University of Amsterdam

*Master System and Network Engineering*

# Automatic SSH public key fingerprint retrieval and publication in DNSSEC

Research Project (1) report

Marc Buijsman, Pascal Cuylaerts
{marc.buijsman, pascal.cuylaerts}@os3.nl

6 February 2011

# Contents

# 1 Introduction

The concept of trust is fundamental in computer network security. Although not everyone is aware of this, encrypted network connections are not much safer than unencrypted ones if the person that initiates such a connection does not explicitly trust that he or she is talking to the intended endpoint.

Take for example an HTTPS connection to a bank's website. If the browser shows that the SSL certificate used for the authentication of the bank's website is valid and that the channel is encrypted (e.g. by showing a lock icon or a green address bar), then one may trust that it really is that bank where he or she is sending sensitive data to (and that no one else can read it).

However, many people do not think about what their trust is actually based on. They implicitly trust the browser's maintainers who ship the browser with a list of what they think are trustworthy certificate authorities (CAs). These are the organisations (third parties) which ultimately need to be trusted since they are the ones signing the certificates, thereby claiming that the website is truly in hands of the associated organisation (the bank in this example).

Unfortunately, someone might not notice that a certificate has been signed by a CA which he or she does not actually trust, but which has been included by the browser's maintainers. If one cannot trust a "secure" connection to another machine, then it cannot be ruled out that an eavesdropper sits in between. Moreover, such a man-in-the-middle could maliciously impair the data flow.

The Secure Shell (SSH) protocol is, like SSL (or TLS), a way to have a secure (encrypted) connection between two computers. It is a protocol used for remotely accessing a machine's command line (shell) with end-to-end encryption. Unlike with SSL as in the example, with SSH one is likely to be confronted with the trust aspect more often. A machine's shell is usually supposed to be accessible by only a few people, whereas websites are aimed at serving many people. Because of this, it makes little sense to purchase a certificate from a CA to secure SSH connections. That is why an SSH client normally involves the user in the authentication process instead.

When one is connecting to a host using SSH for the first time, a so-called fingerprint derived from the remote host's public key is usually presented [1]. In the OpenSSH client this is a hexadecimal presentation of the MD5 hash of the public key. The user can either accept the fingerprint and continue connecting, or refuse it to abort the connection. This step is important. If the user believes that the public key belongs to the private key that is held by the intended host, then it is safe to continue. In the initialisation process the remote host must cryptographically prove that it possesses the private key to authenticate itself.

If the user does not trust the fingerprint, then it would be unwise to accept it. It could be the fingerprint of an eavesdropper for instance. To be able to verify the fingerprint, the user must have had contact with the host's administrator to retrieve it safely. If the user trusts the way the fingerprint has been retrieved, then this person can also trust that he or she is trying to connect to the intended host machine if the presented fingerprint matches the one retrieved out-of-band. The chance that there still is an eavesdropper in the middle is very small since it is hard to generate a public and private key pair with exactly the same fingerprint.

It would however be convenient for a person who is initiating an SSH connection to have a way of verifying the authenticity of a received SSH public key without his or her intervention. Possible human error when comparing fingerprints would also be eliminated. Such a means would require the person (if he or she cares about security) to trust an automated verification process, such that when the key is positively verified he or she can implicitly trust the key to be valid.

If this trust is based on a locally stored list of public keys or fingerprints that was composed by the person him- or herself then a simple automated lookup in this list would suffice. However, this solution is not very scalable. Every person has to compose his or her own list, and keys of previously unknown hosts still have to be verified manually.

The use of the Domain Name System (DNS) offers a better solution, as this is a single database that can be accessed by everyone. An administrator can publish a public key fingerprint in the DNS so that it is instantly publicly available, making it an easy way of distributing fingerprints.

The response to a DNS lookup request can be trusted if DNSSEC (DNS Security [2] [3] [4]) is used. If the retrieved resource record has been signed by an instance that is part of a DNSSEC chain of trust which is ultimately signed by a trusted instance (most commonly the DNS root), then the authenticity of the record can be verified. This would mean that a DNSSEC-validated SSH fingerprint resource record (SSHFP RR [5]) that is tied to a domain name can be trusted to be authenticated by the instance that has the authority over that domain name.

We earlier mentioned that a person would need to contact a host's administrator to retrieve the machine's fingerprint. This could however pose a problem if this person is an organisation's administrator him- or herself. If he or she administers only one machine, then it is not a big deal to walk to the machine, access it directly to retrieve its fingerprint and carrying it back to a workstation. This is the safest way to transport the fingerprint. But if there are many machines of which the fingerprints are yet unknown, then this becomes a cumbersome task.

For someone in such a situation it would be convenient to automate this task. A workstation can be used to collect the fingerprints, which could also push them to the DNS so that other workstations can easily retrieve them as well. When automating this whole process it is inevitable that the a potentially untrusted computer network will be used for the fingerprint retrieval. During our project we investigated a way of retrieving the fingerprints of remote machines securely over an insecure network in the situation where public keys are yet untrusted as a means of host authentication. Such a mechanism of validating a host's fingerprint opens the way for automated fingerprint retrieval and publication in DNSSEC.

## 1.1 Research question

Most of our research was focused on the problem of the insecure connection between an administrator's workstation and a remote machine whose SSH public key is unknown. We have investigated if this channel can be secured, and if so how this can be implemented in a software tool. We have also tried to enable this tool to automatically publish fingerprints in the DNS. This is the practical side of our project; to enable the tool to automatically collect fingerprints in a secure way, the research is a prerequisite for its implementation.

Our research question is:

*How can SSH public key fingerprints be automatically collected from remote machines and published in DNSSEC in a secure way?*

This can be further divided into the following subquestions:

- What are the possible solutions for secure data transfer over an untrusted network?
- Can we make use of existing methods or protocols to realise the possible solutions?
- How can these solutions be implemented in a tool that automates the collection of SSH public keys?
- How can we insert the SSH public key fingerprints into the DNS and sign them using DNSSEC in an automated way?

# 2 Research

In the introduction we explained how DNSSEC can be used to verify the validity of SSH fingerprints and therefore the validity of public keys. If a trust anchor was reached during the DNSSEC-validation of a resource record, then it can be trusted that this record has been authenticated by the instance that has the authority over the concerning domain name. Ultimately, this instance itself needs be trusted as well. If one does not trust that the instance took great care of publishing the correct SSH fingerprint in the DNS, then doing DNSSEC validation makes little to no sense.

A DNS SSHFP record contains a SHA-1 hash (called "fingerprint") of either an RSA or DSA public key [5]; both types can be used in the SSH authentication protocol. The hash is preceded by a number denoting the type of key used (1 for RSA and 2 for DSA) and a number denoting the used hashing algorithm (1 for SHA-1). An example is as follows:

```
domain.com IN SSHFP 2 1 d066788e581f8d91faf1e715954fca596324e851
```

## 2.1 The desired mechanism

We will be describing a mechanism for automatic public key retrieval from remote machines and fingerprint publication in the DNS. We focus for a large part on the situation where the public keys of the remote machines are not certain to belong to those machines. If one uses such a mechanism and he or she wants to be sure that the correct public key fingerprints are published, then there must be a way to verify that a received public key really belongs to the intended machine. After all, there could be an attacker in the middle with whom the actual SSH connection has been set up.

Since in such a situation one cannot be sure whether or not a public key belongs to a certain machine, it cannot be used for the authentication of the machine's identity, even if the machine can prove that it possesses the corresponding private key. It is our goal to collect the public key in such a trustworthy way that it eventually can be used for this purpose. This is necessary for SSH connections where public key cryptography plays a central role in server authentication.

Therefore, some secure mechanism is needed to establish the *authenticity* and the *integrity* of a collected public key. That is, we want to make sure that a public key belongs to the machine with a certain identity, and we want to ensure that its integrity has been preserved during transfer to prevent a possible publication of a wrong or malicious fingerprint into the DNS.

The most secure way to collect public keys would be transporting them out-of-band from each machine separately. This would require a person to physically access these machines one by one to extract the public key. If there are many machines with many administrators, then this task can be simplified by asking each administrator to send their machine's public key in a GPG [6] [7] signed email, for example. However, the senders' GPG public keys first need to be trusted as well. If many machines are under control of a single administrator, this solution may not be workable because he or she still needs to physically access a relatively large number of machines.

In the last case, it would be very convenient to be able to automate the key retrieval process by a computer program without further human intervention needed. This will however need to be done over a potentially insecure network, because there is no other way a computer program can contact a remote machine. What we have here is a classic chicken-and-egg problem. We need to authenticate a machine for which we need its public key and we want the machine to proof that this really is its public key, but then we already need to have authenticated the machine. The machine therefore needs something else than a public and private key pair to be able to identify itself.

## 2.2 Shared secrets

In general, a person that needs to authenticate him- or herself, will need to know something (e.g. a passphrase), have something (e.g. a smartcard), be something (e.g. his fingerprint), do something (e.g. a signature) or a combination of these. The authority that is authenticating this person needs to be able to verify the provided information. In computer security, if two machines need to authenticate one another, they will often know each others public key and use challenge-response authentication combined with public key encryption. An alternative is to have both machines to know some shared secret such that each computer can prove somehow that it knows what the secret is, without revealing it to the outside world.

A shared secret can be seen as a passphrase. Just like passphrases, such a secret needs to stay secret between two parties to prevent a third party from misusing it. Unlike with public and private key pairs, both parties need to protect the secret since they both need to know it to be able to authenticate each other. If only one of the parties needs to authenticate itself to the other using a public and private key pair, then this party needs to protect the private key whereas the other party does not need to protect anything. It does need to know the public key, but since this key is publicly available is does not have to be protected from outsiders.

It could be easy to use a shared secret as a means of authentication in some cases though. A machine specific system identifier can be looked up by the machine itself or by someone having elevated privileges on the machine. A system's Universally Unique Identifier (UUID) for example is a good candidate for a machine identifier (as will be explained later), which is usually only readable by users who have root privileges. We decided to make use of a shared secret since some system identifiers might be listed on hardware inventory lists that are available within an organisation.

Having these numbers on paper already makes a walk to every machine within the organisation, to retrieve the identifiers manually, unnecessary. They could be entered in a computer file straight away. Once this has been done, a program can use this file to perform the automatic public key retrieval process. The only assumption we made is that the identifiers have not been copied by an untrusted party during the identifiers' retrieval process and that the inventory lists are stored safely, something that is important when using them as secrets but which we have not further investigated.

## 2.3 Authentication without shared information

At first, we tried to come up with a protocol that does not need any pre-shared data for the machines to be able to authenticate one another. In this case, there is no shared information that can be used for host authentication. Most of the possible solutions for this problem we have read about consisted of identity-based key agreement schemes that require a trusted third party to act as a key generation center (KGC [8]) that creates key pairs. Apart from the need for a trusted third party, these schemes where too complex for our application.

Methods to detect a man in the middle can also be used, such as the leap-of-faith method [9]. If there is someone in between during the first connection, then he must be in between during all the subsequent connections to prevent the administrator from being warned that the public key has changed. This could be hard to do for the attacker and therefore a second connection can be set up after a certain timespan to see if there will indeed be a warning. If so, then the administrator will know that there was someone in between either during the first connection or the second, making the received data during either connection untrusted.

An administrator could also make assumptions about the network between him or her and the remote host to determine if it will be safe enough to proceed without having the ability to authenticate the received data. Such an assumption can for example be that only the local area network (LAN) will be used which may be considered clear from intruders. Also, since our mechanism needs to be used only once to retrieve public keys, the risk of an attacker being present during the retrieval process is reduced to only one connection for each host. This could be considered an acceptable risk.

However, since there is no information available to authenticate a remote host in these situations, data exchange can never be completely secure. We need information that can be used to authenticate a host to be able to set up a secure session with the host, so ensure that no malicious fingerprints will be published in the DNS. For our mechanism this information will be a pre-shared secret.

# 3 Mechanism design

## 3.1 The key retrieval mechanism

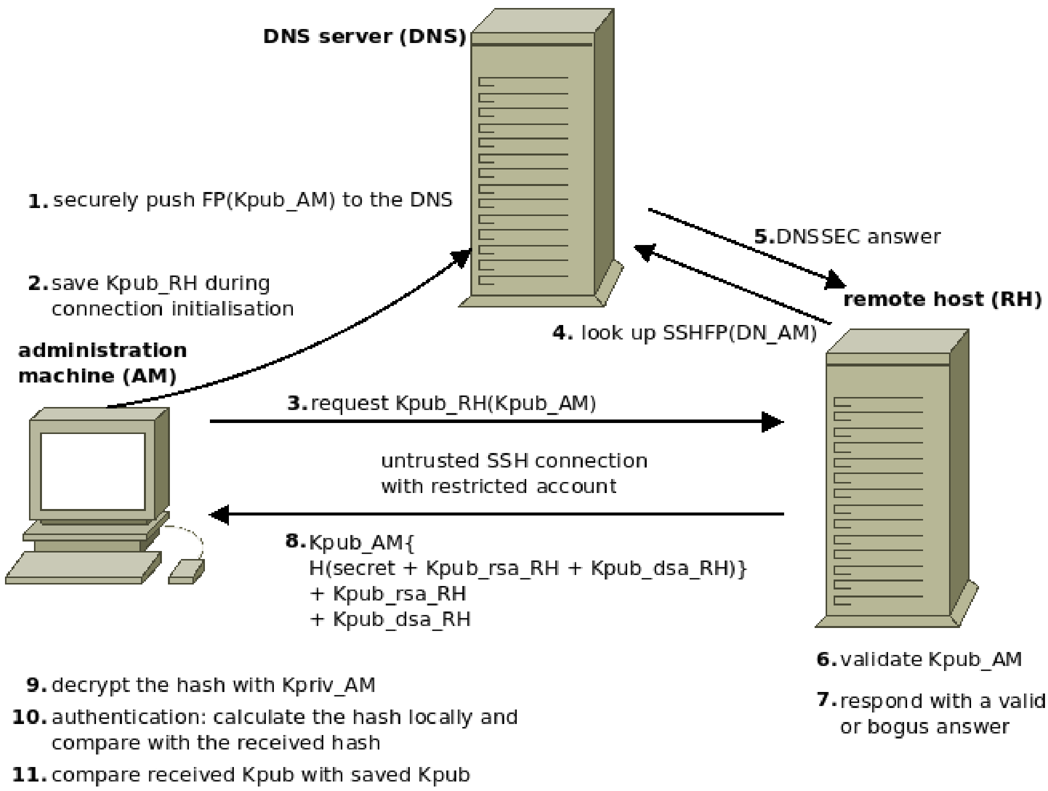


Figure 1: Key retrieval mechanism.

The mechanism we devised to securely retrieve remote hosts' fingerprints and publish them in the DNS (signed using DNSSEC) is illustrated in figure 1. This mechanism assumes that an administrator wants to collect the SSH public keys from a number of remote hosts (RHs) using one administration machine (AM).

To authenticate the responses that the AM will receive from the RHs, a list of shared secrets needs to be available on the AM with an entry for each RH. Because this shared secret is the only means for a remote host to authenticate its identity, this data needs proper protection and must at least be encrypted when stored on disk. Another requirement is that the fingerprint from the AM's SSH public key (`FP(Kpub_AM)`) is stored in the secure domain name system (DNSSEC) (1).

The AM will contact a RH to retrieve its SSH public key (`Kpub_RH`) using SSH. This connection is untrusted and the account used to log in on the RH must have restricted permissions (since the credentials can be read by an eavesdropper). When the connection is being established, the AM will receive `Kpub_AM` and store it temporarily to use at the end of the process (2).

Once the connection has been established, the AM will send a request to the RH to ask for its public key and in this request the AH will include `Kpub_AM` (3). When the RH receives this request, it will look up the SSHFP records in the DNSSEC using the domain name of the AM (4) which needs to be pre-configured on the RH. The SSHFP records (with the associated RRSIGs) in the answer (5) will be validated locally and compared to the fingerprint derived from `Kpub_AM` (6). If the two fingerprints match, the RH will send a response to the AM which includes its secret and SSH public key. If the fingerprints did not match, the RH will respond with a bogus answer (7).

A valid response (8) is built up as follows:

```
Kpub_AM{H(secret + Kpub_rsa_RH + Kpub_dsa_RH)} + Kpub_rsa_RH + Kpub_dsa_RH
```

The secret is concatenated with the RH's RSA and (if present) DSA public keys and this string is hashed. The resulting hash will be encrypted with `Kpub_AM` and then concatenated with the cleartext RSA and DSA public keys of the RH.

Upon retrieval of this response, the AM will decrypt the hash with its private key (`Kpriv_AM`) (9) and calculate its own hash (10) with the received public keys and the secret it has stored locally. If the hashes match the AM can be sure that the response came from the RH he intended to contact and that the response has not been modified on the way back. The hash is therefore used to check the integrity of the public keys that were sent along. Since the secret is incorporated, the keys' authenticity can also be verified.

As an extra security check, the AM can now compare the public key he stored at the beginning of the process with the one he just received. If they do not match, the machine he was communicating with must have been an attacker that was performing a man-in-the-middle attack and who forwarded the request to the actual RH to let it respond with a valid answer. However, the keys' fingerprints can still be published in the DNS if the hashes match since that proves that the answer was not tampered with by the man in the middle.

## 3.2 The key retrieval mechanism under attack

### 3.2.1 Attacker forwards messages

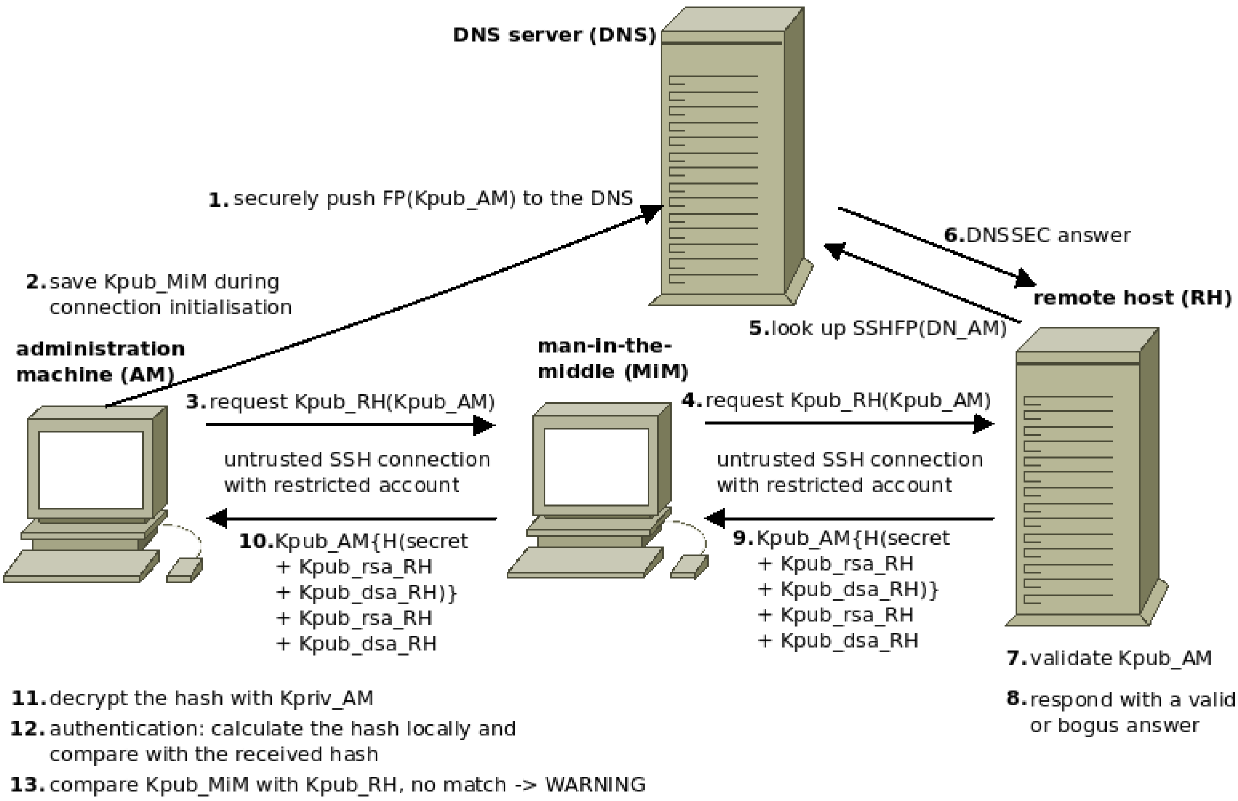


Figure 2: Key retrieval mechanism under MITM attack.

If this mechanism is under a man-in-the middle attack, as illustrated in figure 2, the public key stored at the start (2) will be the one from the man in the middle (MiM). The MiM will just forward the request from the AM (3) to the RH (4) which will validate `Kpub_AM` using DNSSEC (5, 6, 7) and think it is really talking to the AM. As a result it will respond with a valid answer (8) but the hash of the secret concatenated with the public keys will be encrypted with the public key of the AM (9). This makes the intercepted response unreadable for the MiM because he does not know the AM's private key.

After the MiM forwarded the response to the AM (10), the AM will decrypt the hash (11) and calculate the hash itself with the received public keys and the secret it has stored locally (12). If the MiM has not tampered with the public keys and the hashes still match, the AM still does not have a clue that a third party was in the middle, which accepted the SSH connection and saw the login credentials passing by. But because the public key from the host he connected to was stored when the connection was set up (2), he can now check whether it matches the public key from the response. If not, he knows something suspicious happened.

### 3.2.2 Attacker modifies messages

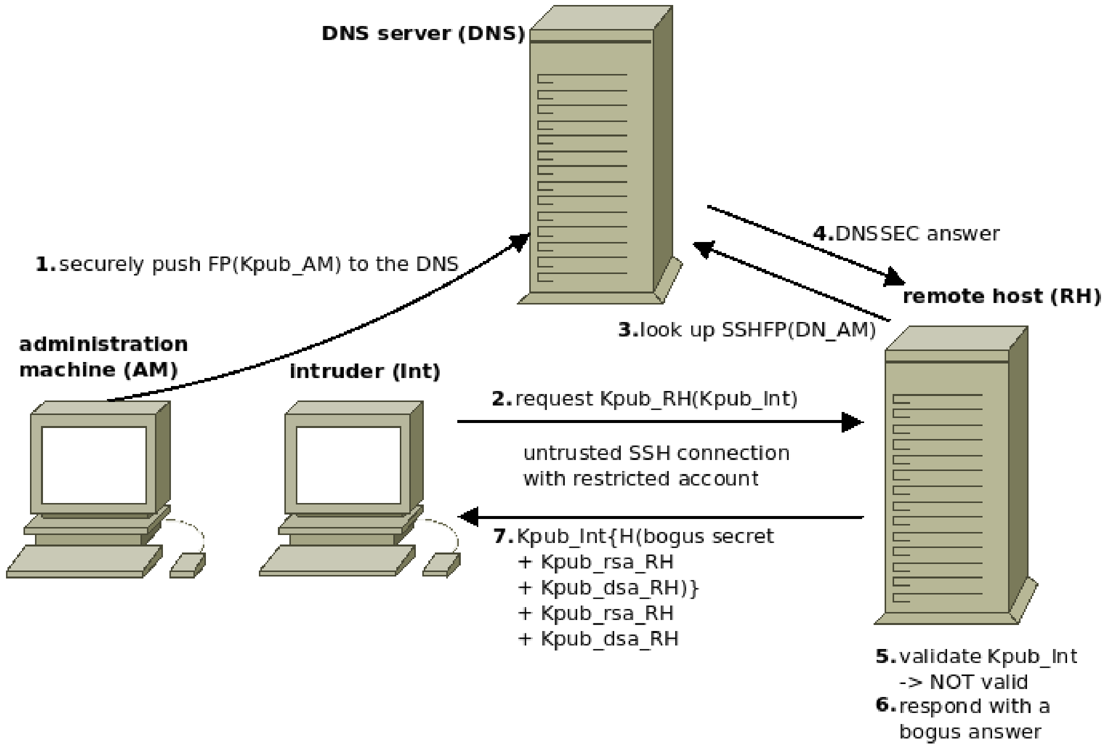


Figure 3: Key retrieval mechanism under attack directly.

When an intruder (Int) manages to log in directly into the RH and requests the secret (2) without forwarding the AM's request as illustrated in figure 3, the RH will notice that the Int is not the AM because the fingerprint of its public key does not match the one he looked up in the DNS (3, 4, 5). As a result, the RH will return a bogus answer (6, 7) encrypted with the Int's public key. This answer will contain a hash from a random string concatenated with the RH's public keys. The Int will be able to decrypt the hash and may assume he received a valid answer. He can now perform offline attacks in an attempt to recover the secret but he will only end up with a random string.

Not sending a response when the fingerprint of the intruder's public key (`Kpub_Int`) does not match the fingerprint found in the DNS would simplify the attacker's job since he would have less hashes to perform attacks on. It also prevents the AM to notice that something is going on, which would be a good thing to know such that the situation can be further investigated.

# 4 Implementation

We implemented this mechanism as a proof of concept with two programs written in Python (listing 1 in A.1.1 and listing 5 in A.2.1) for the Linux OS that will handle the communication between both parties. The programs need to be configured using a configuration file (listing 2 in A.1.2 and listing 6 in A.2.2).

The program that will be executed on the administration machine has two modes of operation. In the normal mode, the program will retrieve the public keys from the remote hosts and push their fingerprints to the DNS in the form of SSHFP records. The second mode takes a list of SSHFP records as input and pushes them directly to the DNS.

## 4.1 Secrets file

In normal mode, the program needs to have access to a file with one line of information for every host that needs to be contacted. This line will have the following format:

```
host.domain.org:4445434C-5700-1050-8034-B7C04F56344A:..CN7084106E00YU.Product Name
```

The first part is the domain name, then a strong secret followed by a weaker secret to which the program can fall back if the strong secret is not available. These are all separated by colons which we believe are acceptable separators for the types of secrets we had in mind.

We chose the system's Universally Unique Identifier (UUID) as the strong secret because of its selection from a large key space, making it hard to guess, and because it is the best information to uniquely identify virtual machines (VMs) and thus to authenticate them. The UUID is usually also listed in the configuration file of a VM which can easily be processed in an automated way to collect the UUIDs from all guest VMs if one has access to the host machine.

The weak secret is a concatenation of the serial number of the system's motherboard and its product name to enlarge the key space. Assuming that a detailed inventory is kept of all hardware used in an organisation's network with this kind of information, it should be easy to generate a list of physical hosts with their secrets. If the UUID of the machine is also listed in the inventory, then that is an advantage, because of the larger key space.

Note that the weak secret is more vulnerable to dictionary attacks. Building up a dictionary of known product names would be easy and a part of the motherboard's serial number also refers to the manufacturer, reducing the possible combinations. Any information that is already available to the administrator can be used to authenticate the remote hosts, and for our proof of concept we considered the serial number and product name identifiers secure enough.

## 4.2 Secret look up at the remote host

The remote host can find out its own secret from the output of the `dmidecode` command. This program will parse the contents of the system management BIOS (SMBIOS) table and present them in a human-readable format. The SMBIOS contains a description of the system's hardware components and other useful information such as serial numbers and details about the BIOS. Dmidecode will access the file `/dev/mem` to access this data. A user that wants access to this file will need elevated permissions.

The values read from the SMBIOS table are not always reliable, because manufacturers can leave values empty or can choose to fill in different kinds of information. The SMBIOS standard [10] is specified by the Distributed Management Task Force (DMTF) and not all the fields of the SMBIOS table are required to be filled in to comply to the standard. The UUID and the Product Name are required fields, but the motherboard's serial number is not required. Although it may be empty according to the standard, we still chose to use the motherboard's serial as part of the secret, because it is a good identifier and most manufacturers seem to fill it in correctly.

Because the information in the secrets file is so critical for the authentication of a host, it should only be stored on disk at the administrator's side with proper encryption. Therefore our program will accept an AES encrypted file and prompt the administrator for the passphrase it needs to decrypt the file.

```
Please provide your credentials for the remote hosts.
Username:
Password:

Please provide the passphrase to decrypt the secrets file.
Passphrase:
```

On the remote hosts a restricted user account must be configured. This account will be used to set up the untrusted SSH connection over which the authenticated key retrieval will take place. Our program will prompt the administrator for these credentials at start up so he will not have to enter them in a configuration file in cleartext.

In order for the program that will be executed on the remote host to be able to read `/dev/mem`, the restricted user account needs to be able to run our program with elevated permissions. Therefore we added a line in the sudoers file `/etc/sudoers` and use sudo when executing the program.

```
untrusted ALL = (root) NOPASSWD: /path/to/program
```

This line means that the user `untrusted` can execute from `ALL` terminals, acting as `root` the program `/path/to/program` without being prompted for a password.

## 4.3 SSH connection

When the credentials have been entered and the secrets file could be decrypted (which is done using `gpg`), the program at the administrator's side will go through the secrets file line by line, creating one SSH session after the other with the host at each of the domain names. To be able to set up an SSH connection it uses a Python module that interfaces with the `libssh2` C library. We created this module ourselves (which we called `sshexec`, see listing 4 in A.1.4), with the basic functionality needed for an SSH session. It has been implemented using the Python C API [11] so that it could be included in the program.

In its current implementation only one SSH connection can exist at a time. It was largely based on an example source file that came with `libssh2`. When a connection has been initiated using the module, it returns the remote host's public key. This key will later be used to check if there was someone eavesdropping on the connection.

Once connected the program will ask the remote host to authenticate its RSA and (if present) DSA public keys using the type of shared secret. If both a strong and a weak secret are listed, then the strong secret will be used. It will do this by executing a command on the remote shell which will initiate the program at the remote machine's side. Once an answer has been received or when the execution timed out, the connection will be closed and the program will continue with the next line in the secrets file after it validated the public keys with the hash if that was sent back.

An answer consists of the encrypted hash concatenated with the RSA and DSA public keys in Base64 encoding, separated by colons (which do not occur in the Base64 encoding scheme). This string is preceded by the response type, which can be "ANSWER", "ERROR" or "WARNING". Only with "ANSWER" a hash will be sent, the others will accompany a human readable message for alert and debug purposes.

An example of an "ANSWER" response is as follows:


```
ANSWER:saDp4JhJNNDttXgu9UidZEZDdq6VInS2Pyt1innR2SZLfBaFuZazzNnsOvW2S9DkV/yng0Aee
t2dLuj1vJH3dVlbAPE4qQWj4uBdCJQE4oSU3A5PjYnedZZYXpCjYQxzFDrKD166yqRUQdtFmpRbgI/bf
i+rEcn1YSUl5pdVjuzQK/B3moYPuScCtj/7o9rn/Yn3auUCC3NzrlmPPibFi94ryLBcAQc3dOYW2N9S2
+OFy1CZfdyRZIemr8g8P+W+gFeTKZEeSiG3GwZxeNuWxmLgkBsu+P4dViHR419dPayfeBTxcVDlT7PLX
e4/t3Q5GnzM4lzT6p478l4TTmBg+w==:AAAAB3NzaC1yc2EAAAABIwAAAQEA+EVTkCxclj1gI2J3HrH2
gkQFgg4dZXBwq6aV4933OVGP6RRcn78RwkF+3zr1jnYhBCelUmePQhmlsZH4ivXWY33XX27JX5ZZjsQ0
wPXXcS8lwCb2pOY4R2+pNKtpOuOM3YWSVXyLCaNIaBWBay+QPFnwyswcJ4o3AUhKuWz1hKUKpHGv1OIs
2nIkyjY2Z1IcLbKlFEswurlWf4lZRqqRkanS7T3UraxtrSC+Hz4aEuB9/WGJ4t/NReXpYBD1m78CgrfX
bjE5LAMWGyR+Rri97KUB2vH/XN/aI7VVOu9ik7gH3PrlaeTsNOUMgSC45TQwiygaGIOuNUZPyx3ISX5K
gQ==:AAAAB3NzaC1kc3MAAACBAImVL4qXUVVOzlqYg/0aGvfXqEW3CIuJ3DcO+ENo9ueKNu9p/RJ8+eZ
bN5vD8bEOwVWvg7/dirheKmMNVMUpDox99b7VaJaUUfY8gZT8OomN7NvBSQ64hWXuHA/xMbGdg6r6YDN
AmOPSnnLR9OkWhLOWKIHn9INU68VtmcC8siGXAAAAFQDE6PYVTjb5XKtn1Uvs/jzYx+TenQAAAIBMBwb
2/A0E6/q/EZzWTp94oGNDDJlVEWd6X7kdgsYjAXM0fk/eH2ri82+7X3JpeGS6LELaBqIhs3hG2HZp9wj
6bp5gLqjc1dWH8IKQpc0xJA/SDGDaH+xKklsolpxqIad/wivMAFo3I/+ch1777K/EKXN4uIzEETMUPLO
mq++nrAAAAIAjOUS3QZGpcpdWMFX8eVDnsrcTvEcRJfgdUJx7pnr0sSX+NNNhTEB8J0XggHg5htfItEp
g2sBfp+Kpr9PpL+e1Gl4VTqNs47jJsadnvQZSRUJ5aZaKeX7VpEpyZxd98Cqcn4B0MLKLs5nEHTHyNoq
QkGVIoGB33+b2WLVa8dTpCg==
```


## 4.4 Local DNSSEC validation

The program at the remote host's side makes use of the `LibUnbound` [12] Python module to do local DNSSEC validation of the RSA public key it receives from the administration machine. The program looks up the SSHFP records of the domain name that was locally configured as the domain name of the administration machine. When the fingerprint of the received public key matches the fingerprint in an SSHFP record, and if that record has been validated using DNSSEC, the program will respond with the shared secret. If the fingerprint could not be validated, a bogus answer will be generated.

This step is important in the sense that it prevents an eavesdropper from discovering the host's secret. The generated hash will be encrypted with the received public key and only if this key belongs to the administration machine, that machine can decrypt the hash using its private key. If the key could not be verified as belonging to the administration machine, then it is possible that an eavesdropper is in between.

When the hash is encrypted by the eavesdropper's public key, he will also be able to decrypt it. By sending a bogus answer when the public key's fingerprint does not match, he will receive an invalid hash, making an offline attack on the hash to discover the secret pointless since it has not been involved in creating the hash. If the eavesdropper forwards the answer to the administration machine it can detect that something is wrong since the hash will not match the one it generates itself.

The DNSSEC validation must be done locally so that the whole validation process does not rely on the "last mile" between the DNS server and the host in which the DNSSEC answer could be forged to look valid when it is actually not. It is therefore necessary to have the certificate of a trust anchor installed at the host which in our case was the DNS root's certificate. One might consider to run Unbound as the local DNS resolver so that the root certificate is automatically updated when its key has been rolled over.

## 4.5 Encryption

As mentioned before a remote host uses the RSA public key of the administration machine to encrypt the hash. We included the `M2Crypto` [13] Python module for encryption functionality. A public key object is created from the RSA exponent and modulus that are extracted from the administrator's public key which is passed on to `M2Crypto` along with the hash to perform the encryption.

RSA "Optimal Asymmetric Encryption Padding" (OAEP) is applied just before the encryption to minimise the chance of a successful cryptographic attack [14]. This also causes the ciphertext to be different each time the same hash is being encrypted, making it impossible for an attacker to find out if an answer from the remote host is actually valid by trying to see if the answer stays the same after multiple identical requests (e.g. with a replay attack). Without the padding a valid answer would not change indeed, whereas a bogus answer is randomly generated at each rejected request.

At the administrator's side, `M2Crypto` is used again to decrypt the hash. The machine's private key is passed to the module, which is the reason why the program must run with root privileges since the private key is not world readable.

## 4.6 Pushing updates to the DNS

In case a list of SSHFP records is provided, the application will immediately try to push the new records to the DNS server, skipping the key retrieval process. Otherwise, the public keys are first retrieved from all the remote hosts whereafter SSHFP records are generated for the trusted keys. To perform dynamic DNS updates, we use `nsupdate` which is part of the package `bind9utils`.

Transaction signatures (TSIG) [15] are used to authenticate the updates. These signatures rely on a shared secret between the administration host and the DNS server. The secret key needs to be configured on the DNS server and the path to the local keyfile also needs to be configured in the configuration file of our application. Hash-based Message Authentication Codes (HMAC), HMAC-SHA512 in our implementation, are then used to ensure authenticity and integrity. We also force `nsupdate` to use TCP instead of UDP to ensure a successful update.

## 4.7 Existing list of SSHFP records

As mentioned before (2 Research), public keys can also be retrieved out-of-band or via encrypted email (GPG). We added the functionality to push an existing list of SSHFP records to the DNS, just by feeding the file to our administration application. The administrator just needs to offer the program a file with valid SSHFP records each on a new line. The help section of the application (listing 3 in A.1.3) shows how to use the arguments.

## 4.8 OpenSSH patch

The result of this whole process is of course more useful if one has a client application that actually looks up the SSHFP records in DNS and does local DNSSEC validation of the answers.

On the website `http://www.dnssec-tools.org/` one can find a whole suite of tools that make use of DNSSEC. First the `DNSSEC-Tools` package will need to be installed, which will install the DNSSEC-Tools resolver and validator libraries and headers on the system. Then OpenSSH [16] [17] can be patched with the patch included in the package. More detailed installation instructions can be found in the `README` file of the package, or on the website.

Once OpenSSH has been patched successful, a new option can be used, `StrictDnssecChecking`, in ssh_config. This option can have the values yes, no and ask. One will also need to enable VerifyHostKeyDNS. This option is already available in the normal version of OpenSSH, but the patch is needed to add validation of the DNS answer using the RRSIG resource records.

When one tries to connect to a host whose fingerprint cannot be validated using DNSSEC, the following warning will be shown:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: UNTRUSTED DNS RESOLUTION FOR HOST KEY!       @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

If the key has also changed since the previous connection (according to the known_hosts file), an even stronger warning will be displayed:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: UNTRUSTED DNS RESOLUTION FOR HOST KEY!       @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
ba:7e:98:3c:42:96:54:b6:67:30:7a:3c:df:fd:33:7d.
Please contact your system administrator.
Add correct host key in /home/<user>/.ssh/known_hosts to get rid of this message.
Offending key in /home/<user>/.ssh/known_hosts:<line number>
RSA host key for host.domain.org has changed and you have requested strict checking.
Host key verification failed.
```

When the public key of the remote host can be trusted, a user will immediately be prompted for his or her credentials and will not be bothered with any message, not even the public key's fingerprint.

## 4.9 System requirements

### 4.9.1 Overview

#### Administration machine

- Python application (listing 1 in A.1.1)
- dependencies (argparse, M2Crypto, libssh2, bind9utils)
- Python interface for libssh2 C library (listing 4 in A.1.4)
- configuration file (listing 2 in A.1.2)
- encrypted secrets file
- shared (with DNS) key file

#### Remote host

- Python application (listing 5 in A.2.1)
- dependencies (argparse, M2Crypto, libunbound)
- configuration file (listing 6 in A.2.2)
- restricted user account
- edited sudoers file (see 4.2 Secret look up at the remote host)

**DNS server**

- SSHFP records for administration machine
- edited `named.conf`
- allow for dynamic updates (`nsupdate`)
- shared (with AM) key in `named.conf`

### 4.9.2 Description

The tools we created were meant as a proof of concept only intended to be used under a Linux OS. The two programs have their own dependencies end these can also have dependencies themselves. We have not tested any configurations other than our own, so it is always possible that one will need to have some library that is not listed in the overview above.

**Dependencies**
One will need to have at least the packages `python`, `python-argparse` and `python-M2Crypto` installed on the administration machine (AM) and the remote hosts (RH). The application at the AM needs `libssh2` in order to set up the SSH connections and `bind9utils` to perform the dynamic updates with `nsupdate`. On the RH, an installation of `libunbound` is required to do the DNSSEC local validation. For our application to be able to use the `libssh2` C library, the included Python interface we have developed needs to be present too.

**Configuration**
For both applications a configuration file is used to adjust the program to a specific implementation. On the RH a restricted user account needs to be configured and the sudoers file needs to be modified to allow the user to run our application with root permissions. For secure dynamic updates, a shared key needs to be present on the AM and the DNS server (in `named.conf`). The AM needs to be allowed to perform updates and the fingerprint of its public key needs to be published in the DNS beforehand.

# 5 Conclusion

The SSH protocol provides an encrypted channel with a remote host in order to securely use its shell. To authenticate the remote host it makes use of public key encryption. During the first connection setup with a remote host, the user of an SSH client program is usually asked to verify the host's public key fingerprint. However, this fingerprint may be unknown to the user. Normally, he or she should retrieve the fingerprint from the remote host's administrator out-of-band and check if it matches the one received over the network. If this is not the case, then a man in the middle could be listening on the line and modify the sent data if the user still accepts the fingerprint and proceeds with the connection.

It would be convenient to have a mechanism that can be used to retrieve and verify a yet untrusted public key without human intervention. In our project we have worked towards a solution in order to make that possible. In the introduction of this report we gave the research question of our project, divided into subquestions. The research question was:

*How can SSH public key fingerprints be automatically collected from remote machines and published in DNSSEC in a secure way?*

By answering the subquestions, the research question can be answered.

**What are the possible solutions for secure data transfer over an untrusted network?**

We wanted to have a way to authenticate data sent by certain remote hosts without the use of their public and private key pairs, since these are yet untrusted in the described situation. We also wanted to automate this process such that it would not be necessary to do this manually. If there are a lot of machines for which this needs to be done, then the solution for this problem offers the possibility of authenticating the hosts' public keys easily.

We have investigated what the possible solutions for this problem are without de need to rely on a trusted third party. We can distinguish two types of solutions: one type where the remote host's identity cannot be verified due to the lack of information about that host, and another type where such information is known such that a host's identity can be established.

The first type of solutions can never be completely secure. The administrator (who is initiating the automatic public key retrieval process) has to make some assumptions about the part of the network he or she uses and determine if it is safe enough to proceed without having the ability to authenticate the received data. Such an assumption can for example be that only the local area network (LAN) will be used which may be considered clear from intruders.

There are also methods for detecting man-in-the-middle attacks, such as the leap-of-faith method. If there is someone in between during the first connection, then he must be in between during all the subsequent connections to prevent the administrator from being warned that the public key has changed. This could be hard to do for the attacker and therefore a second connection can be set up after a certain timespan to see if there will indeed be a warning.

For the second type of solutions it is necessary to have certain information such that a host can be authenticated. As such, data sent by the host can be authenticated by the administrator to come from this host unaltered. It must be trusted that the part of the information that needs to be secret has not fallen into the wrong hands, though. This is the case with a public and private key pair, in which the private key has to be kept secret from everyone else. Since these cannot be used for authentication, we decided that a hard to guess pre-shared secret (e.g. the system's UUID) would be the best alternative.

We made use of shared secrets in our mechanism so that public keys could be authenticated, which subsequently could be used for secure data transfer. By creating hashes of the sent data concatenated with the secret, both the integrity of the data and its authenticity can be verified. By letting the remote host verify the administrator's public key using DNSSEC and using this key to encrypt the hashes, it can be prevented that an eavesdropper does not get to see a hash in which the secret has been involved. If the public key could not be verified, a bogus answer can be sent back. Offline attacks to discover the secret will be pointless for the eavesdropper in that case.

**Can we make use of existing methods or protocols to realise the possible solutions?**

We have seen that most possible solutions to the key retrieval process involve trusted third parties. This is not desirable for this simple application. Soon it became clear that a pre-shared secret was the most feasible solution. The SSH protocol itself can be used in the retrieval mechanism. Using this protocol, an eavesdropper can be detected by comparing the public key received when the SSH connection was initiated and the public key received from the remote host later in the process. If the eavesdropper lets this last key unaltered, the two keys that the administrator received will not match. If he replaces the key with his own key, then there will be a match but then the hash cannot be validated. In both scenarios the administrator will be noticed that something is going on.

The DNS can be used to let the administrator's public key be verified by the remote hosts, by validating the key's fingerprint from the DNS with DNSSEC. If this is done locally and the public key is found to be valid, then it can be certain that a hash encrypted with this public key can only be decrypted by the administrator.

**How can these solutions be implemented in a tool that automates the collection of SSH public keys?**

We combined existing programs and libraries to implement the mechanism we came up with in a solution that requires a program on the administration machine to contact each host and execute of second program on this host in order to retrieve the public keys in a secure way. The mechanism makes use of the methods and protocols mentioned above. Our implementation also made it possible to automate this process for a list of hosts, given their domain name and a shared secret.

**How can we insert the SSH public key fingerprints into the DNS and sign them using DNSSEC in an automated way?**

For the BIND installation we used in our proof of concept, the easiest way of pushing dynamic updates to the DNS server was by using the program `nsupdate`. Authentication of the administration machine was enforced by using a pre-shared key and the updates themselves used transaction signatures to ensure authentication and integrity of the SSHFP resource records that needed to be inserted. The `nsupdate` program also makes sure that the new records are signed using DNSSEC, provided that it can find the private key needed for this process.

# References


[1] Ylonen & Lonvick, *The Secure Shell (SSH) Protocol Architecture*, RFC 4251, January 2006, `http://tools.ietf.org/html/rfc4251`.

[2] Arends, et al., *DNS Security Introduction and Requirements*, RFC 4033, March 2005, `http://tools.ietf.org/html/rfc4033`.

[3] Arends, et al., *Resource Records for the DNS Security Extensions*, RFC 4034, March 2005, `http://tools.ietf.org/html/rfc4034`.

[4] Arends, et al., *Protocol Modifications for the DNS Security Extensions*, RFC 4035, March 2005, `http://tools.ietf.org/html/rfc4035`.

[5] Schlyter & Griffin , *Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints*, RFC 4255, January 2006, `http://tools.ietf.org/html/rfc4255`.

[6] Callas, et al., *OpenPGP Message Format*, RFC 4880, November 2007, `http://tools.ietf.org/html/rfc4880`.

[7] GnuPG project homepage, `http://www.gnupg.org/`.

[8] Al-Riyami & Paterson, (2003). *Certificateless Public Key Cryptography*, University of London.

[9] Arkko, Jari, Nikander & Pekka, (2003). *Weak Authentication: How to Authenticate Unknown Principals without Trusted Parties*, Ericsson Research NomadicLab.

[10] System Management BIOS, DMTF standard DSP0134, July 2010, `http://dmtf.org/standards/smbios`.

[11] Python/C API Reference Manual, `http://docs.python.org/c-api/`.

[12] Unbound documentation, `http://www.unbound.net/documentation/libunbound.html`.

[13] Chandler Wiki: Me Too Crypto, `http://chandlerproject.org/bin/view/Projects/MeTooCrypto`.

[14] *What is OAEP?*, `http://www.rsa.com/rsalabs/node.asp?id=2346`.

[15] Vixie, et al., *Secret Key Transaction Authentication for DNS (TSIG)*, RFC 2845, May 2000, `http://tools.ietf.org/html/rfc2845`.

[16] OpenSSH project homepage, `http://www.openssh.com/`.

[17] OpenBSD. *OpenBSD Reference Manual (SSH).* `http://www.openbsd.org/cgi-bin/man.cgi?query=ssh`.

# A Program code and configuration files

## A.1 For the administration machine

### A.1.1 Application

Listing 1: tool_AM.py

```
#!/usr/bin/python

### imports ###
import ConfigParser     # reading config files
import argparse         # parsing parameters
import subprocess       # spawning new processes
import shlex            # determining the correct tokenization for args
import hashlib          # computing hashes
import sys
import os
import base64           # base64 encoding/decoding
import logging          # will handle the logging of messages
import getpass          # password prompt, input is not printed
from M2Crypto import RSA
sys.path.append("lib")
from sshexec import *   # python code to access libssh's C library

### default parameters ###
logger = None
logfile = "tool_AM.log"
username = ""
password = ""
RH_path_program = "tool_RH.py"
clear_secrets = ""
secrets_path = "secrets/secrets_aes.txt"
SSHFP_list = []
SSHFP_ttl = 1800
DN_DNS = "localhost"
DNS_zone = ""
DNS_update_file = "DNS_update.tmp"
Kpub_RH = ""
private_key_DNS_admin = ""
###

### functions ###
def decryptAES_File(secrets_file, passphrase):
  global clear_secrets
  logger.info("decrypting secrets file \"" + secrets_file + "\"...")
  if os.access(secrets_file, os.F_OK): # if the file exists
    command = subprocess.Popen(shlex.split("gpg --quiet --yes --logger-file /dev/null
        --passphrase " + passphrase + " -d " + secrets_file),stdout = subprocess.PIPE)
    clear_secrets = command.communicate()[0] # put the decrypted file in a global
        variable
    if clear_secrets == "":
      logger.info("wrong passphrase...")
      error_quit("the secrets file could not be decrypted...")
    else:
      logger.info("secrets decrypted...")
  else:
    error_quit("the secrets file \"" + secrets_file + "\" can not be accessed..")

def processList_Of_Hosts():
  logger.info("start processing hosts...")
  global clear_secrets
  records = clear_secrets.splitlines()

  for line in records:
    processHost(line)
  logger.info("all hosts processed...")

```

```python
def processHost(record):
  global username
  global password
  global RH_path_program

  host = record.split(":")[0]
  strong_secret = record.split(":")[1]
  weak_secret = record.split(":")[2]
  logger.info("processing host " + host + "...")

  # which secret can be used?
  secret_type = getSecret_Type(strong_secret, weak_secret)

  # get the public key of the AM
  public_key = getPublic_Key()  # if public key not found -> program exits

  # check parameters
  allOK = True
  if username == "":
    allOK = False
  if password == "":
    allOK = False
  if RH_path_program == "":
    allOK = False
  if host == "":
    allOK = False
  if secret_type == "":
    allOK = False

  # contact host
  if allOK:
    response = getAnswer_From_RH(RH_path_program, host, username, password, secret_type,
         public_key)  # [answers list , exit code]
    if response is None:
      logger.error("no valid answer received from remote host...")
    else:
      resp_list = response[0]
      for resp in resp_list:
        # process answer, rep: <type>:<hash>:<rsa public key>:<dsa public key>
        msg = resp.split(":", 1)[1]
        msg_type = resp.split(":")[0]
        if msg_type == "ERROR":
          logger.error(msg)
          break
        elif msg_type == "WARNING":
          logger.info("WARNING: " + msg)
        elif msg_type == "ANSWER":
          if secret_type == "strong":
            processAnswer(msg, strong_secret, host)
          elif secret_type == "weak":
            processAnswer(msg, weak_secret, host)
          break
  else:
    logger.error("one of the parameters was not set...")

def getSecret_Type(strong, weak):
  secret_type = ""
  if not strong == "":
    secret_type = "strong"
  elif not weak == "":
    secret_type = "weak"
  return secret_type

def getPublic_Key():
  logger.info("locating public key...")
  path_rsa = "/etc/ssh/ssh_host_rsa_key.pub"
  path_dsa = "/etc/ssh/ssh_host_dsa_key.pub"
  if os.access(path_rsa, os.F_OK):
    return readFirst_Line(path_rsa).split()[1]
  elif os.access(path_dsa, os.F_OK):
```

```
    return readFirst_Line(path_dsa).split()[1]
  else:
    error_quit("the SSH public key file could not be accessed...")

def readFirst_Line(path):
  f = open(path, 'r')
  line = f.readline()
  f.close()
  return line

def getAnswer_From_RH(path, host, uname, passwd, secret_type, public_key):
  global Kpub_RH
  answer = None
  IP_list = domainToIPs(host)
  if (len(IP_list) == 0):
    logger.error("domain name could not be resolved to an IP address...")
    return None
  IP = IP_list[0]
  # connect through SSH
    # need to add a timeout here
  logger.info("connecting to " + host + " at " + IP)
  SSH_connection = initConnection(IP)
  Kpub_RH = SSH_connection[0] # put the key in the global variable
  if SSH_connection:
    logger.info("connection established...")
    # log in
    if loginPassword(uname, passwd):
      logger.info("login succeeded...")
      # execute command
      answer = execCommand("sudo " + path + " -s " + secret_type + " -k " + public_key)

      if answer is not None and len(answer[0]) == 0:
        answer = None
      if answer is not None:
        logger.info("response received...")
    else:
      logger.error("login failed; the credentials were not accepted...")
    # disconnect
    closeConnection()
    logger.info("connection closed...")
  else:
    logger.error("failed to set up a connection with the remote host...")
  return answer

def processAnswer(answer, secret, host):
  global Kpub_RH
  key_type = base64.b64decode(Kpub_RH)[4:11]
  logger.info("processing answer...")
  logger.debug("answer:\n" + answer)

  # parse answer  # <hash>:<rsa public key>:<dsa public key>
  untrusted_hash = answer.split(":")[0]
  logger.info("decrypting hash...")
  untrusted_hash = decryptRSA(untrusted_hash, key_type)
  untrusted_rsa_key = answer.split(":")[1]
  untrusted_dsa_key = answer.split(":")[2]

  #compare the public keys
  key_ok = False
  if key_type == "ssh-rsa":
    if Kpub_RH == untrusted_rsa_key:
      logger.info("rsa public key matched...")
    else:
      logger.warning("the public key returned by the remote host doesn't match the key
          used to set up the SSH connection. You may be a victim of a man-in-the-middle
          attack... ")
  elif key_type == "ssh-dss":
    if Kpub_RH == untrusted_dsa_key:
      logger.info("dsa public key matched...")
    else:
```

```
      logger.warning("the public key returned by the remote host doesn't match the key used to set up the SSH connection. You may be a victim of a man-in-the-middle attack... ")

  # calculate the hash with local data
  trusted_hash = makeHash(secret, untrusted_rsa_key, untrusted_dsa_key)
  if trusted_hash == untrusted_hash:
    logger.debug("hash " + trusted_hash + " is trusted...")
    logger.info("hash is TRUSTED...")
    # generate SSHFP records
    makeSSHFP_Records(host, untrusted_rsa_key, untrusted_dsa_key)
  else:
    # warn admin
    logger.warning("the hash received from host \"" + host + "\" is UNTRUSTED! The remote host did NOT proof its knowledge of the secret. You may be a victim of a man-in-the-middle attack, or your public key was not accepted. The retrieved public key(s) won't be pushed to the DNS server..")

def decryptRSA(msg, key_type):
  msg = base64.b64decode(msg)
  Kpriv_AM_path = getPrivate_Key_Path(key_type)
  try:
    key = RSA.load_key(Kpriv_AM_path)
  except:
    error_quit("unable to load private key (wrong permissions?)")
  decrypted_hash = key.private_decrypt(msg, RSA.pkcs1_oaep_padding)
  return decrypted_hash

def makeHash(secret, rsa, dsa):
  data = secret + rsa + dsa
  return hashlib.sha512(data).hexdigest()

def makeSSHFP_Records(hostname, rsa_key, dsa_key):
  global SSHFP_list
  global SSHFP_ttl
  logger.info("generating SSHFP records...")

  # generate SSHFP records
  SSHFP_rsa = hostname + " " + SSHFP_ttl + " IN SSHFP 1 1 " + hashlib.sha1(base64.b64decode(rsa_key)).hexdigest()
  SSHFP_dsa = hostname + " " + SSHFP_ttl + " IN SSHFP 2 1 " + hashlib.sha1(base64.b64decode(dsa_key)).hexdigest()

  logger.info("SSHFP records generated...")
  logger.debug("SSHFP_rsa: " + SSHFP_rsa)
  logger.debug("SSHFP_dsa: " + SSHFP_dsa)

  # collect them in a list
  SSHFP_list.append(SSHFP_rsa)
  SSHFP_list.append(SSHFP_dsa)

def processList_Of_SSHFP_records(path):
  global SSHFP_list

  if os.access(path, os.F_OK):
    f = open(path, 'r')
    contents = f.read()
    logger.debug("list of SSHFP records to push to DNS:\n" + contents)
    for line in contents.splitlines():
      SSHFP_list.append(line)
    f.close()
    logger.info("list read by program...")
  else:
    error_quit("the list of SSHFP records \"" + path  + "\" could not be accessed...")

def testSSHFP_list(SSHFP_list):
  notEmpty = False
  if len(SSHFP_list) > 0:
    notEmpty = True
  else:
```

```
    logger.info("no SSHFP records to be pushed to DNS...")
  return notEmpty

def makeDNS_Update(path, server, zone, SSHFP_list):
  logger.info("generating DNS update in temporary file \"" + path + "\"...")
  f = open(path,"w")
  f.write("server " + server + "\n")
  f.write("zone " + zone + "\n")
  for record in SSHFP_list:
    f.write("update add " + record + "\n")
  f.write("show \n")
  f.write("send \n")
  f.close()

  # just for debugging
  f = open(path,"r")
  logger.debug("update:\n" + f.read())

def pushSSHFP_records(key, DNS_update):
  if os.access(key, os.F_OK):
    logger.info("trying to push SSHFP RR's to the DNS...")
    command = subprocess.Popen(shlex.split("nsupdate -k " + key + " -v " + DNS_update), stdout = subprocess.PIPE)
    #response = command.communicate()[0]
    output = command.communicate()
    response = output[0]

    # test response for errors
    status = "ERROR"
    for line in response.splitlines():
      if "status: " in line:
        status = line.split(",")[1].split(":")[1].strip() # extract the status
    if status == "NOERROR":
      logger.info("DNS update was successful...")
    elif response == "":
      logger.error("DNS update was NOT successful..")
      logger.error("no response from DNS server received or the DNS could not be contacted.")
    else:
      logger.error("DNS update was NOT successful..")
      logger.debug("response:\n" + response)

    # clean up
    os.remove(DNS_update)
    logger.info("temporary file \"" + DNS_update + "\" with DNS update removed...")
  else:
    logger.info("the private key file \"" + key + "\" could not be accessed,the DNS update will not be executed...")

def error_quit(msg):
  logger.error(msg)
  logger.info("program has terminated...")
  sys.exit(1)

def getPrivate_Key_Path(key_type):
  logger.info("locating private key...")
  path_rsa = "/etc/ssh/ssh_host_rsa_key"
  path_dsa = "/etc/ssh/ssh_host_dsa_key"
  if key_type == "ssh-rsa":
    if os.access(path_rsa, os.F_OK):
      return path_rsa
    else:
      error_quit("the SSH private key file could not be accessed...")
  elif key_type == "ssh-dss":
    if os.access(path_dsa, os.F_OK):
      return path_dsa
    else:
      error_quit("the SSH private key file could not be accessed...")

### main program ###
```

```python
def main():
  global logger
  global logfile
  global username
  global password
  global RH_path_program
  global clear_secrets
  global secrets_path
  global SSHFP_list
  global SSHFP_ttl
  global DN_DNS
  global DNS_zone
  global DNS_update_file
  global Kpub_RH
  global private_key_DNS_admin

  # parse arguments #
  prog_description = "This tool can be used to retrieve the SSH public host keys from remote machines and push their fingerprints to a DNS server. If you already have a list of SSHFP records, you can feed them to this program and push them to DNS. This way you can skip the key retrieval process."
  arg_parser = argparse.ArgumentParser(description = prog_description)

  arg_parser.add_argument('-l',
   required = False,
   default = "",
   dest = 'SSHFP_RR_list',
   action = 'store',
   help = 'The path to a list of SSHFP resource records, ready to push to the DNS server.')

  arg_parser.add_argument('-q',
   required = False,
   default = False,
   dest = 'quiet',
   action = 'store_const',
   const = True,
   help = 'Quiet mode. No output will be printed to stdout.')

  arg_parser.add_argument('-v',
   required = False,
   default = False,
   dest = 'verbose',
   action = 'store_const',
   const = True,
   help = 'Verbose mode. Debug info will also be printed to stdout.')

  arg_parser._optionals.title = "flag arguments" # fixes the "optional arguments" in the help
  arguments = arg_parser.parse_args()

  conf = True
  ## configuration ##
  # parse config file #
  config_file = "config/tool_AM.conf"
  config_parser = ConfigParser.RawConfigParser()
  if len(config_parser.read(config_file)) > 0:
    # from config #
    if config_parser.has_option('secrets file', 'path'):
      secrets_path = config_parser.get('secrets file', 'path')

    if config_parser.has_option('remote host', 'path to program'):
      RH_path_program = config_parser.get('remote host', 'path to program')

    if config_parser.has_option('DNS server', 'domain name DNS server'):
      DN_DNS = config_parser.get('DNS server', 'domain name DNS server')

    if config_parser.has_option('DNS server', 'private key admin'):
      private_key_DNS_admin = config_parser.get('DNS server', 'private key admin')

```

```
    if config_parser.has_option('DNS server', 'zone file'):
      DNS_zone = config_parser.get('DNS server', 'zone file')

    if config_parser.has_option('DNS server', 'ttl'):
      SSHFP_ttl = config_parser.get('DNS server', 'ttl')

    if config_parser.has_option('logging', 'path logfile'):
      logfile = config_parser.get('logging', 'path logfile')
  else:
    conf = False

  # from arguments #
  SSHFP_list_path = arguments.SSHFP_RR_list
  quiet = arguments.quiet
  verbose = arguments.verbose

  # configure logging #
  # info levels: DEBUG (10) < INFO (20) < WARNING (30) <  ERROR (40) < CRITICAL (50)
  logger = logging.getLogger("standaard_log")
  logger.setLevel(logging.DEBUG) # lowest level it will log
  ch_stdout = logging.StreamHandler(sys.stdout)
  if verbose:
    ch_stdout.setLevel(logging.DEBUG)
  else:
    ch_stdout.setLevel(logging.INFO)
  fm_stdout = logging.Formatter("%(levelname)s - %(message)s")
  ch_stdout.setFormatter(fm_stdout)

  ch_file = logging.FileHandler(logfile)
  ch_file.setLevel(logging.INFO) # lowest level it will log -> omit DEBUG messages
  fm_file = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")
  ch_file.setFormatter(fm_file)

  if not quiet:
    logger.addHandler(ch_stdout) # log to stdout
  logger.addHandler(ch_file) # log to file

  if SSHFP_list_path == "":
    # prompt user for credentials
    print "\nPlease provide your credentials for the remote hosts."
    username = raw_input("Username: ")
    password = getpass.getpass("Password: ")
    print ""
    print "Please provide the passphrase to decrypt the secrets file."
    Kdecrypt = getpass.getpass("Passphrase: ")
    print ""

  ## program flow ##
  logger.info("program started...")
  if not conf:
    logger.warning("nothing read from configuration file")
  if SSHFP_list_path == "":
    logger.info("no SSHFP list provided, the public keys will be retrieved
        dynamically...")
    # decrypt the secrets file
    decryptAES_File(secrets_path, Kdecrypt)
    # process each host in the secrets file
    processList_Of_Hosts()
    if testSSHFP_list(SSHFP_list):
      # generate the DNS update command
      makeDNS_Update(DNS_update_file, DN_DNS, DNS_zone, SSHFP_list)
      # push the RR's to DNS
      pushSSHFP_records(private_key_DNS_admin, DNS_update_file)
  else:
    logger.info("a list of SSHFP records is provided...")
    # put the list in the global variable
    processList_Of_SSHFP_records(SSHFP_list_path)
    if testSSHFP_list(SSHFP_list):
      # generate the DNS update command
      makeDNS_Update(DNS_update_file, DN_DNS, DNS_zone, SSHFP_list)
```

```
      # push the RR's to DNS
      pushSSHFP_records(private_key_DNS_admin, DNS_update_file)
  logger.info("program has terminated...")

if __name__ == "__main__":
  main()
```

### A.1.2 Configuration file

Listing 2: conf/tool_AM.conf

```
[secrets file]
path=path/to/secrets/file.txt

[remote host]
path to program=path/to/program.py

[DNS server]
domain name DNS server=dns.domain.org
private key admin=path/to/keyfile.private
zone file=zone.domain.org
ttl=1800 ;ttl for the SSHFP records in ms

[logging]
path logfile=path/to/logfile.log
```

### A.1.3 Usage

Listing 3: ./tool_AM.py -h

```
usage: tool_AM.py [-h] [-l SSHFP_RR_LIST] [-q] [-v]

This tool can be used to retrieve the SSH public host keys from remote
machines and push their fingerprints to a DNS server. If you already have a
list of SSHFP records, you can feed them to this program and push them to DNS.
This way you can skip the key retrieval process.

flag arguments:
  -h, --help         show this help message and exit
  -l SSHFP_RR_LIST   The path to a list of SSHFP resource records, ready to
                     push to the DNS server.
  -q                 Quiet mode. No output will be printed to stdout.
  -v                 Verbose mode. Debug info will also be printed to stdout.
```

### A.1.4 Python interface to SSH client functionality

Listing 4: lib/source/sshexec.c

```
/*
 ***********************************************************************************
 * sshexec.c (in) sshexec.so (out)                                                 *
 *                                                                                 *
 * THIS IS A MODIFIED VERSION OF ssh2_exec.c FROM libssh2's EXAMPLE FILES.         *
 * IT WAS MEANT TO BE COMPILED TO A PYTHON MODULE WITH THE FOLLOWING COMMAND:      *
 *                                                                                 *
 * gcc -shared -I/usr/include/python2.6/ -lpython2.6 -lssh2 -o sshexec.so sshexec.c *
 *                                                                                 *
 * SSH module for Python to execute a command on a remote host.                    *
 * At the moment only one connection can exist at a time.                          *
 ***********************************************************************************
 */

#include "libssh2_config.h"
#include <libssh2.h>
#include <Python.h>

#ifdef HAVE_WINSOCK2_H
```

```
# include <winsock2.h>
#endif
#ifdef HAVE_SYS_SOCKET_H
# include <sys/socket.h>
#endif
#ifdef HAVE_NETINET_IN_H
# include <netinet/in.h>
#endif
#ifdef HAVE_SYS_SELECT_H
# include <sys/select.h>
#endif
# ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif
#ifdef HAVE_ARPA_INET_H
# include <arpa/inet.h>
#endif

#include <sys/time.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <ctype.h>
#include <netdb.h>
#include <unistd.h>
#include <pwd.h>
#include <string.h>
#include <time.h>

//#define LIBSSH2_ALLOC(session, count) session->alloc((count), &(session)->abstract)
#define TIMEOUT 20

const char *homedir = "";
int sock;
LIBSSH2_SESSION *session = NULL;
int auth = 0;

/*
 * (c) Daniel Stenberg
 *
 * Found this function at
 * http://www.mail-archive.com/libssh2-devel@lists.sourceforge.net/msg01630.html
 */
size_t _libssh2_base64_encode(const char *inp, size_t insize, char **outptr) {
    //extern LIBSSH2_SESSION *session;
    const char table64[]=
      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    unsigned char ibuf[3];
    unsigned char obuf[4];
    int i;
    int inputparts;
    char *output;
    char *base64data;
    const char *indata = inp;

    *outptr = NULL; /* set to NULL in case of failure before we reach the end */

    if(0 == insize)
        insize = strlen(indata);

    base64data = output = malloc(insize*4/3+4); //LIBSSH2_ALLOC(session, insize*4/3+4);
    if(NULL == output)
        return 0;

    while(insize > 0) {
        for (i = inputparts = 0; i < 3; i++) {
            if(insize > 0) {
                inputparts++;
```

```
                ibuf[i] = *indata;
                indata++;
                insize--;
            } else {
                ibuf[i] = 0;
            }

        }

        obuf[0] = (unsigned char)  ((ibuf[0] & 0xFC) >> 2);
        obuf[1] = (unsigned char) (((ibuf[0] & 0x03) << 4) | \
                                   ((ibuf[1] & 0xF0) >> 4));
        obuf[2] = (unsigned char) (((ibuf[1] & 0x0F) << 2) | \
                                   ((ibuf[2] & 0xC0) >> 6));
        obuf[3] = (unsigned char)   (ibuf[2] & 0x3F);

        switch(inputparts) {
            case 1: /* only one byte read */
                snprintf(output, 5, "%c%c==",
                         table64[obuf[0]],
                         table64[obuf[1]]);
                break;
            case 2: /* two bytes read */
                snprintf(output, 5, "%c%c%c=",
                         table64[obuf[0]],
                         table64[obuf[1]],
                         table64[obuf[2]]);
                break;
            default:
                snprintf(output, 5, "%c%c%c%c",
                         table64[obuf[0]],
                         table64[obuf[1]],
                         table64[obuf[2]],
                         table64[obuf[3]] );
                break;
        }
        output += 4;
    }

    *output = 0;
    *outptr = base64data; /* make it return the actual data memory */

    return strlen(base64data); /* return the length of the new data */
}

static int waitsocket(int socket_fd, LIBSSH2_SESSION *session) {
    struct timeval timeout;
    int rc;
    fd_set fd;
    fd_set *writefd = NULL;
    fd_set *readfd = NULL;
    int dir;

    timeout.tv_sec = 10;
    timeout.tv_usec = 0;

    FD_ZERO(&fd);

    FD_SET(socket_fd, &fd);

    /* now make sure we wait in the correct direction */
    dir = libssh2_session_block_directions(session);

    if(dir & LIBSSH2_SESSION_BLOCK_INBOUND)
        readfd = &fd;

    if(dir & LIBSSH2_SESSION_BLOCK_OUTBOUND)
        writefd = &fd;

    rc = select(socket_fd + 1, readfd, writefd, NULL, &timeout);
```

```
    return rc;
}

static void closesession(void) {
    extern int sock;
    extern LIBSSH2_SESSION *session;
    libssh2_session_disconnect(session, "Normal disconnect");
    libssh2_session_free(session);
    session = NULL;
    close(sock);
}

static int closechannel(LIBSSH2_CHANNEL *channel, unsigned int to) {
    extern int sock;
    int exitcode = 127;
    int rc;
    time_t start;

    // Close channel
    start = time(NULL);
    while ((rc = libssh2_channel_close(channel)) == LIBSSH2_ERROR_EAGAIN) {
        // Time-out?
        if (time(NULL) - start >= to) {
            break;
        }
        waitsocket(sock, session);
    }

    // Get exit status
    if (rc == 0) {
        exitcode = libssh2_channel_get_exit_status(channel);
    }

    libssh2_channel_free(channel);

    return exitcode;
}

static PyObject* py_domainToIPs(PyObject* self, PyObject* args) {
    const char *domain;
    struct hostent *he;
    int i;
    PyObject *ip;
    PyObject *lst;

    // Parse arguments
    if (!PyArg_ParseTuple(args, "s", &domain)) {
        return Py_None;
    }

    // Get addresses for host at domain
    he = gethostbyname(domain);
    if (!he) {
        return PyList_New(0);
    }

    // Count number of addresses
    for (i = 0; he->h_addr_list[i]; i++);
    if (i == 0) {
        return PyList_New(0);
    }

    // Create Python list
    lst = PyList_New(i);

    // Add addresses to list
    i = 0;
    while (he->h_addr_list[i]) {
        ip = PyString_FromString(inet_ntoa(*(struct in_addr*)(he->h_addr_list[i])));
```

```
        if (!ip) {
            return Py_None;
        }
        PyList_SetItem(lst, i, ip);
        i++;
    }

    return lst;
}

static PyObject* py_initConnection(PyObject* self, PyObject* args) {
    extern int sock;
    extern LIBSSH2_SESSION *session;
    const char *ip;
    char *khp = "/.ssh/known_hosts";
    unsigned int to = TIMEOUT;
    time_t start;
    char *kh;
    char check = 0;
    unsigned long hostaddr;
    struct sockaddr_in sin;
    LIBSSH2_KNOWNHOSTS *nh;
    int rc;
    size_t len;
    int type;
    const char *key;
    char *key_base64;
    struct libssh2_knownhost *host;
    PyObject *mismatch = Py_False;
    PyObject *ret;

    // Parse arguments
    if (!PyArg_ParseTuple(args, "s|sI", &ip, &khp, &to)) {
        return Py_None;
    }

    // Check if a session has already been initiated
    if (session) {
        return Py_None;
    }

    if (strcmp(khp, "/.ssh/known_hosts")) {
        kh = khp;
    } else {
        kh = malloc(strlen(homedir)+strlen("/.ssh/known_hosts")+1);
        strcpy(kh, homedir);
        strcat(kh, khp);
        check |= 1;
    }

    // Create socket and connect
    hostaddr = inet_addr(ip);
    sock = socket(AF_INET, SOCK_STREAM, 0);
    sin.sin_family = AF_INET;
    sin.sin_port = htons(22);
    sin.sin_addr.s_addr = hostaddr;
    if (connect(sock, (struct sockaddr*)(&sin),
                sizeof(struct sockaddr_in)) != 0) {
        return Py_None;
    }

    // Create a session instance
    session = libssh2_session_init();
    if (!session) {
        close(sock);
        return Py_None;
    }

    // Tell libssh2 we want it all done non-blocking
    libssh2_session_set_blocking(session, 0);
```

```

    // Start it up. This will trade welcome banners, exchange keys,
    // and setup crypto, compression, and MAC layers
    start = time(NULL);
    while ((rc = libssh2_session_startup(session, sock)) ==
           LIBSSH2_ERROR_EAGAIN) {
        // Time-out?
        if (time(NULL) - start >= to) {
            closesession();
            return Py_None;
        }
    }
    if (rc) {
        closesession();
        return Py_None;
    }

    // Check if the host's key is in the known-hosts file
    nh = libssh2_knownhost_init(session);
    if (!nh) {
        closesession();
        return Py_None;
    }
    key = libssh2_session_hostkey(session, &len, &type);
    libssh2_knownhost_readfile(nh, kh, LIBSSH2_KNOWNHOST_FILE_OPENSSH);
    if (check & 1) {
        free(kh);
    }
    if (key) {
        if (libssh2_knownhost_check(nh, (char *)ip, (char *)key, len,
                                    LIBSSH2_KNOWNHOST_TYPE_PLAIN|
                                    LIBSSH2_KNOWNHOST_KEYENC_RAW,
                                    &host) ==
                                    LIBSSH2_KNOWNHOST_CHECK_MISMATCH) {
            mismatch = Py_True;
        }
    } else {
        closesession();
        libssh2_knownhost_free(nh);
        return Py_None;
    }
    libssh2_knownhost_free(nh);

    // Convert binary key into base64 format and return it
    _libssh2_base64_encode(key, len, &key_base64);
    ret = Py_BuildValue("(s,O)", key_base64, mismatch);
    free(key_base64);
    return ret;
}

static PyObject* py_loginPassword(PyObject* self, PyObject* args) {
    extern LIBSSH2_SESSION *session;
    extern int auth;
    const char *username;
    const char *password;
    unsigned int to = TIMEOUT;
    time_t start;
    int rc;

    // Parse arguments
    if (!PyArg_ParseTuple(args, "ss|I", &username, &password, &to)) {
        return Py_None;
    }

    // Check if there is an active session
    if (!session) {
        return Py_None;
    }

    // Try password login
```

```
    start = time(NULL);
    while ((rc = libssh2_userauth_password(session, username, password)) ==
           LIBSSH2_ERROR_EAGAIN);
        // Time-out?
        if (time(NULL) - start >= to) {
            return Py_False;
        }
    if (rc) {
        return Py_False;
    }

    auth = 1;
    return Py_True;
}

static PyObject* py_loginPublicKey(PyObject* self, PyObject* args) {
    extern LIBSSH2_SESSION *session;
    extern int auth;
    const char *username;
    char *puk = "/.ssh/id_rsa.pub";
    char *pvk = "/.ssh/id_rsa";
    char *pub;
    char *prv;
    char check = 0;
    const char *passphrase = "";
    unsigned int to = TIMEOUT;
    time_t start;
    int rc;

    // Parse arguments
    if (!PyArg_ParseTuple(args, "s|sssI", &username, &puk, &pvk, &passphrase, &to)) {
        return Py_None;
    }

    // Check if there is an active session
    if (!session) {
        return Py_None;
    }

    // Construct path to public key
    if (strcmp(puk, "/.ssh/id_rsa.pub")) {
        pub = puk;
    } else {
        pub = malloc(strlen(homedir)+strlen("/.ssh/id_rsa.pub")+1);
        strcpy(pub, homedir);
        strcat(pub, puk);
        check |= 1;
    }

    // Construct path to private key
    if (strcmp(pvk, "/.ssh/id_rsa")) {
        prv = pvk;
    } else {
        prv = malloc(strlen(homedir)+strlen("/.ssh/id_rsa")+1);
        strcpy(prv, homedir);
        strcat(prv, pvk);
        check |= 2;
    }

    // Try public key login
    start = time(NULL);
    while ((rc = libssh2_userauth_publickey_fromfile(session, username, pub,
                                                      prv, passphrase)) ==
                                                      LIBSSH2_ERROR_EAGAIN) {
        // Time-out?
        if (time(NULL) - start >= to) {
            // Free memory
            if (check & 1) {
                free(pub);
            }
```

```
            if (check & 2) {
                free(prv);
            }
            return Py_False;
        }
    }

    // Free memory
    if (check & 1) {
        free(pub);
    }
    if (check & 2) {
        free(prv);
    }

    // Check if succeeded
    if (rc) {
        return Py_False;
    }

    auth = 1;
    return Py_True;
}

static PyObject* py_execCommand(PyObject* self, PyObject* args) {
    extern int sock;
    extern LIBSSH2_SESSION *session;
    extern int auth;
    const char *command;
    unsigned int to = TIMEOUT;
    time_t start;
    int rc;
    char buffer[0x4000];
    int pos;
    int exitcode;
    int i;
    int j;
    LIBSSH2_CHANNEL *channel;
    int lenanswers = 10;
    char **answers;
    char **temp;
    int numanswers = 0;
    PyObject *lst;

    // Parse arguments
    if (!PyArg_ParseTuple(args, "s|I", &command, &to)) {
        return Py_None;
    }

    // Check if there is an active session and if the user has been logged in
    if (!session || !auth) {
        return Py_None;
    }

    answers = malloc(lenanswers*sizeof(char*));
    if (answers == NULL) {
        return Py_None;
    }

    // Exec non-blocking on the remote host
    start = time(NULL);
    while ((channel = libssh2_channel_open_session(session)) == NULL &&
          libssh2_session_last_error(session,NULL,NULL,0) ==
          LIBSSH2_ERROR_EAGAIN) {
        // Time-out?
        if (time(NULL) - start >= to) {
            if (channel != NULL) {
                closechannel(channel, to);
            }
            free(answers);
```

```
            return Py_None;
        }
        waitsocket(sock, session);
    }
    if (channel == NULL) {
        free(answers);
        return Py_None;
    }

    // Execute command
    start = time(NULL);
    while ((rc = libssh2_channel_exec(channel, command)) ==
          LIBSSH2_ERROR_EAGAIN) {
        // Time-out?
        if (time(NULL) - start >= to) {
            closechannel(channel, to);
            free(answers);
            return Py_None;
        }
        waitsocket(sock, session);
    }
    if (rc != 0) {
        closechannel(channel, to);
        free(answers);
        return Py_None;
    }

    // Loop until all answers have been received
    start = time(NULL);
    for (;;) {
        // Loop until we block
        do {
            rc = libssh2_channel_read(channel, buffer, sizeof(buffer));
            if (rc > 0) {
                i = j = 0;

                // Split answer on newlines and put every substring in the
                // answers array
                while (j < rc) {
                    for (; buffer[j] != '\n' && j < rc; j++);
                    pos = numanswers;
                    numanswers++;

                    // Check if there still is enough memory
                    if (numanswers > lenanswers) {
                        lenanswers *= 2;
                        temp = realloc(answers, lenanswers*sizeof(char*));

                        // If realloc failed, free memory and return
                        if (temp == NULL) {
                            numanswers--;
                            for (i = 0; i < numanswers; i++) {
                                free(answers[i]);
                            }
                            free(answers);
                            closechannel(channel, to);
                            return Py_None;
                        }
                        answers = temp;
                    }
                    answers[pos] = malloc((j-i+1)*sizeof(char));
                    strncpy(answers[pos], &buffer[i], (j-i));
                    answers[pos][j-i] = '\0';
                    j++;
                    i = j;
                }
            }
        }
        while (rc > 0);

```

```
        // This is due to blocking that would occur otherwise so we loop on
        // this condition
        if (rc == LIBSSH2_ERROR_EAGAIN) {
            // Time-out?
            if (time(NULL) - start >= to) {
                closechannel(channel, to);
                for (i = 0; i < numanswers; i++) {
                    free(answers[i]);
                }
                free(answers);
                return Py_None;
            }
            waitsocket(sock, session);
        } else {
            break;
        }
    }

    // Close channel
    exitcode = closechannel(channel, to);

    // Create Python list
    lst = PyList_New(numanswers);

    // Convert answers
    for (i = 0; i < numanswers; i++) {
        PyList_SetItem(lst, i, PyString_FromString(answers[i]));
        free(answers[i]);
    }
    free(answers);

    return Py_BuildValue("(O,i)", lst, exitcode);
}

static PyObject* py_closeConnection(PyObject* self, PyObject* args) {
    extern LIBSSH2_SESSION *session;
    extern int auth;

    // Check if there is an active session
    if (!session) {
        return Py_False;
    }

    closesession();
    auth = 0;

    return Py_True;
}

static PyMethodDef sshexec_methods[] = {
    {"domainToIPs", py_domainToIPs, METH_VARARGS},
    {"initConnection", py_initConnection, METH_VARARGS},
    {"loginPassword", py_loginPassword, METH_VARARGS},
    {"loginPublicKey", py_loginPublicKey, METH_VARARGS},
    {"execCommand", py_execCommand, METH_VARARGS},
    {"closeConnection", py_closeConnection, METH_VARARGS},
    {NULL, NULL}
};

void initsshexec() {
    extern const char *homedir;
    struct passwd *pw;

    (void) Py_InitModule("sshexec", sshexec_methods);

    // Get user's home directory
    pw = getpwuid(getuid());
    homedir = pw->pw_dir;
}
```

## A.2 For the remote host

### A.2.1 Application

Listing 5: tool_RH.py

```
#!/usr/bin/python

### imports ###
import ConfigParser    # reading config files
import argparse        # parsing parameters
import subprocess      # spawning new processes
import shlex           # determining the correct tokenization for args
import hashlib         # computing hashes
import sys
import os
import string
import base64          # base64 encoding/decoding
import random
import math
import struct
from M2Crypto import RSA, DSA
from unbound import ub_ctx, RR_TYPE_SSHFP, RR_CLASS_IN

### default parameters ###
TOOL_CONF = "conf/tool_RH.conf"
RESOLV_CONF = "/etc/resolv.conf"
TRUSTED_KEY = "/etc/unbound/root.key"
HOST_KEYS = "/etc/ssh"
AM_DOMAIN = "localhost"

### functions ###
def warning(msg):
  print "WARNING:" + msg

def error(msg):
  print "ERROR:" + msg
  sys.exit(1)

def answer(digest, rsa_key, dsa_key, am_key):
  print "ANSWER:" + encrypt(digest, am_key) + ":" + rsa_key + ":" + dsa_key
  sys.exit(0)

def encrypt(msg, key):
  key = base64.b64decode(key)
  fields = []

  sb = key[0:4]
  if len(sb) != 4:
    error("bad key")
  sd = struct.unpack(">I", sb)[0]
  type = key[4:4+sd]
  if len(type) != sd:
    error("bad key")

  if type =="ssh-dss":
    error("RSA key required") # DSA cannot be used for encryption/decryption
  elif type != "ssh-rsa":
    error("bad key")

  # Extract exponent and modulus
  s = 4 + sd
  for i in range(2):
    sb = key[s:s+4]
    if len(sb) != 4:
      error("bad key")
    sd = struct.unpack(">I", sb)[0]
    val = key[s+4:s+4+sd]
    if len(val) != sd:
```

```python
      error("bad key")
    fields.append(sb + val)
    s += 4 + sd

  e = fields[0]
  n = fields[1]

  key = RSA.new_pub_key((e, n))

  return base64.b64encode(key.public_encrypt(msg, RSA.pkcs1_oaep_padding))

def getRandomString(length):
  return ''.join(random.choice(string.printable) for x in range(length))

def getSystemUUID():
  command = subprocess.Popen(shlex.split('dmidecode -s system-uuid'), stdout=subprocess.PIPE)
  return command.communicate()[0].rstrip()

def getSystemProductName():
  command = subprocess.Popen(shlex.split('dmidecode -s system-product-name'), stdout=subprocess.PIPE)
  return command.communicate()[0].rstrip()

# not required according to SMBIOS specification
def getMotherboardSerial():
  command = subprocess.Popen(shlex.split('dmidecode -s baseboard-serial-number'), stdout=subprocess.PIPE)
  return command.communicate()[0].rstrip()

def makeHash(secret, rsa_key, dsa_key):
  secret += rsa_key + dsa_key
  return hashlib.sha512(secret).hexdigest()

def getPublicKey_rsa():
  try:
    f = open(HOST_KEYS + '/ssh_host_rsa_key.pub', 'r')
    key = f.readline().split()[1]
  except IOError:
    return ""
  except:
    key = ""

  f.close()
  return key

def getPublicKey_dsa():
  try:
    f = open(HOST_KEYS + '/ssh_host_dsa_key.pub', 'r')
    key = f.readline().split()[1]
  except IOError:
    return ""
  except:
    key = ""

  f.close()
  return key

def getStrong_Secret():
  return getSystemUUID()

def getWeak_Secret():
  # motherboard_serial+system_product_name
  return getMotherboardSerial()+getSystemProductName()

def getBogus_Secret():
  # random string, with padding to minimize collisions
  return "~@$^*)"+getRandomString(128)+"`!#%&("

def getSecretHash(secret_type, rsa_key, dsa_key):
```

```
  secret=""
  if secret_type == "strong":
    secret=getStrong_Secret()
  elif secret_type == "weak":
    secret=getWeak_Secret()
  elif secret_type == "bogus":
    secret=getBogus_Secret()
  else:
    error("wrong type of secret")

  if not secret:
    error("wrong permissions")

  return makeHash(secret, rsa_key, dsa_key)

def checkPublic_Key_AM(key, domain):
  # validate the public key with the SSHFP record

  types = {"ssh-rsa": 1, "ssh-dss": 2}

  try:
    key = base64.b64decode(key)
  except:
    error("bad key")

  # Get key type
  keytype = key[4:11]

  if keytype not in types:
    return False

  keytype = types[keytype]

  # Get key hash
  digest = hashlib.sha1(key).hexdigest()

  # Init Unbound
  ctx = ub_ctx()
  ctx.resolvconf(RESOLV_CONF)

  # Read trusted (root) public key for DNSSEC validation
  if (os.path.isfile(TRUSTED_KEY)):
    ctx.add_ta_file(TRUSTED_KEY)

  # Resolve SSHFP records for the domain name
  status, result = ctx.resolve(domain, RR_TYPE_SSHFP, RR_CLASS_IN)

  # Check if resolving succeeded and if the DNSSEC validation was positive
  if status == 0 and result.havedata and result.secure:
    sshfp = dict()

    # Loop through the resolved SSHFP records
    for record in result.data.address_list:
      fp = record.split(".")

      # Get public key type and digest type
      pub = int(fp.pop(0))
      dig = int(fp.pop(0))

      # Digest algorithm must be SHA1; also no need to compute unused key types
      if dig != 1 or pub != keytype:
        continue

      conv = ""

      # Convert FP from decimal to hexadecimal string
      for num in fp:
        h = hex(int(num))[2:]
        if len(h) == 1:
          h = "0"+h
```

```
        conv += h

      # Store FP
      if pub not in sshfp:
        sshfp[pub] = []
      sshfp[pub].append(conv)

    # See if the fingerprints match
    if digest in sshfp[keytype]:
      return True

  return False

### main program ###
def main():
  global TOOL_CONF
  global RESOLV_CONF
  global TRUSTED_KEY
  global HOST_KEYS
  global AM_DOMAIN

  # parse arguments #
  prog_description = "This tool will return the secret of this machine."
  arg_parser = argparse.ArgumentParser(description=prog_description)
  arg_parser.add_argument('-s',
   choices=['strong', 'weak'],
   required=True,
   dest='type_secret',
   action='store',
   help='The type of secret that must be returned "strong" or "weak".')
  arg_parser.add_argument('-k',
   required=True,
   dest='rsa_public_key',
   action='store',
   help='The client\'s public key.')
  arg_parser.add_argument('-c',
   required=False,
   default=sys.path[0]+"/"+TOOL_CONF,
   dest='path_to_conf',
   action='store',
   help='The path of the configuration file.')

  arg_parser._optionals.title = "flag arguments" # fixes the "optional arguments" in the
       help
  arguments=arg_parser.parse_args()

  ## configuration ##
  # parse config file #
  TOOL_CONF = arguments.path_to_conf
  config_parser = ConfigParser.RawConfigParser()
  if len(config_parser.read(TOOL_CONF)) > 0:
    # from config #
    if config_parser.has_option('administration machine', 'domain_name'):
      AM_DOMAIN = config_parser.get('administration machine', 'domain_name')

    if config_parser.has_option('key files', 'host_keys'):
      HOST_KEYS = config_parser.get('key files', 'host_keys')

    if config_parser.has_option('config files', 'resolv_conf'):
      RESOLV_CONF = config_parser.get('config files', 'resolv_conf')

    if config_parser.has_option('key files', 'trusted_key'):
      TRUSTED_KEY = config_parser.get('key files', 'trusted_key')
  else:
    warning("nothing read from configuration file")

  # from arguments #
  Kpub_AM = arguments.rsa_public_key
  TypeSecret = arguments.type_secret

```

```
  # program flow #
  rsa_key = getPublicKey_rsa()
  dsa_key = getPublicKey_dsa()

  if not rsa_key and not dsa_key:
    error("no host key(s) found")

  if checkPublic_Key_AM(Kpub_AM, AM_DOMAIN):
    # return secret
    answer(getSecretHash(TypeSecret, rsa_key, dsa_key), rsa_key, dsa_key, Kpub_AM)
  else:
    # return bogus answer
    answer(getSecretHash("bogus", rsa_key, dsa_key), rsa_key, dsa_key, Kpub_AM)

if __name__ == "__main__":
  main()
```

### A.2.2 Configuration file

Listing 6: conf/tool_RH.conf

```
[administration machine]
domain_name=admin.domain.org

[key files]
host_keys=/etc/ssh
trusted_key=/etc/unbound/root.key

[config files]
resolv_conf=/etc/resolv.conf
```