Assignment 0x01

Frank Kaiser – 1742945, Jan Martin – 1796943

November 19, 2017

Contents

1 Task 1: Account Recovery

1.1 Most important recommendations for the account recovery process

General

- Don't store passwords in plain text, but checksums/hashes.
- Balance ease of use with challenge.
- Regularly revalidate information
- Minimum time between successful login and recovery

Email Authentication

- Focus on recovering the account - not all features
- Don't send their old password, but a new, randomly generated password. Or a token
- Notify real account holder over all channels available.

Knowledge-Based

- Don't use/allow guessed information
- Factor in changes over time / differences between cultures.
- Changes should require at least as much authority as recovery.

Social

- Don't spam or people won't take part anymore.
- Promising, but on the far end of the effort/securty curve.

Attacker-Driven Analysis of Account Recovery

- placeholder

Multi-Channel Authentication

- Is is an actual channel?

1.2 Example: dotasource.de

Figure 1: ssh-keygen

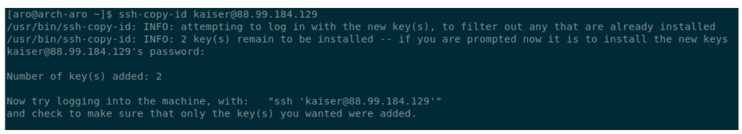

Figure 2: save public key on server

2 Task 2: Public-Key Authentication in SSH

2.1 GNU/Linux

To generate the ssh-key pair I used the command `ssh-keygen` which generates by default a 2048 bit long rsa key.

To copy the key on the server `ssh-copy-id user@88.99.184.129` was used. Since I already had a public key for another server and the exercise was to create one, both keys got uploaded to the server as seen in the picture 2.

The following log-in worked without the user password for the server. Only the optional password for the private ssh key was required. The keys on the server are stored in `~/.ssh/authorized_keys`. See figure 3

Figure 3: location of keys on server

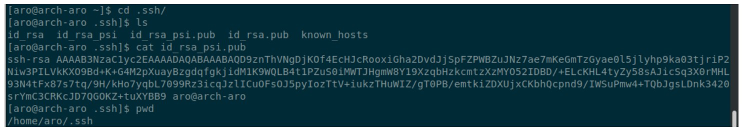

Figure 4: location of keys on local machine

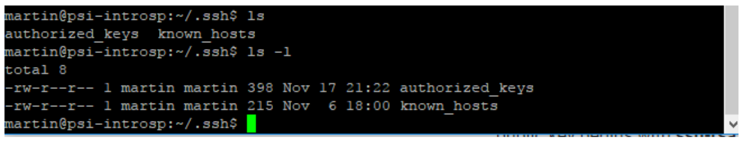

Figure 5: permission check

The ssh-keys on your local machine are by default stored in `~/.ssh/id_rsa.pub` for the public part and the private one in `~/.ssh/id_rsa`. (Provided you did create a rsa key) See figure 4.

2.2 Windows

On a windows machine the same tools from openSSH were not available. That's why the procedure was a little bit different. Here PuTTY was used.

To generate the key the tool `putty-gen` was used. Then we logged into the server via ssh and created the file `authorized_keys` in `~/.ssh/` and pasted the key into it using nano. Lastly, we checked that only we have write access to the file by `ls -l`

In figure 6 you can see the successful login by using the ssh-key pair.

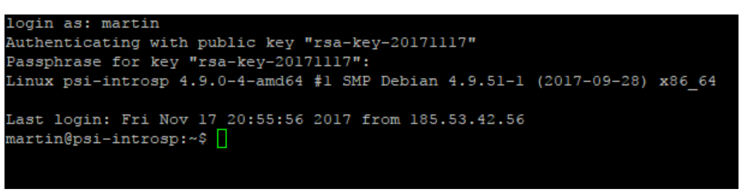

Figure 6: windows ssh-key login

3 Task 3: Buffers in C: The Vigenre Cipher

The source code for the exercise can be found in the following file: Vigenere Cipher

To compile: gcc -Wall vigenere_cipher.c -o "output-name" Optional flag: -DDEBUG

Below is the source code readable:

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int getRotation(char c) {
  /*According to ascii 'A' transfers to 65
  and Z to 90. By subtracting 65 of the char we get
  the rotation. */
  return c - 65;
}

int main(int argc, char *argv[]) {
  char key[256];
  char word[256];

  printf("Type in the key\n");
  fgets(key, sizeof(key), stdin);

  printf("What do you want to encrypt?\n");
  fgets(word, sizeof(word), stdin);

  // Number of char that were not uppercase
  int cntSkipped = 0;
  // length of the key - 1 to know when to start from 0
  int keyLength = strlen(key) - 1; // remove \n
  int keyPosition = 0;             // index of the key word

  for (int i = 0; i < strlen(word); i++) {
              /* Set keyPosition to 0 when end of key is reached
              i is substracted by the number of skipped chars
              so the % operator works as intended */
    if ((i > 0) & ((i - cntSkipped) % keyLength == 0)) {
      keyPosition = 0;
    }
    // ignore lowercases and whitespaces
    if (!isupper(word[i])) {
      cntSkipped++;
      continue;
    }
```

```
    int rotation = getRotation(key[keyPosition]);
    keyPosition++;

#ifdef DEBUG
    printf("%i ", rotation);
#endif

    word[i] = ((word[i] - 'A' + rotation) % 26) + 'A';
  }
  printf("%s", word);

  return 0;
}
```

4 Task 4: Memory Management on the Stack

To get the Success! message you have to do a buffer overflow by giving more than 16 characters as input string. After that you can put input chars on the stack. You can set two variables at once, because memory was allocated for variables 'myvalue' and 'mystring'. The stack is first in, last out, i.e. the first element put into it, will be the last one read. Because of this you have to put in the chars in reverse order. The resulting command looks like this:

```
echo -e "0000000000000000\xef\xbe\xad\xde" | ./memory
```

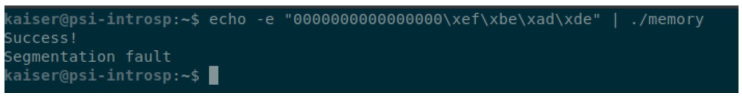

Figure 7: Successful input