Image-based authentication of public keys and applications to SSH

Written by Dmitri Epshtein

Supervised by Hugo Krawczyk

Abstract

We explore the use of human-friendly mechanisms to identify authentic public keys needed to bootstrap security protocols.
These mechanisms improve and complement current server authentication techniques and password security.
A particularly interesting method uses color images as a means for human validation of public keys of remote servers.
This approach raises several interesting technical issues that involve cryptographic considerations as well as image processing issues such as the automatic generation of "random yet sufficiently distinct" images.

An implementation of this approach and its integration in the framework of the popular SSH ("secure shell") protocol is presented.

1 Introduction

To provide full security for network applications working in Client-Server model the three following issues should be resolved:

- Server authentication
- Client authentication
- Data confidentiality and integrity

All three components mentioned above are important to provide real network security. Weakness in any of them will create a gap in the overall security and will allow attackers to succeed in their attacks.

In this work we deal with Server Authentication problems, paying attention to human factors.

First of all let's see why the Server authentication is an important part of network security infrastructure. Former network protocols (e.g. Classical Telnet) didn't provide Server authentication at all. The only security was Client password authentication. The newest Protocols and Applications (e.g. B2B, etc.) have

additional requirements. For these applications it is at least as important to know which server will receive payment as it is to know which Client will make the payment.

Many secure protocols and applications work in asymmetric scenarios, where server authentication is based on a pair of private and public keys, while the client is authenticated based on the user's passwords. It is a critical requirement for security of these protocols, that the server's public key, which is used by the client for encryption and signature verification, be a valid public key for that server.
Usually the server generates a pair of public and private key, saves the private key in a secure place and distributes its public key to the clients via mail, email, software, trusted third parties, etc. This public key then becomes the centerpiece in any protocol that validates the identity of that server.

As an example, consider the establishment of a secure channel between the client and a server through a key exchange algorithm, such as the well-known Diffie-Hellman protocol.
The Diffie-Hellman key exchange provides a shared key that cannot be determined by any eavesdropper to the protocol. However, to protect the protocol against active attackers ("man in the middle") trying to impersonate the server the protocol is usually combined with a public-key authentication technique. In this case the whole security of the exchange depends on the correct validation of the server's public key. We expand on this point next.

1.1 Illustration via Authenticated Diffie-Hellman

For illustration we succinctly describe a Diffie-Hellman exchange as specified by the SSH protocol [SSH-TRANS], and we highlight the point in the process where our work provides a significant improvement, namely, the server's public key verification.

The Server has Kprv/Kpub – a pair of private and public keys;
p is a prime number and g an element of high order mod p.

- Client generates a random number x (1 < x < p), computes e = (g^x mod p) and sends "e" to server.
- Server generates a random number y (1 < y < p), computes f = (g^y mod p).
- Server receives "e" from the Client and computes K = (e^y mod p).
- Server computes H=hash(Kpub || e || f || K)
- Server computes signature "s" on H with its private key Kprv.
- Server sends (Kpub || f || s) to Client.
- **Client verifies Kpub received from the Server**.
- Client computes K = (f^x mod p)
- Client computes H = hash(Kpub || e || f || K)
- Client verifies the signature "s" on H

After this a shared secret K and exchange hash H are used to derive encryption and authentication keys.

If the Client doesn't verify the Server's public key Kpub, it will not be protected against man-in-the-middle attack. In this case the attacker intercepts the messages sent between the server and client and replaces them with its own messages. It plays the role of client in the messages, which it sends to the server, and at same time plays the role of the server in the messages that it sends to the client.
Therefore, a failure to correctly verify Kpub leads to the total insecurity of the system. But how can a human user identify and verify a public key which usually consists of hundreds to thousands of bits?

1.2 Public Key Verification

A validation of the Server key in the public-key infrastructure is one of security issues where the human limitation mentioned above negatively affect security.
Several solutions are currently being used in security systems, but each of them has significant disadvantages.

One of possible solutions is to send the public key of the server signed by a trusted certification authority. The client only needs to know the CA root key, and can verify the validity of all host keys certified by CAs. However, there is no widely deployed key infrastructure available yet on the Internet. In addition, a CA service usually has costs, and it is difficult to imagine that people will be prepared to pay for every possible server on the network, when almost every computer can act as a server.

Another possible solution is for the Client to compare the Server public key with the correct key, which is stored locally on the client machine. In the first connection the client should manually verify and approve the Server public key. This method is reasonable when the client usually accesses its Servers from the same machine, but it is inapplicable in such places as Internet cafes, hotel computers, etc.

Summarizing, Server authentication is an important part of every network security system, and present methods don't provide a good solution in all cases. At this point we want to suggest an improved Server authentication mechanism that will improve exactly those weak points that were discussed above, i.e., human limitations.

1.3 Public Passwords

We build on the concept of "public password" introduced in [HK99]. In our application, a public password is the hashed version of a public key. In order to enable the use of our secure protocols in cases where the client's machine cannot verify the authenticity of the server's public key, the client will be provided with a hashed version of the public key – as its "public password". As with regular public keys, a "public password" requires no secrecy protection but requires integrity. Also, the "public password" should be short enough that a human user can recognize it if it is displayed, or can even type it if requested to do so. But it does not need to be memorized and can be safely written down on a peace of paper, sticker, plastic card, etc. The public password serves as "hand-held certificate" for a public key, which user can conveniently carry with him. Whenever presented with actual public key (after being transmitted to the user's terminal) the user can verify the validity of public key against the hand-held certificate. This enables a human user to participate in protocols that otherwise would be impossible to carry out without a memory

device. This idea can be useful in other scenarios as well. For example this solution is suited for credit-card applications, where public password can be recorded on the credit card itself.
As said, in our application, public password is a hashed value of the server's public key. The length of the public password depends on security level should be provided. In our case the attacker is not looking for collisions in the hash function during the process of key generation, then the public password needs only to resist "second preimage attacks". That is, for given a public key pk, it should be infeasible to find another pk` such that H(pk) = H(pk`). In this case a public password of 60 to 80 bits will suffice for most applications.

1.4 SSH Example

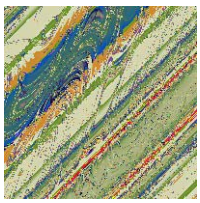
One of the first steps in this direction was done in SSH (Secure Shell) implementation.
When the client machine doesn't have a public key of the Server stored locally, it asks the user to confirm the hashed version of the key (named "fingerprint") received from the Server. The 512/1024 bits of the public key are converted by the MD5 hash function to 128 bits.

These 128 bits are represented to the user in hexadecimal format in this way:
RSA/DSA key fingerprint is:
d7:7d:cf:16:07:3b:5e:17:dc:b7:52:f1:eb:49:37:b1
Are you sure you want to continue connecting (yes/no)?

It is obvious that the presented fingerprint is too difficult for recognition or retyping by the user and it will generally lead to blind user acceptance (user will press Enter key without really verifying the fingerprint).

2 Improved Solution

In our work we want to take the idea of hashed public key for server authentication few steps forward.
Even though public passwords are short enough to be carried by a human being, they usually represent unstructured strings. Thus, for the user to be able to read, recognize or retype the public password, it is advisable to have a human-identifiable format for these passwords. In this work we implement three different representations for public passwords:

- String of English words: *"SCAN TOTE NOON DIE MAID COP"*
- String of Alpha-Numeric words: *"4786 8fqh hnpb"*
- Picture: 

2.1 English Words Format

One of the possible representations for mapping binary strings into easy-to-read (and write) words was introduced in the context of one-time passwords [Hal95]. This solution defines (rfc1760) a dictionary of 2048 words (mostly English words, 2 to 4 letters long) mapping each 11 bit binary string to a different word in the dictionary. Thus, a 66-bit string is represented by six words from this dictionary. An example of how a public password will be represented in this case is:
"SCAN TOTE NOON DIE MAID COP"

2.2 Alpha-Numeric Format

Another way to make the string easier to recognize for user is by decreasing the number of symbols in the string and increasing the range of used symbols to achieve the same security. Alpha-Numeric format is based on 26 letters and 10 digits. We exclude the letters "o" and "l" and the digits "0" and "1" from the symbols set because the user can easily confuse them. The remaining set of 32 symbols is used. Each 5 bits of the hashed public key define one symbol from the set. So 60 bits are converted into 12 symbols, which is presented as three words of four characters each. Such a string (e.g. "*qu24 ih2q sswb*") is secure enough for our purposes and easier for user recognition than the equivalent 15 hexadecimal numbers, and provides the same security.

2.3 Graphical format

One of the most attractive approaches to make long random strings identifiable by human beings is to use a "graphical representation" of these strings. Developing this approach is a main focus of this work.
In Section 3 we present in detail a method that provides public key authentication using computer images.

2.4 User Interface

At the point in which the protocol requires verification of the public key the user's local computer computes a hash of the public key received from the network, converts it to one of formats described above, and then compares the computed value with "public password" held by user. The user can be asked to type the correct public password, or may just be requested to approve a displayed value of the public password. Moreover, after some time some users may be able to recognize the correct value without carrying it with them.
In any case, however, it is important that a user carefully checks the validity of the displayed value. Thus, we designed a user interface that avoids the tendency of users (as mentioned in Section 1.4) to answer every question by simply pressing the Enter key. An example of such an interface based on the mapping of bits to English words as discussed in Section 2.1 follows. In this interface the user is presented with four options from which only one is correct.
The user must choose the correct value from the four values displayed.

(1) SCAN NOON DIE MAID TOTE COP
(2) SCAN TOTE NOON DIE MAID COP
(3) COP TOTE DIE SCAN MAID NOON
(4) TOTE DIE SCAN COP MAID NOON

Which is the appropriate phrase? 1..4

We stress that it is important to carefully choose the three alternative incorrect values. To illustrate this point we show two inappropriate strategies for choosing the alternative strings.

- Too much diversity – All values are very different one from other.

(1) TUM TANK TIP CUBE LID HELM
(2) SCAN TOTE NOON DIE MAID COP !
(3) BANK HANS BIN GOAT JET BEAM
(4) HIGH TUNE REID BARB BONY RAIN

In this case the regular user will remember only the first word "SCAN" and will pick as correct any string that starts with this word. So all the attacker needs to do is to find a public key (with its corresponding private key) that when hashed and mapped to this representation results in a string that starts with "SCAN" e.g. "SCAN GOAT DIE JET TANK COP". Such a search by the attacker requires no more that the generation of 2^11 public keys, and therefore the security of the system is decreased from 2^66 to 2^11.

- Too much similarity – only one word different between the correct value and each of the alternative values.

(1) SCAN BEAM NOON DIE MAID COP
(2) SCAN TOTE NOON DIE MAID COP !
(3) BANK TOTE NOON DIE MAID COP
(4) SCAN TOTE NOON JET MAID COP

In this case users who don't remember the correct value can "reconstruct" the correct choice by just looking at the alternatives. These users could easily succumb to any choice of public key by the attacker. (Thus this method would fail our attempt to defend users from their own "laziness".)

- Our Suggestion

To prevent both problems described above we implemented the following method to determine alternative strings:

- The first alternative string is created from the correct string by the permutation of two random chosen words.
- Each other alternative string is created from the previous one by the permutation of two other randomly chosen words.
- When all alternative strings are created, they are randomly placed from 1 to 4 and represented to the user as shown above.

An example of the result of this strategy are the four options:

(1) SCAN NOON DIE MAID TOTE COP
(2) SCAN TOTE NOON DIE MAID COP
(3) COP TOTE DIE SCAN MAID NOON
(4) TOTE DIE SCAN COP MAID NOON

3 Authentication Through Image

This option is probably the most "user-friendly" way to represent the pubic key to the user and we focus on it in detail in this section.

On one hand, people are very good at identifying geometrical shapes, patterns, and colors, and they can compare two images efficiently and reliably. In addition, people are extremely efficient at recognizing previously seen images.

On the other hand, this option seems the most difficult for attackers to try to foul. There are a huge number of different pictures ($2^{24*100*100}$ for a 100*100 pixel image) although many of them are almost identical. It is difficult to say how many images distinguishable by human beings exist. It is easy to show, however, that the number of easy to differentiate images is well beyond the number of 2^{64} 64-bit hash values of public keys. For example, lets look at the image containing the chosen string. A lot of other images having the same string in different fonts and colors can be created. These images correspond to the same string but can be easily distinguished by human. More than this, such strings can be written in different directions. Hence we can see that for each string there is a correspondent set of different images.
Assuming that the function converting strings to images approximates an uniform distribution in the image field, we can expect that strings will be converted to differentiable images with very high probability.
It is difficult for human to distinguish between two similar strings. When these strings are converted to different images, human can easily recognized them.

Following, there is an example of four images that can be easily remembered and recognized:

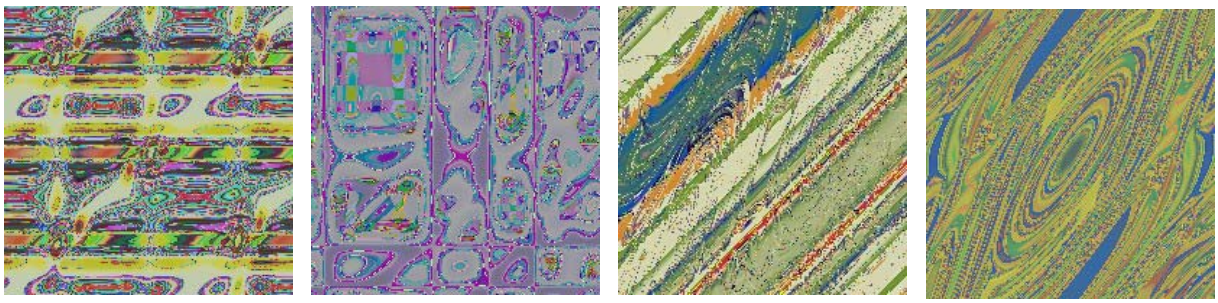


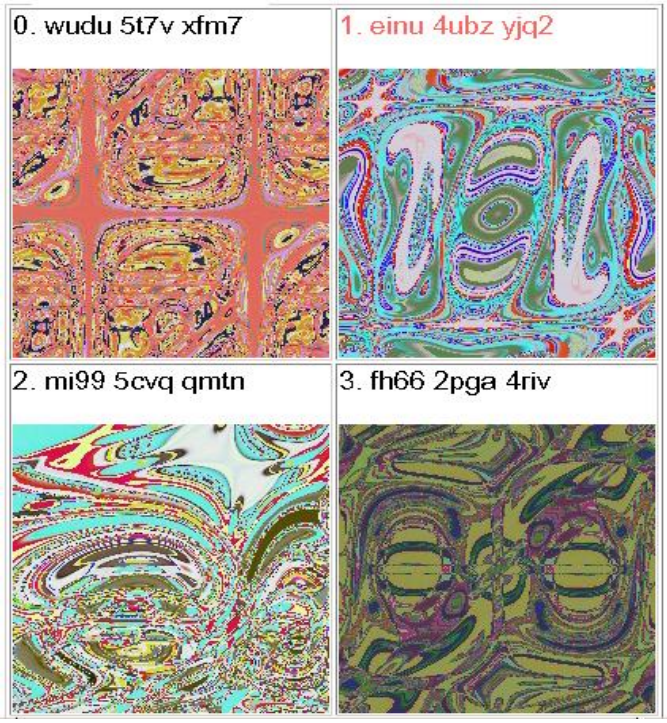
Figure 1: Regular Images

One question we should answer, when implementing this option, is what properties should the images have, in order to be easily recognizable by the user and to be difficult for the attacker to falsify.

We should know how to convert the 64-bit hashed public key into such an image and how to filter the "bad" images from the "good" images. The image creation method should distribute the output uniformly between all images. This will prevent the attacker from creating a similar image from other key. The output image should be easily recognizable by humans. We are going to check quality of the picture, eliminating "bad" images that can be hard to recognize and distinguish from other images.
This check will be done at two points.

The first check is done when the server generates a new pair of private and public keys. If the server wants to support authentication through the image, then only those keys that are mapped to a "good" image should be accepted. If the image created from this public key is not "good", the server shall discard the created pair of keys and generate a new one. After that the generated image should be distributed together with the server's public key to the clients.
The second check is done when the client side generates three alternative images to be presented to the user together with the correct image. Here the client computer will choose a random 64 bit binary string and convert it to an image. If the image is not "good", then another binary string will be randomly picked, until three "good" images will be generated.
As we will see in section **3.3**, most (about 70%) of the images created from randomly chosen 64 bit binary strings using our image creating method (see section **3.2**) are good, so the generation of three good images will require very few attempts. To speed up the generation of alternative images, the client machine could maintain a set of "good" images generated in advance. So it needs only to choose three random images from this set.

The images can be represented for user recognition with a similar user interface as described in section **2.4**:



Which is the appropriate Image?

Figure 2: Example of Authentication through Image

3.1 Image Properties

Some desirable properties that a "good" image should have are presented in [PS99]. We discuss these properties and their relevance and implementation in our context.

3.1.1 Regularity Property

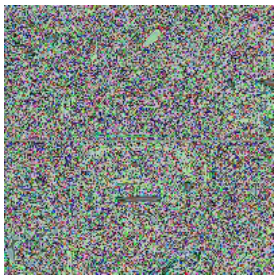
People are good at identifying geometric objects (such as circles, rectangles, triangles, and lines), and shapes in general. Images should consist mainly of recognizable shapes, called "*regular*". If an image is irregular, i.e. does not contain identifiable objects or patterns, or is too chaotic (has white noise), it is difficult for people to identify or recognize it.

[PS99] suggests using a compress algorithm for automatically testing the regularity of an image. If the image is chaotic the compression factor will be very small, since almost every pixel is random. Therefore we can say that an image is regular if the compression factor is above a certain threshold.

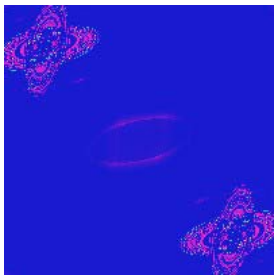
3.1.2 Sufficient Complexity Property

This property should guarantee that the image has enough complexity and can't be easily falsified by the attacker. It says that the image cannot be too simplistic in shapes and patterns, or rely on subtle color differences. Just like the regularity property, we could use compression to detect images that are too simplistic. For example, compression of an image that has all its pixels set to a unique color should result in a very high compression factor. Therefore we can say that an image is not simplistic if the compression factor is under a certain threshold.

Below we show examples of irregular and simplistic images with their compression factors:



18 % - too small compression factor ➔
Not regular image ➔
Difficult to recognize.



94% - too high compression factor ➔
Very simplified image ➔
Easy to falsify.

Figure 3: Images with compression factors out of range

3.1.3 Collision Resistance Property

This property is mandatory for using these images for certain security applications, including ours. It says that it is very difficult to find two different keys, which are represented by the same or very similar images. This is a major problem in itself, how to compare images and when we can say that the images are similar. We should keep in mind that similarity of the compared images should be checked according to properties of the human eye. For the optimal solution, the images should be passed through special filters representing the human eye, and only then should they be

compared.
In our work we implement a simple test. We compare images on a per pixel basis using the normal correlation formula shown below:

$$\mathrm{diff}[\%] = \frac{100 * \sqrt{\sum_{i=1}^{w*h} ((r'_i - r_i)^2 + (g'_i - g_i)^2 + (b'_i - b_i)^2)}}{\sqrt{\sum_{i=1}^{w*h} ((r'_i)^2 + (g'_i)^2 + (b'_i)^2)} + \sqrt{\sum_{i=1}^{w*h} ((r_i)^2 + (g_i)^2 + (b_i)^2)}}$$

Where:
w = width of image in pixels, h = height of image in pixels,
r_i, g_i, b_i – red, green and blue components of the color for pixel "**i**" in the image.
This formula calculate difference between two pictures summarize delta of each colour component for each pixel and normalize the result. So we receive difference between two pictures presented in percents.
Statistical results, received by using this function discussed in session **3.3**

3.2 Image Creation Method

The image creation method used in this work is based on the idea of "*Random Art*" described in [Bau98].

This method creates a W*H image from a 64 bit key, where W is the width of the image in pixels and H is the height of the image in pixels. The image is represented by a W*H array of long words (32 bits). Each long word represents the color of the pixel in the image in RGB format (0x00bbggrr). The least significant byte contains a value for the relative intensity of the color red, the second byte contains a value for the color green, and the third byte contains a value for the color blue. The most significant byte must be zero. The value for a single byte is in the range of 0…255. For example, the value 0x000000FF represents the color red, the value 0x00FF0000 represents the color blue, and 0x0000FF00 represents the color green.
For black and white images red, green and blue components are equal and represent the level of the color gray. For example: 0x00000000 = Black, 0x00FFFFFF = White, and 0x00808080 = Grey.
We use the Z-compress algorithm to filter out irregular and overly simplistic images as explained in 3.1.

This method is based on a set of 16 mathematical functions that convert input color (r, g, b) into output color (r', g', b').
This method is depicted in **Figure 4.**

Each 4 bits of the key define one of the functions from the set. So 64 bits of the key define the 16 functions that will be used to convert this key into an image and the order of their usage. To improve the quality (complexity and regularity) of the created images, each function from the set can't be used more than twice. In this case the next function from the set will replace it.

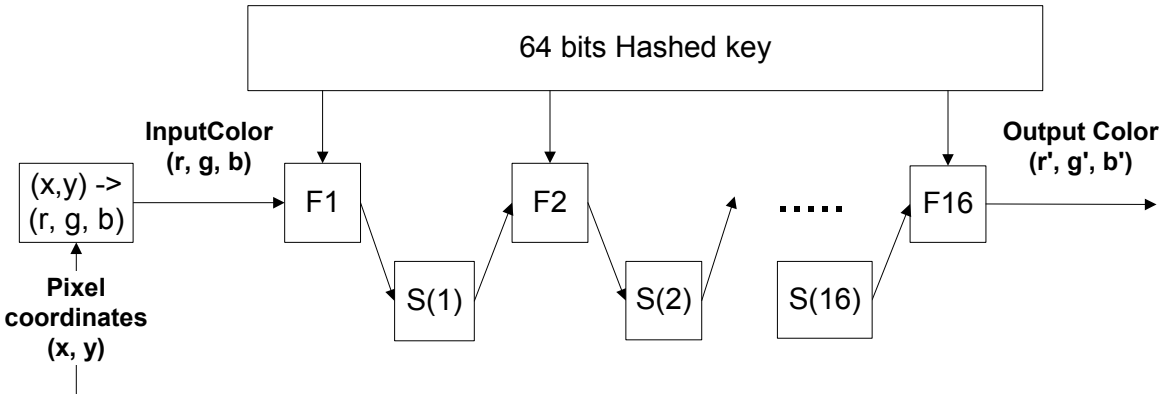
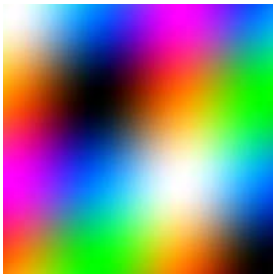


Figure 4: Image Creation Method

Each function from the set works on color values normalized to range [-1, 1], so it continuously converts input colors "c ∈ [-1, 1]" to output colors "c' ∈ [-1, 1]". Some of these functions work on each color component independently and others mix the color components. Examples of functions from the set are shown below:

{r',g',b'}=
log10(4.1 + 4*{r,g,b})

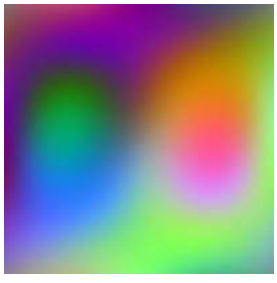
{r',g',b'}=
sin(5*{r,g,b})



{r',g',b'}=
0.8*atan(-3*{r,g,b})

Figure 5: Functions that don't mix color component.

r'=sin(3*r)*cos(2*b)
g'=sin(3*g)*cos(2*r)
b'=sin(3*b)*cos(2*g)



r'= sin(3*r)*sin(3*b)
g'= sin(3*g)*sin(3*r)
b'= sin(3*b)*sin(3*g)

r'=cos(3*r)*sin(2*b)
g'= cos(3*g)*sin(2*r)
b'= cos(3*b)*sin(2*g)

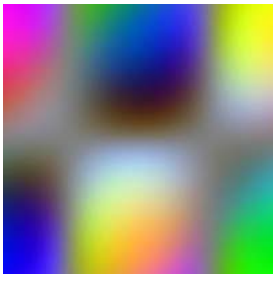


Figure 6: Functions that don't mix color component.

We found these functions empirically, based on few simple rules.

1. The function should convert input value of range [-1, 1] to output value in the same range. This allows the composition of these function one after another without any additional calculations.
2. Output value of these functions should be uniform distributed over the range [-1, 1].

3. The function should significant change input value.
4. Functions should not have commutative property, so F2(F1(x)) != F1(F2(x)).

In general, these functions should create such pictures, that most of them will be "good", i.e. have three properties described above.

The initial value of the color for each pixel in the image is defined according to the coordinates {x, y} of the pixel as: r = x, g = y, b = (x+y)/2 and normalized to a range of [-1, 1].
After that, the color of each pixel goes sequentially through all of the 16 functions defined by the 64-bit key.
To improve the collision resistance of the created images, a special shift (S1...S16) operation takes place after each function operation. The purpose is to prevent from attacker to create a similar picture, using small changes in the 64-bit key. For example, the shift operation helps to provide rule 4 described above.
This shift operation depends not on the previous function, but on the order of operation. All color components are shifted with the same value. For example, after the first function, color components will not be shifted at all. After the second function, color components will be shifted with 0.125. After the last (16^{th}) function, color components will be shifted with 1.875. So the shift value can be defined by the following formula: "shift=2*idx/16", where the *idx* is sequential number of this function execution is in the range [0…15].

3.3 Statistical Results

Now we will try to estimate if this image creation method is suitable for our purposes – image based authentication of a public key. To suit this purpose, images should have three properties mentioned above: regularity, sufficient complexity and collision resistance.
To check regularity and sufficient complexity of the image we used the Z-compress algorithm. After a lot of experiments we defined that images with a compress factor from 25% to 65% have these two properties with very high probability.

We checked 1000 randomly chosen keys to determinate how many "good" and "bad" images were created by this method.

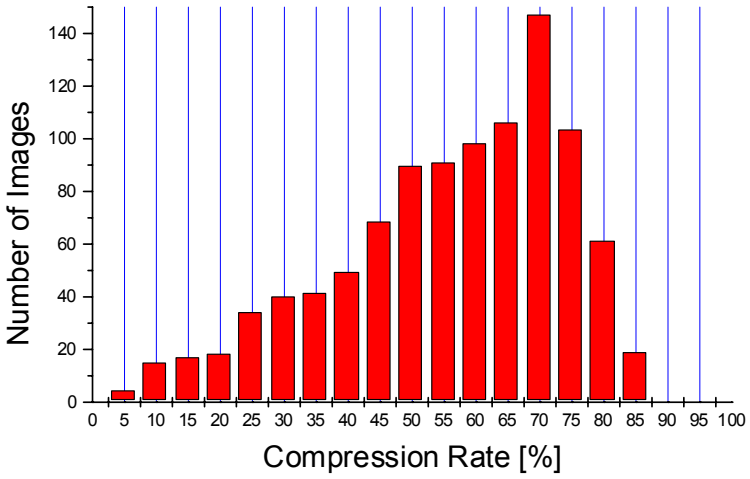


Figure 7: Compression rate distribution

As we can see in the above Figure, most images have a compression factor in the range defined above for “good” images. So in our test 700 of the 1000 images had a compress ratio in the range for “good” images.

The other test was performed to check the collision resistance of created images. One random “good” image was chosen as a reference. Another 1000 random “good” images were generated and compared with the reference image according to the normal correlation formula described above. In our test most images had ~25-40% difference from the reference image and no image had difference less than 15% from the reference image.
These tests indicate that this method of image creation can be used for secure applications.

4 SSH (Secure Shell) Integration

SSH is the protocol that provides secure network services such as telnet, rlogin, etc. over insecure infrastructure.
It consists of three major components:

- Transport layer protocol provides Server Authentication, Confidentiality and Data Integrity.
- User authentication protocol authenticates the Client side to the Server.
- Connection protocol multiplexes encrypted tunnels into several logical channels.

We choose SSH as one of the most widespread network protocols, to prove the security improvements of our public keys authentication method.

Minimal changes were made in the original SSH code to integrate our improvements in the Server authentication method.

Our modified SSH version supports four formats for representing public keys: Fingerprint (for backward compatibility), English Words, Alpha-Numeric, and Image.

The user can choose one of two types of operations:
Retype – applicable for all formats except Image. In this case the user is asked to retype the Server public key in one of the chosen formats.
Confirm – the user is asked to select the correct option from four options, as shown above.

4.1 SSH Framework

The SSH application consists of the following tools:

- ssh_keygen - tool that generate RSA/DSA public/private keys pairs
- sshd - SSH Server (No changes)
- ssh - SSH Client

First of all the public/private keys pair is generated on the Sever side by "ssh_keygen". In the original SSH version a fingerprint of the created public key is displayed. In the updated SSH version all supported formats (Fingerprint, English Words, Alpha-Numeric) are displayed.
If a specific option -v is chosen, visual format will also be displayed.
In this case only a public key that can be converted into a "good" image will be accepted. If the image created from a generated key is "bad" (compress factor is out of range), another private/public key pair will be generated until a good one is found.
The public key of the server should be published everywhere in all supported formats.

SSH server "sshd" should be started on the server side. This application works without any changes, exactly as in the original SSH.

SSH client "ssh" is started on the client side to connect to the server machine.
To create a secured channel between Server and Client, SSH uses the Diffie-Hellman key exchange algorithm. During this algorithm exchange, the Server sends its public

key to the Client, and the Client must verify the public key received from the Server. Here our changes are integrated.
The client has to answer the following questions:

- About the format of public key verification.

(1) Fingerprint
(2) EnglishWords
(3) AlphaNumeric
(4) Visual
(5) Abort
Choose Key format for verification?

- About type of action

(1) Confirm
(2) Retype
(3) Abort
Choose Action for Host verification?

In addition, the choice of above options can be entered via the command line.

For further detailed information about SSH changes see Readme file in Appendix.

5 References

[HK99] Shai Halevi, Hugo Krawczyk. Public cryptography and password protocols. ACM Transactions on Information and System Security, 1999.

[PS99] Adrian Perrig, Dawn Song. Hash Visualization: A New Technique to improve Real-World Security. In Proceedings of the 1999 International Workshop on Cryptographic Techniques and E-Commerce, (CryTEC `99), 1999.

[DP00] Rachna Dhamija, Adrian Perrig. Déjà vu: A User Study. Using Images for Authentication. In the proceedings of the 9th USENIX Security Symposium, 2000.

[Bau98] Andrej Bauer. Gallery of random art. WWW at *http://andrej.com/art/*, 1998.

[Hal95] Neil Haller. The S/KEY One-Time Password System. RFC1760, 1995.

[DH76] W. Diffie, M.E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, IT-22:644–654, 1976.

[SSH-TRANS] Ylonen, T., "SSH Transport Layer Protocol", Internet Draft, draft-ietf-transport-14.txt, July 2001.

6 Appendix

This document describes the changes implemented in the original SSH code to improve the server authentication mechanism.

I. INTRODUCTION:

Server authentication implemented in the current version of SSH uses the following mechanism:

The following flags are relevant for this issue, <> - default:

1. options.strict_host_key_checking ("stricthostkeychecking")
   No (0) - No host authentication (always accept).
   Yes (1) - Use only locally saved host key. (If it is not exist than reject).
   Ask <2> - If locally saved host key is not exist, ask user to confirm received host key (in fingerprint format).

If connection is accepted, the received host key is saved locally in "user_hostfile" or "system_hostfile".

When using default option <2>, the algorithm will work in the following way:

- If Host is known   - Accept connection
- If Host is changed - Check IP address
- If Host is unknown - Ask user to confirm fingerprint (128 bits of HEX string)of received host key.

II. CHANGES:

This project introduces some new features that improve the host authentication mechanism. The new features and changes described below:

- Always ask user to confirm or retype received host key. (Don't check host key saved locally).

- Control for saving received and accepted host key.

- Allow the user to confirm or retype received host key in different formats: English words, Alpha-Numeric, Hexadecimal (Fingerprint) and Visual.

The following changes were implemented in the SSH options :

1. Add new cases for options.strict_host_key_checking ("stricthostkeychecking"):

      confirm (3) - Don't check host key saved locally. (Always ask user to confirm.)
      retype (4) - Don't check host key saved locally. (Always ask user to retype.)

2. Add new option: options.save_host_key ("savehostkey")
   This option allows users to control saving of received and accepted host key.
    Ask <0>   - Ask user about "save"/"don't save". [yes/no]
      Yes (1)   - If host key is accepted, save it in "user_hostfile".
       No (2)   - Don't save received host key.

3. Add new option: options.host_key_format ("hostkeyformat")
   This option allows the user to choose the format of host key
   display to confirm or retype:
       ask   <0> - Ask user about it on-line.
     finger   (1) - Fingerprint format (without changes).
     english (2) - English-Words format (rfc1760).
     alphaNum (3) - Alpha-Numeric format.
     visual   (4) - Visual format (color picture).

III. SUPPORTED FORMATS:

A. English words format: (e.g. TUM TANK TIP CUBE LID HELM)

The host key is converted to a string of English words according to the following algorithm:
1. Pass N-bits of host public key through one-way hash function MD5 and receive binary string of 128 bits size (16 bytes).
2. Convert 16 bytes binary string to 12 English words of 2 to 4 letters each, as described in RFC1760.
3. Display first 6 English words to represent received host key.

B. Alpha-Numeric format: (e.g. qu24 ih3q sswb)

The host key is converted to an Alpha-Numeric string according to the following algorithm:
1. Pass N-bits of host public key through one-way hash function MD5 and receive binary string of 128 bits size (16 bytes).
2. Convert 16 bytes binary string to Alpha-Numeric words of 4 letters each. Each 5 bits of binary(0-31) are converted to 24 letters (a..z exclude l, o) and to 8 digits (2..9).
3. Display first 3 words to represent received host key.

C. Visual format:

The host key is converted to a color picture according to the following algorithm:
1. Pass N-bits of host public key through one-way hash function MD5 and receive binary string of 128 bits size (16 bytes).
2. First 64 bits of the binary string defines 16 mathematical functions from a predefined set (4 bits define one function) and order of their execution. Each function converts input RGB color {r, g, b} into output RGB color {r', g', b').
3. Each pixel of the image gets its initial color according to coordinates {x, y} of the pixel as: r=x, g=y, b=(x+y)/2.
4. After that each pixel goes sequentially through all 16 functions defined by the 64 bits of the binary string. The color received after the 16th function execution will be the color of this pixel in the created image. To improve collision resistance of created images, a special shift operation takes place after each function operation. This shift operation doesn't depend on the previous function operation, but only depends on the order of this operation.

   Only "good" i.e. regular and not simplistic images can be used for public key authentication in a secure application. The Z-compress algorithm is used to detect "good" images and filter "bad" images. To do so, the image is presented as a height*width array of long words, where each long word represents the color of a pixel in RGB format (0x00bbggrr), compressed by the Z-compress library. After that, the compress factor is calculated as:
(1-(compressed_size/(height*width*4))*100 [%])

If compress factor is too small, image is irregular.
If compress factor is too large, image is too simplistic.

Compress factor range for "good" images is defined in the config.h file.

IV. USER VERIFICATION INTERFACE:

If the "confirm" value of "stricthostkeychecking" option is chosen, the user should confirm that the appropriate string or picture is displayed. To avoid a situation where the user answers every question by pressing the Enter key, the special user interface is used:

Each time when the user should confirm the received host key, four different options are displayed. The appropriate one is placed in a random place within the range 1 through 4. The wrong options are similar to the correct one, so the user must choose carefully.
Note: For visual format the wrong pictures are chosen randomly.

For example, following are four Host key options in English word format:

(1) SCAN NOON DIE MAID TOTE COP
(2) SCAN TOTE NOON DIE MAID COP
(3) COP TOTE DIE SCAN MAID NOON
(4) TOTE DIE SCAN COP MAID NOON

There are different methods for choosing these alternative strings. In this project we used the permutation function that randomly replaces four of six English words or two of three Alpha-Numeric words. Other methods to choose alternate strings can be easily adopted.

For Visual format three "good" images are generated. Each image is created from a randomly chosen 64 bit binary string. Netscape is used to show all four images to the user. The user must choose the correct image and type the number of the selected image.

The following changes were implemented in the key generation process (ssh-keygen):

When a new private-public key pair is generated, all supported formats (Fingerprint, English Words, Alpha-Numeric) will be displayed.
The Color image is generated and displayed only if the special option -v is selected by user. In this case only the public/private key pair, that can be used to create a "good" image will be accepted. Otherwise another pair will be generated. In this SSH version, a temporary JPG file is generated to display visual format to the user and Netscape is used to display this file.

For example: => ssh-keygen -b 512

```
512 bits, dimae@tochna12
Key Fingerprint    : 42:9c:f1:03:b6:82:49:2d:36:e9:0c:b2:95:4d:23:8d
Key English Words  : TUM TANK TIP CUBE LID HELM
Key Alpha Numeric  : qu24 ih2q sswb
```

The following changes were implemented in the connection procedure (ssh):

1. Skip checking of received host key with locally saved one, if option stricthostkeychecking = confirm | retype.

2. Ask a user to confirm or retype received host key in one of display formats as mentioned above.

V. FILES:

The following files were changed:

Makefile	- Added new files "Format\wordput.c" and "Format\wordconvert.c".
key.c	- Added a new function "key_digest(..)" to generate MD5 digest of the key.
readconf.c, readconf.h	- Added new options as described above.
sshconnect.c	- Changed "check_host_key(..)" function to work with new options. For more details see Framework session below.
ssh-keygen.c	- After key generation, display all possible formats of new created key: Fingerprint, English Words, Alpha-Numeric. All these key formats are displayed when -y (print_public) or -l (print_fingerprint) options are used.

The following files were added:

zlib\	- Library that implements Z-compress algorithm.
jpeg-6b\	- Library to read and write JPEG file format.
Format\wordput.h Format\wordput.c	- Contains English words table (2048 words) defined in rfc1760 and functions to convert binary string to English and Alpha-Numeric words.
Format\config.h	- Configuration parameters for all supported formats can be easily changed:

#define ALT_PHRASE	4	- Number of different options displayed from which user can select.
#define ENGLISH_WORDS	6	- Number of 2-4 letter English words displayed for key in English Word format.
#define ALPHA_NUM_WORDS	3	- Number of 4 symbol words displayed for key in Alpha-Numeric format.
#define ENGLISH_WORDS_SWAP	4	- Number of words would be randomly shuffled to generate alternative strings for 6 words of English words format.
#define ALPHA_NUM_WORDS_SWAP	2	- Number of words would be randomly shuffled to generate

```
                                        alternative strings for 3 words
                                        of Alpha-Numeric format.

    #define MAX_RGB_COMPRESS_FACTOR   25  - Minimal compress factor
                                            for "good" RGB image.

    #define MIN_RGB_COMPRESS_FACTOR   65  - Maximal compress factor
                                            for "good" RGB image.

    #define MAX_BW_COMPRESS_FACTOR    30  - Maximal compress factor
                                            for "good" GREY image.

    #define MIN_BW_COMPRESS_FACTOR    80  - Minimal compress factor
                                           for "good" GREY image.

    #define KEY_LENGTH                64  - Hashed Key length used to
                                            create image.

    #define IMAGE_WIDTH               100 - Width of image will be
                                            generated.

    #define IMAGE_HEIGHT              100 - Height of image will be
                                            generated.
```

Format\keyvisual.h
Format\keyvisual.c — These files implement the visual representation of a binary string. The set of 16 mathematical functions in "FuncArray" defines the image creation method.

The main function implemented in this file *"buildJpegFile()"* builds JPEG file contained image created from 64 bits of hashed public key. If generated image is out of "good" compress factor range value 1 is returned and file will not be generated. If value 0 is returned "good" image is generated and valid JPEG file is created.

Format\keyformat.h
Format\keyformat.c — These files implement public functions and should be used by the original SHH code to improve server authentication.

Function "int user_check_host_key(Key* pKey, int format, int action)" called during server authentication process. It asks the user to confirm or retype received host public key in one of the formats described above according to "hostkeyformat" and "stricthostkeychecking" options respectively.
If "format == 0 (ask)" the function asks the user about desirable display format.
If "action == 2 (ask)" the function asks the user about desirable action ("confirm" or "retype").

VI. OTHER FORMATS:

It is simple enough to add new format to be supported. For each supported format, a set of three functions should be implemented:

- int getKey<english/alphaNum/other>Format(Key *pKey,
                                          char** text_buf);

This function converts host key to text string ready for display in desirable format.

- int get<english/alphaNum/other>Alt(char *str, int n, char **alt);
  This function generates array of size "n" of strings alternative to appropriate string "str" in desirable format.

- int readKey<english/alphaNum/other>Format(char* text_buf, int size)
  This function reads from console string accordingly with desirable format and check validity of the string.

To add the new key representation format to be supported, the following changes should be made:
1. Add new value for host_key_format ("hostkeyformat") option in readconf.c file.
2. Implement three functions described above.
3. Add new case in switch statement, implemented in function user_check_host_key().

VII. FRAMEWORK:

Server side will first generate pair of public-private RSA/DSA keys using "ssh-keygen" program. When keys are generated, strings generated from the public key accordingly with all supported formats will be displayed.

Current version supports the following formats:
- Fingerprint (128 bits HEX format),
- English words (6 words of 2-4 letters English words),
- Alpha-Numeric (3 words of 4 letters).
- Visual (100*100 pixels picture)

All these formats are based on at least 64 bits and can be considered secure enough.

Visual format will be generated and displayed only when user chooses special option -v. If created image is not "good" enough (too simplified or not regular) the other pair of keys will be generated while "good" image is not generated.

Note: To print previously generated key in all these formats use "ssh-keygen" program with options: -y (print_public) or -l (print_fingerprint) or -v (generate and display image)

Client should know at least one of these formats before connection setup will be started.

To achieve maximum benefit from the new improved server authentication mechanism, the user on the client side should run the "ssh" program with the following options:

- stricthostkeychecking = confirm|retype - Always ask user to confirm or retype host key.

- hostkeyformat = ask - Ask user for host key display format on-line.

- savehostkey = no - Don't save accepted host key locally.

When client side receives host key during Diffie-Helman key exchange algorithm the user is asked about host key representation format:

(1) Fingerprint
(2) EnglishWords

(3) AlphaNumeric
(4) Visual
(5) Abort

Choose Key format for verification?

After that the user is asked about type of verification action Retype or Confirm (no Retype for Visual format):

(1) Confirm
(2) Retype
(3) Abort

Choose Action for Host verification?

If the Confirm option is chosen, four different (but similar) options are displayed. The user should choose the appropriate one. For example:

For English words format:

(1) SCAN NOON DIE MAID TOTE COP
(2) SCAN TOTE NOON DIE MAID COP
(3) COP TOTE DIE SCAN MAID NOON
(4) TOTE DIE SCAN COP MAID NOON

Which is the appropriate phrase?

For Alpha-Numeric format:

(1) 4786 8fqh hnpb
(2) hnpb 4786 8fqh
(3) hnpb 8fqh 4786
(4) 8fqh hnpb 4786

Which is the appropriate phrase?

For Fingerprint format (without changes):

RSA/DSA key fingerprint is:
f1:0a:52:5f:e7:25:c7:98:6e:19:6d:8d:5c:c9:e0:a2.

Are you sure you want to continue connecting (yes/no)?

If the user chooses the appropriate option, the host key is accepted and the connection process will continue.

If the "Retype" option is chosen, the user is asked to retype the string presented user host key in the chosen format:

For English words format:

Please type the Host key in English format (6 words): W1 W2...W6
->

For Alpha-Numeric format:

Please type the Host key in AlphaNum format (3 words): w1 w2 w3
->

For Fingerprint format:

Please type fingerprint of the Host key (16 bytes): X1:X2:...:X16
->

If the option "savehostkey" == "ask", the user will be asked the following question:

Do you want to save the Host Key permanently in the list of known hosts: (yes/no)?

If the option value is "yes", the Host key will be saved automatically.
If the option value is "no", the Host key will not be saved.

All other functionality of the SSH code remains unchanged.