BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2018

Bc. Martin Sehnoutka

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

AUTOMATIC VERIFICATION OF SOFTWARE PACKAGES WITH HELP OF DNS

AUTOMATICKÉ OVĚŘOVÁNÍ SOFTWAROVÝCH BALÍKŮ ZA POMOCÍ DNS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. Martin Sehnoutka
SUPERVISOR VEDOUCÍ PRÁCE	doc. Ing. Jan Jeřábek, Ph.D.

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Martin Sehnoutka ***ID:*** 154865
Ročník: 2 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Automatické ověřování softwarových balíků za pomocí DNS

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s protokolem DNS, rozšířením DNSSEC, ostatními rozšiřujícími záznamy a asymetrickou kryptografií. Analyzujte a porovnejte způsoby, jakými jsou podepsány a verifikovány softwarové balíky v Linuxových distribucích. Zaměřte se především na balíky formátu RPM a distribuci Fedora. Zaměřte se na možné využití systému DNS pro účely automatické verifikace integrity a autenticity softwarových balíků a navrhněte konkrétní řešení, které s použitím dat uložených v DNS databázi ověří autentičnost veřejného klíče. Zvolené řešení implementujte ve k tomu účelu vhodném programovacím jazyce. Vytvořte testovací prostředí, které použijete pro zhodnocení zátěže a správnosti vašeho řešení.

DOPORUČENÁ LITERATURA:

[1] JEŘÁBEK, Jan. Pokročilé komunikační techniky. verze 2017. Brno: Vysoké učení technické v Brně, 2015. ISBN 978-80-214-4713-4.

[2] DNF package manager (software) [online]. GitHub: 2017. Poslední změna 21.8.2017 [cit. 22.8.2017]. Dostupné z: https://github.com/rpm-software-management/dnf

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 21.5.2018

Vedoucí práce: doc. Ing. Jan Jeřábek, Ph.D.
Konzultant: Ing. Tomáš Hozza

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRACT

This master's thesis deals with the problem of secure software distribution. An enhancement for the current state is proposed with the help of the domain name system which is used as a storage for verification keys. These keys are necessary for integrity verification of packages downloaded using a package manager. Furthermore, an extended version is proposed, which takes into account also repository metadata. Both versions are implemented using the Python programming language and integrated into the dnf package manager. This implementation is then tested in a virtual environment, discussed and evaluated in terms of its performance.

KEYWORDS

Software distribution, DNS, DNSSEC, package manager, PGP, Fedora, dnf, Virtualization, Ansible

ABSTRAKT

Tato diplomová práce se zabývá problémem bezpečné distribuce software. Je navrženo zlepšení s pomocí doménového systému, který je použit pro uložení verifikačních klíčů, potřebných pro ověření integrity balíků stáhnutých pomocí správce balíků. Navíc je navržena rozšířená verze, které se zabývá zabezpečením metadat repositářů. Obě verze jsou implementovány v jazyce Python a integrovány do správce balíků dnf. Tato implementace je otestována ve virtuálním prostředí, diskutována a zhodnocena z hlediska způsobené zátěže.

KLÍČOVÁ SLOVA

Distribuce software, DNS, DNSSEC, správce balíků, PGP, Fedora, dnf, virtualizace, Ansible

SEHNOUTKA, Martin. *Automatic verification of software packages with help of DNS*. Brno, 2018, 83 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications. Advised by doc. Ing. Jeřábek Jan, Ph.D.

Vysázeno pomocí balíčku `thesis` verze v4.41; `http://latex.feec.vutbr.cz`

DECLARATION

I declare that I have written the Master's Thesis titled "Automatic verification of software packages with help of DNS" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno . . . .

author's signature

ACKNOWLEDGEMENT

Rád bych poděkoval vedoucímu diplomové práce panu doc. Ing. Janu Jeřábkovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Rovněž bych rád poděkoval konzultantovi panu Ing. Tomáši Hozzovi za zpětnou vazbu k mé práci.

Brno . . . .

author's signature

SIX
research centre
sensor, information and communication systems

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

ACKNOWLEDGEMENT

Research described in this Master's Thesis has been implemented in the laboratories supported by the SIX project; reg. no. CZ.1.05/2.1.00/03.0072, operational program Výzkum a vývoj pro inovace.

Brno . . . .

author's signature

MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

2007–13
OP Výzkum a vývoj
pro inovace

Tato práce vznikla jako součást klíčové aktivity KA6 - Individuální výuka a zapojení studentů bakalářských a magisterských studijních programů do výzkumu v rámci projektu OP VVV Vytvoření double-degree doktorského studijního programu Elektronika a informační technologie a vytvoření doktorského studijního programu Informační bezpečnost, reg. č. CZ.02.2.69/0.0/0.0/16_018/0002575.

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

Projekt je spolufinancován Evropskou unií.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

INTRODUCTION

Linux distributions are trying to make their desktop versions more user-friendly by using nice looking interface and simplifying common tasks, like software installation, user management or online accounts management. But these changes must not sacrifice security of the whole system. For example verification key, a concept from public-key cryptography should be verified before usage even though a common user has no clue about its existence. The same applies to the cloud environment, where hundreds of operating system instances can be deployed at the same time using automation tools. These instances should not skip verification, just because there is no one to perform it.

In this thesis I propose a Domain Name System (DNS) aided technique for public key verification. In Linux distributions, it is common to use public keys for verifying integrity of software packages. However, the problem of integrity verification is non-trivial as on the Internet anyone can claim to be someone else. Thus it is necessary to prove authenticity of a given public key, in other words, that it indeed belongs to the identity it claims to belong. Finally, the proof of the authenticity of a given information has to arrive from a trusted source. In this case, the root zone key is used as a trust anchor and the Doman Name System SECurity extension (DNSSEC) decentralized chain of trust is used to extend the trust from root zone key to a signing key, thus the key can be trusted as much as the DNSSEC system itself. Using the same technique a method for automatic key revocation and repository metadata verification is proposed.

In the text, the necessary theoretical background is described, such as asymmetric cryptography, DNS, software distribution, and development. Based on this a solution to the problem stated in this thesis assignment is proposed and implemented. In the end, performance is discussed as well as possibilities of server-side deployment.

1 THEORETICAL BACKGROUND

In this chapter, all necessary theoretical concepts are introduced starting from symmetric and asymmetric cryptography, followed by Pretty Good Privacy, introduction to domain name system and domain name system security extension and finally package formats and managers used in Linux distributions.

1.1 Cryptography

When there are two parties which want to share a secret information, they can either meet in person or they can send this information through a public channel, such as the Internet or a post service. In case of the public channel, the information is put at risk of being revealed. This is where cryptography comes to rescue; it provides the possibility to hide the secret from third parties, but still, allows for sending it over the channel.

1.1.1 Terminology

Cryptography is the science of hiding secrets from third parties.

Cryptoanalysis is the science of breaking cryptosystems. It is the opposite of cryptography. Although it might seem that breaking something is not an academic discipline, it is actually of critical importance nowadays, because without cryptoanalysis, we would never know if our cryptosystem is really secure. In other words, cryptosystem is considered secure, if there are no known attacks. [5]

Plain text is the original message in human or machine-readable format. For example, a text in Czech language or a JSON encoded structure. Every symbol in this message is an element of a plain text alphabet.

Cipher text is the processed message, which is not readable without previous decryption. This message has its own alphabet, which can be, but does not have to be, the same as the plain text alphabet. For instance, every message, either encrypted or plain text, on a computer will be stored as a series of bits. Each bit comes from the set $0, 1$. Thus the alphabet is the same of both encrypted and plain messages.

1.1.2 Symmetric cryptography

The basic property of symmetric cryptography is that both sides use the same key to encrypt and decrypt the communication. This type of cryptography is very effective in terms of performance and it is also system proven by centuries of usage. Unfortunately shared key cannot be used to verify authenticity of any message.

1.1.3 Asymmetric cryptography

In asymmetric cryptography or in other words public-key cryptography as opposed to symmetric, a key pair is used instead of a single symmetric key. One key is kept private and one can be publicly available, thus the names private key k_{pr} and public key k_{pub}. The idea is depicted in figure 1.1.

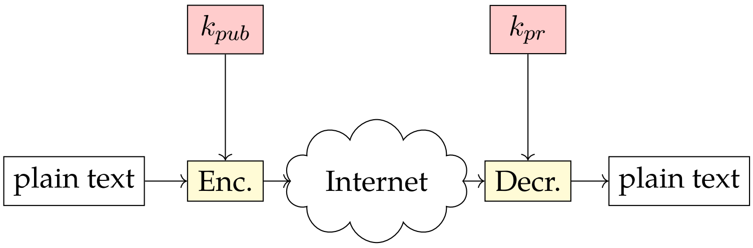

Fig. 1.1: Asymmetric cryptography

Construction of such public-key schemes require a one-way function f as defined in equation 1.1[5]:

$$y = f(x) \textit{ is easily computable,} \tag{1.1}$$

$$x = f^{-1}(y) \textit{ is very hard to compute.} \tag{1.2}$$

The result of this scheme is that only one user or entity posses the private key and only this particular entity can prove its identity using this key. This property is called nonrepudiation and cannot be achieved using symmetric algorithms. Thus asymmetric cryptography can be used for both message encryption and signing (Section 1.1.8). Another advantage is the possibility of constructing key establishment protocols, which are very useful when communicating over an insecure channel, such as the Internet.

Two well-known problems used in public-key cryptography schemes are discrete logarithm, used in Diffie-Hellman key exchange, and integer factorization problem used in Rivest, Shamir, Adleman (RSA). Elliptic curve schemes are gaining in popularity nowadays, e.g. in Bitcoin cryptocurrency network. All three schemes can be used for key establishment, nonrepudiation by using digital signature and message encryption.

Finally, a single public key is not enough, because there is no way to prove, that this key belongs to its owner. That is why we need to construct chains of trust, where we trust one entity (e.g. a DNS root server or certification authority) and if this entity verifies an identity of another one, we trust it as well. This principle will allow us to automatically verify software packages using DNSSEC chain of trust as described in section 1.2.6.

1.1.4 Integer factorization and RSA cryptosystem

One of the one-way functions used in public-key cryptography is the integer factorization problem. It is easy to multiply two prime numbers, but given the result of this multiplication, it is very difficult to compute the prime numbers. The first step is to construct a cryptosystem [5].

1. Choose two large prime numbers p and q

2. Compute the modulus n:
$$n = p \cdot q. \tag{1.3}$$
3. Compute $\Phi(n)$:
$$\Phi(n) = (p-1)(q-1). \tag{1.4}$$
4. Select the public exponent $e \in \{1, 2, ..., \Phi(n) - 1\}$ such that:
$$gcd(e, \Phi(n)) = 1. \tag{1.5}$$
5. Compute the private key d, so that this equation holds:
$$d \cdot e = 1 \; mod \; \Phi(n). \tag{1.6}$$

The result of this construction is the key pair of public key: $k_{pub} = (n, e)$ and private key: $k_{pr} = (d)$. With these key, we can encrypt an integer x, where $0 < x < n - 1$:

$$y = x^e \; mod \; n. \tag{1.7}$$

The decryption works on the same principle:

$$x = y^d \; mod \; n. \tag{1.8}$$

It is common for the numbers x, y, d and n to be at least 2048 bits at the time of writing this thesis [6]. Finally, the problem of braking this cryptosystem is in the equation 1.3, where two large prime numbers are multiplied and the attacker knows only the result number n.

1.1.5 Discrete logarithm problem and Diffie–Hellman key exchange

Another hard problem, that can be used to construct asymmetric cryptosystems is the discrete logarithm problem. The basic structure here is a group $(\mathbb{Z}_p^*, \cdot)$, where $\mathbb{Z}^*$ denotes a set of positive integers without zero and p is a prime. This group has special properties beneficial for the use in cryptography [5]. One of these properties is the existence of so-called generator element α:

$$\alpha^i = j \; mod \; p; \; where \; i \in \{1, 2, ...p - 1\}, \tag{1.9}$$

which can be used to "generate" all elements j in the set of $\mathbb{Z}_p^*$. More formally, the element α is said to have order equal to the cardinality of $\mathbb{Z}_p^*$:

$$ord(\alpha) = p - 1 = |\mathbb{Z}_p^*|. \tag{1.10}$$

This property is used in the Diffie-Hellman key exchange (DHKE), which start by choosing p and α and publishing these two values. The purpose of the key exchange is to construct a secret key using an unsecured channel. These two entities are commonly referred to as Alice and Bob. DHKE protocol between Alice and Bob is depicted in Figure 1.2 and the key derived on both sides it the same, because:

$$B^a = (\alpha^b)^a = \alpha^{ab} = (\alpha^a)^b = A^b \; mod \; p. \tag{1.11}$$

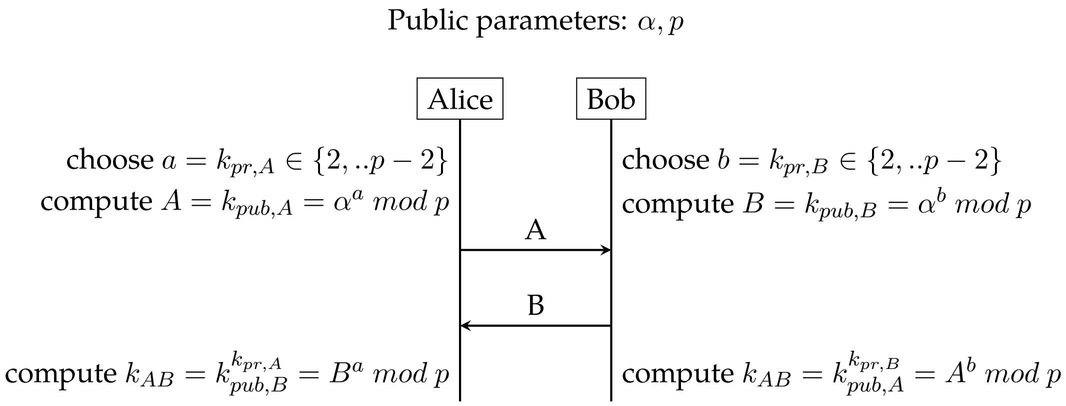
Public parameters: α, p
Alice
Bob
choose $a = k_{pr,A} \in \{2, ..p-2\}$
compute $A = k_{pub,A} = \alpha^a \, mod \, p$
choose $b = k_{pr,B} \in \{2, ..p-2\}$
compute $B = k_{pub,B} = \alpha^b \, mod \, p$
A
B
compute $k_{AB} = k_{pub,B}^{k_{pr,A}} = B^a \, mod \, p$
compute $k_{AB} = k_{pub,A}^{k_{pr,B}} = A^b \, mod \, p$

Fig. 1.2: Diffie-Hellman key exchange

With this setup, an attacker knows α, p, A and B and his task is to derive a and b. He can do it using the discrete logarithm, which will be very hard to compute:

$$a = log_\alpha A \, mod \, p. \tag{1.12}$$

DHKE is not the only usage of the discrete logarithm problem, it can also be used for Elgamal encryption and the Digital Signature Algorithm (DSA).

1.1.6 Elliptic curves

Cryptosystems based on elliptic curves use the same discrete logarithm problem as described in the previous section, but the group is defined over an elliptic curve as opposed to a set of integers. The elliptic curve is a set of points (x, y) which fulfill equation:

$$y^2 = x^3 + a \cdot x + b \, mod \, p \tag{1.13}$$

together with an imaginary point of infinity P_∞.

In order to create a group an operation "$+$" is defined over points in the curve.

Of course in cryptography, a finite structure is needed, so in this case the curve is defined over a prime field, where p is the prime. The number of points on the curve is denoted as $\#E$.

Finally the difficult problem in this case is finding an integer d such that:

$$\underbrace{P + P + .. + P}_{\text{d times}} = dP = T, \tag{1.14}$$

where P is a generator point on the curve and T is a point resulting from successive application of the "$+$" operation on P with itself.

With this knowledge, key-exchange protocol over elliptic curves can be defined in a similar fashion as the DHKE. Figure 1.3 shows the required steps in Elliptic Curve Diffie–Hellman Key Exchange (ECDH). Again using the group properties it is possible to prove, that keys on both sides will be the same [5].

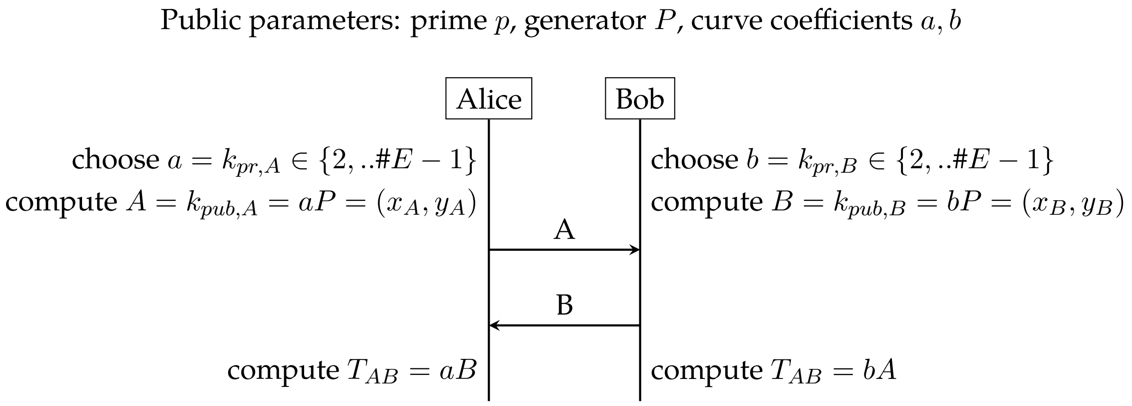
Public parameters: prime p, generator P, curve coefficients a, b
Alice
Bob
choose $a = k_{pr,A} \in \{2, ..\#E - 1\}$
compute $A = k_{pub,A} = aP = (x_A, y_A)$
choose $b = k_{pr,B} \in \{2, ..\#E - 1\}$
compute $B = k_{pub,B} = bP = (x_B, y_B)$
A
B
compute $T_{AB} = aB$
compute $T_{AB} = bA$

Fig. 1.3: Elliptic Curve Diffie–Hellman Key Exchange

One of the biggest advantages of ECC is that it requires shorter keys than classic cryptography based on integer factorization or discrete logarithm problem. For long-term security one should use at least 163 bits key [7].

1.1.7 Hash functions

A hash function is a function which takes arbitrarily long input (x) and produces a number of fixed length (y)[6]. Its main purpose is to ensure integrity of the given input x:

$$y = h(x). \tag{1.15}$$

It is important to note that it does not take any keys, so the hash value is not bound to any identity, but it is nevertheless an important building block for more complicated cryptographic protocols, such as Transport Layer Security (TLS) or Pretty Good Privacy (pgp). There are a lot of different types of hash functions, but in this thesis, I will focus on cryptographic hash functions. These are characterized by some special properties. First of all the function should be only one-way, meaning that it is impossible to obtain the original input from a hash value (so-called preimage resistance [5]). It should also be very hard to find a collision, that is two different messages with the same hash value. But it is important to mention, that there is simply no such function without collisions since the function is mapping an arbitrarily long input to a fixed length output. Finally given a message it should be hard to find a different one with the same hash value (second preimage resistance [5]). For example: given an Extensible Markup Language (XML) document, it should not be possible to find a different one with the same hash in a reasonable time frame.

1.1.8 Signatures

Signatures are a mechanism that is used to prove authorship of a message. It uses public-key cryptography, because in this scheme only one entity is in possession of a private key, thus authenticity of the message can be verified and cannot be rejected by the private key owner (non-repudiation). Real-world signatures algorithm works by first hashing the message and then ”encrypting” it with the private key. The signature can be verified using a public key (Figure 1.4).

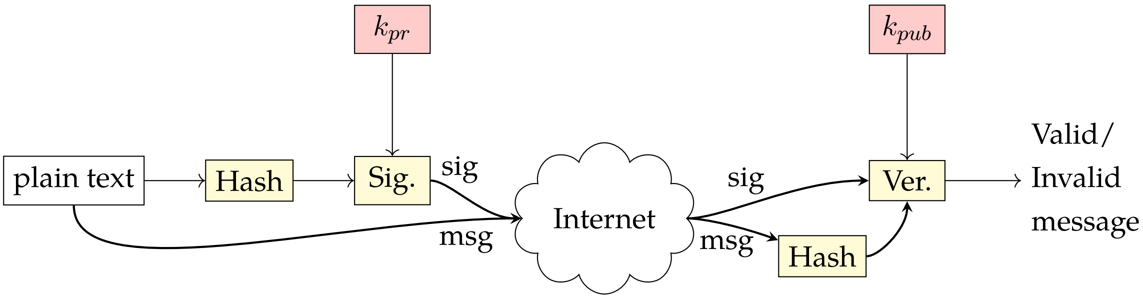

Fig. 1.4: Signatures in asymmetric cryptography

1.1.9 Certificates

In the last few sections many algorithms of cryptography were introduced, but how can a user trust a public key found on the Internet? Certificates are a way to bind an identity to a public key [6]. The identity is verified by a certification authority, which in turn digitally signs the identity information together with a public key forming a certificate. Users obtain a trusted copy of a public key of the certification authority together with their operating system or internet browser. Using this trust, they can in turn trust the identity written in the certificate.

1.1.10 Pretty Good Privacy

pgp is a computer software that provides end-to-end privacy and authenticity for data communication, such as electronic mail. It is defined as a protocol in RFC 4880 [14]. PGP uses both symmetric and asymmetric cryptography. The former one is used for message encryption. The second one is used for signing and secure transmission of a symmetric decryption key. The whole protocol is depicted in figure 1.5

Although it was designed as a program for message encryption, it can be also used as a general purpose tool for encryption and signing. RPM packages are signed using PGP keys.

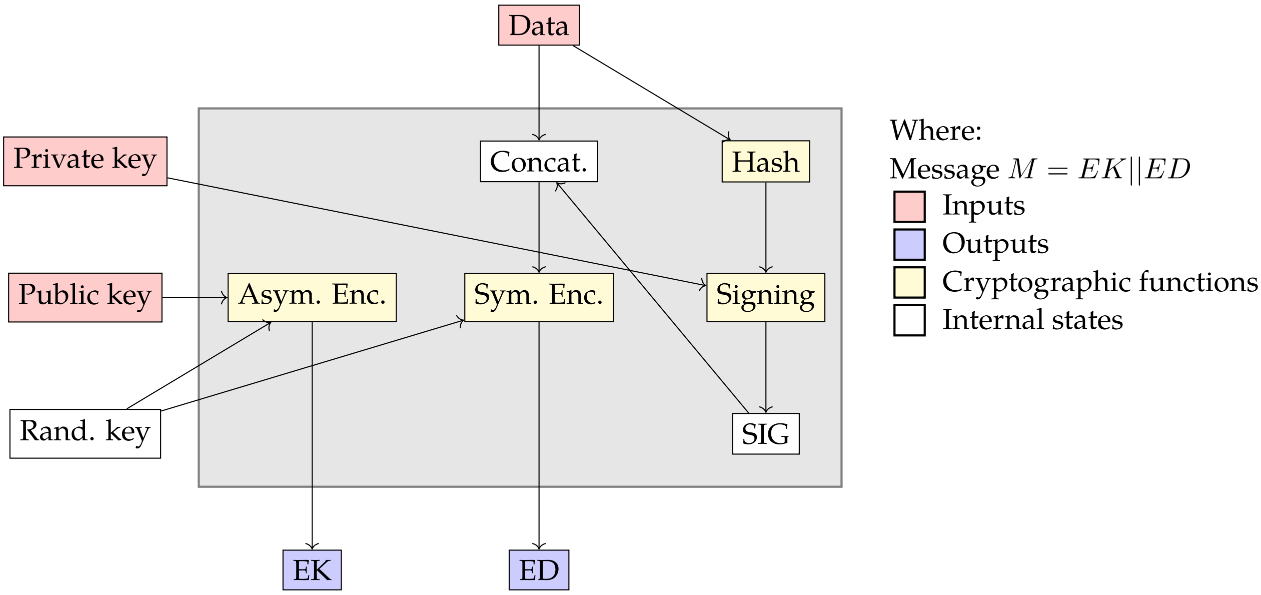

Fig. 1.5: PGP message encryption

1.2 Domain Name System

The most prominent feature of the Domain Name System (DNS) is a translation from easy to memorize names to numerical addresses which are then used by a computer. Before DNS was introduced, files like */etc/hosts*[3] could have been used, where it is possible to define mappings from names to IP addresses manually; this file can be distributed using FTP protocol so that all nodes in a network know the names. This approach, however, suffers from its monolithic nature and is not suitable for large-scale networks such as the Internet. A new, decentralized and scalable protocol had to be introduced. The DNS meets these requirements and creates a hierarchical naming scheme for computers available over the Internet or another network.

Very simple example of a DNS usage is depicted in Figure 1.6. Let us explore what happens when the client wants to communicate with the ”example.com” web server. The client’s software issues a DNS query which is sent to the local DNS resolver. The resolver takes the name in the query and it starts sending it to the authoritative servers or it splits the domain name into labels [1] and contacts appropriate authoritative DNS servers while sending only minimal amount of labels. It starts the lookup with the root server whose IP address is already known [2]. The root server sends an IP address of a server responsible for the ”.com” domain. The same happens in case of the server responsible for the ”.com” domain; when the server receives a query for ”example.com”, it responds with an address of a server responsible for the domain. Finally, the authoritative server for ”example.com”

---

[1] Assuming that it follows RFC 7816 [4]

[2] IP addresses of the root servers can be found in the source code for DNS resolvers. For example in *lib/dns/rootns.c* in Bind DNS server

domain sends the IP address of the ”example.com” web server to the resolver, which forwards it to the client.

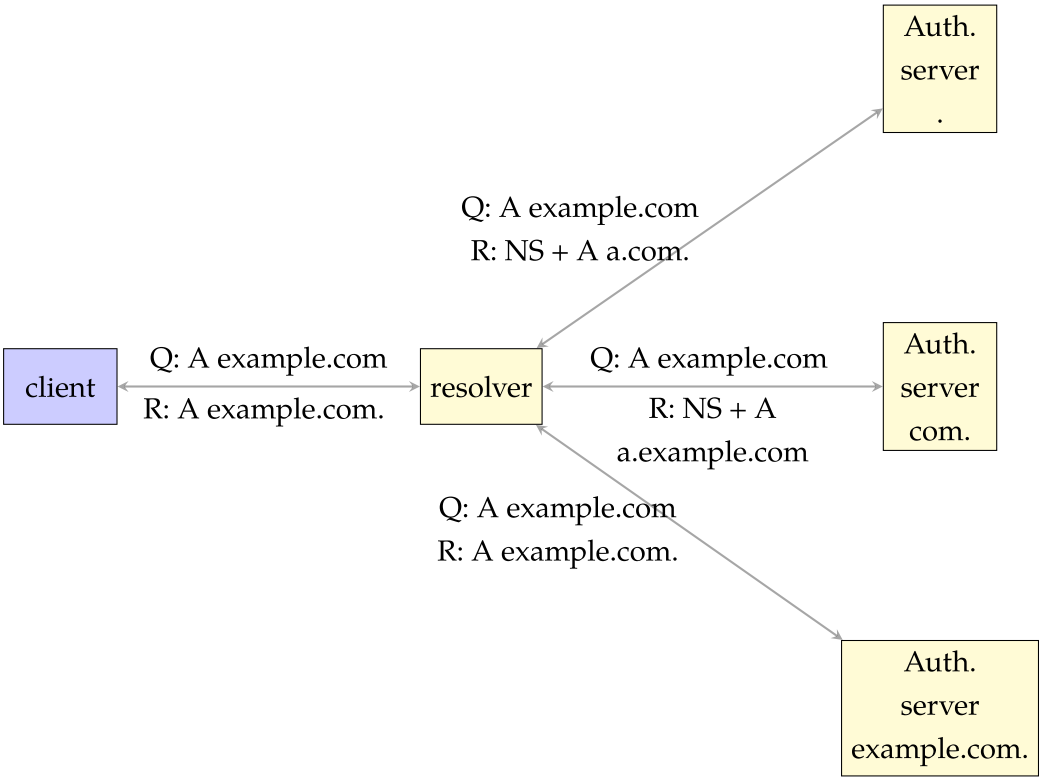

Fig. 1.6: Example of Domain Name System in use

As can be seen, there is a lot of new terminologies involved. First of all, I will describe different kinds of servers involved, then I will focus on messages and data structures used for communication and storage of DNS data.

1.2.1 Authoritative DNS Server

Authoritative servers hold the actual data in the Domain Name System. Each server is responsible for its own domain, let us say, that it is responsible for ”example.com”. Information such as IPv4 address, IPv6 address or domain name of its mail server can be stored in this type of server.

Widely used implementations are Bind, Knot DNS or Power DNS, where Bind is the referential implementation based on all available RFCs.

1.2.2 DNS Resolver

This type of server is usually responsible for serving DNS data in a local network or directly on a client’s computer. It performs query resolution and caching, optionally validation.

1.2.3 Resource records

Resource records (RR) are basic elements for information storage in the DNS. It contains a domain name, associated data and additional metadata about the record itself. Every RR is of a certain type; DNS defines many different types and new ones are introduced in separate RFCs. Following is a list of basic RR types, more will be introduced throughout this chapter.

Tab. 1.1: Resource record types

A	IPv4 address of a domain name
AAAA	IPv6 address of a domain name
NS	Domain name of a name server responsible for the queried domain name
TXT	Arbitrary human-readable data

1.2.4 Additional resource record types

New resource records can be introduced in additional RFCs. Following is a list of relevant types, which are used to store information, that can be used for verifying authenticity of a certain entity. Where entity can be a server, user or service and information can be a certificate, fingerprint or public key.

Tab. 1.2: Additional resource record types

SSHFP	A fingerprint of a public key associated with a domain name [8].
CERT	Certificate containing a public key, identity information and a signature. This RR can be used without DNSSEC, because it contains its own signature and it is up to client to decide about its credibility. [9]
OPENPGPKEY	OpenPGP public key [10]

1.2.5 DNS messages

DNS protocol also defines the structure of messages that are sent over the wire. A message is either a query or a response. The response usually contains one or more resource records corresponding to the query.

After introducing the concept of resource records, I could turn the figure 1.6 into a sequence diagram to depict order of queries and responses involved in typical DNS communication 1.7

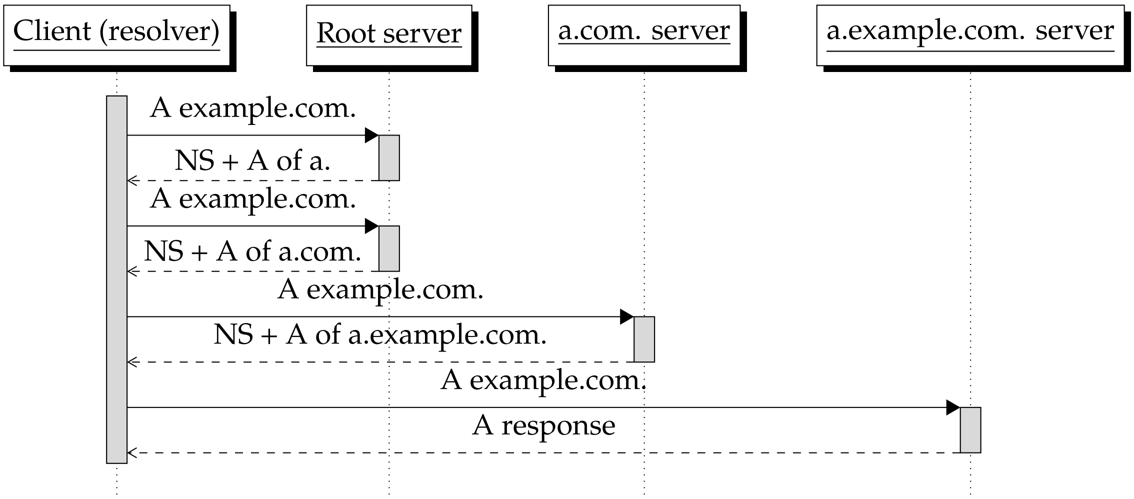

Fig. 1.7: Sequence diagram of a DNS query

1.2.6 Security extension - DNSSEC

DNSSEC adds a layer of authenticity over DNS[3][12]. The operation of DNSSEC is based on public key cryptography and digital signatures. Each set of resource records is signed with so-called Zone Signing Key (ZSK) and the result is stored in a RRSIG record. This proves authenticity of the resource record set. In order to validate the signature, public part of the ZSK is stored in a DNSKEY record. Again this RR set is signed and the signature stored in a RRSIG record, but as opposed to other signatures, this one is made with a private part of Key Signing Key (KSK). This key is different from the ZSK and is usually larger [11], because it is not changed as often as the ZSK. The reason to make this key bigger is that it is shared with a parent zone in form of Delegation Signer (DS) record. This record contains a hash of the public part of the KSK. By creating a DS record a relation between parent zone a child zone is created. If DNSSEC is correctly deployed, this relation exists between all zones from the root zone down to the last existing zone creating a chain-of-trust.

Figure 1.8 shows how a communication between client and server looks like when DNSSEC is enabled.

When a clients software receives all necessary messages, it can start with verification itself. Figure 1.9 shows the relation between different kinds of resource records.

The procedure above describes the way an existing resource record is verified, but there must also be a way to provide authenticated denial of existence. For this purpose, NSEC [13] and NSEC3 [15] resource records were introduced. The former one works by returning the next secure domain. For example a nameserver, that defines A records for ″api″ and ″www″, when requested for ″bugzilla″ would return NSEC record containing ″www″. The unfortunate side effect of this solution is revealing information about the

---

[3] The name is unfortunate, because it does not add layer of security.

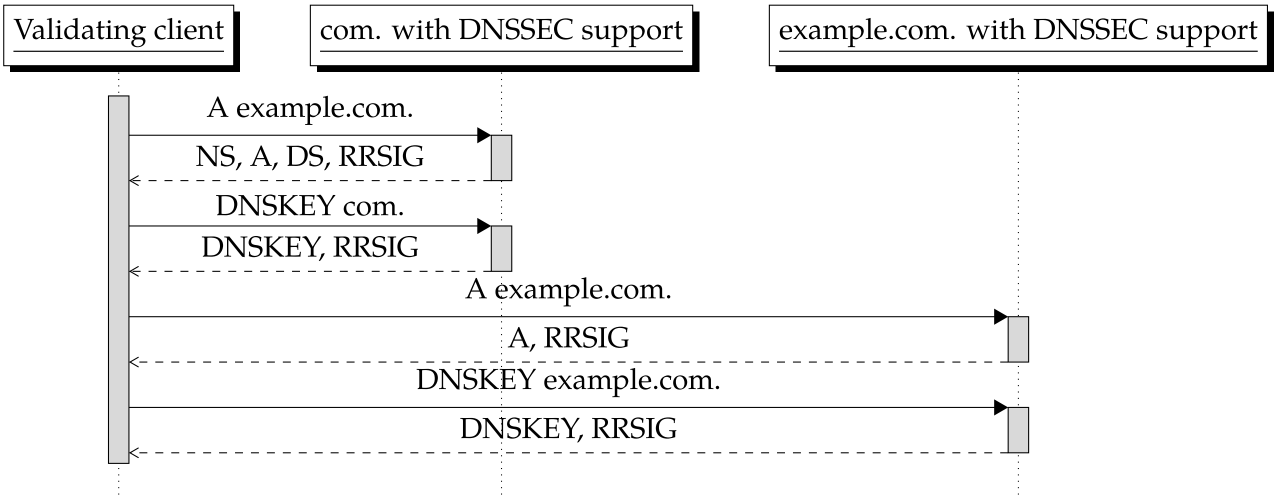

Fig. 1.8: Sequence diagram of a DNS query

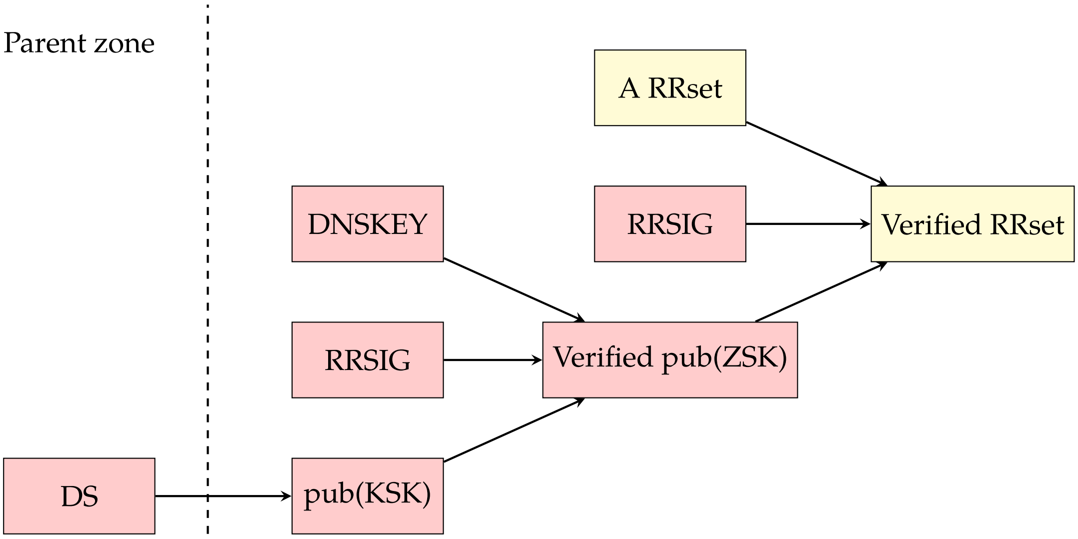

Fig. 1.9: Verification of a RR set using DNSSEC records

zone content. One can basically gather the whole content of the zone without any prior knowledge. This practice is called a zone enumeration and was one of the reasons for introduction of NSEC3 record [15]. As opposed to NSEC, NSEC3 utilizes a cryptographic hash function with respect to privacy. This way zone enumeration is prevented, but the original aim is preserved.

1.3 Software distribution

Software distribution deals with the problem of delivering software to the end user. Many different methods were established during past few decades. For instance, most Microsoft

Windows users would probably go to the Internet and downloaded the latest installer directly from a vendor of the desired software. On the other hand, it is common for users of Unix-like operating systems to download a source code a compile it themselves, e.g. using well-known sequence:

```
./configure && make && make install
```

Both of these approaches suffer from the fact, that it is up to the user or the installed software to check for available updates and even worse, it's only up to the user to verify integrity and authenticity of downloaded software.

An answer to these issues is straightforward, use one single repository where software is published and verified by maintainers and updates are performed by a client software, usually referred to as a package manager. These repositories are nowadays found even in mainstream operating systems like Android or iOS. The only difference is that they are called in a fancy way such as "App Store" or "Play Store".

In the next few sections, several package formats and managers will be introduced. I will focus on Linux distributions and their ecosystem. Nonetheless, there are way more methods to distribute software like programming language specific packages (e.g. Pip for Python or Crates for Rust), Docker images for server-side applications or Flatpaks for Graphical User Interface (GUI) applications just to mention a few.

1.4 Packages in Fedora based distributions

These distributions are based on RPM packages. It used to stand for Red Hat Package Manager, but was changed to RPM Package Manager (a recursive acronym), because it is used outside Red Hat ecosystem as well. RPM is both a file format for software packages and a set of tools used to manage them, but it provides very low-level interface not suitable for common users. For many years Yellowdog Updater, Modified (YUM) used to be the tool for package installation, updates and dependency management in Fedora, but since version 22, there is completely new Dandified Yum (DNF) package manager, but YUM is still in use in Red Hat Enterprise Linux (RHEL) and all its derivatives like CentOS or Scientific Linux. In this thesis, I will focus on DNF, since it is the future of Fedora distribution.

1.4.1 RPM file format

The RPM package format is not the first attempt to create a format like this, but it is the first one to fulfill all requirements we have. First of all, it should be possible and easy to install, remove and verify installation of a package. The installation process should not involve any complicated steps to minimize human errors. The verification is also a vital part of the installation process to ensure that all files were installed properly. The second goal is to make package maintainer life easier. Package maintenance is not the most exciting job and one package maintainer is usually responsible for more than just one package, sometimes even thousands of packages, so the process of creation and maintenance should be as easy as possible. This is related to the usage of an original source code. It is desired to keep downstream changes minimal, which in turn makes it easier to update a package. Last but not least, it should be possible to build one source package for multiple architectures, so you don't have to keep more versions of the same package for different platforms, e.g. x86_64[4] or s390[5]. [1]

It was decided that RPM would use a custom file format instead of using already existing formats, e.g. a tar archive compressed using gzip. The file format consists of four parts:

- The lead
- The signature
- The header
- The archive

The lead was used to store information used internally by RPM in previous versions. Nowadays it is not used any more by RPM and the header is used instead. Nonetheless, it is still used by utilities like **file** to determine file format. The main reason to introduce header over lead was its inflexibility. Since the lead is a simple data structure defined like this:

```
struct rpmlead_s {
    unsigned char magic[4];
    unsigned char major;
    unsigned char minor;
    short type;
    short archnum;
    char name[66];
    short osnum;
    short signature_type;       /*!< Signature header type (RPMSIG_HEADERSIG) */
    char reserved[16];      /*!< Pad to 96 bytes -- 8 byte aligned! */
};
```

there is simply no way to extend this structure in a binary compatible way. On the other hand, the header structure can store arbitrary data because it only holds a pointer to actual data as opposed to the lead. An RPM package can also contain more headers than just one and it usually does, because the signature is also wrapped into a header structure. The last part of a package is obviously a source code, which is compressed using GNU zip.

---

[4] x86_64 is the 64-bit version of the x86 instruction set

[5] Processor architecture for IBM mainframes

1.4.2 Signature

As mentioned above, the signature is implemented as a header structure. It is based on the archive and header only, the lead and signature itself are not signed, nor they are part of any checks based on the information stored in the signature. While this might sound strange at first glance, it actually is reasonable. Because the lead is not used internally any more, its modification would, at worst, result in unknown file format. Almost the same applies to the signature itself. Modification of its content would result in an invalid package, which will be rejected by a package manager.

The **rpm** utility can be used to examine RPM packages and see if they come with a signature and what is the algorithm used to create them:

```
$ rpm --query --info kernel
Name        : kernel
Version     : 4.12.5
Release     : 300.fc26
Architecture: x86_64
Install Date: Tue 15 Aug 2017 01:22:45 PM CEST
Group       : System Environment/Kernel
Size        : 0
License     : GPLv2 and Redistributable, no modification permitted
Signature   : RSA/SHA256, Wed 09 Aug 2017 08:33:30 PM CEST, Key ID 812a6b4b64dab85d
Source RPM  : kernel-4.12.5-300.fc26.src.rpm
Build Date  : Mon 07 Aug 2017 06:32:54 PM CEST
Build Host  : bkernel01.phx2.fedoraproject.org
Relocations : (not relocatable)
Packager    : Fedora Project
Vendor      : Fedora Project
URL         : http://www.kernel.org/
Summary     : The Linux kernel
Description :
The kernel meta package
```

As you can see, the signature algorithm is RSA/SHA256 which was described in the section about cryptography (1.1).

1.4.3 Creation of a RPM package

As described above, RPM is a custom format for software distribution and as such, it needs specific tooling. Every package is built from its sources, usually in form of an archive, and a ”spec” file, which defines steps to build this package and its content. When both the sources and the spec file are ready, **rpmbuild** tool can be used to create an RPM package.

Details of this procedure are beyond the scope of this thesis, but one step is important: package signing.

Package signing is pretty straightforward because it is very common. A packager starts by creating his public/private key pair using the **gpg** utility. Then he specifies which key to use in the *~/.rpmmacros* file and finally, he can run:

```
rpm --addsign <filename .rpm>
```

The presence of a signature can be verified by querying the SIGPGP header using the **rpm** utility:

```
rpm --query --package --queryformat "%{SIGPGP:xml}\n" <filename .rpm>
warning: <filename .rpm>: Header V4 RSA/SHA256 Signature, key ID b3971e8d:
NOKEY
<base64>iQFJBAABCAAzFiEE0QR2K01VfXvpYguDDtF8k70XHo0FAloZMloVHHBhY2thZ2Vy
QGV4YW1wbGUuY29tAAoJEA7RfJOzlx6N/lwIAIw0/moxkhoEu6BnKGbNA1CcJ74w
Leh1MNhGn8iS1bL75Geqb2+irFhBpyiIlXDJt5qysUgTVHZ2isFVw+s3AazmdXu1
eeDjT2689x9+6uneLKIQC4ml0++pn1P9mEjxpl+pqH04Kb7VFae/1nIr7iv4bJwF
NlXKF+j4iiXpLafJgQEPs3UzJA3iTbTRR0GeLTwUlRf00uRVGl/KciaaKcGakMdF
YPysW/9gGWkkg5uejCEMrskEBUPwgVcVzjtBzCMwFb66YUp53TYYYqJjPlxuCQFN
VYNusxfFr5CAYKRI9ZvCMzv6l/0eyVe1sSqikCPI0X18NMU6Wc5luuPB47o=
</base64>
```

1.4.4 RPM repository

createrepo utility is a tool for repository creation. It goes through all packages in a directory and creates metadata to be used by dnf or other package managers. These are stored in a *repodata* directory. The main file is *repomd.xml* which contains timestamps , hashes and other meta data about other files. **-primary.xml.gz* contains information and hashes of packages in the repository. The whole idea is captured in Figure 1.10. The repository metadata file, *repomd.xml*, can be optionally signed, but this is not usually used [20], anyway it is possible simply by using the **gpg** utility:

```
gpg --detach-sign --armor repodata/repomd.xml
```

This time a private key from a local keyring will be used.

1.5 Packages in various Linux distributions

The term Linux distribution usually refers to a collection of open source software, that creates a fully functioning operating system based on the Linux kernel. Many of these distributions have developed their own way to distribute software packages. This usually involves a client-side application to handle packages, file format for storing them and a server-side infrastructure.

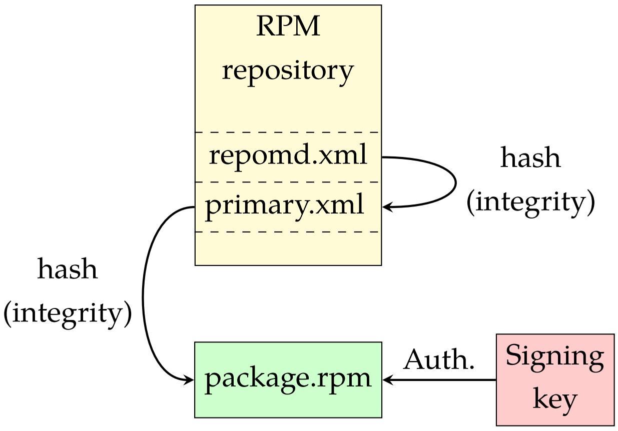

Fig. 1.10: Important elements in the RPM repository

1.5.1 Gentoo

Although Gentoo is widely known for their source packages, they also have a specification for package distribution in the binary format [16]. The package format is made of two parts:

- An archive containing files to be installed on the system stored in .tar.bz2 format (where "tar" is a program to compose more files into a single one and bzip2 is a compression algorithm)
- xpak archive with package metadata

This reflects the Unix philosophy of using simple solutions and composing them together into more complicated ones.

On the server side, where repositories reside, file and directory structure is defined so, that a package manager knows where to look for information. The most important file, from this thesis perspective, is a "Manifest". It contains a list of packages alongside with their hashes to provide integrity verification and an optional cryptographic signature of this list so that the authenticity can be also verified [17]. Example of such file can be found in Listing 1.

As you can see, the manifest contains three different hashes for each package and the signature is simple ascii armored string. To sum up Gentoo's approach to package verification: packages are hashed to provide integrity verification, these hashed are signed to provide authenticity and the list of signing keys on the official Gentoo website serves as the trust anchor.

Other files, that can be found in Gentoo repository are not related to verification; you can find there for example "ebuild" file, which contains scripts to be run during installation and configuration of this package, and an XML encoded metadata with information about the package and its packager.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

DIST iptables-1.4.21.tar.bz2 547439 SHA256 ... SHA512 ... WHIRLPOOL ...
DIST iptables-1.6.0.tar.bz2 608288 SHA256 ... SHA512 ... WHIRLPOOL ...
DIST iptables-1.6.1.tar.bz2 620890 SHA256 ... SHA512 ... WHIRLPOOL ...

BEGIN PGP SIGNATURE-----

Version: GnuPG v2.0.16 (GNU/Linux)

iEYEARECAAYFAkzXIv0ACgkQ/ejvha5XGaOREwCdH3qqFMNPmrZNLvzhv0jmM5QD
9r4AoPBm/72TYh+x3LTDn+0n9OhBlYiz
=Snqo

END PGP SIGNATURE-----
```

Listing 1: Manifest containing list of packages with hashes

1.5.2 Debian and all its derivatives

Debian is one of the oldest and most widely known distributions and its derivatives like Ubuntu and Linux Mint are also very popular. All of them are using the **deb** package format. As in Gentoo, this file format is not a custom one. It is a composition of different archives, that are compressed with **xz**, **gzip** or similar utility. On the top level, the **deb** package consist of 3 parts, that are composed into one archive using **ar** [18]:

- debian-binary - basically a version number
- control archive - metadata and installation scripts
- data itself

One can examine the file format quite easily with a standard Linux shell:

```
# Download the package
$ wget http://http.us.debian.org/debian/pool/main/f/fish/fish_2.4.0-1_amd64.deb
# Unpack it
$ ar xv fish_2.4.0-1_amd64.deb
x - debian-binary
x - control.tar.gz
x - data.tar.xz
# Extract the control part
$ tar xvfz control.tar.gz
./
./control
./md5sums
```

```
./postinst
./postrm
./preinst
./prerm
```

It is possible to sign each package individually, but many distributions including Debian itself have this option disabled and instead verify the file with repository metadata [19].

Since the package signing is disabled by default, it is a bit more difficult to sign and verify **deb** packages. Signing can be done with the **debsigs** utility which will sign each part in the **ar** archive separately and store the result alongside these files. On the other hand, verification can be performed using the **debsig-verify** utility, but it requires non-trivial, manual configuration.

The approach taken by the Debian project is similar to that of Gentoo. A repository is described by a file with metadata. This file contains a list of packages with associated hashes and the whole file is signed. Public keys can be found on the Debian website, for example here: `https://ftp-master.debian.org/keys/archive-key-9.asc`.

1.6 Common patterns in package managers

As can be seen in the previous sections, there are some patterns in the software distribution systems that are common for groups of different package managers. These patterns can be classified based on used cryptographic operations and target file [20]. Considering a hypothetical repository with metadata file and content to be served to users (packages), the designer of this system may decide to protect this system by some cryptographic means. The least secure method is to sign only the packages [20], whereby package I mean both its content and its metadata including dependencies. A better option is to sign the repository metadata file, which contains hashes for the packages and thus ensures their integrity. Also, both packages and the repository metadata file can be signed to allow secure installation of packages from a repository as well as standalone packages (packages not coming from a trusted repository). Finally, the authenticity can rely solely on TLS certificates and avoid using signatures at all.

Also the trust must be bootstrapped somewhere. This can be done using secure HTTP connection or using DNS security extension as discussed in later chapters.

1.7 The Update Framework

Since the patterns tend to repeat and there is a lot of different repositories, which are used by Linux users nowadays (Flatpak, PyPI, npm, crates.io or Ruby Gems just to name a few), it is reasonable to attempt to extract these common pieces and construct a framework, that can be reused by these existing repositories and those to come. One such attempt is The Update Framework (TUF)[21]. It does so by presenting generic repository structure and key management scheme, that can be used with arbitrary content served.

The TUF design is based on *roles*. Each role has a certain responsibility and a file, where it stores its data (Figure 1.11 shows the overall idea of this system). The first role is the root role. It signs the root.txt file, where keys of other top-level roles are specified. The timestamp role is responsible for signing the timestamp.txt file, where the latest version of release.txt is specified. The release role signs and fills in the release.txt file, which contains the latest versions of all metadata files except for the timestamp.txt. Finally, targets.txt specifies available targets, which in terms of SW package distribution would be the software packages themselves.

TUF is responsible for downloading and verifying the content in a secure way, the rest is up to the package manager.

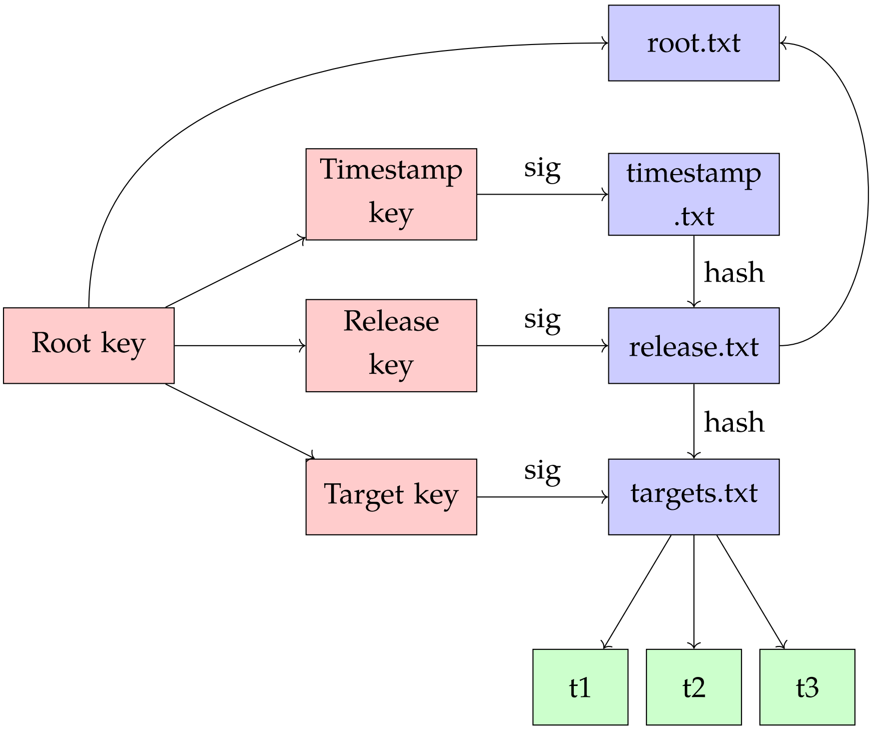

Fig. 1.11: Diagram of The Update Framework

1.8 Bootstrapping a trust chain

One problem with the authenticity verification is that you have to bootstrap the trust somewhere. In other words, you have to start trusting something at some point, so that you can build on top of this trust. To make things easier, let me assume, that a client obtained an ISO image containing an operating system from reliable source and the trust starts here. On the image, there will most likely be a public key of the DNS root zone and certificates of few trusted authorities. These are the trust anchors, that can be used to validate other parties, such as repositories with software packages.

1.9 Possible security threats for software distribution systems

The software distribution system is an essential part of every operating system and as such, it is also a target of possible attacks from the outside world. Different attacks can be categorized based on how the attacker wants to harm the victim [20] and how is he going to invade the distribution pipeline.

First of all, let me introduce the concept of a mirror. Distributions with a lot of users cannot handle the traffic of updates with only one server, thus they allow the community to create mirrors. These are servers, that are supposed to serve the exact same content as the official repository does. For example in Fedora distribution, users in the Czech Republic can download packages from the Brno University of Technology, Masaryk University or the CZ.NIC domain registry.

It was important to introduce this concept because it can be used as one of the attack mechanisms and as opposed to others, it does not really require any computer attack, more like a social engineering. Finally, the attacker can start by using one of these methods:

- Create its own mirror and wait for users to start using it,
- man-it-the-middle attack on a user with the goal of redirecting requests to different servers,
- intrusion of the official repository servers.

Once he has successfully performed one of these attacks, he can harm the user in different ways based on the protection, which the package manager is able to provide. These categories are based on assumption, that package managers can sign package content, package metadata, repository metadata or nothing.

- Arbitrary package - Installation of arbitrary package.
- Replay Attack - Providing an old package with known vulnerabilities. This package can be correctly signed, because it comes from the official distribution, but is outdated.
- Freeze Attack - Prevent the user from installing necessary security updates by providing old repository metadata.
- Extraneous Dependencies - Modifying package dependencies in order to install his own package alongside a regular package.
- Endless Data - Crash the user's system by sending an endless stream of data instead of any response.

Signatures of package content and metadata can provide protection against installation of arbitrary package and dependency. Repository metadata with trusted signature can protect the same as signatures of package content and metadata, but also from the Reply Attack, because the attacker cannot create a metadata file on his own. Unfortunately, the Freeze Attack is still possible. The Endless Data attack can be mitigated in the package manager software and does not require help of cryptographic algorithms.

The Update Framework introduced in section 1.7 is supposed to protect against all of these attacks by providing a hierarchy of cryptographic hashes and signatures with a timeout, implemented using a short validity of the signature on the *timestamp.txt* file.

1.10 Alternatives to using DNSSEC

An obvious alternative to using DNSSEC for public key validation is secure TLS connection with a party, that has a certificate signed by a trusted certification authority. This approach is basically used today because a user is supposed to check the signing key by hand and he can do so by entering the repository website, which will probably contain information about the key and will be available over a secure HTTP connection. The advantage of using DNSSEC over web application is that two distinct chains of trust will be used, thus making it even harder for an attacker to control all security mechanisms.

The second option would be a usage of classical key servers. The problem with this approach is that PGP system is meant as peer-to-peer, but we need global authority trusted by anyone who is running some Linux distribution, e.g. Fedora.

2 ANALYSIS OF DEVELOPMENT TOOLS

2.1 Implementation of the dnf package manager

Dnf interacts with many other tools and libraries. Most notably it uses **rpm** for package manipulation and **gpg** for key management. The basic idea is captured in Figure 2.1.

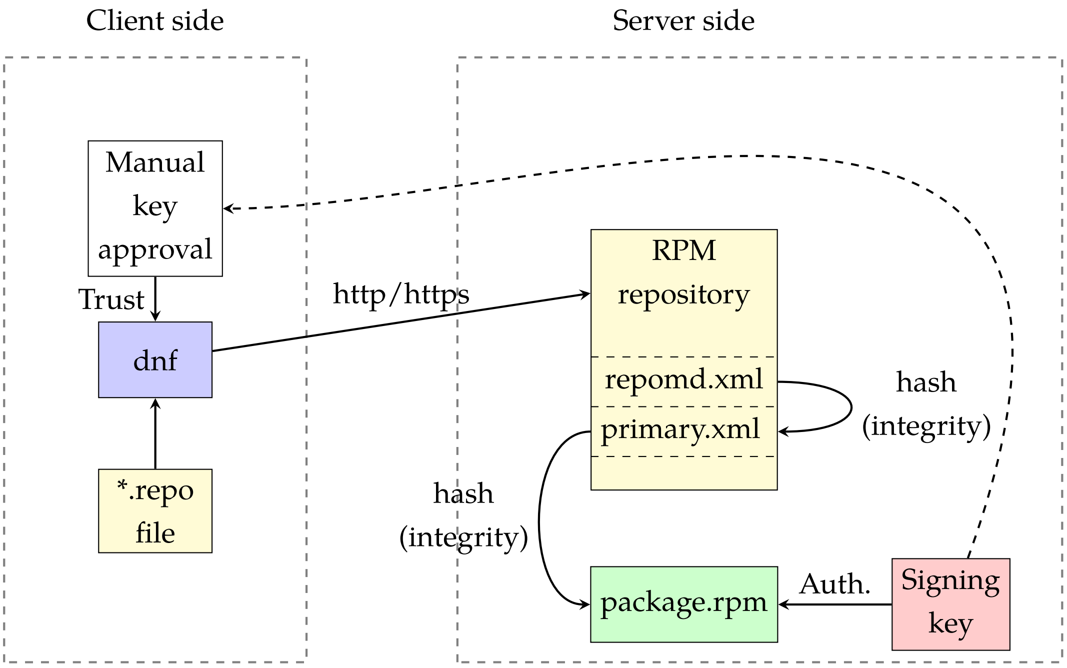

Fig. 2.1: How DNF works together with RPM package repository

The crucial information here is, that the **.repo* file includes an URL of a key, that is supposed to be used for signing in this repository. These files are located in the */etc/yum.repos.d/* directory are specify repositories, which the dnf should use. Their format is a simple ini file, like this:

```
[google-chrome]
name=google-chrome
baseurl=http://dl.google.com/linux/chrome/rpm/stable/x86_64
enabled=1
gpgcheck=1
gpgkey=https://dl.google.com/linux/linux_signing_key.pub
```

As you can see, the file contains URL of both the repository and the verification key.

2.1.1 Package integrity verification

Once a new signing key is encountered during a package installation, the user is prompted for a confirmation about its validity. When the validity is confirmed, the key is stored in a dnf specific storage as a special package with no content named ”gpg-pubkey-<some hash>”. It can be seen from this output, where all keys are listed (rpm -qa) an only the information about its name is displayed (xargs rpm -qi together with grep):

```
$ rpm -qa 'gpg-pubkey*' | xargs rpm -qi | grep -o -E "gpg\(.*<"
gpg(Fedora 26 Primary (26) <
gpg(@rust_playground (None) <
gpg(Google Inc. (Linux Packages Signing Authority) <
gpg(msehnout_Neovim-qt (None) <
gpg(Google, Inc. Linux Package Signing Key <
```

The key validity is never checked again and it is up to the user to follow information from the upstream community about possible security breaches.

The whole process as a diagram can be seen at Figure 2.2. Two nodes are emphasized, because they will be reffed to from the solution proposal chapter (3).

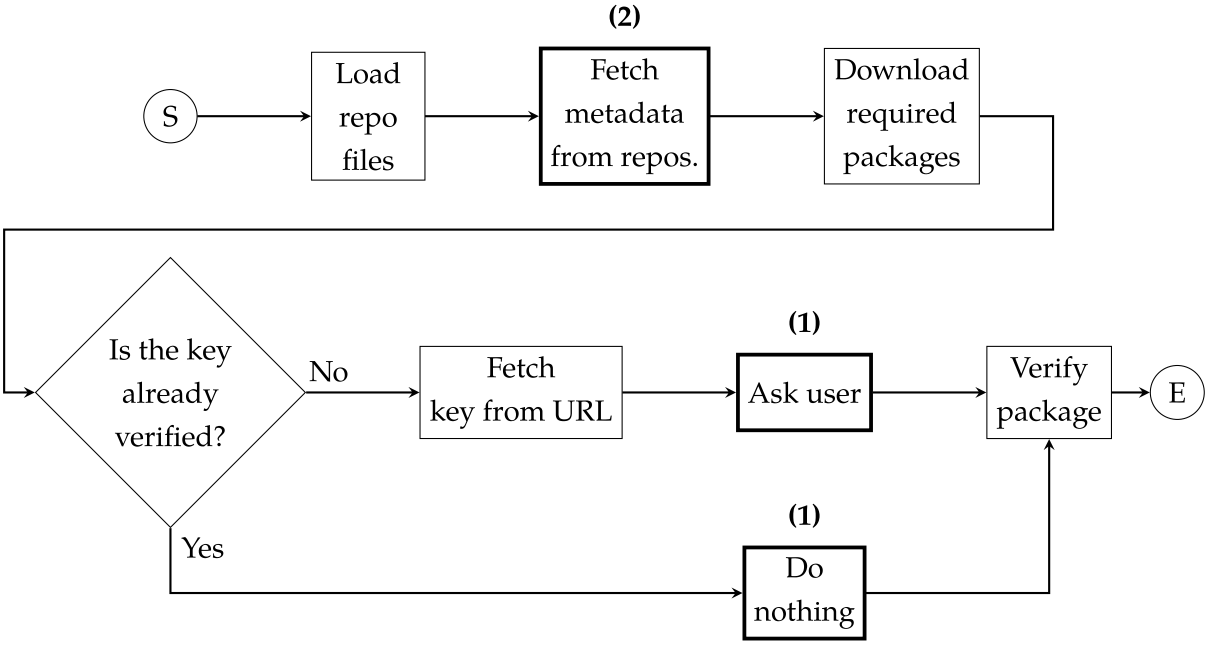

Fig. 2.2: New package installation process

2.2 Virtual Machines

Traditional approach is to run only one operating system on a sing computer, but this approach is very restricting in both interactive and server usage. For example, it is a waste of resources to buy new hardware for each application a company would like to deploy. Of course these applications could be run on the same system, but it comes with a risk of interference between unrelated applications. Also one system is not necessarily capable of running all applications, because these can have different requirements of programming libraries, their versions etc. Users, mainly developers, also need a way to test their applications in different environments than their own operating system. For example a developer running macOS is developing an application for server running CentOS. All of these use cases can be fulfilled with a virtualization.

The fundamental idea of the virtualization is, that on the very bottom level a host is running. This is usually a physical machine. On top of the host, a virtual machine manager (or hypervisor) runs and creates an interface for virtual machines, which is identical to the host interface [23]. These virtual machines are also known as guests.

Many different types of virtualization have been developed during past few decades, but three of them are especially useful for programming.

type 1 a general purpose operating system, that also provides a hypervisor functionality. For instance RHEL with Kernel Virtual Machine (KVM).

type 2 application running on top of a standard OS providing hypervisor features. e.g. VM ware Workstation or Oracle Virtual Box

Application containment this is not a virtualization technology, but rather technique providing virtualization-like features by isolating applications from the host OS. Linux Containers (LXC) and Docker are examples of this approach.

2.2.1 Reproducible development environments

As mentioned before, virtual machines are a useful tool for developers, because it can create an isolated development environment, which every developer in a team can instantiate. Vagrant (`https://www.vagrantup.com/`) is a tool, that can be used for exactly this purpose. By creating a simple text file, *Vagrantfile*, anyone can define an operating system, a set of installed libraries, local files and global configuration, so that everybody who wants to work on the same project can have the same system running on their desktop. This is especially useful in open-source communities and when working with networking software.

2.3 Programming languages

In this section I will briefly discuss programming languages, which are suitable for this project.

One important criteria is the language in which dnf is written. The frontend is written in the Python language, as opposed to the libdnf, which is written in C[1].

---

[1] As of November 2017 it seems to being rewritten in C++. Source: `https://github.com/rpm-software-management/libdnf`

Other languages can be in theory used as well, but it is undesirable to use languages which are not compiled to a machine code or use different runtime than the Python one. This limitation comes from the need to minimize container size[2]. This excludes languages like Go, Java or C# and leaves only languages like C++, Rust and of course Python. Unfortunately Rust is not very popular among Fedora developers, although its usage would definitely bring some benefits, because C-like languages are widely known for their error-prone nature. Typical memory bugs like buffer overflow or use-after-free were one of the most frequent vulnerabilities in the past [22] and still are at the time of writing this thesis [24].

---

[2] Docker containers are important part of Fedora project nowadays

3 PROPOSED SOLUTION

3.1 Goals summary

The primary goal of this project is automation of integrity verification in the process of RPM packages distribution. Users should not be prompted for questions which they cannot answer in an informed and responsible way (see Listing 2). Also there are a lot of tools which automate system deployment and these typically skip any verification, as there is no way to perform it. This automation has also a potential to increase security, because it will provide a simple way to revoke a key, that has been compromised.

```
Importing GPG key 0x46CD093F:
 Userid     : "msehnout_neovim (None) <msehnout#neovim@copr.fedora.org>"
 Fingerprint: 5E84 DA77 A5C0 121E D09C 60B9 4603 C1E8 46CD 093F
 From       : https://copr-be.cloud.fedora.org/msehnout/neovim/pubkey.gpg
Is this ok [y/N]: y
Key imported successfully
```

Listing 2: Example of dnf prompting user for decision about key validity

In this case, the automation is achieved by using DNS for key storage together with DNSSEC to prove authenticity of these keys and also to bootstrap the trust with the root zone key already present in an operating system. With this system in place, when a user wants to download a package, dnf can use secure HTTP connection to obtain the package and DNS to authenticate the verification key, thus using two different chains of trust.

3.2 Overall design of the system for automatic package integrity verification

Let me recapitulate the role of a user running Fedora with dnf and a web server providing RPM packages repository. Given this scenario, when the user wants to install a new package from a new repository, he creates a _*.repo_ file specifying an URL of the RPM repository and an URL of a verification key. With this file in place, the user issues a dnf command to install a package from this repository. dnf downloads a repository metadata followed by the package. When the package is to be installed, dnf checks its keyring for this key and since it is not found, it is downloaded from the provided URL and dnf asks the user to manually verify this key. The user then verifies this key by some means, he consider appropriate, e.g. a website with a valid TLS certificate saying, that this key indeed belongs to this repository.

Following the classification from the last paragraph, the user is responsible for key verification, dnf for downloading repository metadata and packages and the server for serving RPM repository over some application layer protocol, usually http. My goal is to free the user from this responsibility and use dnf and server side instead. For this purpose dnf will gain a new capability of communicating with the DNS system and verifying GPG keys through it. On the other side, the existing DNS infrastructure will now have to contain additional information. The whole concept is captured in the figure 3.1. Server side is represented by the "Repository" node and "Domain name system" group of nodes; client side contains dnf package manager and operates on a RPM package downloaded from the repository.

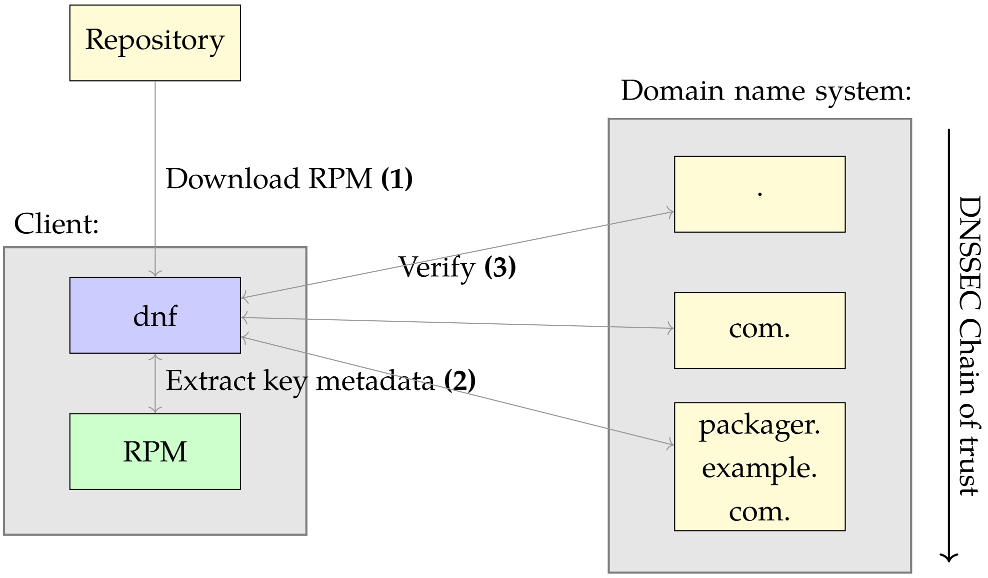

Fig. 3.1: Verification of public key derived from RPM package using DNSSEC

3.2.1 Minimal version

Since the RPM ecosystem is widely used it is not desirable to make substantial changes to it. For this reason, I decided to propose two versions of a solution. The first one, minimal, is supposed to automate the current behavior, whereas the second one is also meant to improve overall security by taking inspiration from TUF. In order to put these versions into context, they fit into the diagram at Figure 2.2. The minimal one fits into the slot number **1**, whereas extended spans over both **1** and **2**.

In the minimal version, only automation of the key verification is implemented. The two following sections define server side storage and approach taken by dnf, in order to decide about key validity.

3.2.2 Server side

RPM repositories can contain software from multiple sources and signed with different keys, although it is not usual, it is possible and the design of automated key verification must cover this scenario. Therefore the DNS server responsible for serving certain key is not necessarily on the same domain as the repository. Consequently the design for server side deals with used RR for key storage and domain name on which the RR will be available. As the key uniquely identifies source of a package, it can be used as a base for deriving the domain name.

OPENPGPKEY resource record can be used to store a PGP key in a domain name database and it is proposed as a solution for this application, as it seems to fit best the requirements, because it stores only the key itself and nothing more. As opposed to the CERT resource record, which could in theory be used as well, but the CERT record contains a whole certificate, thus requiring a trusted signature from a certification authority, but the idea in this thesis is to use DNSSEC chain of trust, not just another CA.

Now an algorithm is needed, that takes a public key as an input and returns a domain name on which the key can be found. RFC 7929 [10], section 3 is dealing with exactly the same problem, so the solution can be reused.

The resulting domain name, on which the RR will be stored, looks like this:

```
<packager name>.<tag>.<domain name>
```

Where:

1. Packager name is constructed from the left-hand side of the input email address. It is first pre-processed and then hashed using SHA2-256 algorithm. Finally only 28 leading octets are taken to create the label.
2. Tag is defined as *"_openpgpkey"* in RFC 7929, but I propose not to reuse this tag and use RPM specific tag instead, specifically *"_rpmpkgsignkey"*, nevertheless it is not very important and the implementation can easily handle both cases. Also note, that the label contains underscore, which might be a problem for some DNS implementation [25], but these should be fixed instead of avoiding the underscore character.
3. Domain name part of the email address is left as is and appended to the end of the domain name.

3.2.3 Client (dnf) side

On the client side, the resulting software must query the domain name derived from the key. It must also verify all signatures starting from the trust anchor all the way down to the signature of RR holding the key. This part of the solution will be integrated with the dnf package manager, as such it should be optional and possibly distributed separately from the dnf package. These requirements come from the fact, that dnf is already used in production systems and this enhancement must not break anything.

The key verification require handling of few edge cases, which are captured in the Figrure 3.2. These will be reflected in the proposed API, that is described in the following chapter.

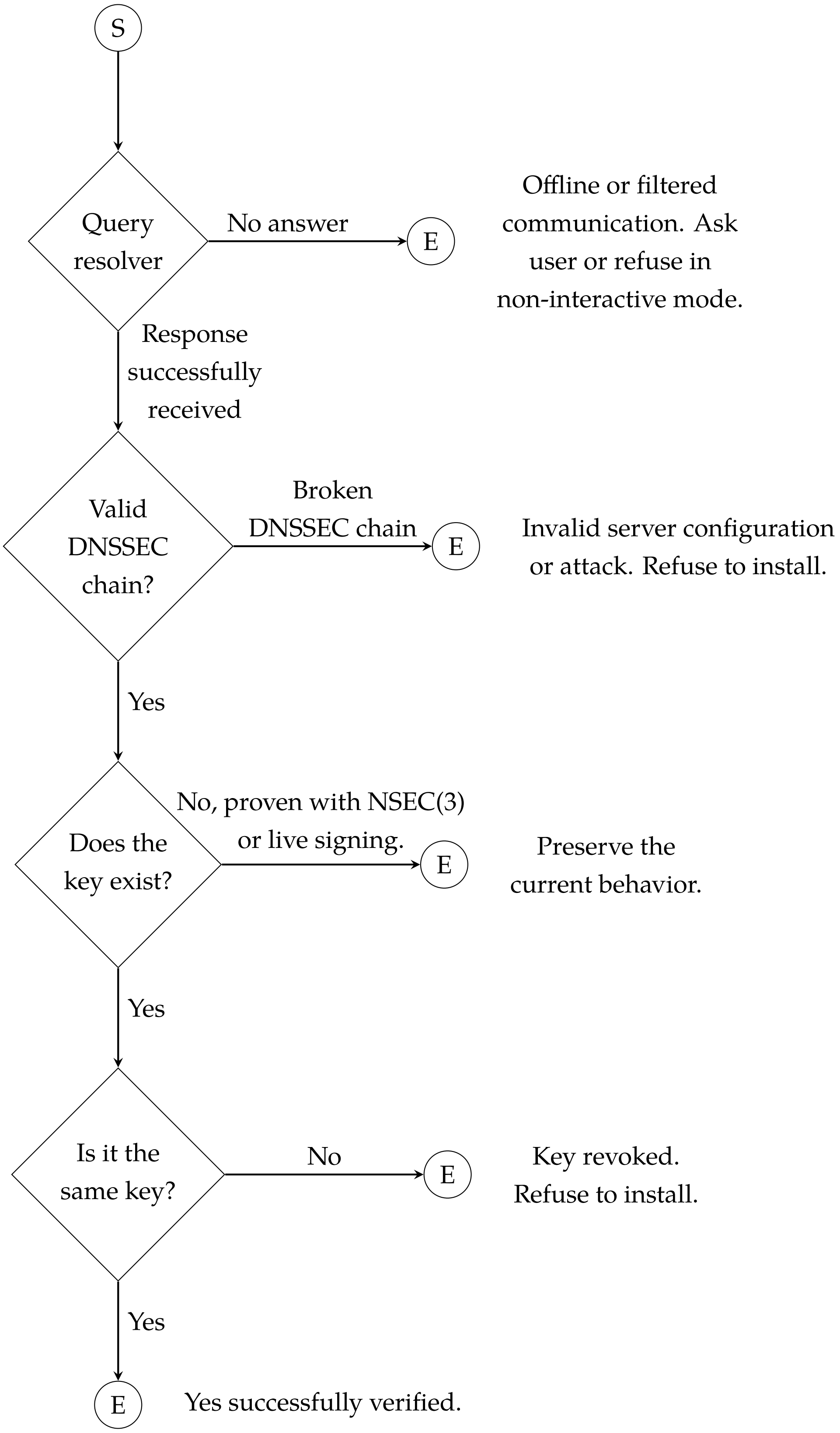
S
Query resolver
No answer
E
Offline or filtered communication. Ask user or refuse in non-interactive mode.
Response successfully received
Valid DNSSEC chain?
Broken DNSSEC chain
E
Invalid server configuration or attack. Refuse to install.
Yes
Does the key exist?
No, proven with NSEC(3) or live signing.
E
Preserve the current behavior.
Yes
Is it the same key?
No
E
Key revoked. Refuse to install.
Yes
E
Yes successfully verified.

Fig. 3.2: Key verification algorithm using DNSSEC

3.2.4 Extended version

The minimal version is supposed to fulfill the stated goal with minimal changes to the current system, but as described in the paper about TUF [21], overall security can be further increased. In this section, I propose to implement system similar to the TUF, but using DNS for key storage and DNSSEC for signing and bootstrapping trust.

In this proposal, the root key is replaced with a ZSK, which is already trusted and the zone is used for trusted data storage. The release key is stored in the zone at domain derived using the same algorithm as with RPM signing key, but with a different tag, e.g. *"_rpmmdsignkey"*. Metadata hash and timestamp are also stored in the zone, for instance in TXT record type. This way, authenticity and integrity of metadata is ensured. The target key is used for package signing instead as opposed to the TUF. Also note, that the timestamp key is omitted, but can be added the same way as release key. Figure 3.3 shows this version in context.

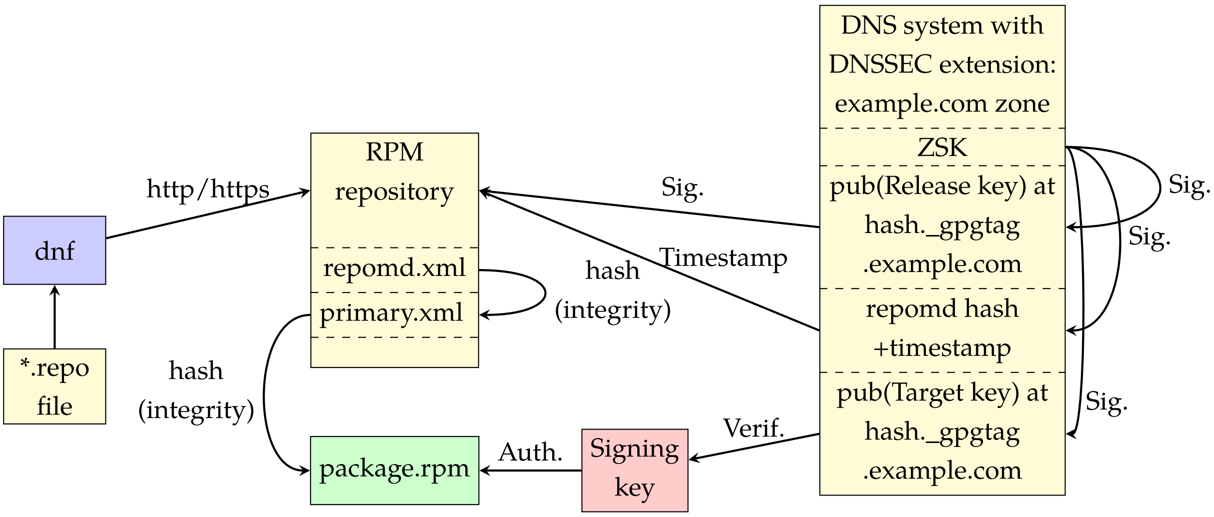

Fig. 3.3: Extended version of automatic verification system

4 IMPLEMENTATION

In this chapter, a specific implementation is described and discussed. The high-level idea has already been mentioned in the previous one. The implementation is written in the Python programming language as proposed in the previous chapter. It is split into two parts: a minimal and an extended one. Since this project is mainly about integrating existing technologies together, it is necessary to take advantage of existing libraries and executables, that can perform desired steps.

In the beginning, I cover the events that happen during a single dnf transaction. Then I put my solution into the context of this transaction and describe the implementation itself.

4.1 Flow of a transaction in the dnf package manager

As mentioned in the chapter about dnf implementation (2.1), it is a compound of many different projects. It uses its dependencies for the low-level package manipulation, dependency resolution or network communication via various protocols. As such, it is not exactly easy to read and contribute to, but I will try to briefly and visually describe what is going on during a transaction. The essential transaction is package installation.

When the installation process begins, the list of required packages from the user must be transformed into a set of packages with specific versions and all their dependencies. Since only one version of a package can be installed at a time, this task is not trivial, but eventually, the result is again a list of specific packages from specific repositories. All of these are then downloaded, their signatures are verified and in case they are valid, they are installed into the system. The whole idea is covered in Figure 4.1.

The signature verification process can be further divided as depicted in Figure 4.2. The verification itself is performed by the RPM library and keys are loaded from the RPM database. Therefore dnf only asks RPM to verify the package and in case it fails, new keys can be installed and the verification process is attempted again. The keys are specified in a *.repo file as described in section 2.1. The following step where new keys are installed and verified is the place where the DNSSEC extension will be applied.

4.2 Suitable Python libraries

First of all, a DNS library is needed. Ideally, this library should be already present in the Fedora distribution and provide easy-to-use and well-documented API. Unfortunately, there is no such option, but libundound has at least the Python API. As the name suggests it comes from the same upstream community as the Unbound resolver does. It provides both C and Python API and both are available as RPM packages. At the time of writing, there is almost no documentation except for a few examples [27]. Nevertheless, after some debugging and code exploration I was able to come up with implementation, that can use the library API to fetch the public key from the DNS database and verify it using

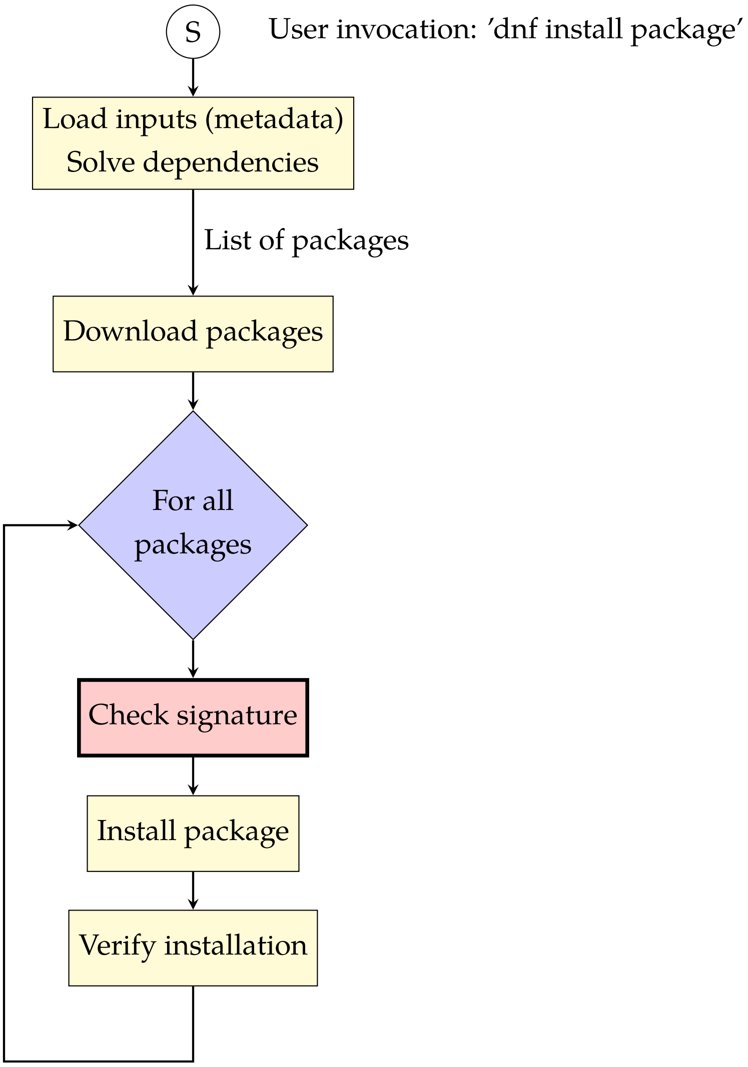

Fig. 4.1: High-level overview of a dnf install transaction

DNSSEC. I tested the library against my own, testing, authoritative DNS servers that were configured in a way to trigger either correct or bogus result.

Second important component is a PGP library, although it is used only for development purposes, because dnf takes care of PGP by itself. For this purpose, either PGP Made Easy (pgpme) or gnupg library can be used. I chose gnupg library because it provides better documentation. Unfortunately, the Fedora package containing this library is outdated and defective so I had to package it myself.

4.3 Library implementing the minimal version

Its purpose is to verify a given pair of a public key and an email address by contacting DNS servers and acquiring the public key on a domain derived from the email address. The whole process is secured with the DNSSEC extension otherwise it is not trusted. The

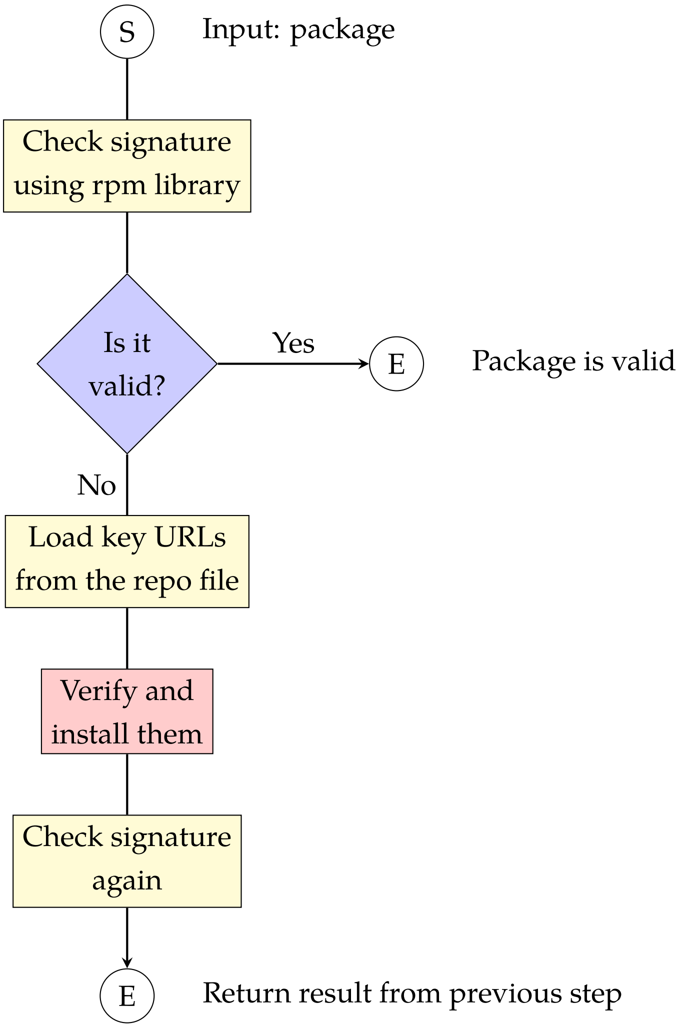
S
Input: package
Check signature using rpm library
Is it valid?
Yes
E
Package is valid
No
Load key URLs from the repo file
Verify and install them
Check signature again
E
Return result from previous step

Fig. 4.2: Signature check during the transaction

implementation can be divided into few logical steps which are depicted in Figure 4.3. The final step, *query DNS*, has already been specified in Figure 3.2.

The public API is really simple, because it was decided to keep it synchronous. That means, there is no thread spawned in background to perform in-advance key verification. UML diagram of the public-facing API of the library is in Figure 4.4. The main class implementing the algorithm is **DNSSECKeyVerification** which has only one public facing static[1] method for key verification. The input argument is of type **KeyInfo** that is in turn only email address and public key. Both are stored as regular strings. Return value of the method is of type **Validity**, which defines all possible outputs defined by algorithm in Figure 3.2.

The complete source code in form of a patch is available in attachment A.1. The file *dnf/dnssec/dnsseckeyverification.py* contains an implementation of the class described above.

---

[1] The class does not need to be instantiated for the method to be used.

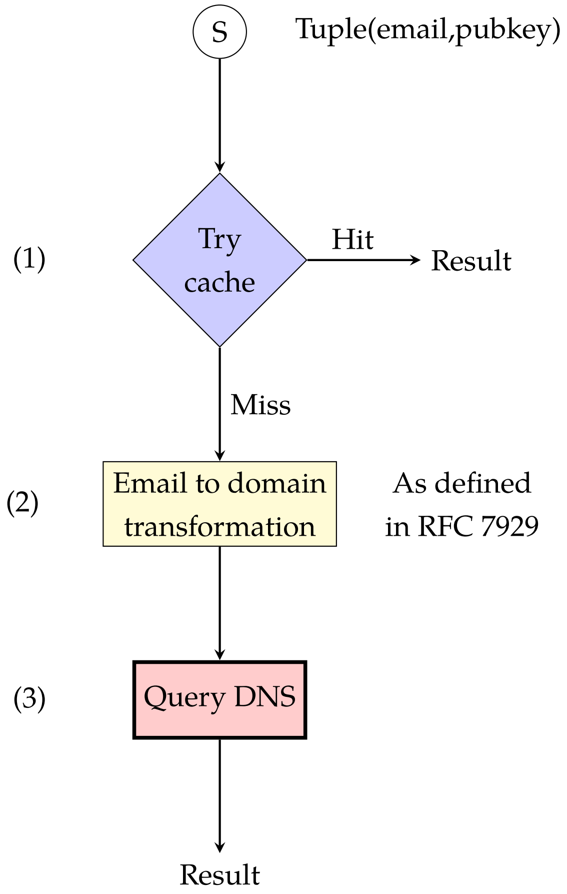

Fig. 4.3: Algorithm implemented by the key verification library

In case of a cache hit, the result is evaluated in the **__cache_hit** method (step 1 in Figure 4.3). Otherwise, the input email address is transformed in the **email2location** function (step 2) and then the DNS system is queried in the **__cache_miss** method (step 3). The transformation is a standalone function because it is not specific to this application.

The **RpmImportedKeys** class creates a list of imported keys. This list is then used to verify each key separately. Since RPM does not really provide any convenient API for listing keys, this class is implemented by calling **rpm** utility directly and parsing its output. First, a list of packages containing public keys is loaded in the **__load_package_list** method, then this list of names is turned into a list of **KeyInfo** objects (method **__pkg_list_into_keys** which in turn just iterates over the list and calls **__pkg_name_into_key** on each element.

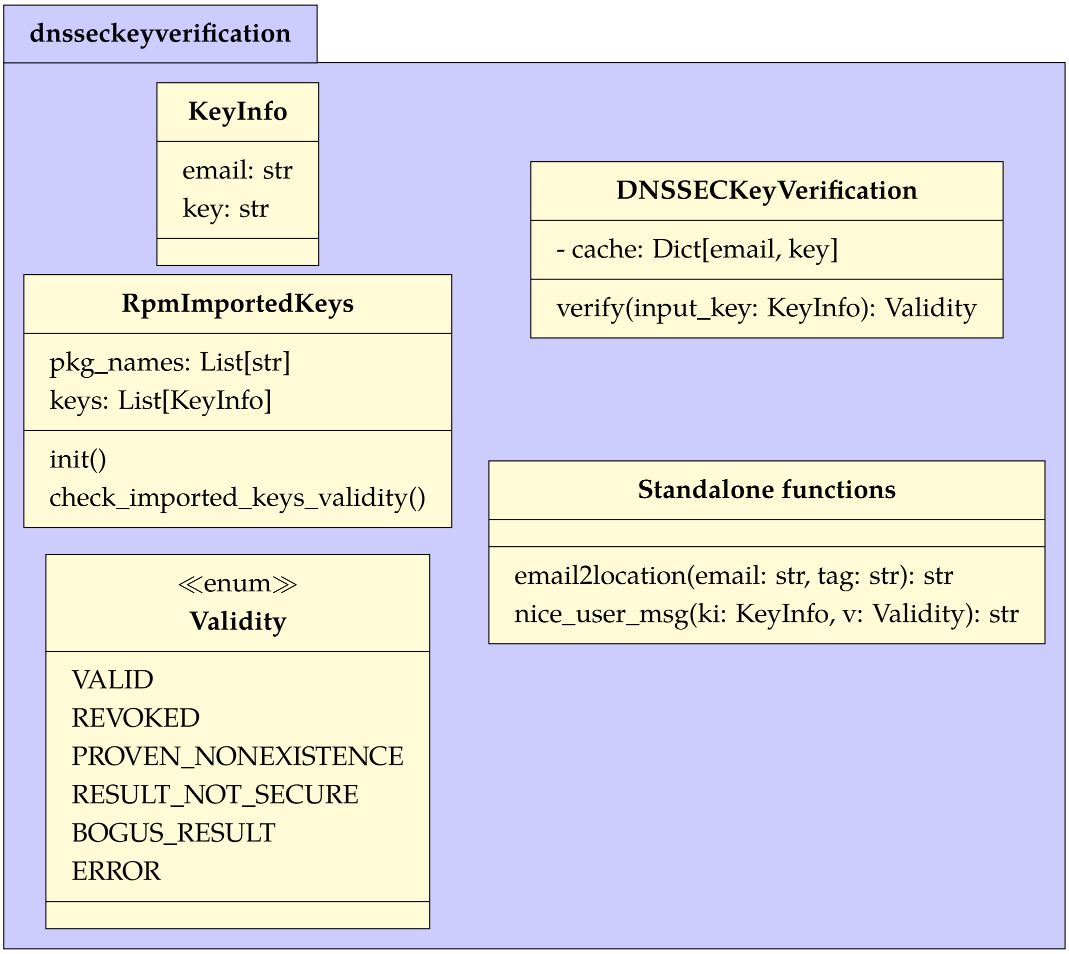

Fig. 4.4: UML diagram of the library implementing minimal version

4.4 Embedding the library into dnf

There are two main use-case scenarios for the minimal version; the first and the most useful one is new key installation. When a package is to be installed from a repository that has not been used before, dnf checks the repository file for a URL with verification key. If the URL is present, the key is downloaded and the user is requested for an approval of this key. This is where the DNSSEC extension can provide the user with an advice regarding the trustworthiness of the key.

Following this specification, it is necessary to use the library in the place where the key gets downloaded. After some code exploration, I found that the method *”dnf.Base._get_key_for_package”* is responsible for this task. In this method, two code paths can benefit from the extension. The first one is an interactive mode, where a message gets printed so that the user can decide based on the data found in DNS (Listing 3). The second one is a non-interactive mode, where ”yes” is assumed to be the default answer for any question. In this case, the key is tested and approved if it is valid or its non-existence is proven (Listing 4).

```
# DNS Extension: create a key object, pass it to the verification class
# and print its result.
dns_input_key = dnssec.KeyInfo.from_rpm_key_object(info.userid, info.raw_key)
dns_result = dnssec.DNSSECKeyVerification.verify(dns_input_key)
logger.info(dnssec.nice_user_msg(dns_input_key, dns_result))
```

Listing 3: Library usage for interactive sessions

```
if self.conf.assumeyes:
    # DNS Extension: We assume, that the key is trusted in case it is valid,
    # its existence is explicitly denied or in case the domain is not signed
    # and therefore there is no way to know for sure (this is mainly for
    # backward compatibility)
    if dns_result == dnssec.Validity.VALID or \
            dns_result == dnssec.Validity.PROVEN_NONEXISTENCE:
        rc = True
        logger.info(dnssec.any_msg("The key has been approved."))
    else:
        rc = False
        logger.info(dnssec.any_msg("The key has been rejected."))
```

Listing 4: Library usage for non-interactive sessions

The second usage scenario is a key revocation. Before each transaction, all installed verification keys are compared with their counterparts published in the DNS database. If the key matches, it is considered valid and left untouched. On the contrary, if the key is different, it has probably been revoked and thus should be removed from the RPM database and not considered safe any more. This functionality is encapsulated in the **RpmImportedKeys** class as described earlier.

A new key could theoretically be distributed using DNS, but I am currently focused on verification only. Also most of the RPM repositories provide their keys over HTTP(S), so the new key can be automatically downloaded and verified again using DNS.

4.5 Library implementing the extended version

The extended version stores verification key for the metadata signature (as described in section 1.4.4) and hash of the *repomd.xml* file together with its timestamp. The key can be stored and verified using the same procedure as described in the previous section. For the remaining part, I wrote server side and client side scripts. The former one is able to create a RR Set consisting of TXT records, that hold the information about the metadata file. This RR Set is then signed with ZSK as the rest of the zone. Output of the script can be seen in Listing 5. I decided to use multiple TXT records where each one holds only one key-value pair. This format is based on RFC 1464 [28]. An alternative approach would be storage of all pairs in a single TXT record using some serialization syntax such as JSON.

```
; b6359151f847834a2dd3dbcd20e631393826ba90999b40ce8c00afd54daf2b35 <- digest
; JSON: {"alg": "sha256", "hash": "b6359151f847834a2dd3dbcd20e631393826ba909\
; 99b40ce8c00afd54daf2b35", "ts": "24/03/2018", "val": "9d"}
$ORIGIN repomd.example.com.
@ IN TXT alg=sha256
@ IN TXT hash=b6359151f847834a2dd3dbcd20e631393826ba90999b40ce8c00afd54daf2b35
@ IN TXT ts=24/03/2018
@ IN TXT val=9d
```

Listing 5: Zone file holding hash and timestamp of the repomd.xml file

The client side is again represented by a single public function, which does all the work (The code is available in Attachment A.2). The idea here is again very simple as can be seen from Figure 4.5 (function **verify_md**). A file name and a domain is given to the function, it creates a hash of the local file using the algorithm depicted in Figure 4.6 (function **__hash_local_file**). The algorithm is using hash class available in the standard Python library. In the next step, it again uses *libundound* resolver to query DNS for the RR set created by the server side script (Figure 4.7) (function **__load_from_dns**). These records are processed into a dictionary holding their values and used to verify the metadata file. UML diagram of this module is in Figure 4.8.

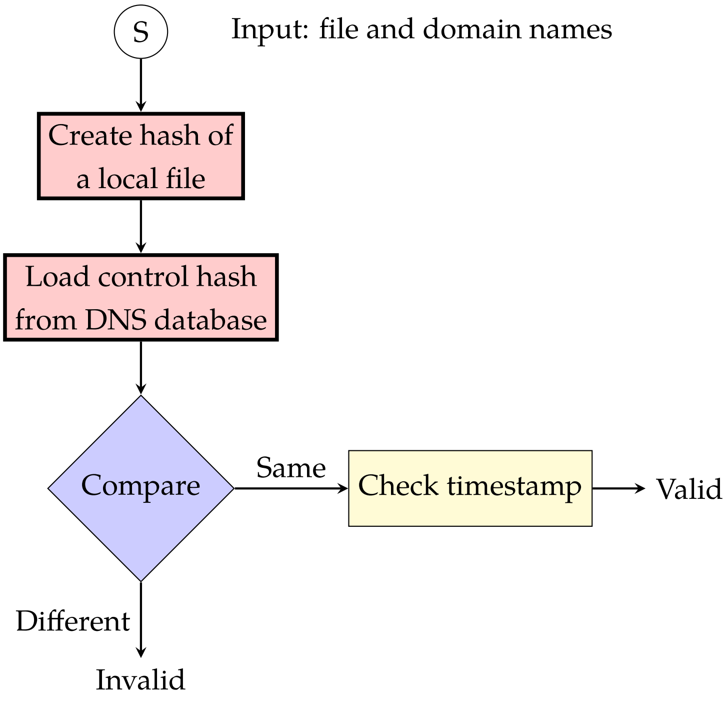
S
Input: file and domain names
Create hash of a local file
Load control hash from DNS database
Compare
Same
Check timestamp
Valid
Different
Invalid

Fig. 4.5: Metadata verification function

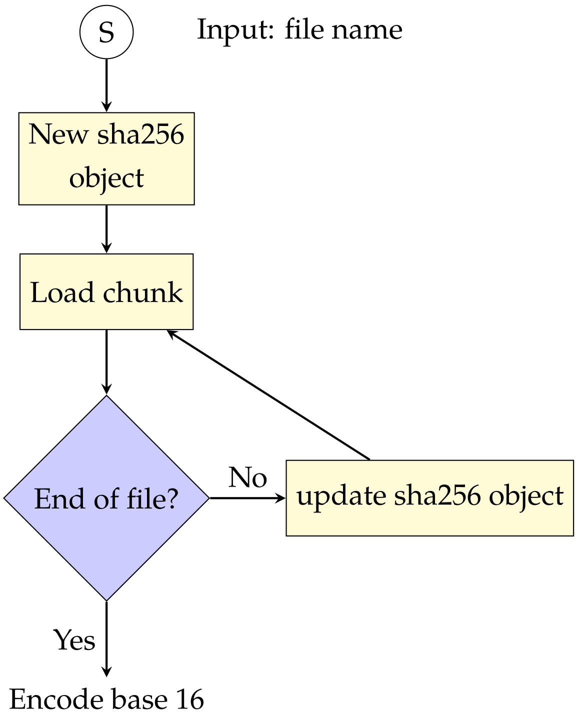

Fig. 4.6: Calculate hash of local file

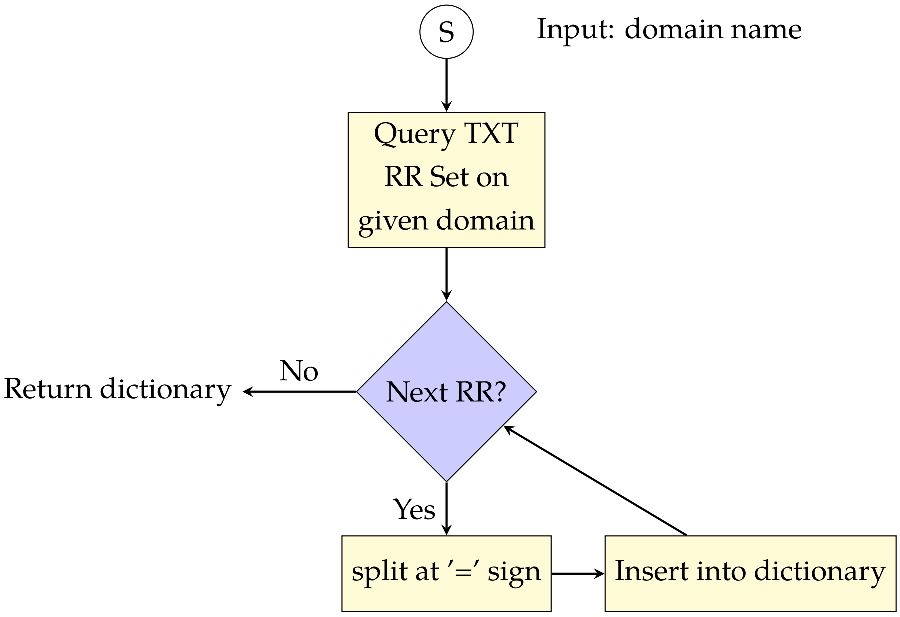

Fig. 4.7: Get information about metadata from DNS

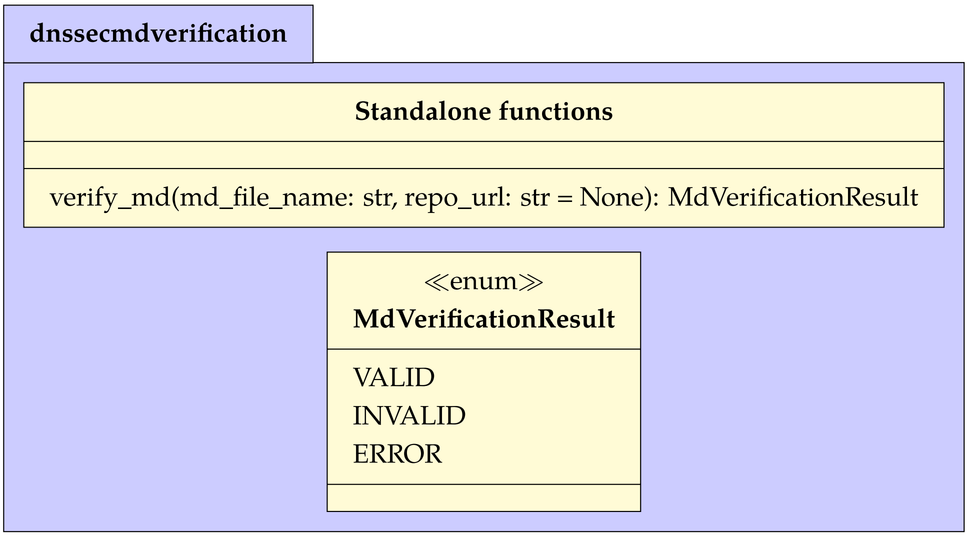

Fig. 4.8: UML diagram of the library implementing extended version

5 DEVELOPMENT AND TESTING ENVIRONMENT

When developing a piece of software into already working environment and especially in networking, the program needs to communicate with many different ones in order to perform its duty. Therefore it is convenient for programmers to create an isolated and reproducible development environment in which the software can be created without dealing with changes of the other components.

In this chapter, I will describe the configuration of the virtual environment used for development of this thesis. In the beginning, the concept of init system is briefly discussed, because it is needed for spawning DNS servers. The next section describes configuration management that is not based on shell scripts. Suitable virtualization tools are chosen in the following section and in the final two, all of these tools are put together in order to spin up a reproducible testing environment and run the extension.

5.1 Running services in modern Linux environment

A service (also known as a daemon) is a non-interactive process running in an operating system. It is usually started during the boot time by the init system. Examples of such services include public-facing DNS server or HTTP server as well as internal systems like virtualization daemon or Inter-Process Communication (IPC) bus.

Traditionally the init system executed shell scripts located in the */etc/init.d* directory which specified necessary steps required to run services. This approach can be thought of as the imperative approach to system configuration because the system administrator is required to write a script that consists of a series of steps.

The traditional way was very flexible but it required a lot of work with shell scripts, which is error prone, and did not allow for modern features such as parallel service start-up. In order to improve the boot process, a new init system called **systemd** [30] was created. It requires only a simple configuration file for a service to be specified. This file has a syntax similar to INI files [31] as can be seen from Listing 6. As opposed to the previous approach, this one can be thought of as a declarative one, because it does not specify steps required to run a service, it only describes the service and **systemd** takes care of the rest.

5.2 Server configuration with automation engine

Similarly to the init scripts, the system configuration was traditionally done using shell scripts which were written by the system administrators. Although this way works fine, it requires a lot of scripting and is not very reusable. As a result, a lot of automation tools appeared during past few years. I am going to mention only Ansible [32] because it seems to be the most popular one in the Fedora community.

Let me describe how Ansible works in few sentences. Given a scenario with one workstation and multiple servers a system administrator installs Ansible to his workstation and

```
[Unit]
Description=Some simple daemon

[Service]
Type=forking
ExecStart=/usr/sbin/my-simple-daemon -d

[Install]
WantedBy=multi-user.target
```

Listing 6: Example of a systemd service file

writes a configuration file called a playbook. He can now use the playbook with Ansible in order to configure all the servers without the need to prepare them in any way. This works because Ansible is using SSH in the background and thus not need any additional software on the server side. Compared to the traditional approach, Ansible takes care of the tasks that were written by hand such as command execution, error handling or abstracting over different distributions.

The configuration files, playbooks, can be further encapsulated into roles. A role contains a unit of configuration that is tight together in some way. For example a role for Apache server configuration. These roles can be in turn executed from another playbooks creating a hierarchy of configuration files. Listing 7 is an example of an Ansible playbook with two tasks.

```
---
- name: Install list of packages
  dnf: name={{item}} state=installed
  with_items:
      - fish          # For friendly interactive usage :)
      - nc            # Debugging
      - tmux          # Essential tool for terminal
- name: Set ip forwarding on in /proc and do not reload the sysctl file
  sysctl:
      name: net.ipv4.ip_forward
      value: 1
      sysctl_set: yes
```

Listing 7: Example of a Ansible playbook

5.3 Choosing virtual environment

Virtual machines are nowadays gaining on popularity and there are a lot of tools that can help with setting up a virtual machine (VM); as an example, I would mention Vagrant, which was described in the theoretical section. My original plan was to use it, unfortunately the official Fedora Vagrant box is based on the Fedora cloud image, which is a minimal image missing a lot of packages including support for any language except of English and encoding except for UTF-8 (see bug report [29]). This prevented me from installing some packages and continue my work. Therefore I decided to use the full server image which works just fine. Unfortunately there is no Vagrant box made from this image, so I had to either create one, or use plain virtual machine. I decided to pick the later option as the configuration was done using simple Ansible playbook anyway so all I had to do was to synchronize directories between the host and guest operating systems by hand. For this purpose **lsyncd** was used. It is a daemon, that watch directories on the host system and synchronize any changes made in files in these directories.

5.4 Virtual machine configuration

Inside the VM a simulation of the Domain Name System is built. There is a root server with DNSSEC support, top-level domain servers (e.g. com.) and few 2nd level domains. (e.g. example.com.). All of these servers are using the ”bind” authoritative DNS server implementation, they are started using the systemd init system and each of them has one address on the loopback interface. (Figure 5.1)

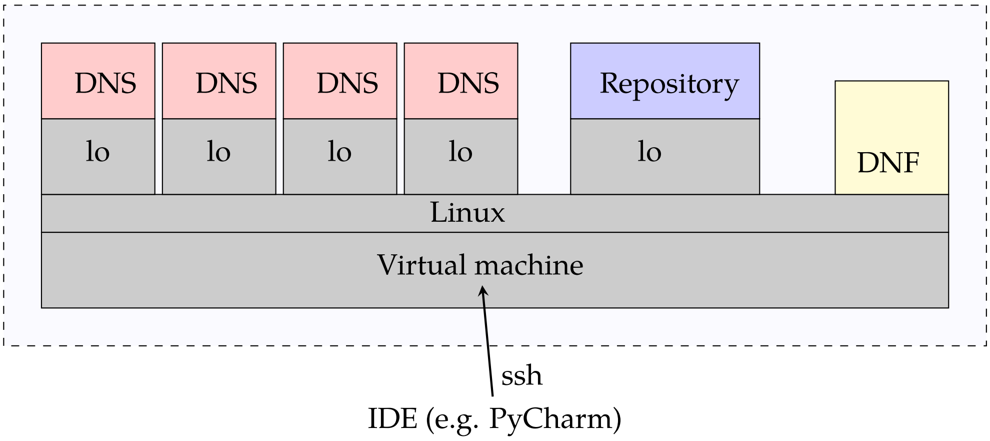

Fig. 5.1: Virtual machine configuration for development and testing purposes

As mentioned before, the whole environment is configured using Ansible. I chose this system because it uses simple, declarative approach to system administration. The main file here is the top level playbook (as explained in section about Ansible) and is defined in Listing 8. An example of a role can be seen in Figure 9. This role is used to spin up all the

DNS servers in the testing environment. Following the role definition, systemd service files are used to spin up name servers. Service files also use simple declarative approach as opposed to init scripts (Listing 10).

The two remaining roles, fedora and repository, are used to configure the system and spin up an HTTP server that provides testing packages.

```
---
- hosts: vm
  gather_facts: false
  become_user: root
  become: yes
  roles:
      - fedora
      - dns
      - repository
```

Listing 8: Ansible playbook to configure testing environment

```
---
- name: Run DNS servers
  systemd: state=restarted name={{item}}
  with_items:
    - root-server
    - com-server
    - example-com-server
    - wrongconfig-com-server
    - notsigned-com-server
    - resolver
```

Listing 9: Ansible role for DNS hierarchy configuration

```
[Service]
Type=forking
Environment=NAMEDCONF=/vagrant/com-server/named.conf
ExecStart=/usr/sbin/named -u named -c ${NAMEDCONF}
```

Listing 10: systemd service file for Bind nameserver

5.5 Testing the implementation using the virtual environment

Once the environment is up and running the implementation can be tested by adding a new trust anchor into the Unbound so that it uses testing root servers when starting iterative lookup.

A testing transaction can by started by executing the dnf script with DNSSEC extension. In the beginning, all imported keys are verified:

```
$ bin/dnf-3 --repo=diploma-thesis install test-good-sig -y -d9
...
DNSSEC extension: Testing already imported keys for their validity.
DNSSEC extension: Key associated with identity fedora-27@fedoraproject.org
                  was tested with result: Validity.PROVEN_NONEXISTENCE
DNSSEC extension: Key associated with identity msehnout#python-gnupg
                  @copr.fedorahosted.org was tested with result:
                  Validity.PROVEN_NONEXISTENCE
```

Then the downloaded metadata are verified:

```
...
repo: downloading from remote: diploma-thesis, _Handle: metalnk: None,
      mlist: None, urls ['http://127.0.0.1/f27/'].
Diploma thesis testing repository     912 kB/s | 1.0 kB     00:00
{'hash': 'b6359151f847834a2dd3dbcd20e631393826ba90999b40ce8c00afd54daf2b35',
 'ts': '24/03/2018', 'alg': 'sha256', 'val': '9d'}
DNSSEC extension: Repository metadata considered: MdVerificationResult.VALID
...
```

Using the metadata file, dependencies are calculated and appropriate packages are downloaded:

```
--> Starting dependency resolution
---> Package test-good-sig.x86_64 0.1.0-1.fc27 will be installed
--> Finished dependency resolution
timer: depsolve: 6 ms
Dependencies resolved.
============================================================
 Package        Arch    Version       Repository      Size
============================================================
Installing:
 test-good-sig  x86_64  0.1.0-1.fc27  diploma-thesis  9.9 k

Transaction Summary
============================================================
Install  1 Package
...
```

Once the package is downloaded, RPM tries to verify its signature and since the key is not imported it will fail and a new key is imported:

```
DNSSEC: keyurl=http://127.0.0.1/keys/packager.asc
DNSSEC extension: Key for user packager@example.com is valid.
Importing GPG key 0x59E08E43:
 Userid     : "RPM Packager (The guy who creates packages) <packager@
              example.com>"
 Fingerprint: CC14 FBDE B7E9 02A4 6D8B 2374 C292 9F55 59E0 8E43
 From       : http://127.0.0.1/keys/packager.asc
DNSSEC extension: The key has been approved.
Key imported successfully
```

In this manner, all use cases were tested by hand. Of course for production use case it will be beneficial to automate these tests so that they can be integrated into a build pipeline.

I have also included a video presentation of the testing environment in action. It is available in the attachment as a file with name *demo.flv* (see attachment B). It shows how to run a virtual machine, configure it using the provided automation tools and how to try the extended dnf.

6 DEPLOYMENT

Although not directly mentioned in the assignment of this thesis, deployment on the server side is also important because without wide adoption of the server side the client side extension will not be useful. A really nice use case for the key verification using DNS is in the Fedora Community Build Service (COPR). It is a place where users can create their own, private repositories and distribute either their software, or software that is not available in the Fedora main repository. It is common for one user to have multiple repositories from COPR enabled at the same time, so it is a perfect candidate for deploying automatic key verification. It will also be a good way to measure the impact on an infrastructure because every repository will have its own key and there are circa 70k repositories.

6.1 Dynamic Updates in the Domain Name System

Although the DNS system was originally meant as a static database where records are written by hand a standard for dynamic updates was created in RFC 2136 [33]. This allows for updates without restarting the authoritative DNS server. An update can be applied using the **nsupdate** utility which is available as part of the Bind 9 DNS server implementation. Its usage is fairly simple, it takes a file with server specification and updates to be applied (Listing 11).

```
server 127.0.0.1
update add www.example.com 3600 A 127.0.0.5
send
```

Listing 11: Example of a dynamic DNS update

6.2 Extending COPR with DNS service

COPR architecture consist of a builder backend, keygen service and a web frontend [26]. The DNS aided key verification can be implemented as a service that communicates with other COPR services over a Web API and as an output, it sends dynamic DNS updates to the authoritative DNS server responsible for the copr.fedorainfracloud.org. domain. The idea is depicted in Figure 6.1.

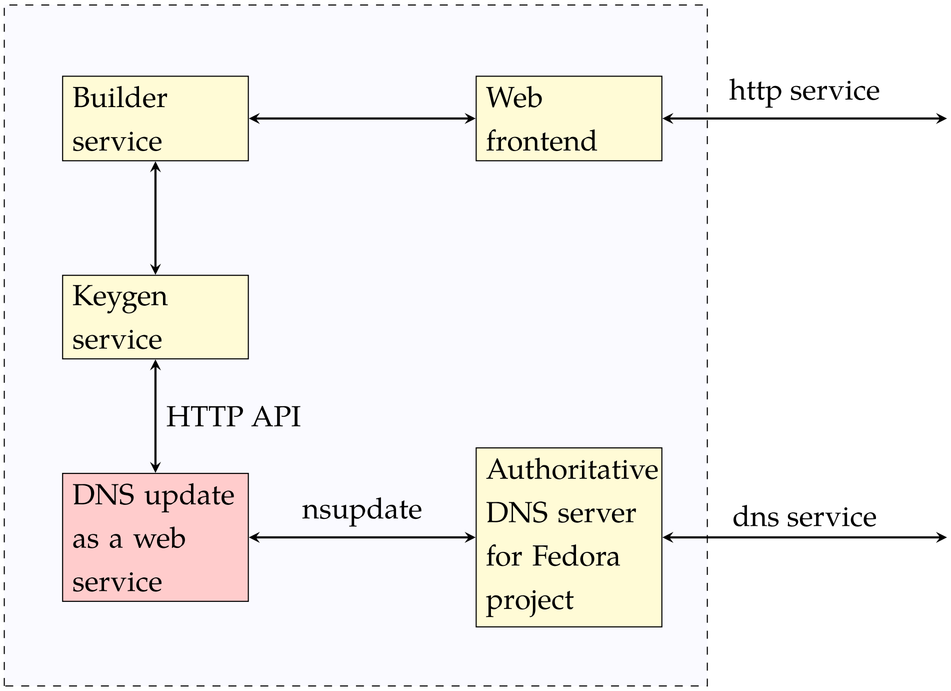

Fig. 6.1: COPR repository with DNS frontend

7 PERFORMANCE EVALUATION

This project is dealing with package management, which is probably one of the most important parts of an operating system. Therefore it is undesirable to introduce long delays or substantial computational load. In this chapter, few performance aspects of the implemented system will be described and measured.

As in previous chapters, the evaluation can be split into client and server side. On the client side, dnf package manager is explored, whereas, on the server side, the focus is on an authoritative DNS server.

7.1 Impact of the extension on the client side

The dnf package manager was extended with two libraries, from which both depend on libundound and few other modules from the standard Python library. The performance can be considered from various ways, I am going to describe these four: computational heavy tasks, memory consumption, additional dependencies and IO heavy operations.

As far as the computations are considered, the extension consist of few parsing sections and cryptographic operations. All of these are pretty fast given that one transaction can take tens of seconds, even minutes[1]. The memory consumption will increase especially by dynamic linking of the libundound library, but again its size is negligible compared to gigabytes of memory that is usually available nowadays.

```
# Size of libundound shared library (that is a file to be loaded into
# the memory space of the process that wants to use it)
$ du -h /usr/lib64/libunbound.so.2.5.8
1.2M    /usr/lib64/libunbound.so.2.5.8
```

Libundound is also the only new dependency introduced and it is available in official Fedora repositories, so the only remaining problem is IO operations.

The most expensive operation will be a lookup of all necessary resource records that are needed to perform DNSSEC validation. The best case scenario is a local caching resolver in which case most of the queries will go only over the loopback interface. This case was measured in a virtual environment, where all DNS servers are running locally and the result is depicted in Figure. On the other hand, the worst case scenario is an iterative lookup without any cache using authoritative servers on the Internet. As far as the result from Figure 7.1 are considered, the time delay caused by the DNS lookup is negligible given that the fastest transactions take at least few tenths of a second (see Figure 7.2 where single package installation from a local repository is tested).

---

[1] Especially metadata download is extremely slow because those in official repositories are usually 10MB in size

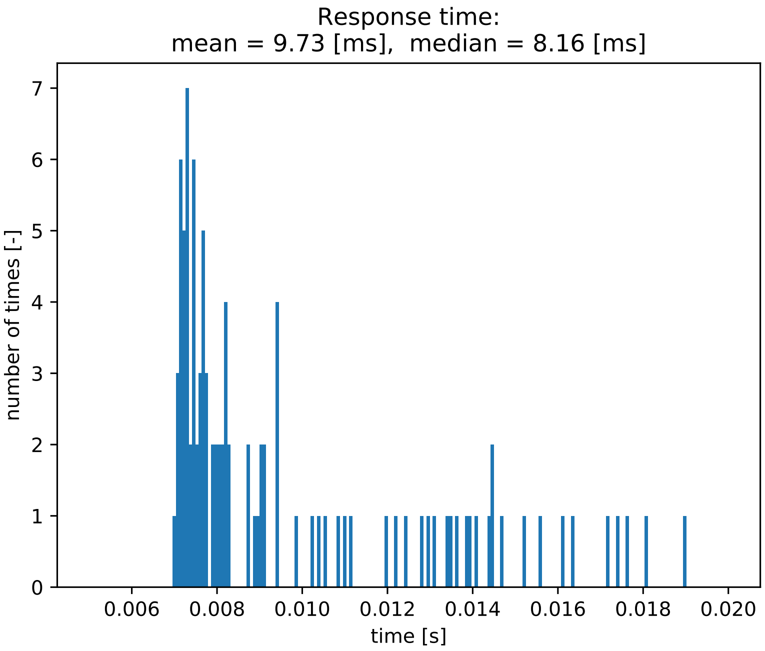

Fig. 7.1: Best case scenario for DNS lookup - all queries goes only over loopback

7.1.1 Comparison of extended dnf with its default version

In this test case, time of a single transaction was measured. On one side the default dnf version from plain installation was tested, on the other my development version with both minimal and extended version in use. The transaction consisted of a single package installation from a single enabled repository. In case of modified dnf, there were two keys checked for revocation before the transaction and one was automatically accepted during the transaction. The result can be seen in Figure 7.2. The difference is circa 50ms in this scenario. The source code of this test can be seen in Listing 12. Please note, that the tested executable is passed as an argument. More specifically it is available as $1 variable because that is the way in which shell scripts pass arguments.

```
run_test() {
   # Run dnf with single repository enabled, install a package and measure time
   RESULT=$({ time $1 --repo=diploma-thesis install test-good-sig -y; } 2>&1 |
               grep '^real')
   printf "${1} ${RESULT}\n"
   # Clean up
   rpm -e gpg-pubkey-59e08e43-5a96cdec &> /dev/null
   dnf --repo=diploma-thesis remove test-good-sig -y &> /dev/null
}
```

Listing 12: Bash script for dnf transaction time measurement

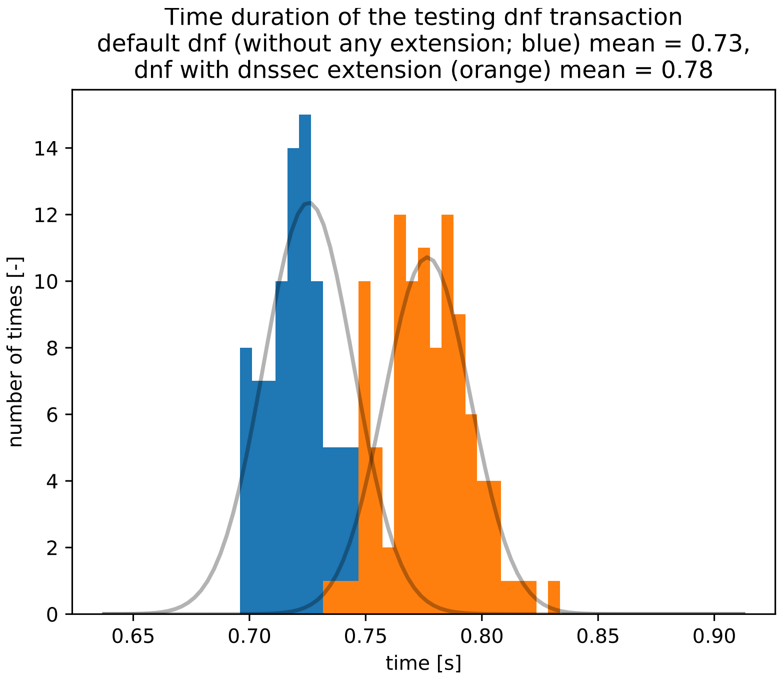

Fig. 7.2: Testing dnf transaction - comparison of default and extended version

7.2 Impact of the extension on the server side

On the server side, all information is stored inside a DNS server. Except for some additional administrative burden, there should be no problem. Response time is given by the data structure used for resource record storage. For this purpose, a structure with reasonably quick lookup time is appropriate; for example, a red-black tree. Therefore additional records will not affect the server's response time. The only metric, that can be affected is a memory usage, but it is tricky to measure since we have multiple approaches to do so. Let me divide them into two groups: a white box and a black box approaches. In the white box case, I know how much memory is used by the process and I know what it is used for. The downside of this method is that either the process needs to have built-in memory analyzer or I need to run it with a memory profiler. The black box approach, on the other hand, does not take into account the purpose of allocated memory, it just states how much memory is the process using. I am going to use the black box method, more specifically how much memory is currently residing in the Random Access Memory (RAM). This information can be obtained from the Linux kernel via its interface in the /proc pseudo-filesystem [2]. More specifically in the parameter named Virtual memory Resident Set Size (VmRSS) which states how much RAM is the process using for heap allocations, memory mapped files and shared memory [34].

---

[2] It is considered pseudo because files in this directory are not real files on a disk, they are memory structures in the Linux kernel mapped into the filesystem.

7.2.1 Test case for the Bind9 authoritative server

I created 100 000 random OPENPGPKEY records and then submitted all of them into the running DNS server. A file with an update looks like in Listing 13. It is using the **nsupdate** utility, that was described in section 6.1.

```
server 127.0.0.1
update add dyn1.example.com. 3600 TYPE61 \# 1024
0a990008ac37...<more data>..2884da0eef1
send
```

Listing 13: File with a dynamic update for authoritative DNS server

For each record, one such file was created. Then I used a simple Bash script to upload them all and measure memory usage for each iteration. The script can be seen in Listing 14.

```
# Get PID of running Bind9
PID=$(pgrep named)
echo "Running memory consumption test with PID=${PID}"
# Create an empty file for results
printf '' > result
# Run the test for 100 000 keys
for i in $(seq 0 100000)
do
  # Insert the key into the server using dynamic DNS updates
  nsupdate "b/${i}"
  printf "${i} "
  # Measure memory usage and append it to the results file
  grep VmRSS "/proc/${PID}/status" | tee --append result
done
```

Listing 14: Bash script for memory usage measurement

The result is depicted in Figure 7.3. Starting from approximately 15MB, the memory usage raised to 140MB when storing 100 000 OPENPGPKEY records. Given the fact that available memory nowadays is at least few GB this should not expose any trouble to the system administrators. The test was run in a virtual machine with CentOS7 Core and bind package version 9.9.4-51.el7_4.2.x86_64.

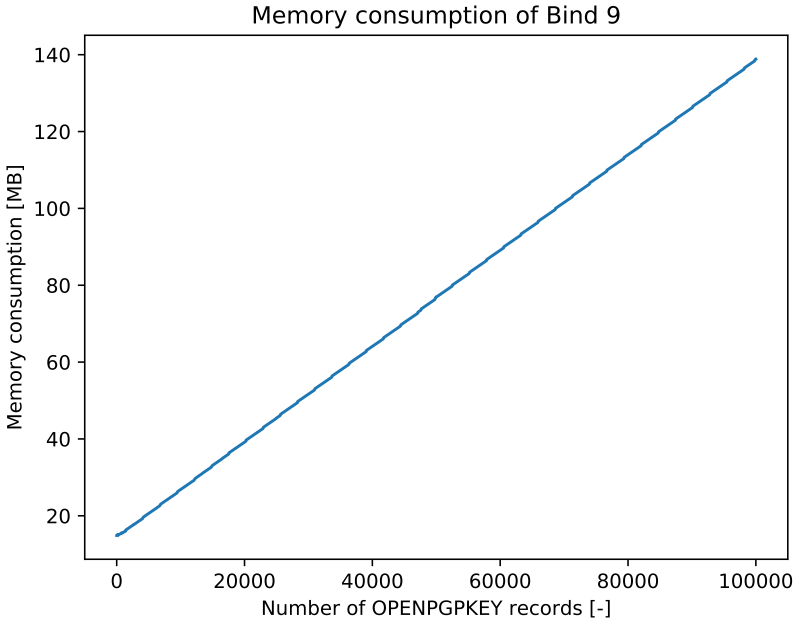

Fig. 7.3: Memory consumption of Bind9 given 0 to 100 000 OPENPGPKEY resource records

8 CONCLUSION AND FUTURE WORK

In this thesis, the problem of safe software distribution is addressed as well as its implementation in Fedora Linux. In the beginning of this thesis theoretical background is described, starting from cryptography, followed by DNS with DNSSEC extension and finally software distribution and possible attacks against package managers.

The current implementation of DNF package manager is examined for the way verification keys are handled. Development tools like programming languages and virtualization are also briefly mentioned.

I propose a key verification mechanism based on the DNS system with the DNSSEC extension. I describe necessary components to make this system work on both client and server side. Two versions are described: minimal version and extended one. The minimal one defines used resource record type and domain on which a verification key would be stored and steps to be taken by a package manager in order to decide about key validity. In the extended version, repository metadata and their verification key are also considered for inclusion in the DNS database to prevent certain kinds of attacks.

The proposed methods were implemented using the Python programming language. Each one of them is implemented as a single-file library which is then integrated into the code of the dnf package manager. The code is described in the chapter about implementation (4) and it is available on the attached CD. A testing environment was created using virtualization technologies and configuration engine Ansible. All necessary techniques were described in Chapter 5. Configuration files are also available on the CD. In Chapter 6 a method for server-side deployment is described as a case study for Fedora Community Build Service. Finally, the performance of the whole solution is discussed and measured in Chapter 7.

With this thesis and the source codes, I can cooperate with the upstream communities in order to push this system into production systems. Once the client-side support is in place, it will be necessary to inform the public about the possibility to publish GPG keys in DNS so that owners of RPM repositories are aware of it.

If this system proves itself useful, it can relieve users from manual key verification and help to secure automatic server deployments, as well as graphical tools, build on top of package managers, as these skip verification as well. Nevertheless, it is important to work on DNSSEC availability to the end users, because in some networks DNS resolvers might not provide necessary resource records. There are projects that provide open DNS resolvers over TLS, e.g. Cloudflare 1.1.1.1 server, but it is not possible to use only public DNS resolvers as they do not provide information about internal networks, for example, when working in a corporate.

BIBLIOGRAPHY

[1] EDWARD C. BAILEY. *Maximum RPM: taking the Red Hat package manager to the limit*. Rev. 1.0. Durham, NC: Red Hat Software, 1997. ISBN 18-881-7278-9.

[2] JEŘÁBEK, Jan. *Pokročilé komunikační techniky*. 5. 2. 2015. Brno, 2015.

[3] Hosts_(file). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-09-13]. Dostupné z: https://en.wikipedia.org/wiki/Hosts_(file)

[4] *RFC 7816: DNS Query Name Minimisation to Improve Privacy*. March 2016. Internet Engineering Task Force (IETF).

[5] PAAR, Christof a Jan. PELZL. *Understanding cryptography: a textbook for students and practitioners*. New York: Springer, c2010. ISBN 978-3-642-04101-3.

[6] SMART, Nigel. *Cryptography: an introduction*. 3rd ed. London [u.a.]: McGraw-Hill, 2003. ISBN 978-007-7099-879.

[7] ZEMAN, Václav. *Elliptic Curve Cryptography - MKRI slides*. Brno, 2016.

[8] RFC 4255: Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 2006 [cit. 2017-11-07]. Dostupné z: https://tools.ietf.org/html/rfc4255

[9] RFC 4398: Storing Certificates in the Domain Name System (DNS). *RFC online* [online]. California: Internet Engineering Task Force (IETF), 2006 [cit. 2017-11-07]. Dostupné z: https://tools.ietf.org/html/rfc4398

[10] RFC 7929: DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 2016 [cit. 2017-11-07]. Dostupné z: https://tools.ietf.org/html/rfc7929

[11] How DNSSEC works. *Blog CloudFlare* [online]. San Francisco: CloudFlare, 2017 [cit. 2017-10-16]. Dostupné z: https://www.cloudflare.com/dns/dnssec/how-dnssec-works/

[12] RFC 2535: Domain Name System Security Extensions. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 1999 [cit. 2017-11-04]. Dostupné z: https://tools.ietf.org/html/rfc2535

[13] RFC 3845: DNS Security (DNSSEC) NextSECure (NSEC) RDATA Format. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 2004 [cit. 2017-11-04]. Dostupné z: https://tools.ietf.org/html/rfc3845

[14] RFC 4880: OpenPGP Message Format. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 2007 [cit. 2017-10-24]. Dostupné z: https://tools.ietf.org/html/rfc4880

[15] RFC 5155: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 2008 [cit. 2017-11-04]. Dostupné z: https://tools.ietf.org/html/rfc5155

[16] Binary package guide. *Gentoo wiki* [online]. Global: Gentoo Foundation, 2017 [cit. 2017-11-15]. Dostupné z: https://wiki.gentoo.org/wiki/Binary_package_guide#Understanding_the_binary_package_format

[17] Manifest. *Gentoo wiki* [online]. Global: Gentoo Foundation, 2017 [cit. 2017-11-16]. Dostupné z: https://wiki.gentoo.org/wiki/Repository_format/package/Manifest

[18] *Manual page - deb(5)*. Dostupné také z: http://man7.org/linux/man-pages/man5/deb.5.html

[19] HOWTO: GPG sign and verify deb packages and APT repositories. *Packagecloud blog* [online]. San Francisco: packagecloud.io, 2014 [cit. 2017-11-16]. Dostupné z: https://blog.packagecloud.io/eng/2014/10/28/howto-gpg-sign-verify-deb-packages-apt-repositories/

[20] CAPPOS, Justin, Justin SAMUEL, Scott BAKER a John H. HARTMAN. A look in the mirror: Attacks on Package Managers. In: *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*. New York, New York, USA: ACM Press, 2008, s. 565-574. DOI: 10.1145/1455770.1455841. ISBN 9781595938107. Dostupné také z: http://portal.acm.org/citation.cfm?doid=1455770.1455841

[21] SAMUEL, Justin, Nick MATHEWSON, Justin CAPPOS a Roger DINGLEDINE. Survivable key compromise in software update systems. In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*. New York, New York, USA: ACM Press, 2010, s. 61-72. DOI: 10.1145/1866307.1866315. ISBN 9781450302456. Dostupné také z: http://portal.acm.org/citation.cfm?doid=1866307.1866315

[22] YOUNAN, Yves. *Efficient countermeasures for software vulnerabilities due to memory management errors*. Leuven, 2008. Ph.D. thesis. Katholieke universiteit Leuven.

[23] SILBERSCHATZ, Abraham., Peter B. GALVIN a Greg. GAGNE. *Operating system concepts*. Ninth edition. Hoboken, NJ: Wiley, 2013]. ISBN 978-111-8063-330.

[24] Security vulnerabilities fixed in Firefox 56. *Mozilla Foundation Security Advisory* [online]. Mountain View, California: Mozilla, 2017 [cit. 2017-11-08]. Dostupné z: https://www.mozilla.org/en-US/security/advisories/mfsa2017-21/

[25] Systemd issue tracker. *Github repository* [online]. 2017 [cit. 2017-11-30]. Dostupné z: https://github.com/systemd/systemd/issues/6426

[26] Developer Documentation. *COPR Documentation* [online]. 2017 [cit. 2017-12-03]. Dostupné z: https://docs.pagure.org/copr.copr/developer_documentation.html#developer-documentation

[27] PyUndound documentation. *Unbound homepage* [online]. [cit. 2018-05-09]. Dostupné z: https://www.unbound.net/documentation/pyunbound/index.html

[28] RFC 1464: Using the Domain Name System To Store Arbitrary String Attributes. *RFC online* [online]. California: Internet Engineering Task Force (IETF), 1993 [cit. 2018-05-09]. Dostupné z: https://tools.ietf.org/html/rfc1464

[29] The base-cloud Vagrant images set ASCII locale when client does not use en_US.utf8/C.utf8. *Red Hat Bugzilla* [online]. North Carolina, United States: Red Hat, 2017 [cit. 2018-05-09]. Dostupné z: https://bugzilla.redhat.com/show_bug.cgi?id=1432426

[30] Systemd. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2018 [cit. 2018-05-09]. Dostupné z: https://en.wikipedia.org/wiki/Systemd

[31] INI File. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-05-09]. Dostupné z: https://en.wikipedia.org/wiki/INI_file

[32] Ansible in depth. In: *Ansible Whitepaper* [online]. 2017 [cit. 2018-05-09]. Dostupné z: https://www.ansible.com/resources/whitepapers/ansible-in-depth

[33] Dynamic Updates in the Domain Name System (DNS UPDATE). *RFC online* [online]. California: Internet Engineering Task Force (IETF), 1997 [cit. 2018-05-09]. Dostupné z: https://tools.ietf.org/html/rfc2136

[34] Proc(5). *Linux man page* [online]. [cit. 2018-05-11]. Dostupné z: http://man7.org/linux/man-pages/man5/proc.5.html

LIST OF SYMBOLS, PHYSICAL CONSTANTS AND ABBREVIATIONS

DNS	Domain Name System
GUI	Graphical User Interface
RPM	RPM Package Manager
YUM	Yellowdog Updater, Modified
DNF	Dandified Yum
RHEL	Red Hat Enterprise Linux
IP	Internet Protocol
ZSK	Zone Signing Key
KSK	Key Signing Key
RSA	Rivest, Shamir, Adleman
DHKE	Diffie-Hellman key exchange
DSA	Digital Signature Algorithm
ECDH	Elliptic Curve Diffie–Hellman Key Exchange
TLS	Transport Layer Security
pgp	Pretty Good Privacy
DS	Delegation Signer
API	Application Programming Interface
pgpme	PGP Made Easy
VM	virtual machine
KVM	Kernel Virtual Machine
TUF	The Update Framework
XML	Extensible Markup Language
URL	Uniform Resource Locator
COPR	Fedora community build service
RAM	Random Access Memory
VmRSS	Virtual memory Resident Set Size
IPC	Inter-Process Communication

LIST OF APPENDICES

A PATCHES FOR DNF PACKAGE MANAGER

In open source development, it is common to send proposed features as patches. That is a file containing changes to the project. My contribution was split into the minimal and extended part.

A.1 Minimal version

```
From d7e262df7dc95dac908175e9ce45d19283cc0673 Mon Sep 17 00:00:00 2001
From: Martin Sehnoutka <msehnout@redhat.com>
Date: Fri, 11 May 2018 10:32:26 +0200
Subject: [PATCH] dns extension for automatic key management

---
 dnf/CMakeLists.txt                  |   1 +
 dnf/base.py                         |  19 +++-
 dnf/cli/main.py                     |   3 +
 dnf/dnssec/CMakeLists.txt           |   2 +
 dnf/dnssec/dnsseckeyverification.py | 211 ++++++++++++++++++++++++++++++++++++
 dnf/dnssec/libunbound.conf          |   4 +
 6 files changed, 239 insertions(+), 1 deletion(-)
 create mode 100644 dnf/dnssec/CMakeLists.txt
 create mode 100644 dnf/dnssec/dnsseckeyverification.py
 create mode 100644 dnf/dnssec/libunbound.conf

diff --git a/dnf/CMakeLists.txt b/dnf/CMakeLists.txt
index 88f9103a..26144f19 100644
--- a/dnf/CMakeLists.txt
+++ b/dnf/CMakeLists.txt
@@ -9,3 +9,4 @@ ADD_SUBDIRECTORY (conf)
 ADD_SUBDIRECTORY (rpm)
 ADD_SUBDIRECTORY (yum)
 ADD_SUBDIRECTORY (db)
+ADD_SUBDIRECTORY (dnssec)
diff --git a/dnf/base.py b/dnf/base.py
index cce1bbcb..f1e124e8 100644
--- a/dnf/base.py
+++ b/dnf/base.py
@@ -37,6 +37,7 @@ import dnf.comps
 import dnf.conf
 import dnf.conf.read
 import dnf.crypto
+import dnf.dnssec.dnsseckeyverification as dnssec
 import dnf.drpm
 import dnf.exceptions
 import dnf.goal
@@ -2166,6 +2167,12 @@ class Base(object):
                     logger.info(msg, keyurl, info.short_id)
                     continue
```

```
+                # DNS Extension: create a key object, pass it to the verification class
+                # and print its result as an advice to the user.
+                dns_input_key = dnssec.KeyInfo.from_rpm_key_object(info.userid, info.raw_key)
+                dns_result = dnssec.DNSSECKeyVerification.verify(dns_input_key)
+                logger.info(dnssec.nice_user_msg(dns_input_key, dns_result))
+
                 # Try installing/updating GPG key
                 info.url = keyurl
                 dnf.crypto.log_key_import(info)
@@ -2173,7 +2180,17 @@ class Base(object):
                 if self.conf.assumeno:
                     rc = False
                 elif self.conf.assumeyes:
-                    rc = True
+                    # DNS Extension: We assume, that the key is trusted in case it is valid,
+                    # its existence is explicitly denied or in case the domain is not signed
+                    # and therefore there is no way to know for sure (this is mainly for
+                    # backward compatibility)
+                    if dns_result == dnssec.Validity.VALID or \
+                            dns_result == dnssec.Validity.PROVEN_NONEXISTENCE:
+                        rc = True
+                        logger.info(dnssec.any_msg("The key has been approved."))
+                    else:
+                        rc = False
+                        logger.info(dnssec.any_msg("The key has been rejected."))

                 # grab the .sig/.asc for the keyurl, if it exists if it
                 # does check the signature on the key if it is signed by
diff --git a/dnf/cli/main.py b/dnf/cli/main.py
index 519c5533..e7900378 100644
--- a/dnf/cli/main.py
+++ b/dnf/cli/main.py
@@ -42,6 +42,8 @@ import os
 import os.path
 import sys

+import dnf.dnssec.dnsseckeyverification as dnssec
+
 logger = logging.getLogger("dnf")


@@ -87,6 +89,7 @@ def _main(base, args, cli_class, option_parser):

     # our core object for the cli
     base._logging._presetup()
+    dnssec.RpmImportedKeys.check_imported_keys_validity(logger)
     cli = cli_class(base)

     # do our cli parsing and config file setup
diff --git a/dnf/dnssec/CMakeLists.txt b/dnf/dnssec/CMakeLists.txt
new file mode 100644
```

```
index 00000000..37170b09
--- /dev/null
+++ b/dnf/dnssec/CMakeLists.txt
@@ -0,0 +1,2 @@
+FILE(GLOB dnssec *.py)
+INSTALL (FILES ${dnssec} DESTINATION ${PYTHON_INSTALL_DIR}/dnf/dnssec)
diff --git a/dnf/dnssec/dnsseckeyverification.py b/dnf/dnssec/dnsseckeyverification.py
new file mode 100644
index 00000000..a98a313a
--- /dev/null
+++ b/dnf/dnssec/dnsseckeyverification.py
@@ -0,0 +1,211 @@
+from enum import Enum
+import unbound
+import hashlib
+import base64
+import re
+import subprocess
+
+
+def email2location(email_address, tag="_openpgpkey"):
+    # type: (str, str) -> str
+    """
+    Implements RFC 7929, section 3
+    https://tools.ietf.org/html/rfc7929#section-3
+    """
+    split = email_address.split("@")
+    if len(split) == 2:
+        # Take the first part of the email address (local)
+        # and apply the algorithm.
+        local = split[0]
+        domain = split[1]
+        hash = hashlib.sha256()
+        hash.update(local.encode('utf-8'))
+        digest = base64.b16encode(hash.digest()[0:28])\
+            .decode("utf-8")\
+            .lower()
+        # Compose and return the result
+        return digest + "." + tag + "." + domain
+    else:
+        return "Error"
+
+
+# Possible results of the verification process
+class Validity(Enum):
+    VALID = 1
+    REVOKED = 2
+    PROVEN_NONEXISTENCE = 3
+    RESULT_NOT_SECURE = 4
+    BOGUS_RESULT = 5
+    ERROR = 9
```

```
+
+
+# Just a placeholder class to denote explicit
+# denial of the key existence using the Python's
+# type system.
+class NoKey:
+    pass
+
+
+# Data class that encapsulates email and key
+class KeyInfo:
+    def __init__(self, email=None, key=None):
+        self.email = email
+        self.key = key
+
+    @staticmethod
+    def from_rpm_key_object(userid, raw_key):
+        # type: (str, bytes) -> KeyInfo
+        # The key object from the RPM library comes in an unfortunate
+        # text format. It is necessary to parse it in order to work with
+        # it.
+        # This method can be thought of as a constructor from the RPM
+        # key object.
+        email = re.search('<(.*@.*)>', userid).group(1)
+        key = raw_key.decode('ascii').split('\n')
+        start = next(i for i in range(0, len(key))
+                     if key[i] == '-----BEGIN PGP PUBLIC KEY BLOCK-----')
+        stop = next(i for i in range(0, len(key))
+                    if key[i] == '-----END PGP PUBLIC KEY BLOCK-----')
+        cat_key = ''.join(key[start + 2:stop - 1]).encode('ascii')
+
+        ret_val = KeyInfo()
+        ret_val.email = email
+        ret_val.key = cat_key
+        return ret_val
+
+
+class DNSSECKeyVerification:
+    # Mapping from email address to b64 encoded public key or NoKey in case of proven nonexistence
+    __cache = {}
+    # type: Dict[str, Union[str, NoKey]]
+    # The type signature says, that the email is of string type as well as the key.
+
+    @staticmethod
+    def __cache_hit(key_union, input_key_string):
+        # type: (Union[str, NoKey], str) -> Validity
+        # Return result from the cache
+        if key_union == input_key_string:
+            # In this case, the key of type string and is the same
+            return Validity.VALID
+        elif key_union is NoKey:
```

```
+            # As opposed to this case where it is of NoKey type
+            return Validity.PROVEN_NONEXISTENCE
+        else:
+            # It is string, but a different one
+            return Validity.REVOKED
+
+    @staticmethod
+    def __cache_miss(input_key):
+        # type: (KeyInfo) -> Validity
+        # Define constant, that is missing in the libunbound API
+        RR_TYPE_OPENPGPKEY = 61
+        # Create the stub resolver and query given domain for the OPENPGP
+        # RRSet in the Internet class.
+        ctx = unbound.ub_ctx()
+        ctx.config("/etc/dnf/libunbound.conf")
+        status, result = ctx.resolve(email2location(input_key.email),
+                                     RR_TYPE_OPENPGPKEY, unbound.RR_CLASS_IN)
+        # Implement algorithm, that is described in the chapter about proposed
+        # solution.
+        if status != 0:
+            return Validity.ERROR
+        if result.bogus:
+            return Validity.BOGUS_RESULT
+        if not result.secure:
+            return Validity.RESULT_NOT_SECURE
+        if result.nxdomain:
+            return Validity.PROVEN_NONEXISTENCE
+        if not result.havedata:
+            return Validity.ERROR
+        else:
+            data = result.data.as_raw_data()[0]
+            dns_data_b64 = base64.b64encode(data)
+            if dns_data_b64 == input_key.key:
+                return Validity.VALID
+            else:
+                return Validity.REVOKED
+
+    @staticmethod
+    def verify(input_key):
+        # type: (KeyInfo) -> Validity
+        # Public API method, first try the cache and in case the record
+        # is missing, query the DNS system and store the result.
+        key_union = DNSSECKeyVerification.__cache.get(input_key.email)
+        if key_union is not None:
+            return DNSSECKeyVerification.__cache_hit(key_union, input_key.key)
+        else:
+            result = DNSSECKeyVerification.__cache_miss(input_key)
+            if result == Validity.VALID:
+                DNSSECKeyVerification.__cache[input_key.email] = input_key.key
+            elif result == Validity.PROVEN_NONEXISTENCE:
+                DNSSECKeyVerification.__cache[input_key.email] = NoKey()
+            return result
+
+
+def nice_user_msg(ki, v):
+    # type: (KeyInfo, Validity) -> str
+    # Print nice message about key validity
+    prefix = "DNSSEC extension: Key for user " + ki.email + " "
+    if v == Validity.VALID:
+        return prefix + "is valid."
+    else:
+        return prefix + "has unknown status."
+
+
+def any_msg(m):
+    # type: (str) -> str
+    # Prefix any message
+    return "DNSSEC extension: " + m
+
+
+# Class that encapsulates keys already imported into the RPM database
+class RpmImportedKeys:
+    def __init__(self):
+        # Load packages with keys from RPM database
+        self.pkg_names = RpmImportedKeys.__load_package_list()
+        # Convert them into KeyInfo type
+        self.keys = RpmImportedKeys.__pkgs_list_into_keys(self.pkg_names)
+
+    @staticmethod
+    def __load_package_list():
+        # type: () -> List[str]
+        # Since there is no API for this purpose, just call the 'rpm' executable
+        # and parse its output.
+        p1 = subprocess.Popen(["rpm", "-q", "gpg-pubkey"], stdout=subprocess.PIPE)
+        out = p1.communicate()[0]
+        keys = out.decode().split('\n')
+        return [x for x in keys if x.startswith('gpg-pubkey')]
+
+    @staticmethod
+    def __pkg_name_into_key(pkg):
+        # type: (str) -> KeyInfo
+        # Load output of the rpm -qi call
+        p1 = subprocess.Popen(["rpm", "-qi", pkg], stdout=subprocess.PIPE)
+        info = p1.communicate()[0].decode().split('\n')
+        # Parse packager email
+        packager = [x for x in info if x.startswith('Packager')][0]
+        email = re.search('<(.*@.*)>', packager).group(1)
+        # Parse gpg key
+        pgp_start = [n for n, l in enumerate(info)
+                     if l.startswith('-----BEGIN PGP PUBLIC KEY BLOCK-----')][0]
+        pgp_stop = [n for n, l in enumerate(info)
+                    if l.startswith('-----END PGP PUBLIC KEY BLOCK-----')][0]
+        pgp_key_lines = list(info[pgp_start + 2:pgp_stop - 1])
+        pgp_key_str = ''.join(pgp_key_lines)
+        # Compose KeyInfo structure and return it
+        return KeyInfo(email, pgp_key_str.encode('ascii'))
+
+    @staticmethod
+    def __pkgs_list_into_keys(packages):
+        # type: (List[str]) -> List[KeyInfo]
+        return [RpmImportedKeys.__pkg_name_into_key(x) for x in packages]
+
+    @staticmethod
+    def check_imported_keys_validity(logger):
+        # For each key in the list run the verification process
+        keys = RpmImportedKeys()
+        logger.info(any_msg("Testing already imported keys for their validity."))
+        for key in keys.keys:
+            result = DNSSECKeyVerification.verify(key)
+            logger.info(any_msg("Key associated with identity " + key.email +
+                        " was tested with result: " + str(result)))
diff --git a/dnf/dnssec/libunbound.conf b/dnf/dnssec/libunbound.conf
new file mode 100644
index 00000000..dc1b5f6a
--- /dev/null
+++ b/dnf/dnssec/libunbound.conf
@@ -0,0 +1,4 @@
+server:
+   verbosity: 0
+   qname-minimisation: yes
+
--
2.14.3
```

A.2 Extended version

```
From ee6b5d04fe492f3351dcd94cd3a45a19b0fde5fd Mon Sep 17 00:00:00 2001
From: Martin Sehnoutka <msehnout@redhat.com>
Date: Thu, 17 May 2018 13:00:44 +0200
Subject: [PATCH] extended version

---
 dnf/dnssec/dnssecmdverification.py | 77 ++++++++++++++++++++++++++++++++++++++
 dnf/repo.py                        |  9 +++++
 2 files changed, 86 insertions(+)
 create mode 100644 dnf/dnssec/dnssecmdverification.py

diff --git a/dnf/dnssec/dnssecmdverification.py b/dnf/dnssec/dnssecmdverification.py
new file mode 100644
index 0000000..5e620b8
--- /dev/null
+++ b/dnf/dnssec/dnssecmdverification.py
```

```
@@ -0,0 +1,77 @@
+from typing import Union, Dict, List
+from enum import Enum
+import unbound
+import hashlib
+import base64
+
+
+def __load_from_dns(repo_url: str) -> Dict[str, str]:
+    """
+    Private function to load key, value pairs from DNS.
+    """
+    repo_url = repo_url if repo_url is not None else "repomd.example.com"
+    # List of expected keys
+    KEYS = ['alg', 'hash', 'ts', 'val']
+    # Create unbound context => validating stub resolver
+    ctx = unbound.ub_ctx()
+    ctx.config("/etc/dnf/libunbound.conf")
+    # Resolve given domain and obtain TXT RRSet
+    status, result = ctx.resolve(repo_url, unbound.RR_TYPE_TXT, unbound.RR_CLASS_IN)
+    if status != 0:
+        print("error communicating with DNS server")
+    else:
+        data = result.data.as_raw_data()
+        structured_result = {}
+        # Iterate over each RR in the set
+        for d in data:
+            # Just encoding
+            key_val = d.decode('ascii')
+            # Parse k,v pair
+            key, val = key_val.split('=')
+            # Insert into the resulting dictionary if
+            # the key is known
+            for k in KEYS:
+                if key.endswith(k):
+                    structured_result[k] = val
+
+        print(structured_result)
+        return structured_result
+
+
+# Possible outputs of the verification process
+class MdVerificationResult(Enum):
+    VALID=0,
+    INVALID=1,
+    ERROR=2
+
+
+def __hash_local_file(md_file_name: str) -> str:
+    """
+    Function implementing the hashing functionality needed
+    for verification purposes.
+    """
+    with open(md_file_name, 'rb') as f:
+        m = hashlib.sha256()
+        while True:
+            chunk = f.read(2048)
+            if chunk:
+                m.update(chunk)
+            else:
+                break
+
+        digest = base64.b16encode(m.digest()).decode('utf-8').lower()
+        return digest
+
+
+def verify_md(md_file_name: str, repo_url: str = None) -> MdVerificationResult:
+    """
+    This is the only function, that is part of the public API (in Python private
+    functions are prefixed with underscores).
+    """
+    dns_dict = __load_from_dns(repo_url)
+    hash = __hash_local_file(md_file_name)
+    if hash == dns_dict['hash']:
+        return MdVerificationResult.VALID
+    else:
+        return MdVerificationResult.ERROR
+
diff --git a/dnf/repo.py b/dnf/repo.py
index 75a11a4..574b512 100644
--- a/dnf/repo.py
+++ b/dnf/repo.py
@@ -925,6 +925,15 @@ class Repo(dnf.conf.RepoConf):
             msg = _("Failed to synchronize cache for repo '%s'") % (self.id)
             raise dnf.exceptions.RepoError(msg)
         self._expired = False
+        # DNSSEC: self._repomd_fn contains path to the local repomd.xml file.
+        # Load necessary libraries.
+        import dnf.dnssec.dnssecmdverification as dnsmd
+        import dnf.dnssec.dnsseckeyverification as dnssec
+        # Run metadata verification, turn it into a message and inform the user
+        # about the result.
+        logger.info(dnssec.any_msg("Repository metadata considered: " + \
+                    str(dnsmd.verify_md(self._repomd_fn))))
+
         return True

     def _md_expire_cache(self):
--
2.14.3
```

B CONTENT OF THE ATTACHED CD

The CD contains complete source code for the dnf package manager and the libraries that were developed as part of this thesis as opposed to attachement A.1 and A.2 that contains only patches (difference between the original and modified version, not the complete code). It also contains configuration for the testing environment. The configuration is available in form of a Makefile that can be used against a fresh installation of Fedora 27 Server image. It is, however, necessary to change the IP address in settings.lua and hosts files to reflect the new address.

- `/` ..... CD root directory
  - `ansible.cfg` ..... Ansible configuration file
  - `configuration` ..... Configuration for all services (DNS, HTTP)
    - `com-server`
    - `example-com-server`
    - `lighttpd`
    - `local-repo`
    - `notsigned-com-server`
    - `README.md`
    - `resolver`
    - `root-server`
    - `unbound`
    - `wrongconfig-com-server`
  - `demo.flv` ..... Video demonstration of the virtual environment and dnf extension
  - `gen-lsyncd-conf.py` ..... Script to generate lsyncd configuration
  - `hosts` ..... File containing IP address of the server
  - `keyring` ..... Directory with GPG keyring
  - `Makefile` ..... Configuration file for the **make** command
  - `packages` ..... Source code for testing RPM packages
    - `rpms` ..... Binary RPMs
    - `test-good-sig` ..... Sources for the first package
    - `test-revoked-key` ..... Sources for the second package
  - `playbook.yml` ..... The main Ansible Playbook
  - `README.md`
  - `roles` ..... Directory containing Ansible roles
    - `dns`
    - `fedora`
    - `repository`
  - `run-sync.sh` ..... Bash script to run lsyncd
  - `settings.lua` ..... lsyncd configuration
  - `src` ..... Source codes
    - `dnf` ..... dnf sources
    - `lib` ..... The libraries written for this thesis
    - `tests` ..... Sources of the performance tests
  - `unit-files` ..... systemd unit files
    - `com-server.service`
    - `example-com-server.service`
    - `notsigned-com-server.service`
    - `resolver.service`

    - `root-server.service`
    - `wrongconfig-com-server.service`
- `Vagrantfile` .... Vagrant configuration file