When the test was run for the first time, the parameter “DS” (RR name) was given as a string.

Class *DNSChecker* was initialized with following parameters:

```
dsc = DNSChecker("dnssec.fi", "fi.")
```

The zone selected for the querying is “dnssec.fi” and the parent of this zone is “fi.”. The Parent domain’s (fi.) list of servers were omitted from initialization, as it was an optional parameter, so a quick check was made that the servers were automatically resolved by the *__getParentServers* function:

```
In [92]: print dsc.parentservers  
['193.166.4.1', '194.146.106.26', '156.154.100.26', '77.72.229.253', '194.0.1.14',  
'87.239.127.198', '156.154.101.26', '156.154.102.26', '156.154.103.26']
```

Before testing the function, the data was queried with the “dig” tool:

```
$ dig fi. NS
```

and the answer

```
....
;; ADDITIONAL SECTION:
a.fi.                86350 IN    A      193.166.4.1
....
```

Now the query to fi-domain's nameserver for verifying the correct DS:

```
$ dig @193.166.4.1 dnssec.fi. DS
```

```
;; ANSWER SECTION:
dnssec.fi.          21600 IN    DS     59650 8 1
DF58C2A099A71F3348462472416743EA7BF5E9A3
dnssec.fi.          21600 IN    DS     59650 8 2
BD0F7575BB5F3503FABF2CDD3093F02E1DD40E4FF0F859B5A1B47DB3
14B32893
```

The TTL value here is “21600” and that is the value which the `getRR TTL` function should return.

```
In [83]: print dsc.getRR TTL("DS")
21600
```

The value is correct and test is passed - the function works as intended. However, it was noticed that with other nameservers, the function sometimes failed and nameservers returned response with “*rcode FORMERR*”. At first, there was not obvious reason but the response indicated that something was wrong with the message that was sent, so it was required to troubleshoot the issue by looking at the actual DNS message creation in the code.

```
q = dns.message.make_query(self.zone, rr)
```

Here “*rr*” is string representation of the record queried, and therein lays the problem. The *Message* object’s *make_query* function does in fact expect the resource record to be converted in *rdata*, or, in *int* format. The “*Dns.rdatatype.from_text()*” function provides the *rdata* representation so the following change was added to make the conversion:

```
q = dns.message.make_query(self.zone, dns.rdatatype.from_text(rr))
```

It is still not fully understood why most of the time queries worked even with string representation, but the change made the function more reliable in the testing. The same correction was made to the other functions where DNS messages were used, e.g., *checkDS*.

Querying new DS record from the parent servers

DS record querying from the parent servers was done with the same instance of the class *DNSChecker* as in the previous example of fetching the TTL.

First data were obtained with “*dig*” from the same zone “*dnssec.fi*”:

```
$ dig @193.166.4.1 dnssec.fi. DS
```

```
:: ANSWER SECTION:
```

```
dnssec.fi.          21600 IN      DS      59650 8 1
```

```
DF58C2A099A71F3348462472416743EA7BF5E9A3
```

```
dnssec.fi.          21600 IN      DS      59650 8 2
```

```
BD0F7575BB5F3503FABF2CDD3093F02E1DD40E4FF0F859B5A1B47DB3  
14B32893
```

Then this DS record was used as the parameter for the function *checkDS*. As it is known that these records are the same function, they should return “True”.

```
In [80]: print dsc.checkDS("59650 8 1  
DF58C2A099A71F3348462472416743EA7BF5E9A3")  
True
```

The returned value is correct and the test is passed – the function works as intended. But what is going to happen if the DS records are not matching? To verify this, a parameter was given a random value string pretending to be a DS record in the dnssec.fi zone.

```
In [81]: print dsc.checkDS("59650 8 1  
DF58C2A099A71F3348462472416743NOTVALID")  
False
```

The function returned the value “False” which was expected in this test case.

3.7.2 NameSurfer API

The only function of the class *NSClient* is to delete the specified record from the NameSurfer instance. First, a test installation of NameSurfer Suite was implemented on the virtual machine. Then there were created a test domain “example.com” and the generated NameSurfer API (NSAPI) key. The test zone was signed with the zone signing key and key set with the key signing key.

Zone overview

Type:	Primary	Serial#:	2012120228								
Zone status:	enabled	Last modified:	Sun Dec 2 21:00:12 2012								
Nodes:	6	DNSSEC status:	enabled								
A:	5	NSEC3 status:	disabled								
Master NS:	ns1.example.com N/A	DNSSEC Keys:	<table border="1"> <tr> <th>Tag</th> <th>Type</th> <th>Role</th> <th>In use</th> </tr> <tr> <td>30822</td> <td>RSA/SHA-512</td> <td>Key signing key</td> <td>Yes</td> </tr> </table>	Tag	Type	Role	In use	30822	RSA/SHA-512	Key signing key	Yes
Tag	Type	Role	In use								
30822	RSA/SHA-512	Key signing key	Yes								
Name servers (NS):	ns1.example.com N/A	Remote secondary name servers:	N/A Modify								

Filter: none a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 Show NSEC3 records

Name	Address	MX	HINFO	Aliases	RP	Text	Hidden text	SRV
example.com	ZONE ROOT						hidden_text_for_test	
host0.example.com	A IPv4 172.16.40.0							
host1.example.com	A IPv4 172.16.40.1					testtext	hidden_text_for_test	
host2.example.com	A IPv4 172.16.40.2							
host4.example.com	A IPv4 172.16.40.4							
host5.example.com	A IPv4 172.16.40.5							

Figure 8. Zone "example.com" in NameSurfer Suite

```

$ORIGIN example.com.
@      IN      SOA      ns1.example.com. katja.katja.com. (
                2012120525 28800 7200 604800 86400 )
NS     ns1
;--    HTXT    test
;--    HTXT    hidden_text_for_test
;--    ALSO_NOTIFY  ns3.example.com
;--    ALSO_NOTIFY  172.16.40.132
;--    NSEC3PARAM  1 0 1 71C0E6
DNSKEY 257 3 10
Av85w29IRnesjhlKNs6YyH4tyzIA9S2jKg98jdToV/x8zWqRRApER0wz6wpw1ez0tg5VP/HWm6AD4tQPFPJNBb6007vkGBt/Zxb5HuKkUUnnXFoYub7Xf3yOufnfvs44Imi
gwVc1L63r2KUoMRpNd/44Bg5JGsOobXb3iqcBMvu+uA0=
RRSIG  DNSKEY 10 2 86400 20121231000000 20121205233010 30822 example.com
GvixV4F2s+gwiWwcTs/V1b7dk07VJraW6tEF5UbtmGe2oxotoL6gW6YT/Ld41HKUAYTuTSrVcfGaiFBWUdcYo09fzdY4vqVQGPjU+1SMDpX4KW+dm60D05xGialcaecSk0
Tes5ncJrxoTfGGdm3YC5kWetYI51pamoQsb9hiL8=
host0      A      172.16.40.0
host1      A      172.16.40.1
;--    HTXT    "hidden text"
;--    TXT     text
;--    TXT     testtext
;--    HTXT    hidden_text_for_test
host2      A      172.16.40.2
host4      A      172.16.40.4
host5      A      172.16.40.5

```

Figure 9. Exported zone's master file with key signing key

Below is placed the code used for deleting DNSKEY resource record:

```

def __modifyRR(self, operation, zone, node, type, data, ttl="", view="") :
    procedurename = "ns.dns.update"
    timestamp = str(int(time.time()))

    hash = hashlib.sha256()
    hash.update( procedurename \
                 + "&" + self.keyname \
                 + "&" + timestamp \
                 + "&" + operation \
                 + "&" + zone \

```

```

        + "&" + view \
        + "&" + node \
        + "&" + ttl \
        + "&" + type \
        + "&" + data \
        + "&" + self.transactionkey );

    try:
        myServer = Server(self.serveraddr)

        return myServer.ns.dns.update( self.keyname, timestamp, operation, zone,
view, node, ttl, type, data, hash.hexdigest() )

    except Fault, v:
        logging.error("Error occured while processing XML-RPC method call:",v)
        return -1

```

Here *keyname* is the name of the NSAPI key and *transactionkey* is the key's value used for authentication, *serveraddr* is the address and the port number of the virtual server where NameSurfer Suite is installed. Then there are parameters, such as zone name and the node from which RR will be deleted, type of the record and its value. TTL and view name are optional and those fields were left empty.

After the code of class *NSClient* was copied to iPython, the following command lines were run to initialize the object and to run the deletion function:

```

nsc =
NSClient("apitest","wfUseKRzeVn3H6N5HTCPI6Ft0A2R2e5OurMnh0fJlm0=", "http://19
2.168.0.6:8057/RPC2")

```

```

In [96]: print nsc.delRR("example.com","example.com","DNSKEY","257 3 10
Av85w29IRnssjihKNs6YyH4tyzIA9SZjKg98jdToV/x8zWqRRApER0wz6wpwlez0tg5VP/

```

HWm6AD4tQPHFJNBb6007vkGBt/Zxb5HuKkUUUnnXFoYuB7Xf3yOufnfvsa4ImigwVc1L63r2KUsMRpNd/44Bg5JGsOobXb3iqcBMvu+uA0=)

0

According to [3], the response code “0” received after script running means that there is no error. Verifying the result by exporting zone’s master file:

```
$ORIGIN example.com.
@           IN           SOA      ns1.example.com. katja.katja.com. (
                2012120529 28800 7200 604800 86400 )
                NS       ns1
;-          HTXT      test
;-          HTXT      hidden_text_for_test
;-          ALSO_NOTIFY ns3.example.com
;-          ALSO_NOTIFY 172.16.40.132
                NSEC3PARAM 1 0 1 71C0E6
host0      A         172.16.40.0
host1      A         172.16.40.1
;-          HTXT      "hidden text"
                TXT      text
                TXT      testtext
;-          HTXT      hidden_text_for_test
host2      A         172.16.40.2
host4      A         172.16.40.4
host5      A         172.16.40.5
```

The DNSKEY record was removed from the zone and the test was passed.

3.7.3 Web Service client

To be able to test the script against registry’s Web Service, Ficora provided access to the test environment with a personal user account and key.

[webservice]

user = Andreeva

key = xxxxxxxx

url = https://domainws.ficora.fi/fiDomainTest/DomainName.svc?wsdl

This test service has almost the same functionality as the production one except that additions and changes are not saved or propagated to the real

system. Registered Ficora clients can access this service free of charge. Testing was started by using some of the simple Web Service functions.

Testing basic Web Service functions

The first step to try could be pinging the Web Service; this can be done even without user authentication.

The Ping service is an interface with no message to be sent, and returns a simple response. Developers can use this service to test that the connection to the test/production server functions without generating messages and calculating a MAC code [14]

Initialization of the class *WSClient*:

```
wsc=WSClient("dnssec.fi", "Andreeva", "xxxxxxxxxxxxx",
"https://domainws.ficora.fi/fiDomainTest/DomainNamexxxxxxx.svc?wsdl")
```

Using the *ping* function of this class:

```
In [3]: print wsc.ping(5)
5
```

The *Ping* function returned “5” as it was expected, so the test is passed.

Another functionality that is easy to test was listing the user’s domains.

The Renew Domain Name service can be used to retrieve a list of domain names available for renewal. The service returns a list of the domain names in control of the ISP (i.e., the ISP holds their authorization key or ISP has control over domain name through name server) and are expiring within x days. [14]

This request already requires a MAC to be calculated, so the *listDomains* function gives a possibility to properly test user authentication.

As the service's name suggest, by default it does not return all the user's domains but only those that will expire within N amount of days. But by setting the N number high enough, for example 10 000, will in practice get the full list.

After initializing the class as in the previous test case, the printing domain list:

```
In [8]: wsc.listDomains()
Out[8]:
(renewable_domains_response){
  domains = None
  validation_errors = None
  code = True
}
```

As it can be seen from the example above, the response code is correct and there are no errors with MAC or authentication. This confirms that the function `__createMac` works correctly. The value "None" for the variable `domains` means that this Web Service user "Andreeva" has no domains. The test is passed and the function works as intended.

Registering domain

As the previous example showed, the first speed bump occurred - test user "Andreeva" does not have any domains. To resolve this issue, there seemed to be two options: register a new domain for this user with the Web Service function `registerDomain`, or ask Ficora's support to create a test domain for user "Andreeva".

Registering domain with the `registerDomain` function

While the domain registration was not in the scope of the script, the function *createRegisterDomainRequest* was created under the class *WSCClient* just to tackle the issue with the test domain.

```
def createRegisterDomainRequest(self) :
    """
    Domain registration function for debugging purpose
    """
    contactInfo = ["Katja Co", "Katja", "Andreeva", "Streetname 1" \
                  , "00200", "Helsinki", "123456789" \
                  , "katja@katja", "fi-FI", "Department", "FI"]

    request = self.client.factory.create('ns1:apply_request')

    request.name = self.zone
    request.valid_applicant_confirmation = "true"
    request.based_on_person_name = "false"
    request.person_name_registration_id = "0"
    request.person_name_registration_number = "0"
    request.domain_name_holder_company_type = "1"
    request.domain_name_holder_business_id = "12345671"
    request.domain_name_holder_person_id = "0"
    request.electronic_notification_approval = "true"
    request.data_publishing_approval = "false"

    nameserver = self.client.factory.create('ns1:nameserver')
    request.domain_name_holder_domicile_in_finland = "true"

    nameserver.name = "ns1.katja"
    nameserver.ipaddress = "1.1.1.1"
    request.nameservers.nameserver.append(nameserver)

    request.domain_validity_period_in_months = "36"

    contact = self.client.factory.create('ns1:contact')
```

```

contact.type.set("0")
contact.company = contactInfo[0]
contact.first_names.string.append(contactInfo[1])
contact.last_name = contactInfo[2]
contact.postal_address = contactInfo[3]
contact.postal_code = contactInfo[4]
contact.postal_office = contactInfo[5]
contact.phone = contactInfo[6]
contact.email = contactInfo[7]
contact.language_code = contactInfo[8]
contact.department = contactInfo[9]
contact.country = contactInfo[10]
contact.ContactId = "0"

request.contacts.contact.append(contact)

request.context.user_name = self.username

request.context.timestamp = self.__getTimeStamp()

hashvals = request.name \
            +request.valid_applicant_confirmation \
            +request.based_on_person_name \
            +request.person_name_registration_id \
            +request.person_name_registration_id \
            +request.domain_name_holder_company_type \
            +request.domain_name_holder_business_id \
            +request.electronic_notification_approval \
            +request.data_publishing_approval \
            +nameserver.name \
            +nameserver.ipaddress \
            +contact.type.get() \
            +".join(contactInfo) \
            +contact.ContactId \

```

```

        +self.username \
        +request.context.timestamp \
        +request.domain_validity_period_in_months \

    +request.domain_name_holder_domicile_in_finland

    mac = self.__createMac(hashvals)
    request.context.mac = mac

    return request

```

As it seen from the example below, this function requires a lot of additional information such as organization name and contact information that is not needed for sending a DS record.

```
In [12]: regdomainrequest = c.factory.create('ns1:apply_request')
```

```
In [13]: print regdomainrequest
```

```

(apply_request){
  name = None
  valid_applicant_confirmation = None
  based_on_person_name = None
  person_name_registration_id = None
  person_name_registration_number = None
  domain_name_holder_company_type = None
  domain_name_holder_business_id = None
  domain_name_holder_person_id = None
  electronic_notification_approval = None
  data_publishing_approval = None
  nameservers =
    (ArrayOfnameserver){
      nameserver[] = <empty>
    }
  contacts =
    (ArrayOfcontact){

```

```

        contact[] = <empty>
    }
    context =
        (context){
            user_name = None
            mac = None
            timestamp = None
        }
    domain_validity_period_in_months = None
    domain_name_holder_domicile_in_finland = None
}

```

For this reason it was decided to “hard-code” the required parameters into the function.

Ficora’s Web Service documentation section 3.3.2 describes how to calculate a MAC for *createRegisterDomainRequest* request [14]. Those values were stored in the variable *hashvals* and passed to the *__createMac* function.

Below is the request created by the function and ready to be sent to the Web Service:

```

//In [36]: wsc.createRegisterDomainRequest()
Out[36]:
(apply_request){
    name = "dnssec.fi"
    valid_applicant_confirmation = True
    based_on_person_name = False
    person_name_registration_id = "0"
    person_name_registration_number = "0"
    domain_name_holder_company_type = "0"
    domain_name_holder_business_id = None
    domain_name_holder_person_id = "0110810013"
    electronic_notification_approval = True
    data_publishing_approval = "false"
}

```

```

nameservers =
  (ArrayOfnameserver){
    nameserver[] =
      (nameserver){
        name = "ns1.katja"
        ipaddress = "1.1.1.1"
        ipv6address = None
      },
  }
contacts =
  (ArrayOfcontact){
    contact[] =
      (contact){
        type =
          (contact_type){
            value = "0"
          }
        company = "Katja Co"
        first_names =
          (ArrayOfstring){
            string[] =
              "Katja",
          }
        last_name = "Andreeva"
        postal_address = "Streetname 1"
        postal_code = "00200"
        postal_office = "Helsinki"
        phone = "123456789"
        email = "katja@katja"
        language_code = "fi-FI"
        department = "Department"
        organizationid = None
        country = "FI"
        ContactId = None
      },
  }

```

```

    }
context =
  (context){
    user_name = "Andreeva"
    mac = "0a6e5d3becaef1b3414bfd8cccba3cc1b00ba55a"
    timestamp = "2012-12-06T14:25:22.4600000Z"
  }
domain_validity_period_in_months = 36
domain_name_holder_domicile_in_finland = "true"
}

```

Then this request was sent to the Web Service with the function `sendDomainRequestregister` and the following response was received:

```

In [37]: wsc.sendDomainRegisterRequest()
Out[37]:
(apply_response){
  code = False
  timestamp = 2012-12-06 16:25:14.218857
  webdomain_validation_errors =
    (ArrayOferror_message){
      error_message[] =
        (error_message){
          description = "Current user can not be authenticated by the system"
          code = "AUTHENTICATION_FAILED"
        },
      }
    }
  contact_validation_errors = None
  nameserver_validation_errors = None
  nameserver_technical_errors = None
}

```

According to the error message, it seems that the user's authentication failed, which would mean in turn that the MAC value sent was not valid. But after

double-checking with Ficora's specification in Section 3.3.2 of the documentation [14] it did not reveal any mistakes in the code. There is a chance that some fields require either specifically empty string or a null value but that is not clarified in the documentation. In addition, it is not obvious in which cases numeric values should be of type "int" or just a string. Regardless of the domain registration problem, the logic of the MAC calculation in the script has been proven to be valid when tested with the *listDomain* function.

Troubleshooting domain creation with Ficora

After a few failed attempts to create a test domain under user account "Andreeva" using the web service API, an email explaining the situation was sent to Ficora and any possible assistance with message syntax troubleshooting was requested. It was suggested that Ficora could either check if the message and mac sent was correctly formatted or if Ficora could manually create a test domain under the account "Andreeva" to avoid whole domain creation troubleshooting. However, the reply from Ficora stated there were some challenges at the moment creating new domains in test system, and unfortunately the test web service interface does not work 1:1 as the production one. Ficora provided a testing tool called *macman* to help validation of the web service request message, but the Windows application did not run properly and could not be used for validation at the time of testing.

According to Web Service Testing [19], another big difference between test and production environment is that

It functions identically to the actual production version, but the changes and additions are not updated. [19]

The quoted description makes it sound like the test environment is "read"-only. Any changes made in it will not be recorded and taken into use. Thus, even if the registration would work and web service returned "True", if the domain does not actually get created there will not be a domain to test DS uploading.

Further input from the usage of *macman* tool is still been expected from Ficora in addition to a clarification if the web service is actually writable or read-only. However, since the time for this project was running out, it was decided to proceed with other parts of the script development.

Uploading DS records

DS upload testing was understandably quite crippled, as there was not a test domain available for uploading in Ficora's system. Nevertheless, it was possible to test that the upload message was created correctly and that no unexpected errors were received from the web service.

In the *ipython* prompt, the the DS upload message was created with the following DS key parameter:

```
In [2]: wsc =
WSClient("dnssec.fi", "Andreeva", "xxxxxxxx", "https://domainws.ficora.fi/fiDomainTest/
DomainNameWS_FicoraDomainNameWS.svc?wsdl")
```

```
In [3]: req = wsc.createAddDSRequest("60273 10 2
6d2489c4d3539b3eab512166be7d495befe5389fa8581b786de5c042d2ffed23")
```

```
In [4]: print req
(AddDsRecordRequest){
  Context =
    (context){
      user_name = "Andreeva"
      mac = "b8fb262ec39623bebf0195f57a04e2a651fd7a16"
      timestamp = "2012-12-06T22:43:26.8200000Z"
    }
  DomainName = "dnssec.fi"
  AuthorizationKey = "xxxxxxxx "
  DsRecords =
    (ArrayOfDsRecord){
      DsRecord[] =
        (DsRecord){
          KeyTag = "60273"
          Algorithm = "10"
          DigestType = "2"
          Digest =
            "6d2489c4d3539b3eab512166be7d495befe5389fa8581b786de5c042d2ffed23"
```

```

    },
  }
}

```

The response message from web service where there, of course, was not the test domain “dnssec.fi” was the following:

```

(WSAddDsRecordResponse){
  Code = False
  TimeStamp = 2012-12-07 00:44:06.818873
  WebdomainValidationErrors =
  (ArrayOferror_message){
    error_message[] =
    (error_message){
      description = "Current user can not be authenticated by the system"
      code = "AUTHENTICATION_FAILED"
    },
  }
}
}

```

The fact that the error was again authentication-related was somehow confusing, but it could be a generic error message when there is no matching data at the web service’s end.

3.7.5 Testing the final script

For the final script testing, some changes were needed to be made since there is no test domain where to upload the DS key.

From the script’s *main* function:

```

#Checking if DS key was uploaded successfully
  #if resp.Code == False:
  if True == False:

```

With this code the web service’s response for DS uploading is set to be always true and this gives possibility to finalize the testing. For simulating the existence of test domain, “dnssec.fi” was created in NameSurfer Suite and signed with ZSK and KSK. In addition, a new KSK was introduced to zone. The old key

signing key will be given as parameter to the script to be removed from the zone after the new DS is published. As the DS record could not be uploaded to the actual fi-root, an already published DS was obtained with the “dig” utility from the “real” “dnssec.fi” zone. After DS had uploaded when the script starts checking when the DS is published in the parent domain, it should see it immediately.

Parameters given to the script from command-line:

```
$python DSUploader.py -z dnssec.fi -k "59650 8 1
DF58C2A099A71F3348462472416743EA7BF5E9A3" -p 5 -d "257 3 10
Av858BIMmUbrwKQfS0j0DAcwWuwbg3+1N70BpC6Cpd8O/2A1rw97/sNfC6WR1QFd
VIBWx304wHdOXH1Q04YxBBi109b6uDkBFClAxA/nLh6p05iiJi4CdEp/rE/iREWy9XRD
iUtdzOJs1Jn75o078wXYASviSAwfm/oM0LHwX7Uosh8="
```

Entries from the script’s log during script execution:

```
INFO:root:DS request sent, polling parent for new DS
INFO:root:DS found from parent servers.
INFO:root:Waiting for 43200 seconds before removing DS from NameSurfer
```

Because the DS given to the script is exactly the same as the one in the zone “dnssec.fi”, the script immediately found it from the parent servers and began to wait double the DS record’s TTL (2*21600 seconds) until it removed the old key signing key from NameSurfer.

```
INFO:root:Key 257 3 10
Av858BIMmUbrwKQfS0j0DAcwWuwbg3+1N70BpC6Cpd8O/2A1rw97/sNfC6W
R1QFdVIBWx304wHdOXH1Q04YxBBi109b6uDkBFClAxA/nLh6p05iiJi4CdEp/r
E/iREWy9XRD iUtdzOJs1Jn75o078wXYASviSAwfm/oM0LHwX7Uosh8=
removed from NameSurfer
```

The script was executed successfully without any error or warnings given. This means that test is passed and the code works as expected (except minor issues related to domain existence in the Ficora system).

4. Conclusion

As increasingly critical services start to operate on the Internet and data becomes more sensitive, cyber criminals are motivated to abuse the vulnerabilities and weaknesses in the infrastructure raises. In the pre-DNSSEC world, there was no way to verify the authenticity of DNS responses. The figurative “phone-book” of the Internet was and still partly remains vulnerable to spoofing and “man-in-the-middle” attacks. DNS, being a so fundamental and essential part of the Internet infrastructure solution to securing it, cannot come soon enough.

Validation in DNSSEC is based on the “chain of trust”. An authority higher in hierarchy can always validate a lower one. But without a functioning key exchange implementing this “chain of trust” and DNSSEC itself is cumbersome. Agreeing on the standard methods and best practices of the key exchange and then DNS management tools implementing those is still work in progress. Although DNSSEC needs to be implemented starting from top to bottom in DNS hierarchy, there is already a need to think about the key exchange not only between registrars and registries, top level domains and second level domains, but also between any parent and child zone. This requires that DNS management tools universally implement standardized key exchange mechanisms.

This project was an attempt to bring the key exchange automation to Name Surfer Suite for zones with “.ff” parent following the best practices introduced in the theory part of this thesis, specifically in section “2.4 DNSSEC Key Exchange”. Automation was implemented as described in Section “3.3 Proposed functionality of the Key Exchange Mechanism”. The existing Ficora’s Web Service API was utilized in combination with DNSPython. Integration to NameSurfer was accomplished using its public API. The result of this work is the *DSUploader* – (Appendix 3) and its supporting libraries (Appendix 1 and Appendix 2).

Based on the results of the script testing phase, presented in Section “3.7 *Testing*”, further troubleshooting needs to be done in the web service test environment. Lack of the test data in the web service test environment prevents key upload testing so the script cannot be properly verified yet. However, the groundwork with the Web Service’s API implementation in Python is completed. Once the technical issues with the testing environment have been resolved and script’s key upload functionality is fully verified, the project can move to the next phase where the *DSUploader* script’s Python classes are used to integrate the key exchange mechanism within the NameSurfer Suite itself.

REFERENCES

- [1] Albitz, P.; Liu, C. 2006 "DNS and BIND" Fifth edition, USA: O'Reilly Media Inc.
- [2] Arends, R.; Huston, G.; Michaelson, G., Wallström, P. 2010 "Protocol Rolling Over DNSSEC Keys", The Internet Protocol Journal, Volume 13, No 1, Cisco Systems, http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_13-1/ipj_13-1.pdf
- [3] Bound, J., Rekhter, Y., Thompson, S., Vixie, P. April, 1997 "Dynamic Updates in the Domain Name System (DNS UPDATE)" (RFC 2136), <http://www.ietf.org/rfc/rfc2136.txt>
- [4] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. 15 March, 2001 "Web Services Description Language (WSDL) 1.1", W3C, <http://www.w3.org/TR/wsdl>
- [5] Courtay, O.;Guette, G. 18 January, 2005 "Requirements for Automated Key Rollover in DNSSEC", <http://tools.ietf.org/html/draft-ietf-dnsop-key-rollover-requirements-02>
- [6] Gieber, R.; Kolkman, O. September 2006 "DNSSEC Operational Practices" (RFC 4641), NLnet Labs, <http://www.ietf.org/rfc/rfc4641.txt>
- [7] Honnelbeck, S. May 2001 "Extensible Provisioning Protocol", Verisign, Inc, <http://tools.ietf.org/html/draft-hollenbeck-epp-00>
- [8] Honnelbeck, S. September 2002 "Generic Registry-Registrar Protocol Requirements" (RFC 3375), Verisign, Inc, <http://tools.ietf.org/html/rfc3375>
- [9] Internet System Consortium, 2009 "BIND 9.5 Administrator Reference Manual", Internet System Consortium, <http://www.isc.org/software/bind/documentation/arm95#id2543755>
- [10] Jansen, J. 11 September, 2008 "DNSSEC Key Maintenance Analysis" version 1.0, NLnet Labs, http://ws.edu.isoc.org/workshops/2008/cctldams/Documentation/DNSSEC_Key_maintenance.pdf
- [11] Kolkman, O. 4 July, 2009 "DNSSEC How to, a tutorial in disguise" Revision 134, http://nlnetlabs.nl/publications/dnssec_howto/
- [12] Lehtonen, J. December, 2011 "Reference Manual Nixu NameSurfer Suite 7 Series", Nixu Software.
- [13] Lehtonen, J. December, 2011 "Reference Manual Nixu NameSurfer Suite 7 Series – Appendix A NSAPI RPC References", Nixu Software.

- [14] Lehvälaiho, K. May, 2011 “Ficora Domain Name System Web Service functionality service description”, Ficora.
- [15] Mohan, R. 06 July, 2010 “Five Strategies for Flawless DNSSEC Key Management and Rollover”, SecurityWeek, <http://www.securityweek.com/five-strategies-flawless-dnssec-key-management-and-rollover>
- [16] Mockapetris, P., November, 1987 “Domain Names – implementation and specification” (RFC 1035), Network Working Group, <http://www.ietf.org/rfc/rfc1035.txt>
- [17] RFC 3225, <http://www.ietf.org/rfc/rfc3225.txt>
- [18] RFC 3658, <http://tools.ietf.org/html/rfc3658>
- [19] RFCs 4033-4035, <http://www.rfc-archive.org/index-rfc-4001-4100.php>
- [20] TFC 3655, <http://tools.ietf.org/rfc/rfc3655.txt>
- [21] “Testing of Web Service” (Ficora Web Service) Consulted on 10.12.2012. <https://domain.fi/info/en/index/palveluntarjoajille/webservice/testaaminen.html>

Appendix 1 “FicoraWS.py” code

```

from suds.client import Client
import time, xml.utils.iso8601, hashlib, logging

class WSClient:

    def __init__(self, zone, username, authorizationkey, url) :
        self.username = username
        self.authorizationkey = authorizationkey
        self.url = url
        self.zone = zone
        try:
            self.client = Client(url)
        except:
            logging.error("Failed to connect to host %s"
%self.url)

    #Create webservice request for adding new DS key to root
    def createAddDSRequest(self, dsr) :
        """
        Create new DsRequest
        @param dsr: DS key
        @type dsr: string
        @return: DsRequest object
        """
        dsrec = self.client.factory.create('ns1:DsRecord')
        ds = dsr.split(" ")

        dsrec.KeyTag = ds[0]
        dsrec.Algorithm = ds[1]
        dsrec.DigestType = ds[2]
        dsrec.Digest = ds[3]

        request =
self.client.factory.create('ns1:AddDsRecordRequest')
        request.Context.user_name = self.username
        request.AuthorizationKey = self.authorizationkey
        request.DomainName = self.zone

        request.DsRecords.DsRecord.append(dsrec)

        request.Context.timestamp = self.__getTimeStamp()

        hashvals =
''.join([self.username,request.Context.timestamp,self.zone, \
        ds[0],ds[1],ds[2],ds[3]])
        request.Context.mac = self.__createMac(hashvals)

        return request

    #Create MAC authentication value using the values
    def __createMac(self, vals) :
        m = hashlib.sha1()
        vals = vals+self.authorizationkey
        vals.encode('utf-8')
        m.update(vals)
        return m.hexdigest()

```

```

def __getTimeStamp(self) :
    ts = xml.utils.iso8601.tostring(time.time())
    ts = ts[:22]+"0000"+ts[22:]
    return ts

#Send the request to webservice
def addDS(self, ds) :
    """
    Sends an AddDsRecordRequest to web service
    @param r: An AddDsRecordRequest
    @type r: suds.sudsobject.AddDsRecordRequest
    @return: WSAddDsRecordResponse
    """
    r = self.createAddDSRequest(ds)
    return self.client.service.AddDsRecord(r)

def listDomains(self) :

    request =
self.client.factory.create('ns1:renewable_domains_request')
    request.DaysToExpiration = "10000"

    request.context.user_name = self.username

    request.context.timestamp = self.__getTimeStamp()
    hashvals = '10000'+self.username+request.context.timestamp
    mac = self.__createMac(hashvals)
    request.context.mac = mac

    return self.client.service.RenewableDomains(request)

def ping(self, i) :
    return self.client.service.Ping(i)

def createRegisterDomainRequest(self) :
    """
    Domain registration function for debugging purpose
    """
    contactInfo = ["Katja Co", "Department",
"Katja","Andreeva","Streetname 1" \
                    , "00200","Helsinki","123456789" \
                    , "katja@katja","fi-FI","FI"]

    request = self.client.factory.create('ns1:apply_request')

    request.name = self.zone
    request.valid_applicant_confirmation = True
    request.based_on_person_name = False
    request.person_name_registration_id = "0"
    request.person_name_registration_number = "0"
    request.domain_name_holder_company_type = "0"
    #request.domain_name_holder_business_id="12345671"
    request.domain_name_holder_person_id="0110810013"
    request.electronic_notification_approval = True
    request.data_publishing_approval = "false"

    nameserver = self.client.factory.create('ns1:nameserver')
    request.domain_name_holder_domicile_in_finland = "true"

    nameserver.name = "ns1.katja"

```

```

nameserver.ipaddress = "1.1.1.1"
request.nameservers.nameserver.append(nameserver)

request.domain_validity_period_in_months = 36

contact = self.client.factory.create('nsl:contact')

contact.type.set("0")
contact.company = contactInfo[0]
contact.department = contactInfo[1]
contact.first_names.string.append(contactInfo[2])
contact.last_name = contactInfo[3]
contact.postal_address = contactInfo[4]
contact.postal_code = contactInfo[5]
contact.postal_office = contactInfo[6]
contact.phone = contactInfo[7]
contact.email = contactInfo[8]
contact.language_code = contactInfo[9]
contact.country = contactInfo[10]
#contact.ContactId = "0"

request.contacts.contact.append(contact)

request.context.user_name = self.username

request.context.timestamp = self.__getTimeStamp()

hashvals = request.name \

+str(request.valid_applicant_confirmation) \
    +str(request.based_on_person_name) \
    +request.person_name_registration_id \
    +request.person_name_registration_number
\
    +request.domain_name_holder_company_type
\
    +request.domain_name_holder_person_id \

+str(request.electronic_notification_approval) \
    +request.data_publishing_approval \
    +nameserver.name \
    +nameserver.ipaddress \
    +contact.type.get() \
    +''.join(contactInfo) \
    +self.username \
    +request.context.timestamp \

+str(request.domain_validity_period_in_months)

    mac = self.__createMac(hashvals)
    request.context.mac = mac

    return request

def sendDomainRegisterRequest(self):
    r = self.createRegisterDomainRequest()
    return self.client.service.Apply(r)

```

Appendix 2. “NSAPI.py” code.

```

from xmlrpclib import Server, Fault
import hashlib, logging
import time

class NSClient:

    def __init__ (self, keyname, transactionkey, serveraddr) :

        self.keyname = keyname
        self.transactionkey = transactionkey
        self.serveraddr = serveraddr

    def __modifyRR(self, operation, zone, node, type, data, ttl="",
view="") :
        procedurename = "ns.dns.update"
        timestamp = str(int(time.time()))

        hash = hashlib.sha256()
        hash.update( procedurename \
            + "&" + self.keyname \
            + "&" + timestamp \
            + "&" + operation \
            + "&" + zone \
            + "&" + view \
            + "&" + node \
            + "&" + ttl \
            + "&" + type \
            + "&" + data \
            + "&" + self.transactionkey );

        try:
            myServer = Server(self.serveraddr)

            return myServer.ns.dns.update( self.keyname,
timestamp, operation, zone, \
            view, node, ttl, type, data, hash.hexdigest() )

        except Fault as err:
            logging.error("Error occured while processing XML-RPC
method call: %d" % err.faultCode)
            logging.error("Fault string: %s" % err.faultString)

            return -1

    def addRR(self, zone, node, type, data, ttl="", view="") :
        return self.__modifyRR("ADD",zone,node,type,data,ttl,view)

    def delRR(self, zone, node, type, data, ttl="", view="") :
        return self.__modifyRR("DELETE", zone, node, type, data,
ttl, view)

```

Appendix 3. “DSUploader.py” code

```

import sys, time, dns, dns.query, dns.resolver, ConfigParser,
argparse, logging
from FicoraWS import WSClient
from NSAPI import NSClient

class DNSChecker :

    def __init__(self, zone, parentdomain, parentservers=None) :
        self.zone = zone
        self.parentdomain = parentdomain
        self.parentservers = parentservers

        if not self.parentservers:
            self.parentservers =
self.__getParentServers(self.parentdomain)
        else:
            self.parentservers = parentservers.split(",")

    #Method to resolve parentdomain's nameservers in case those are
not provided in cfg file
    def __getParentServers(self, domain) :
        parentservers = []
        answer = dns.resolver.query(domain,dns.rdatatype.NS)
        for i in answer.response.additional:
            if i.rdtype == 1:
                parentservers.append(i.items[0].to_text())
        return parentservers

    #Query master for existence of new DS
    def checkDS(self, ds) :
        found = False
        q = dns.message.make_query(self.zone, dns.rdatatype.DS)
        for server in self.parentservers:
            if found == True: break
            response = dns.query.udp(q,server,30)
            for set in response.answer:
                for dsr in set:
                    if dsr.to_text().lower() == ds.lower():
                        found = True
                        break

        return found

    def getRRRTTL(self, rr) :
        q = dns.message.make_query(self.zone,
dns.rdatatype.from_text(rr))
        ttl = None
        for server in self.parentservers:
            response = dns.query.udp(q,server,30)
            for set in response.answer:
                ttl = set.ttl
            if ttl is None:
                for set in response.additional:
                    ttl = set.ttl
            if ttl is not None: break
        return ttl

def main():

```

```

#Enabling logging
logging.basicConfig(filename="DSUploader.log",
level=logging.INFO)

#Reading more or less static parameters from config-file
config = ConfigParser.RawConfigParser()
config.read("default.cfg")

username = config.get("webservice","user")
authorizationkey = config.get("webservice","key")
url = config.get("webservice","url")
parentdomain = config.get("domain","parent")
parentservers = config.get("domain","parentservers")
if len(parentservers) < 1:
    parentservers = None
nsaddress = config.get("namesurfer","serveraddr")
nskeyname = config.get("namesurfer","keyname")
nskey = config.get("namesurfer","transactionkey")

#More often changing parameters such as zone and key
#are read from command-line arguments:
parser = argparse.ArgumentParser()
parser.add_argument("-z", dest="zone", required=True, help =
"Zone name")
parser.add_argument("-k", dest="dskey", required=True, help =
"DS key as \
    space delimited text fields \'KEYTAG ALGORITHM DIGESTTYPE
DIGEST\'")
parser.add_argument("-p", dest="pollfreq", default=1800,
type=int, help = \
    "Polling frequency for checking the root for new DS.
Default 1800 seconds.")
parser.add_argument("-d", dest="deletekey", help = "Old key to
be deleted from \
    NameSurfer (OPTIONAL).")

cli = parser.parse_args()
zone = cli.zone
ds = cli.dskey
pf = cli.pollfreq
oldkey = cli.deletekey

#First get the current DS keys TTL - this will be needed later
checker = DNSChecker(zone, parentdomain, parentservers)
ttl = checker.getRRRTTL("DS")
#Creating new webservice client object
ws = WSClient(zone, username, authorizationkey, url)
#Sending the new DS key to webservice
resp = ws.addDS(ds)
#Checking if DS key was uploaded successfully
if resp.Code == False:
    logging.error("Error Response: %s" \

%resp.WebdomainValidationErrors.error_message[0].description)
else :
    logging.info("DS request sent, polling parent for new DS")
    while True:
        if checker.checkDS(ds) is True: break

```

```
logging.info("DS not found. Checking again after %s
seconds" %pf)
    time.sleep(pf)
    logging.info("DS found from parent servers.")
    #here comes nasu script to delete old DS
    if oldkey is not None and len(oldkey)>0:
        logging.info("Waiting for %s seconds before removing
DS from NameSurfer" %(ttl*2))
        time.sleep(ttl*2)
        nsc = NSClient(nskeyname, nskey, nsaddress)
        if nsc.delRR(zone, zone, "DNSKEY", oldkey) is not -
1:
            logging.info("Key \'%s\' removed from
NameSurfer" %oldkey)
if __name__ == "__main__":
    main()
```