

Assignment 0x01

Frank Kaiser – 1742945, Jan Martin – 1796943

November 19, 2017

Contents

1	Task 1: Account Recovery	2
1.1	Most important recommendations for the account recovery process	2
1.2	Example: dotasource.de	2
2	Task 2: Public-Key Authentication in SSH	5
2.1	GNU/Linux	5
2.2	Windows	6
3	Task 3: Buffers in C: The Vigenre Cipher	7
4	Task 4: Memory Management on the Stack	9

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.
- Balance ease of use with challenge.
- Regularly revalidate information like current email address, phone numbers etc.

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 allow easily guessed information.
- Plan for cultural differences.
- Changes should require at least as much authority as recovery.

Social

- Don't spam or otherwise alienate the users social contacts required for information.

Multi-Channel Authentication

- Is only useful if independent, so sending a SMS to a phone that is used to recover the account is not helpful.

1.2 Example: dotasource.de

As test object I used the web site www.dotasource.de.

This is what the Lost Password page looks like:

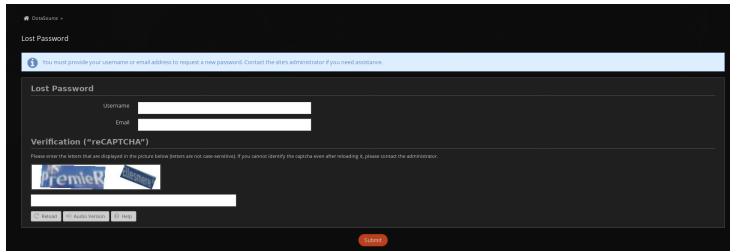


Figure 1: Lost Password

When entered a non-existent username or E-mail address the page responds that the user does not exist or no account with that e-mail is registered.

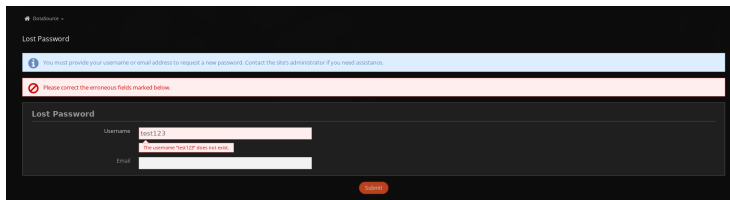


Figure 2: User does not exist

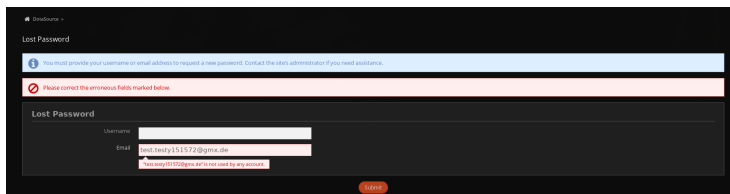


Figure 3: E-mail not used by an account

When entering correct information an email is sent to the account's address, containing a link for recovery.

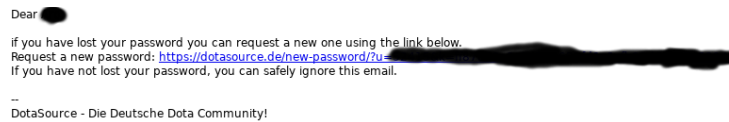


Figure 4: Reset Link

When clicked on the link another email with a new, random generated password in plain text is sent. Sessions that are still logged in don't lock you out.

But for a new login the password is in immediate effect. To change the password again, you have to login using the given password and visit the account management site.

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.

```
aro@arch-aro ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/aro/.ssh/id_rsa): /home/aro/.ssh/id_rsa_psi
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/aro/.ssh/id_rsa_psi.
Your public key has been saved in /home/aro/.ssh/id_rsa_psi.pub.
The key fingerprint is:
SHA256:E/pB4YMB3MhgF0a0LrcTliMzLw23/T/IMFPctS9sgDo aro@arch-aro
The key's randomart image is:
+---[RSA 2048]-----+
| .o+.+. . |
| .. B .+ . . |
| = .. =o . . |
|   o .o+ o   |
|   . . . S = . |
| * O E. O+ = . |
| % * .. B .   |
| . * . . . . |
| . . . . . . |
+---[SHA256]-----+
```

Figure 5: ssh-keygen

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 6.

```
aro@arch-aro ~]$ ssh-copy-id kaiser@88.99.184.129
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 2 key(s) remain to be installed -- if you are prompted now it is to install the new keys
kaiser@88.99.184.129's password:

Number of key(s) added: 2

Now try logging into the machine, with: "ssh 'kaiser@88.99.184.129'"
and check to make sure that only the key(s) you wanted were added.
```

Figure 6: save public key on server

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 7

```
aro@arch-aro ~]$ ssh kaiser@88.99.184.129
linux psi-introsp 4.9.0-4-amd64 #1 SMP Debian 4.9.51-1 (2017-09-28) x86_64

Last login: Fri Nov 17 20:32:43 2017 from 188.194.245.11
kaiser@psi-introsp:~$ cd .ssh/
kaiser@psi-introsp:~$ ssh ls
authorized_keys known hosts
kaiser@psi-introsp:~$ ssh cat authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCOuXv87Yg14grvXefISyBpILx2XV5BRJ46fVpGrKcNsria0Z3HAK5V0JUSMAxTE0JIIImy8y0lVOT
0HkrKsPer+26t4KKUHQge7n00a-gld8f0CvW4say9G0DX/0Eed22/T8TD4Log4Ta13St+q0vml8fauTgmKv9Cul1/Lhr2UF1Moc1utq8s3NwNTH
e+77Q0Phn0Csaye4+BCUWTF59waFgB55106wsgRe5lKMsro5ImaGusf24Ml f6Ea5oz/w75s6F2rtkDh703YI1yh6b+auMvrt0DnExDq+IPVlEDtpKaa
+8QSTECTKHvbulUnZEqsGLxLlHN+3TLThvvgMN08PaSkSRVze9HjMVF3eVyp8V2LYnqD/Tv73KUSZvZc327TFN56lKc693sVHFYKtCKT2mc9l+SLN/9uau
UDFN5xLqTata29GQVKKAKZDX0mKUBSqdM/LH8NorwyW7u3L4XchZu57FhcwktfJlCtLl4mCwmN+Xr/80PffDyZpn/wnu0XdrRogUSog6y1f337bxxB
6s+JZ1UUDF29HqElyBPp0lcur7C500eXqWfYwVXbwa4deGdeu4d4XoaZpc9Wq1a/s1Na0bW9Mj0dPj7B9s+45VEghl05Eo+H09MFXA1Eb661a0HT
3bVjCyl+u0pde0= frank-phillip.kessler@stud.uni-bamberg.de
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQ9znThVng0jK0f4eChJCRoox1Gha2DvdJjSpfZPwBzUjN27ae7Kc0mTzGyae015jlyhp9ka03tjr1P2
Niw3PLVKKX09Bd+K+G4M2pxuayBzgdqfgkjdM1K9WOLB4t1PZuS0lMWTJHgm8Y19XzqbHkcmTzxMY052IDBD/+ELKHL4tyZyS8sAJ1csq3X0rPHL
9JNAtfX877Tq/9H/H07yqBL7099Rz31cQJ2l1Cu0F5035py1ozTTV+1ukZThwI2/gT0PB/entk1ZDXUjXCKbh0cpnd9/1WSuPmw4+T0bJgsLdnk3420
ryvC3CRKc3D70G0KZ4tUyYB9 aro@arch-aro
kaiser@psi-introsp:~$ ssh ls
```

Figure 7: location of keys on server

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 8.

```
[aro@arch-aro ~]$ cd .ssh/
[aro@arch-aro .ssh]$ ls
id_rsa  id_rsa.pub  id_rsa.pub  id_rsa.pub  known_hosts
[aro@arch-aro .ssh]$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDAQ92nTHVNg0jK0f4eCHJCRoox1Gha2DvdJjSpFZPWBZuJNz7ae7mKeOmTzGyae0l5jlyhp9ka83tjr1P2
NlW3PILVX0098d+K+G4H2pXuy8zgdfgkjl0M1K9WQLB4t1PZuS0lMWTJHgm8Y19XzqbHkcmTzXZY052IDBD/+ELckH4tyZyS8SA1c5q3X8rPHL
93Mitx8Z7tq9W/km07qkL7899Rz2icd2l1lc0F503py1ozTtV+1ukzTHuW1Z/gT0PB/entk1Z0XUjxCKBhQcpnd9/1W5uPmw4+T0bJgsLbnk3428
srymC3CRKc1D70GOKZ+tuXY8B9 aro@arch-aro
[aro@arch-aro .ssh]$ pwd
/home/aro/.ssh
```

Figure 8: location of keys on local machine

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`

```
martin@psi-introsp:~/.ssh$ ls
authorized_keys  known_hosts
martin@psi-introsp:~/.ssh$ ls -l
total 8
-rw-r--r-- 1 martin martin 398 Nov 17 21:22 authorized_keys
-rw-r--r-- 1 martin martin 215 Nov  6 18:00 known_hosts
martin@psi-introsp:~/.ssh$
```

Figure 9: permission check

In figure 10 you can see the successful login by using the ssh-key pair.

```
login as: martin
Authenticating with public key "rsa-key-20171117"
Passphrase for key "rsa-key-20171117":
Linux psi-introsp 4.9.0-4-amd64 #1 SMP Debian 4.9.51-1 (2017-09-28) x86_64

Last login: Fri Nov 17 20:55:56 2017 from 185.53.42.56
martin@psi-introsp:~$
```

Figure 10: 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:

```
#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 subtracted 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);

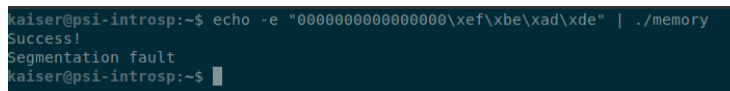
    return 0;
}

```


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
```

A terminal window with a dark background. The prompt is 'kaiser@psi-introsp:~\$'. The command entered is 'echo -e "0000000000000000\xef\xbe\xad\xde" | ./memory'. The output shows 'Success!' on the first line, 'Segmentation fault' on the second line, and the prompt 'kaiser@psi-introsp:~\$' on the third line.

```
kaiser@psi-introsp:~$ echo -e "0000000000000000\xef\xbe\xad\xde" | ./memory
Success!
Segmentation fault
kaiser@psi-introsp:~$
```

Figure 11: Successful input