EVIL

Evil Corp. Penetration Test

by Justus Uurtimo

Start of testing: March 30, 2022

End of testing: April 6, 2022

Contents

1 Executive Summary

In this penetration test the Evil company was examined for security-relevant weaknesses. The kind of testing was Gray-box, this is the kind where some specific information about the internals of the system is given. The company provided different assets for the assignments. I completed the assignments and created this report based on my findings.

These assignments were:

- Assignment 1: Password cracking with John the Ripper and Hashcat
  - For this assignment a few files were provided. These files contained hashes that were suspected of containing weak passwords, I managed to crack them using JohnTheRipper and hashcat combained with password dictionary. View 3.1 for more information.
- Assignment 2: RSA key-cracking
  - RSA-public key was provided. It was suspeced that this key might be prone to cracking, due to its small modulus. I managed to crack the key fairly easily. Please refer 3.2 for more information.
- Assignment 3: RSA file-message decryption
  - This assignment was a continuation of the previous assignment and the factories acquired from that assignment was used to re-create the private key and decrypt the encrypted file. Please refer 3.3 for more information.
- Assignment 4: RSA file-message forged signature
  - In this assignment a file was provided that was to be signed with the private key that was forged in the previous exercise. After that I verified the signature with the original public key to make sure it was valid. Please refer 3.4 for more information.
- Assignment 5: DSA key-cracking
  - In this assignment, it was suspected that two files were signed with the same ECDSA private key (which is ok), but a nonce was reused. From these signed

files I managed to recreate the private key. Plese refer 3.5 for more information.

- Assignment 6: DSA file-message forged signature
  - This assignment was a continuation of the previous assignment. In this assignment I forged a signature with the private key that I recovered in the previous assignment. Please refer 3.6 for more information.
- Assignment 7: Simple MD5 collision
  - In this assignment I collided files and created two slightly different files that have the same MD5 hash value. Please refer 3.7 for more information.
- Assignment 8: MD5-collission 'weaponized'
  - This is a continuation assignment of the previous assignment. In this assignment I utilize MD5-collision to create to python files that will have the same MD5-hash value but that will behave very differently. Please refer 3.8 for more information.
- Assignment 9: DSA file-message decryption
  - In this assignment I was tasked to decrypt a message using a DSA private key that should be aquired from the assignment 5. However I believe this is impossible for more information on why please refer 3.9.

2 Vulnerability overview

Table 2.1 depicts all assignments done for the penetration test report. They are categorized in the order of completion. Risks are assessed with the possible impact and how easy they are to take advantage off. Please note that the assignments listed here are the unique weaknesses found. Some assignments are continuations of previous ones and they are not listed here since they are in the report to show what implications could occur if any of these vulnerabilities are taken advantage of.

Risks	Asset	Vulnerability	Section	Page
High	Assignment 1	Weak hashes	3.1	5
High	Assignment 2	RSA-key with very small modulus value	3.2	10
High	Assignment 5	Nonce re-use	3.5	17
High	Assignment 8	MD5-Collision	3.7	22

Table 2.1: Vulnerability overview

3 Results

In this chapter, the vulnerabilities found during the penetration test are presented. All the issues are in the order of solving them. All contain the following information

- Brief description.
- CVSS Base Score when it is applicable here for details.
- Business impact.
- References to classifications, such as CWE
- Steps to reproduce.

Also the remediation recommendations are given for each issue found during the penetration test.

3.1 Password cracking with Hashcat and John the Ripper

3.1.1 General information

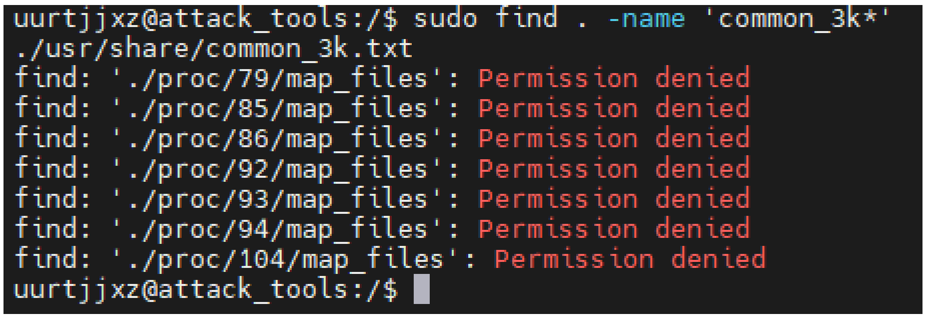
Password cracking is a process where the attacker will try to gain access to unknown or forgotten passwords via applications or programs. There are a few different ways to go about the password cracking. The attacker can utilize dictionary attacks, in which the attacker will utilize a pre-determined lists that can have been gathered from previous data breaches, dictionaries, common-password-lists etc. The attacker will then try to compare these words to the hashed password and try and find a match. The attacker could also use rainbow table - attack, where the attacker has a list of most common passwords and their hashed version, making finding the correct password much faster. There are also rainbow-tables that have all possible passwords for a certain lengths and their hashes.

In this case I am using John the Ripper and Hashcat to crack the passwords. John the ripper is a free, open-source, command-based application, that supports massive list of different cipher and hash types. It is one of the most frequently used password testing and breaking programs (concise 2017). Hashcat is a program that positions itself as the world's fastest password cracker. Hashcat is also free and open-source and it offers techniques from bruteforce attacks to hybrid mask with wordlists. What makes the tool truly universal is the fact that it supports over 300 hash types (Jancis 2022).

CWEs of this vulnerability are CWE-916: Use of Password Hash With Insufficient Computational Effort, since the passwords were easily cracked and CWE-521: Weak Password Requirements since the passwords found were quite weak and possessed small amount of entropy.

3.1.2 Proof of concept

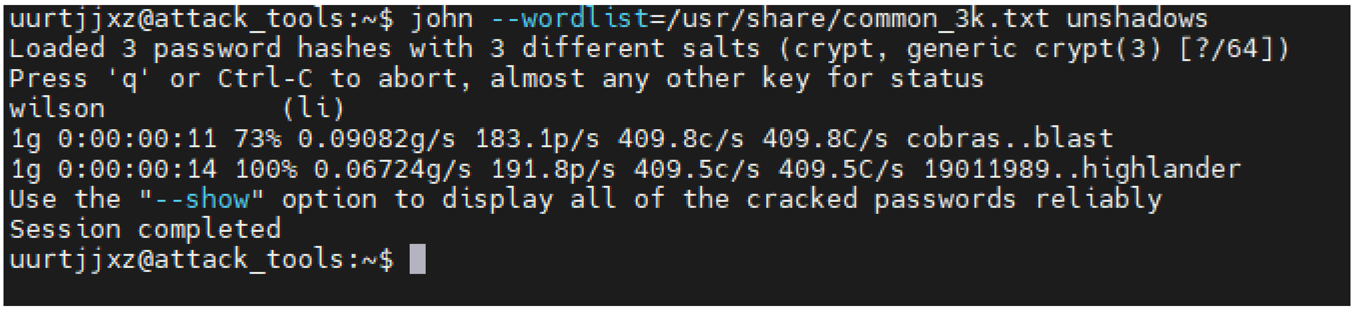
First I need to find the password dictionary. I used the command at the root directory "/" :
sudo find . -name ’common_3k*’



```
uurtjjxz@attack_tools:/$ sudo find . -name 'common_3k*'
./usr/share/common_3k.txt
find: './proc/79/map_files': Permission denied
find: './proc/85/map_files': Permission denied
find: './proc/86/map_files': Permission denied
find: './proc/92/map_files': Permission denied
find: './proc/93/map_files': Permission denied
find: './proc/94/map_files': Permission denied
find: './proc/104/map_files': Permission denied
uurtjjxz@attack_tools:/$
```

Figure 3.1: Find the passlist

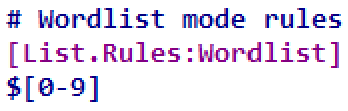
As can be seen from the image 3.1 the list is located at
./usr/share/common_3k.txt
It seems that I dont have permissions to use that file in that directory,
so I copied it to my directory with command:
sudo cp ./usr/share/common_3k.txt ./home/uurtjjxz/common_3k.txt
Next I combine the passwd and shadow files with command:
unshadow /etc/passwd.kybs2004 /etc/shadow.kybs2004 > unshadows
After this its time to use the John the ripper. First I run the ripper normally and then with some modified rules.
For the first run I used the command:
john –wordlist=/usr/share/common_3k.txt unshadows
With this command the ripper found the password "wilson" that belongs to user "li", as can be seen from figure 3.2



```
uurtjjxz@attack_tools:~$ john --wordlist=/usr/share/common_3k.txt unshadows
Loaded 3 password hashes with 3 different salts (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
wilson           (li)
1g 0:00:00:11 73% 0.09082g/s 183.1p/s 409.8c/s 409.8C/s cobras..blast
1g 0:00:00:14 100% 0.06724g/s 191.8p/s 409.5c/s 409.5C/s 19011989..highlander
Use the "--show" option to display all of the cracked passwords reliably
Session completed
uurtjjxz@attack_tools:~$
```

Figure 3.2: First ripper run

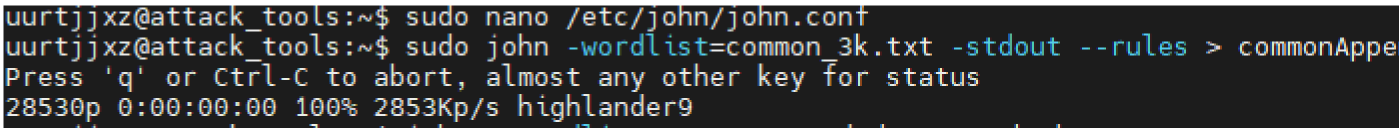
Next I need to modify the rules a bit. I quickly found out that simply creating my own rules would not work here as stated in (jfoug 2016). This means that I would need to modify the List.Rules:Worldlist, or simply create my own set with the same name. For this time I opted on creating my own. I did this by commenting out the original header of List.Rules:Worldlist and then simply writing it again at the top with only my single rule. This seemed to be an easier approach since my rule would be so short.



```
# Wordlist mode rules
[List.Rules:Wordlist]
$[0-9]
```

Figure 3.3: New Ripper rule

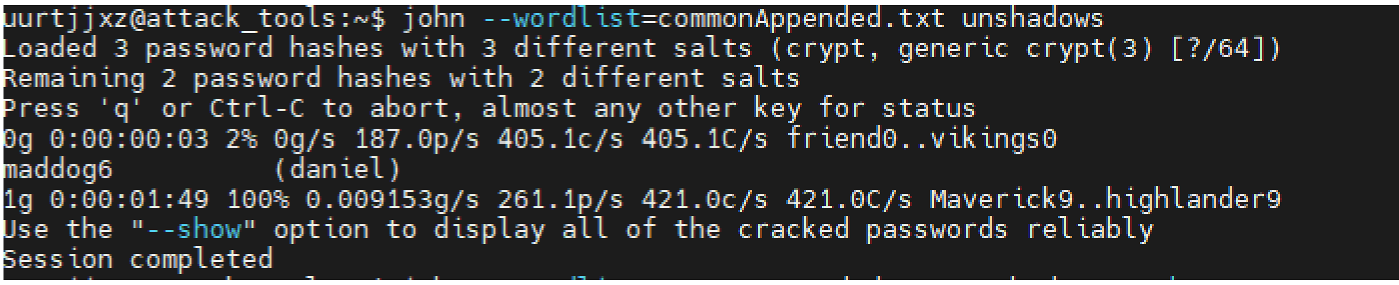
After this I created a new wordlist with the new rule. For this, I used the command:
sudo john -wordlist=common_3k.txt -stdout –rules > commonAppended.txt



```
uurtjjxz@attack_tools:~$ sudo nano /etc/john/john.conf
uurtjjxz@attack_tools:~$ sudo john -wordlist=common_3k.txt -stdout --rules > commonAppe
Press 'q' or Ctrl-C to abort, almost any other key for status
28530p 0:00:00:00 100% 2853Kp/s highlander9
```

Figure 3.4: New wordlist from rules

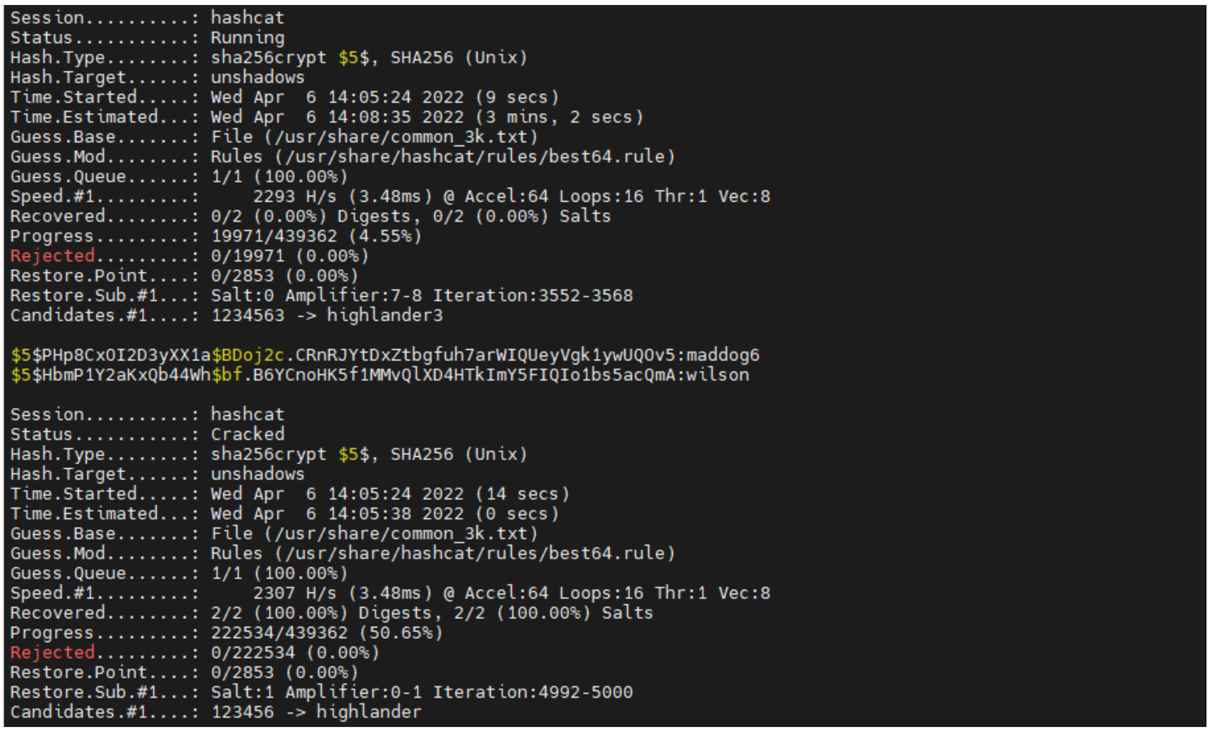
After this I ran the ripper again, but this time with the new wordlist.
john –wordlist=commonAppended.txt unshadows
And as can be seen from the figure 3.5, another password and account combination can be found with this.



```
uurtjjxz@attack_tools:~$ john --wordlist=commonAppended.txt unshadows
Loaded 3 password hashes with 3 different salts (crypt, generic crypt(3) [?/64])
Remaining 2 password hashes with 2 different salts
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:03 2% 0g/s 187.0p/s 405.1c/s 405.1C/s friend0..vikings0
maddog6          (daniel)
1g 0:00:01:49 100% 0.009153g/s 261.1p/s 421.0c/s 421.0C/s Maverick9..highlander9
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Figure 3.5: Second ripper run

Next it was time to crack the passwords using hashcat. I ran the hashcat with the following command:
hashcat -m 7400 -a 0 -r /usr/share/hashcat/rules/best64.rule unshadows common_3k.txt



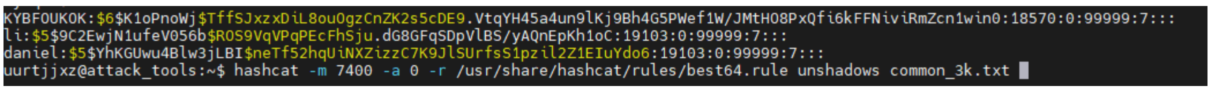
```
Session..........: hashcat
Status...........: Running
Hash.Type........: sha256crypt $5$, SHA256 (Unix)
Hash.Target......: unshadows
Time.Started.....: Wed Apr  6 14:05:24 2022 (9 secs)
Time.Estimated...: Wed Apr  6 14:08:35 2022 (3 mins, 2 secs)
Guess.Base.......: File (/usr/share/common_3k.txt)
Guess.Mod........: Rules (/usr/share/hashcat/rules/best64.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     2293 H/s (3.48ms) @ Accel:64 Loops:16 Thr:1 Vec:8
Recovered........: 0/2 (0.00%) Digests, 0/2 (0.00%) Salts
Progress.........: 19971/439362 (4.55%)
Rejected.........: 0/19971 (0.00%)
Restore.Point....: 0/2853 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:7-8 Iteration:3552-3568
Candidates.#1....: 1234563 -> highlander3

$5$PHp8CxOI2D3yXX1a$BDoj2c.CRnRJYtDxZtbgfuh7arWIQUeyVgk1ywUQOv5:maddog6
$5$HbmP1Y2aKxQb44Wh$bf.B6YCnoHK5f1MMvQlXD4HTkImY5FIQIo1bs5acQmA:wilson

Session..........: hashcat
Status...........: Cracked
Hash.Type........: sha256crypt $5$, SHA256 (Unix)
Hash.Target......: unshadows
Time.Started.....: Wed Apr  6 14:05:24 2022 (14 secs)
Time.Estimated...: Wed Apr  6 14:05:38 2022 (0 secs)
Guess.Base.......: File (/usr/share/common_3k.txt)
Guess.Mod........: Rules (/usr/share/hashcat/rules/best64.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:     2307 H/s (3.48ms) @ Accel:64 Loops:16 Thr:1 Vec:8
Recovered........: 2/2 (100.00%) Digests, 2/2 (100.00%) Salts
Progress.........: 222534/439362 (50.65%)
Rejected.........: 0/222534 (0.00%)
Restore.Point....: 0/2853 (0.00%)
Restore.Sub.#1...: Salt:1 Amplifier:0-1 Iteration:4992-5000
Candidates.#1....: 123456 -> highlander
```

Figure 3.6: Hashcat Run

As it can be seen from the image the hashcat was successful in obtaining the passwords but does not seem to show the corresponding account name for that password. Although since I am in possession of the original hash I could just compare the cracked password hashes to the hashes in files so connect the password to the correct account. This is because the account names are not hashed in the files as can be seen from the image 3.7



```
KYBFOUKOK:$6$K1oPnoWj$TffSJxzxDiL8ou0gzCnZK2s5cDE9.VtqYH45a4un9lKj9Bh4G5PWef1W/JMtHO8PxQfi6kFFNiviRmZcn1win0:18570:0:99999:7:::
li:$5$9C2EwjN1ufeV056b$ROS9VqVPqPEcFhSju.dG8GFqSDpVlBS/yAQnEpKh1oC:19103:0:99999:7:::
daniel:$5$YhKGUwu4Blw3jLBI$neTf52hqUiNXZizzC7K9JlSUrfsS1pzil2Z1EIuYdo6:19103:0:99999:7:::
uurtjjxz@attack_tools:~$ hashcat -m 7400 -a 0 -r /usr/share/hashcat/rules/best64.rule unshadows common_3k.txt
```

Figure 3.7: Original hash

3.1.3 Proposed solutions

For this issue the easiest solution would be to implement some basic rules for passwords. These rules can include that the password must contain at least certain amount of characters and it cant contain multiable same characters in row. Also advising the users that the password should not be found in any dictionaries. Also it is worth noting that common number substitions of certain letters, such as using "4" instead of "A" or "a" " does not increase the security of the password. The more entropy the better.

The salt used for the hashes should be larger. Even with 128-bit salt, two users with same passwords would have different hashes, which would make the cracking take much longer and generating tables for all salts would take astronomical amount of time (Jancis 2022).

The service should also use key stretching, which would make even weaker passwords more secure and much more time consuming for the attacker to crack. One way of conducting key stretching is to require repeated hashing so that it is usable for legitimate user but very time consuming for malicious actor (Cook 2019).

3.2 RSA key-crack

3.2.1 General information

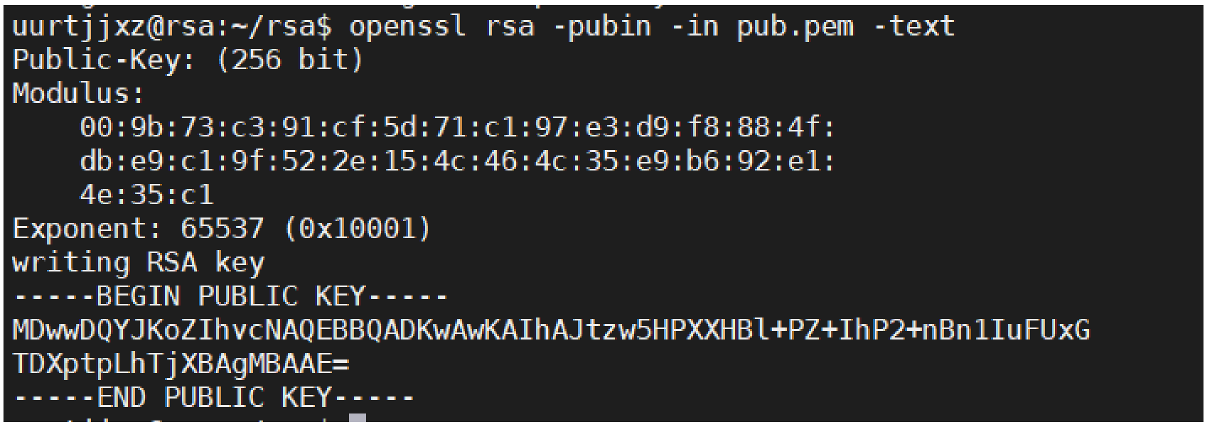
RSA is based on multiplication of two prime numbers, often marked as p and q, to give the modulus value, which is often marked as n. If the modulus is cracked, the attacker is able to crack the decryption key. This means that the attacker would be able to create their own private key and decrypt messages (Doyle 2019).

The CWE categorization for this issue is: CWE-326: Inadequate Encryption Strength. This is because RSA is a valid method for encryption but the key used here had a very weak modulus length.

Business impact: Essentially anyone with the private key of the owner can impersonate the owner and decrypt private data, gain unauthorized access to systems or even generate fake signatures that will appear authentic to others. I will demonstrate the forging of new private key from recovered primes in 3.3 and forging of the signature with said forged private key in 3.4.

3.2.2 Proof of concept

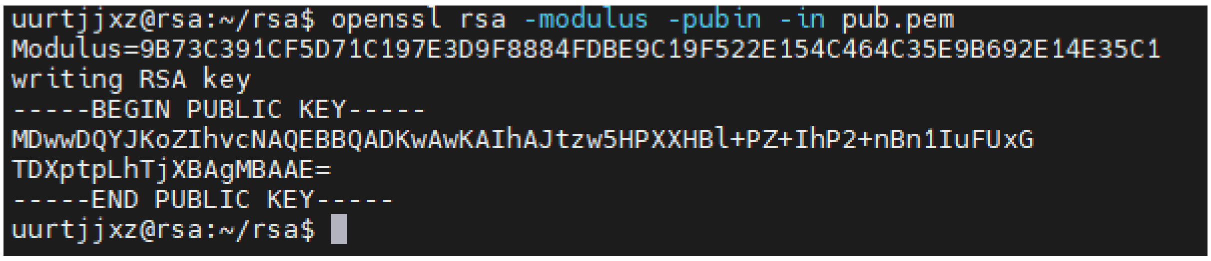
First I check information from the key. For this i used the command:
openssl rsa -pubin -in pub.pem -text
This will take in a public key and print the key in text. As can be seen from the image below the key is 256 bit public key.



```
uurtjjxz@rsa:~/rsa$ openssl rsa -pubin -in pub.pem -text
Public-Key: (256 bit)
Modulus:
    00:9b:73:c3:91:cf:5d:71:c1:97:e3:d9:f8:88:4f:
    db:e9:c1:9f:52:2e:15:4c:46:4c:35:e9:b6:92:e1:
    4e:35:c1
Exponent: 65537 (0x10001)
writing RSA key
-----BEGIN PUBLIC KEY-----
MDwwDQYJKoZIhvcNAQEBBQADKwAwKAIhAJtzw5HPXXHBl+PZ+IhP2+nBn1IuFUxG
TDXptpLhTjXBAgMBAAE=
-----END PUBLIC KEY-----
```

Figure 3.8: RSA key information

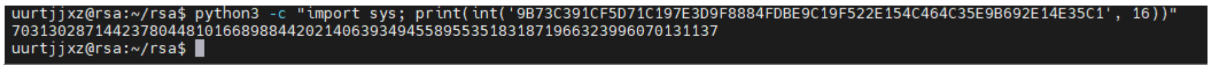
Then I start the exercise by getting the modulus. I will use the openssl rsa for this and used the following command to achieve this:
openssl rsa -modulus -pubin -in pub.pem
In this command:
"-modulus" will print the RSA key modulus.
"-pubin" = will tell the operation to expect a public key as input file.
"-in" will set the input file as parameter, in our case it is pub.pem.



```
uurtjjxz@rsa:~/rsa$ openssl rsa -modulus -pubin -in pub.pem
Modulus=9B73C391CF5D71C197E3D9F8884FDBE9C19F522E154C464C35E9B692E14E35C1
writing RSA key
-----BEGIN PUBLIC KEY-----
MDwwDQYJKoZIhvcNAQEBBQADKwAwKAIhAJtzw5HPXXHBl+PZ+IhP2+nBn1IuFUxG
TDXptpLhTjXBAgMBAAE=
-----END PUBLIC KEY-----
uurtjjxz@rsa:~/rsa$
```

Figure 3.9: RSA modulus

Next, I will change the hex value to base 10 number. For this I will use the following command:
python3 -c "import sys; print(int('9B73C391CF5D71C1
97E3D9F8884FDBE9C19F522E154C464C35E9B692E14E35C1', 16))"
(I had to put a line break in the middle, since it otherwise got clipped out, but the command should be in one line)
This gives us the base 10 number, which is as follows:
base 10 of hex =
703130287144237804481016689884420214063934945589553518318719663
23996070131137

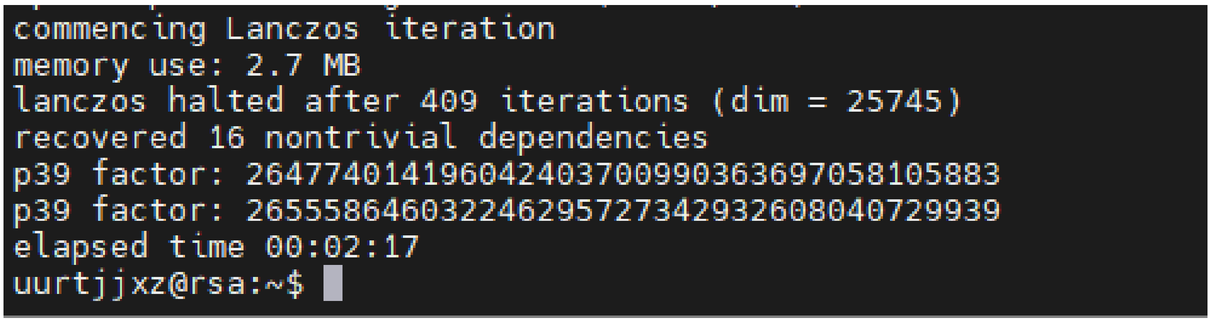


```
uurtjjxz@rsa:~/rsa$ python3 -c "import sys; print(int('9B73C391CF5D71C197E3D9F8884FDBE9C19F522E154C464C35E9B692E14E35C1', 16))"
70313028714423780448101668988442021406393494558955351831871966323996070131137
uurtjjxz@rsa:~/rsa$
```

Figure 3.10: Hex to base 10

After this I can take the two factors using the following command: *msieve -v 70313028714423780*
448101668988442021406393494558955351831871966323996070131137
(I had to put a line break in the middle, since it otherwise got clipped out, but the command should be in one line)

The result of this was:
p39 factor: 264774014196042403700990363697058105883
p39 factor: 265558646032246295727342932608040729939



```
commencing Lanczos iteration
memory use: 2.7 MB
lanczos halted after 409 iterations (dim = 25745)
recovered 16 nontrivial dependencies
p39 factor: 264774014196042403700990363697058105883
p39 factor: 265558646032246295727342932608040729939
elapsed time 00:02:17
uurtjjxz@rsa:~$
```

Figure 3.11: Sieve result

3.2.3 Proposed solutions

The main reason that this key could be cracked in such short amount of time is that the key modulus was very small (256-bits). Given that RSA-100 that has 100 decimal digits (330 bits) has been factored in 1991 and this key had even smaller modulus. So the first step would be to use RSA with much bigger modulus. For example to create RSA private key with 1024-bit modulus I could write:
openssl genrsa -out test2.pem 1024
This is equivalent to RSA-309 and that has not been cracked yet.

Also it is worth noting that a big issue with RSA-keys is that they often rely on random number generators for determine the prime numbers used (p and q). But the random rumber generators are often not that random and when many people use the same generators there are going to be overlaps on primes used on the public keys. A good example of this was when in 2012 6 million public keys were collected from the internet and researchers managed to crack 12,934 of them (Lenstra et al. 2012). These were all RSA-keys that shared "random" prime numbers with other keys.

3.3 RSA file-message decryption

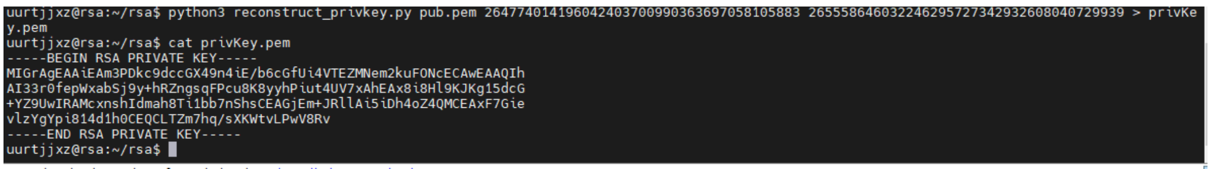
3.3.1 General information

When RSA-key is cracked and the attacker manages to extract the two prime factors (p and q), the attacker is then able to re-construct the private key used for the message encryption. When private key is reconstructed the attacker is able to read any messages encrypted by the actual owner of the original private key.

Business impact: The business impact of this issue is related to 3.2 as this is basically just a demonstration what the attacker could do with the acquired primes.

3.3.2 Proof of concept

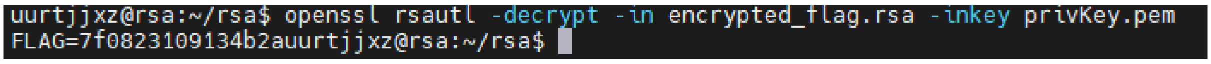
First I will re-construct the private key and save it in a file named privKey.pem
The command used for this is:
python3 reconstruct_privkey.py pub.pem
264774014196042403700990363697058105883265
558646032246295727342932608040729939 > privKey.pem



```
uurtjjxz@rsa:~/rsa$ python3 reconstruct_privkey.py pub.pem 264774014196042403700990363697058105883 265558646032246295727342932608040729939 > privKe
y.pem
uurtjjxz@rsa:~/rsa$ cat privKey.pem
-----BEGIN RSA PRIVATE KEY-----
MIGrAgEAAiEAm3PDkc9dccGX49n4iE/b6cGfUi4VTEZMNem2kuFONcECAwEAAQIh
AI33r0fepWxabSj9y+hRZngsqFPcu8K8yyhPiut4UV7xAhEAx8i8Hl9KJKg15dcG
+YZ9UwIRAMcxnshIdmah8Ti1bb7nShsCEAGjEm+JRllAi5iDh4oZ4QMCEAxF7Gie
vlzYgYpi814d1h0CEQCLTZm7hq/sXKWtvLPwV8Rv
-----END RSA PRIVATE KEY-----
uurtjjxz@rsa:~/rsa$ 
```

Figure 3.12: re-construct the private key

Here the pub.pem is the public key.
264774014196042403700990363697058105883 and
265558646032246295727342932608040729939 are the P and Q prime factors computed from the MODULUS of the public key.
After this I will decrypt the file with the command:
openssl rsautl -decrypt -in encrypted_flag.rsa -inkey privKey.pem
Here the -decrypt indicates that I am decryption. -in takes in the input file, which in our case is the encrypted file. -inkey takes in the private key that we just reconstructed.
This gives me the flag: 7f0823109134b2a which I returned to this exercise.



```
uurtjjxz@rsa:~/rsa$ openssl rsautl -decrypt -in encrypted_flag.rsa -inkey privKey.pem
FLAG=7f0823109134b2auurtjjxz@rsa:~/rsa$
```

Figure 3.13: Get the flag

3.3.3 Proposed solutions

Since the forging of malicious private key is quite trivial matter after the original key has been cracked there are no solutions that only apply to this issue. As such the solutions are the same as the ones found in key-cracking 3.2.

3.4 RSA file-forged signature

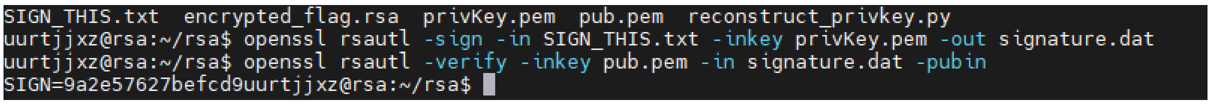
3.4.1 General information

If the attacker is able to reconstruct the private key, s/he is then able to also forge signatures and thus appear as the original owner of the private key. When the attacker is in control of the forged key forging a signature is very trivial task.

Business impact: The business impact of this issue is related to 3.2 as this is basically just a demonstration what the attacker could do with the acquired primes.

3.4.2 Proof of concept

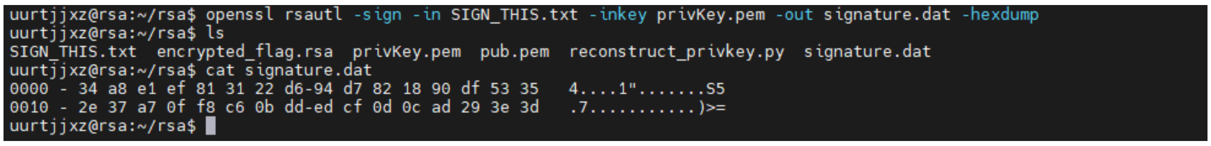
NB! I used the private key that I made in the last exercise
So first I do the signature and for that I used the command:
openssl rsautl -sign -in SIGN_THIS.txt -inkey privKey.pem -out signature.dat
After this I will verify the signature. I do this with:
openssl rsautl -verify -inkey pub.pem -in signature.dat -pubin
This gives us a result that matches the contents of the file SIGN_THIS.txt which indicates that the signature was successful.



```
SIGN_THIS.txt  encrypted_flag.rsa  privKey.pem  pub.pem  reconstruct_privkey.py
uurtjjxz@rsa:~/rsa$ openssl rsautl -sign -in SIGN_THIS.txt -inkey privKey.pem -out signature.dat
uurtjjxz@rsa:~/rsa$ openssl rsautl -verify -inkey pub.pem -in signature.dat -pubin
SIGN=9a2e57627befcd9uurtjjxz@rsa:~/rsa$ 
```

Figure 3.14: Sign and verify

Next I will create the same thing but instead use hexdump to get the hexadecimal format of the signature. This was done with this command:
openssl rsautl -sign -in SIGN_THIS.txt -inkey privKey.pem -out signature.dat -hexdump
Whole hexdump:
0000 - 34 a8 e1 ef 81 31 22 d6-94 d7 82 18 90 df 53 35 4....1".......S5
0010 - 2e 37 a7 0f f8 c6 0b dd-ed cf 0d 0c ad 29 3e 3d .7...........)>=
Hexdump that was returned for the exercise:
34 a8 e1 ef 81 31 22 d6-94 d7 82 18 90 df 53 35 2e 37 a7 0f f8 c6 0b dd-ed cf 0d 0c ad 29 3e 3d



```
uurtjjxz@rsa:~/rsa$ openssl rsautl -sign -in SIGN_THIS.txt -inkey privKey.pem -out signature.dat -hexdump
uurtjjxz@rsa:~/rsa$ ls
SIGN_THIS.txt  encrypted_flag.rsa  privKey.pem  pub.pem  reconstruct_privkey.py  signature.dat
uurtjjxz@rsa:~/rsa$ cat signature.dat
0000 - 34 a8 e1 ef 81 31 22 d6-94 d7 82 18 90 df 53 35   4....1".......S5
0010 - 2e 37 a7 0f f8 c6 0b dd-ed cf 0d 0c ad 29 3e 3d   .7..........)>=
uurtjjxz@rsa:~/rsa$ 
```

Figure 3.15: Signature Hexdump

3.4.3 Proposed solutions

Since the forging of the signature when a private key is forged, is quite trivial matter and as such are no solutions that only apply to this issue. As such the solutions are the same as the ones found in key-cracking 3.2.

3.5 DSA key-cracking

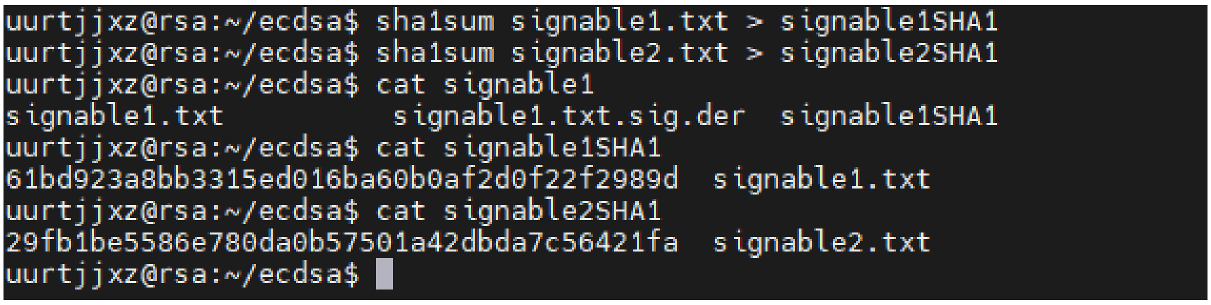
3.5.1 General information

In this exercise two files have been signed with the same ECDSA (Elliptic Curve Digital Signature Algorithim). ECDSA is a specific form of DSA. DSA is a common digital signature scheme, that is defined by three algorithims: key generation, signing and verifying. However in this case the nonce is reused, which will allow the recovery of the original private key from the signed files. If the attacker manages to recreate the private key s/he will be able to forge signature and appear as the owner of the original private key.

CWE of this issue is: CWE-323: Reusing a Nonce, Key Pair in Encryption

3.5.2 Proof of concept

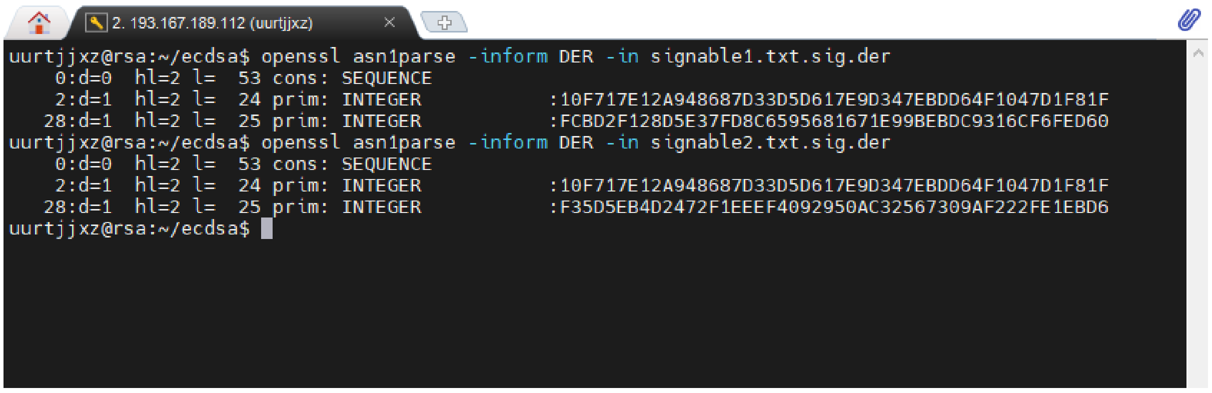
First I need the SHA1 hashes from the two sign-able files. I did this with the following commands:
sha1sum signable1.txt > signable1SHA1 and
sha1sum signable2.txt > signable2SHA1



```
uurtjjxz@rsa:~/ecdsa$ sha1sum signable1.txt > signable1SHA1
uurtjjxz@rsa:~/ecdsa$ sha1sum signable2.txt > signable2SHA1
uurtjjxz@rsa:~/ecdsa$ cat signable1
signable1.txt          signable1.txt.sig.der  signable1SHA1
uurtjjxz@rsa:~/ecdsa$ cat signable1SHA1
61bd923a8bb3315ed016ba60b0af2d0f22f2989d  signable1.txt
uurtjjxz@rsa:~/ecdsa$ cat signable2SHA1
29fb1be5586e780da0b57501a42dbda7c56421fa  signable2.txt
uurtjjxz@rsa:~/ecdsa$
```

Figure 3.16: SHA1 hashes

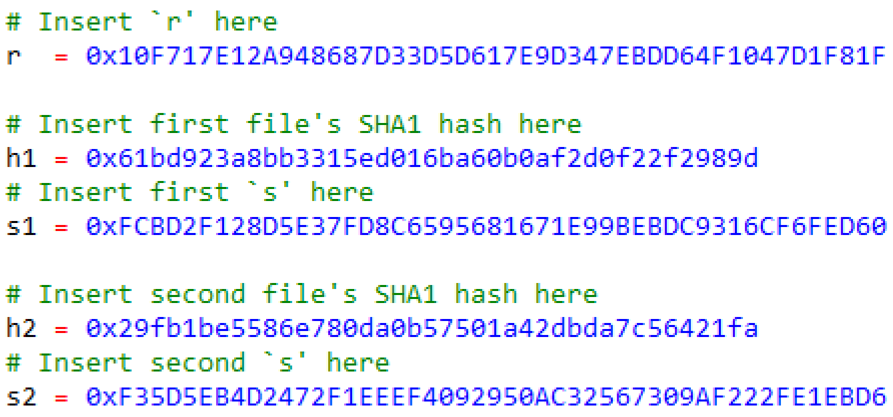
Next I needed to get the primes from the two files. I used the following commands for this:
openssl asn1parse -inform DER -in signable1.txt.sig.der
openssl asn1parse -inform DER -in signable2.txt.sig.der



```
uurtjjxz@rsa:~/ecdsa$ openssl asn1parse -inform DER -in signable1.txt.sig.der
    0:d=0  hl=2 l=  53 cons: SEQUENCE
    2:d=1  hl=2 l=  24 prim: INTEGER           :10F717E12A948687D33D5D617E9D347EBDD64F1047D1F81F
   28:d=1  hl=2 l=  25 prim: INTEGER           :FCBD2F128D5E37FD8C6595681671E99BEBDC9316CF6FED60
uurtjjxz@rsa:~/ecdsa$ openssl asn1parse -inform DER -in signable2.txt.sig.der
    0:d=0  hl=2 l=  53 cons: SEQUENCE
    2:d=1  hl=2 l=  24 prim: INTEGER           :10F717E12A948687D33D5D617E9D347EBDD64F1047D1F81F
   28:d=1  hl=2 l=  25 prim: INTEGER           :F35D5EB4D2472F1EEEF4092950AC32567309AF222FE1EBD6
uurtjjxz@rsa:~/ecdsa$
```

Figure 3.17: Primes

As it can be seen from the image 3.17, I get hexacii number that is common in both commands. This is the *r* that is the result of nonce reuse. The different hexacii numbers are the primes.
With this information I can now modify and run the ecdsa_recover.py and get the nonce and the exponent. This also produces the DSA private key that is used in later exercises. After the modification the values become as follows:
r = 0x10F717E12A948687D33D5D617E9D347EBDD64F1047D1F81F
h1 = 0x61bd923a8bb3315ed016ba60b0af2d0f22f2989d
s1 = 0xFCBD2F128D5E37FD8C6595681671E99BEBDC9316CF6FED60
h2 = 0x29fb1be5586e780da0b57501a42dbda7c56421fa
s2 = 0xF35D5EB4D2472F1EEEF4092950AC32567309AF222FE1EBD6



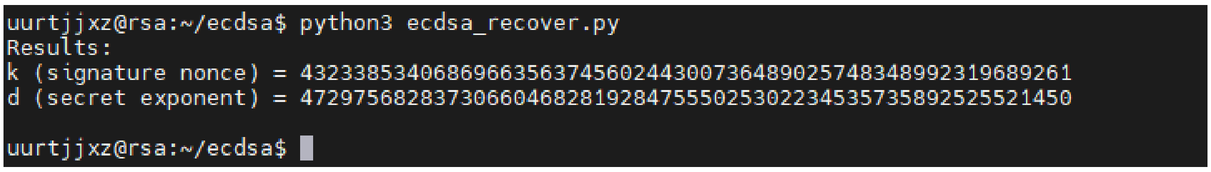
```
# Insert `r' here
r   = 0x10F717E12A948687D33D5D617E9D347EBDD64F1047D1F81F

# Insert first file's SHA1 hash here
h1  = 0x61bd923a8bb3315ed016ba60b0af2d0f22f2989d
# Insert first `s' here
s1  = 0xFCBD2F128D5E37FD8C6595681671E99BEBDC9316CF6FED60

# Insert second file's SHA1 hash here
h2  = 0x29fb1be5586e780da0b57501a42dbda7c56421fa
# Insert second `s' here
s2  = 0xF35D5EB4D2472F1EEEF4092950AC32567309AF222FE1EBD6
```

Figure 3.18: Values after modifying

After this I can simply run the python script with command:
python3 ecdsa_recover.py
Which produces the results of:
k= 4323385340686966356374560244300736489025748348992319689261
d= 4729756828373066046828192847555025302234535735892525521450



```
uurtjjxz@rsa:~/ecdsa$ python3 ecdsa_recover.py
Results:
k (signature nonce) = 4323385340686966356374560244300736489025748348992319689261
d (secret exponent) = 4729756828373066046828192847555025302234535735892525521450

uurtjjxz@rsa:~/ecdsa$
```

Figure 3.19: Results

3.5.3 Proposed solutions

The most obivious solution for this issue is to deter from reusing the nonce and choosing the nonce unpredictably. Also it is utmost importance that the ECDSA is implemented correctly. A good example of bad implementation is CVE-2022-21449, where the bad implementation would allow an attacker to access protected data with empty (full of zeroes) signature. It is also worth noting that storing the nonce anywhere is a huge security risk, as should it leak it would make the ECDSA completely useless.

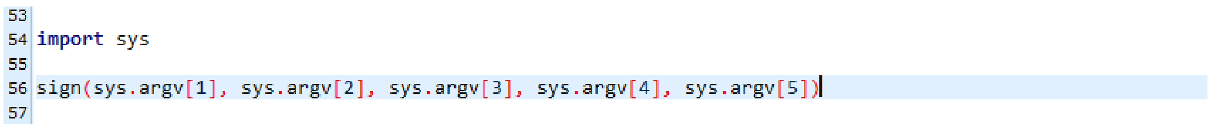
3.6 DSA file-message forged

3.6.1 General information

As mentioned in the section before 3.5, should the private key fall into the wrong hands it would enable the attacker to forge signatures and thus appear as the original owner of the private key. This could, for example, allow the attacker to pass malicious software as legitimate one and access data that should otherwise be protected.

3.6.2 Proof of concept

First I need to modify the ecdsa_sign.py. With quick Googling around it can be determined that the added rows are as follows:
import sys
sign(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4], sys.argv[5])



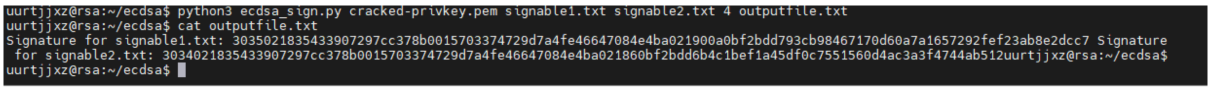
```
import sys

sign(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4], sys.argv[5])
```

Figure 3.20: last Lines

Next I need to run the python script, and as stated in the exercise I simply need to fill in the correct parameters:
python3 ecdsa_sign.py cracked-privkey.pem signable1.txt signable2.txt 4 outputfile.txt
After this the output file is created the following signatures can be obtained:
Signature for signable1.txt:
3035021835433907297cc378b0015703374729d7a4fe46647084
e4ba021900a0bf2bdd793cb98467170d60a7a1657292fef23ab8e2dcc7

Signature for signable2.txt:
3034021835433907297cc378b0015703374729d7a4fe46647084
e4ba021860bf2bdd6b4c1bef1a45df0c7551560d4ac3a3f4744ab512



```
uurtjjxz@rsa:~/ecdsa$ python3 ecdsa_sign.py cracked-privkey.pem signable1.txt signable2.txt 4 outputfile.txt
uurtjjxz@rsa:~/ecdsa$ cat outputfile.txt
Signature for signable1.txt: 3035021835433907297cc378b0015703374729d7a4fe46647084e4ba021900a0bf2bdd793cb98467170d60a7a1657292fef23ab8e2dcc7 Signature
 for signable2.txt: 3034021835433907297cc378b0015703374729d7a4fe46647084e4ba021860bf2bdd6b4c1bef1a45df0c7551560d4ac3a3f4744ab512uurtjjxz@rsa:~/ecdsa$
uurtjjxz@rsa:~/ecdsa$
```

Figure 3.21: run the script

3.6.3 Proposed solutions

This issue is very closely related to 3.5 and as such the solutions are the same.

3.7 MD5-collision simple

3.7.1 General information

Hash functions are a basic part of any modern cryptography. Hashes can be used to verify digital signatures, files authenticity, passwords, etc. Hash functions all have three fundamental properties, that are: They must convert digital information to a fixed length of hash value in ease, the information contained in the hash must be impossible to derive just from the hash value and it must be computationally impossible to find two different files that share a hash (Thompson 2005).

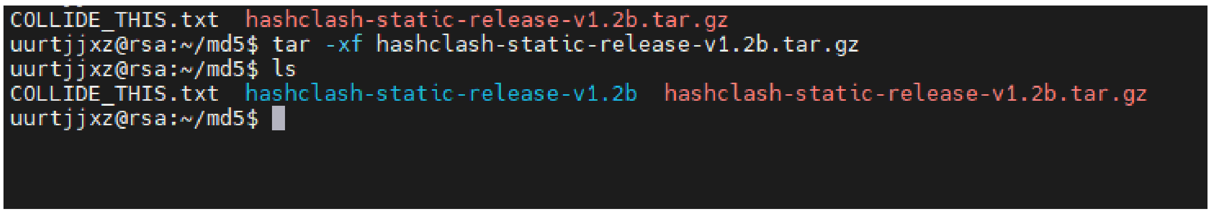
A collision is when you find two different files with the same hash. In summer 2004, it was demonstrated that it was possible to create MD5 collisions, where two slightly different messages had the same hash (Thompson 2005).

In this exercise I will demonstrate this by using fastcoll, to create two different files that have slightly different contents, but have the same MD5 hash.

This issue could be categorized as CWE-328: Use of Weak Hash, since the hash is used in a context where it is not secure.

3.7.2 Proof of concept

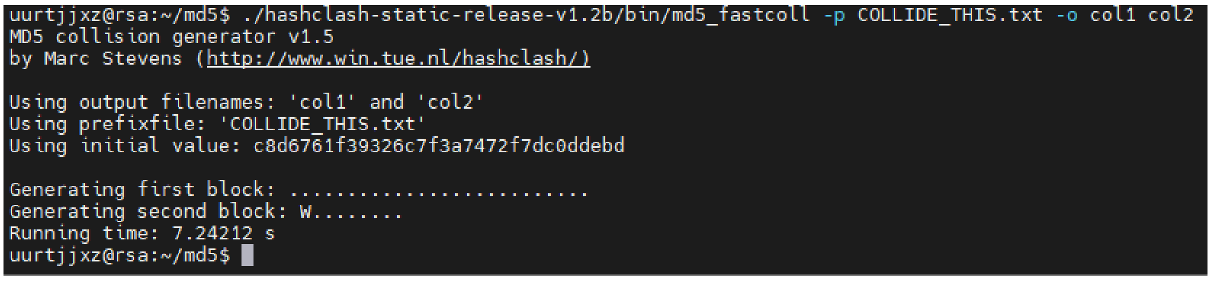
First I unzip the tar-file. This is .tar.gz which just means it is a tar archive that is also been compressed with Gzip. The file can be unzipped with command:
tar -xf hashclash-static-release-v1.2b.tar.gz
The tar command will auto-detect the compression type and extract the archive.



```
COLLIDE_THIS.txt  hashclash-static-release-v1.2b.tar.gz
uurtjjxz@rsa:~/md5$ tar -xf hashclash-static-release-v1.2b.tar.gz
uurtjjxz@rsa:~/md5$ ls
COLLIDE_THIS.txt  hashclash-static-release-v1.2b  hashclash-static-release-v1.2b.tar.gz
uurtjjxz@rsa:~/md5$ 
```

Figure 3.22: unzipped

Next I use fastcoll to create a collision that creates two files with different contents but the same MD5 value. I did this with the following command:
./hashclash-static-release-v1.2b/bin/md5_fastcoll -p COLLIDE_THIS.txt -o col1 col2



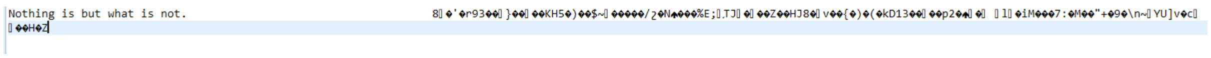
```
uurtjjxz@rsa:~/md5$ ./hashclash-static-release-v1.2b/bin/md5_fastcoll -p COLLIDE_THIS.txt -o col1 col2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'col1' and 'col2'
Using prefixfile: 'COLLIDE_THIS.txt'
Using initial value: c8d6761f39326c7f3a7472f7dc0ddebd

Generating first block: .........................
Generating second block: W........
Running time: 7.24212 s
uurtjjxz@rsa:~/md5$
```

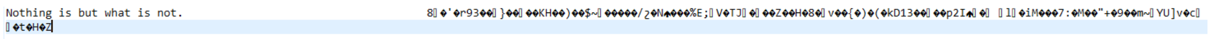
Figure 3.23: Collision

With this command the two new files will have the same prefix as COLLIDE_THIS.txt but different contents. This can also be seen in images 3.24 and 3.25 compared to the original 3.26



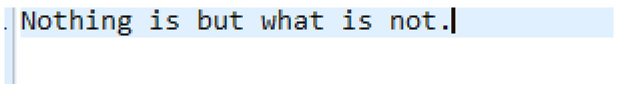
```
Nothing is but what is not.                    8▯�'�r93��▯}��▯��KH5�)��$~▯�����/¿�N▲���%E;▯.TJ▯�▯��Z��HJ8�▯v��{�)�(�kD13��▯��p2�▲▯�▯ ▯1▯�iM���7:�M��"+�9�\n~▯YU]v�c▯
▯��H�Z
```

Figure 3.24: Col1 content



```
Nothing is but what is not.                    8▯�'�r93��▯}��▯��KH��)��$~▯�����/¿�N▲���%E;▯V�TJ▯�▯��Z��H�8�▯v��{�)�(�kD13��▯��p2I▲▯�▯ ▯1▯�iM���7:�M��"+�9��m~▯YU]v�c▯
▯�t�H�Z
```

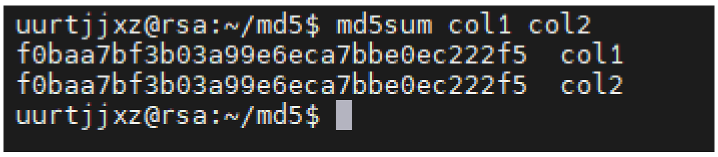
Figure 3.25: Col2 content



```
Nothing is but what is not.
```

Figure 3.26: Original content

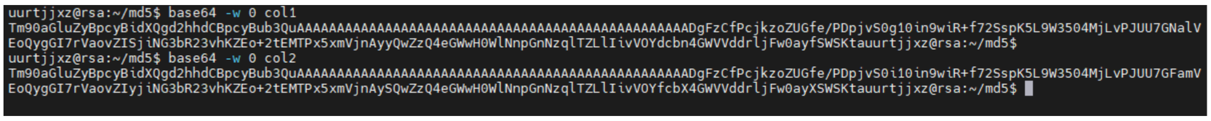
After this I will check that the two created files have the same md5 values. I do this with command:
md5sum col1 col2



```
uurtjjxz@rsa:~/md5$ md5sum col1 col2
f0baa7bf3b03a99e6eca7bbe0ec222f5  col1
f0baa7bf3b03a99e6eca7bbe0ec222f5  col2
uurtjjxz@rsa:~/md5$
```

Figure 3.27: MD5_Sum Check

It can seen from image 3.27 that the MD5 values are the same.
Lastly I convert the files to base64 encoded strings (please note that the ending of the strings "uurtjjxz@rsa: md5$" is not part of the base64 string:
base64 -w 0 col1
base64 -w 0 col2



```
uurtjjxz@rsa:~/md5$ base64 -w 0 col1
Tm90aGluZyBpcyBidXQgd2hhdCBpcyBub3QuAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADgFzCfPcjkzoZUGfe/PDpjvS0g10in9wiR+f72SspK5L9W3504MjLvPJUU7GNalV
EoQygGI7rVaovZISjiNG3bR23vhKZEo+2tEMTPx5xmVjnAyyQwZzQ4eGWwH0WlNnpGnNzqlTZLlIivVOYdcbn4GWVVddrljFw0ayfSWSKtauurtjjxz@rsa:~/md5$
uurtjjxz@rsa:~/md5$ base64 -w 0 col2
Tm90aGluZyBpcyBidXQgd2hhdCBpcyBub3QuAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADgFzCfPcjkzoZUGfe/PDpjvS0i10in9wiR+f72SspK5L9W3504MjLvPJUU7GFamV
EoQygGI7rVaovZIyjiNG3bR23vhKZEo+2tEMTPx5xmVjnAySQwZzQ4eGWwH0WlNnpGnNzqlTZLlIivVOYfcbX4GWVVddrljFw0ayXSWSKtauurtjjxz@rsa:~/md5$
```

Figure 3.28: base64 of The files

3.7.3 Proposed solutions

Since MD5 is not collision resistant. This means that it should not be used for any applications that rely on collision resistance. For example MD5 is unsuited for SSL certifications and digital signatures, since they are especially reliant on having unique hash values. In 2008 a group of researched managed to create fake SSL certificate validities (Sotirov et al., n.d.).

3.8 MD5-collission ’weaponized’

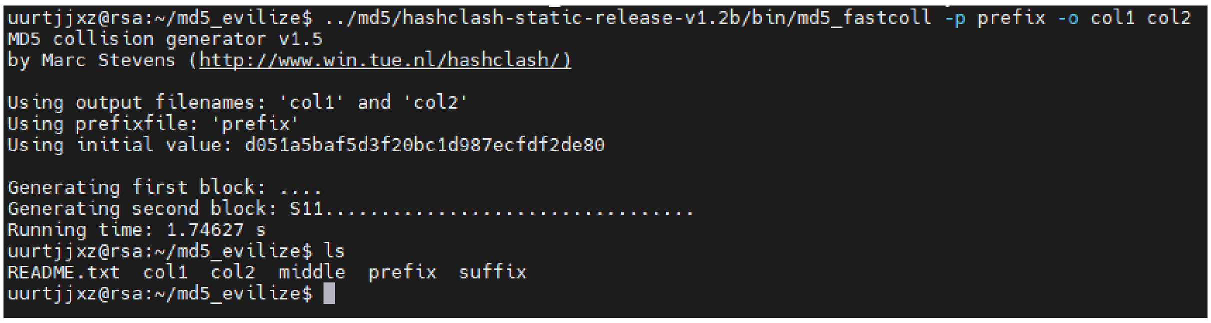
3.8.1 General information

View 3.7 for general information on MD5-collision.

In this exercise I will utilize fastcoll to create two python files. These python files will have the same MD5-hash values but will be completely different in their functionality.

3.8.2 Proof of concept

I had already unzipped the required files in the MD5 simple - exercise so I am skipping that part here.
I run the md5_fastcoll with command:
../md5/hashclash-static-release-v1.2b/bin/md5_fastcoll -p prefix -o col1 col2
This creates two new files col1 and col2 in the folder md5_evilize



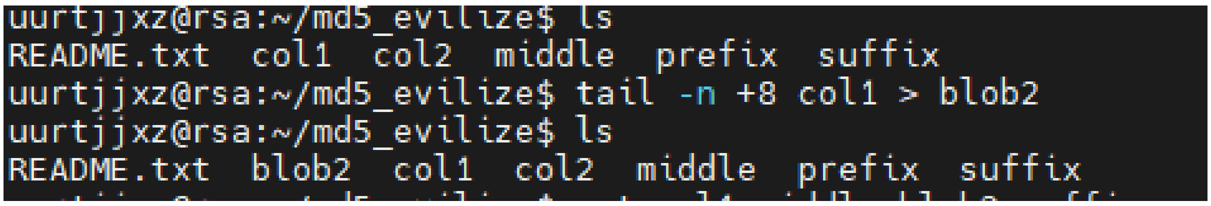
```
uurtjjxz@rsa:~/md5_evilize$ ../md5/hashclash-static-release-v1.2b/bin/md5_fastcoll -p prefix -o col1 col2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'col1' and 'col2'
Using prefixfile: 'prefix'
Using initial value: d051a5baf5d3f20bc1d987ecfdf2de80

Generating first block: ....
Generating second block: S11...............................
Running time: 1.74627 s
uurtjjxz@rsa:~/md5_evilize$ ls
README.txt  col1  col2  middle  prefix  suffix
uurtjjxz@rsa:~/md5_evilize$
```

Figure 3.29: Collision

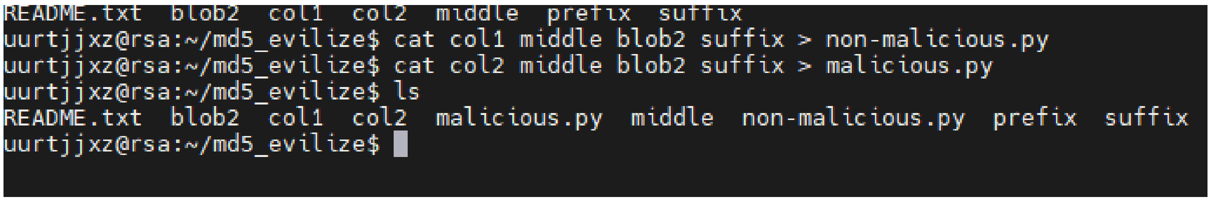
After this I make the blob2 with command:
tail -n +8 col1 > blob2



```
uurtjjxz@rsa:~/md5_evilize$ ls
README.txt  col1  col2  middle  prefix  suffix
uurtjjxz@rsa:~/md5_evilize$ tail -n +8 col1 > blob2
uurtjjxz@rsa:~/md5_evilize$ ls
README.txt  blob2  col1  col2  middle  prefix  suffix
```

Figure 3.30: tail

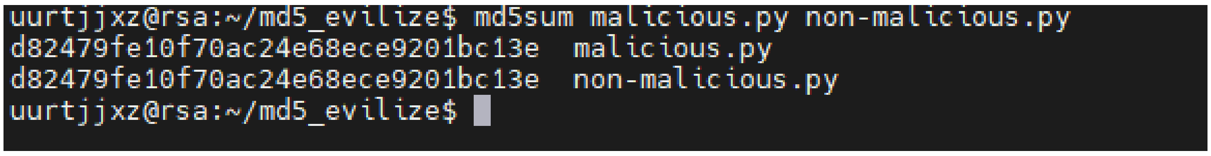
After this I create the two python files that will have the same md5:
cat col1 middle blob2 suffix > non-malicious.py
cat col2 middle blob2 suffix > malicious.py



```
README.txt  blob2  col1  col2  middle  prefix  suffix
uurtjjxz@rsa:~/md5_evilize$ cat col1 middle blob2 suffix > non-malicious.py
uurtjjxz@rsa:~/md5_evilize$ cat col2 middle blob2 suffix > malicious.py
uurtjjxz@rsa:~/md5_evilize$ ls
README.txt  blob2  col1  col2  malicious.py  middle  non-malicious.py  prefix  suffix
uurtjjxz@rsa:~/md5_evilize$
```

Figure 3.31: create python files

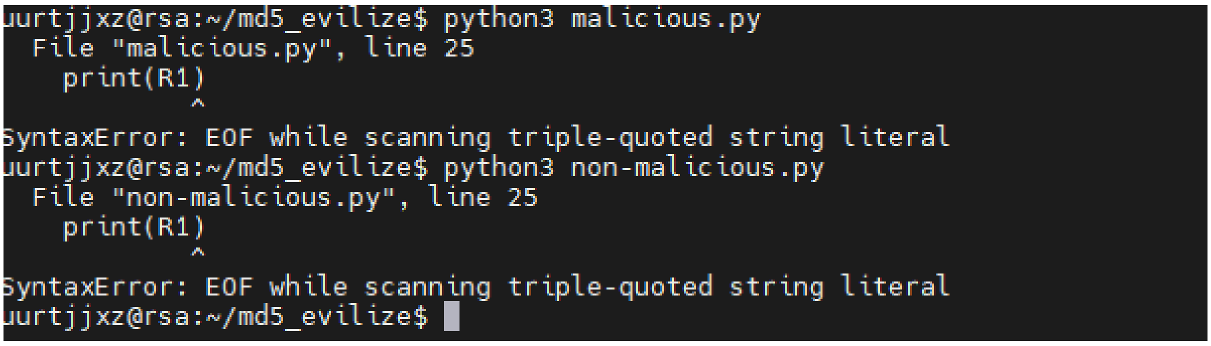
After this I check that the MD5 values of the files are the same with:
md5sum malicious.py non-malicious.py
And as it can be seen from the image below the values are the same.



```
uurtjjxz@rsa:~/md5_evilize$ md5sum malicious.py non-malicious.py
d82479fe10f70ac24e68ece9201bc13e  malicious.py
d82479fe10f70ac24e68ece9201bc13e  non-malicious.py
uurtjjxz@rsa:~/md5_evilize$
```

Figure 3.32: md5 of pythons

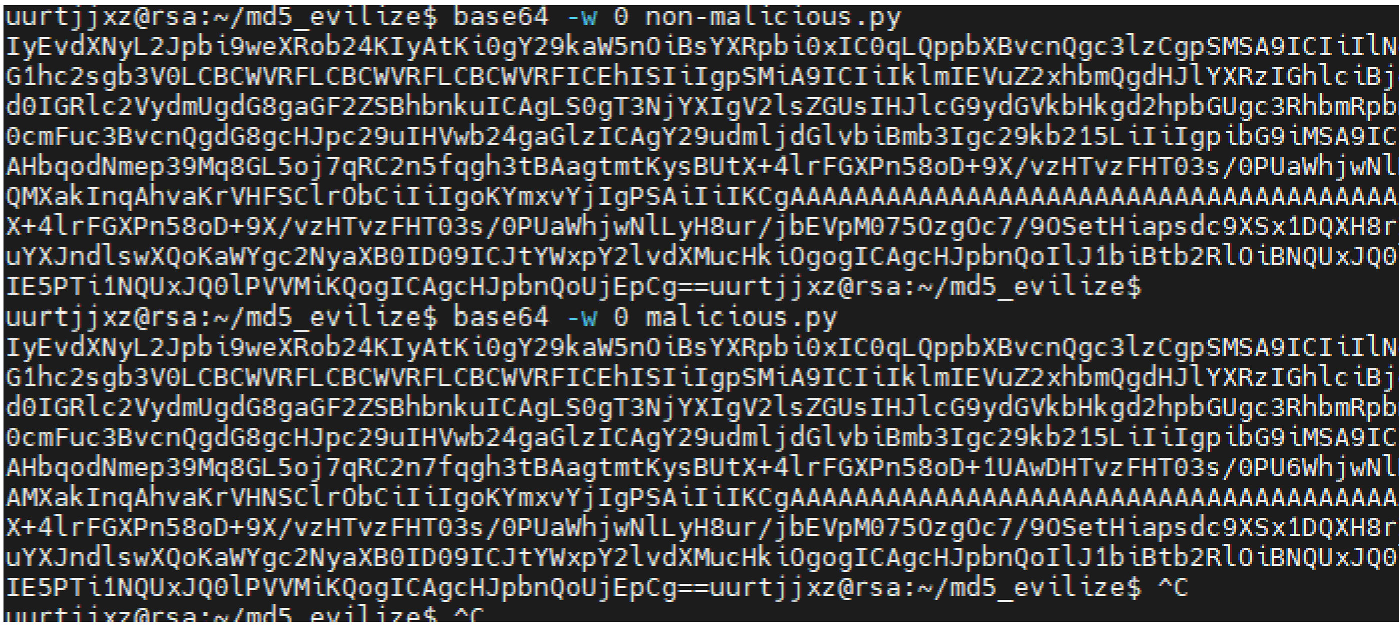
After this I tried to run the python files but faced the syntax error each time (I tried to re-do the collisions multiable times but each time it was the same) :(



```
uurtjjxz@rsa:~/md5_evilize$ python3 malicious.py
  File "malicious.py", line 25
    print(R1)
            ^
SyntaxError: EOF while scanning triple-quoted string literal
uurtjjxz@rsa:~/md5_evilize$ python3 non-malicious.py
  File "non-malicious.py", line 25
    print(R1)
            ^
SyntaxError: EOF while scanning triple-quoted string literal
uurtjjxz@rsa:~/md5_evilize$
```

Figure 3.33: run pythons

Lastly I made the base64 values, please not that the ending of uurtjjxz..etc is not part of the value.



```
uurtjjxz@rsa:~/md5_evilize$ base64 -w 0 non-malicious.py
IyEvdXNyL2Jpbi9weXRob24KIyAtKi0gY29kaW5nOiBsYXRpbi0xIC0qLQppbXBvcnQgc3lzCgpSMSA9ICIiIlN
G1hc2sgb3V0LCBCWVRFLCBCWVRFLCBCWVRFICEhISIiIgpSMiA9ICIiIklmIEVuZ2xhbmQgdHJlYXRzIGhlciBj
d0IGRlc2VydmUgdG8gaGF2ZSBhbnkuICAgLS0gT3NjYXIgV2lsZGUsIHJlcG9ydGVkbHkgd2hpbGUgc3RhbmRpb
0cmFuc3BvcnQgdG8gcHJpc29uIHVwb24gaGlzICAgY29udmljdGlvbiBmb3Igc29kb215LiIiIgpibG9iMSA9IC
AHbqodNmep39Mq8GL5oj7qRC2n5fqgh3tBAagtmtKysBUtX+4lrFGXPn58oD+9X/vzHTvzFHT03s/0PUaWhjwNl
QMXakInqAhvaKrVHFSClrObCiIiIgoKYmxvYjIgPSAiIiIKCgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
X+4lrFGXPn58oD+9X/vzHTvzFHT03s/0PUaWhjwNlLyH8ur/jbEVpM075OzgOc7/9OSetHiapsdc9XSx1DQXH8r
uYXJndlswXQoKaWYgc2NyaXB0ID09ICJtYWxpY2lvdXMucHkiOgogICAgcHJpbnQoIlJ1biBtb2RlOiBNQUxJQ0
IE5PTi1NQUxJQ0lPVVMiKQogICAgcHJpbnQoUjEpCg==uurtjjxz@rsa:~/md5_evilize$
uurtjjxz@rsa:~/md5_evilize$ base64 -w 0 malicious.py
IyEvdXNyL2Jpbi9weXRob24KIyAtKi0gY29kaW5nOiBsYXRpbi0xIC0qLQppbXBvcnQgc3lzCgpSMSA9ICIiIlN
G1hc2sgb3V0LCBCWVRFLCBCWVRFLCBCWVRFICEhISIiIgpSMiA9ICIiIklmIEVuZ2xhbmQgdHJlYXRzIGhlciBj
d0IGRlc2VydmUgdG8gaGF2ZSBhbnkuICAgLS0gT3NjYXIgV2lsZGUsIHJlcG9ydGVkbHkgd2hpbGUgc3RhbmRpb
0cmFuc3BvcnQgdG8gcHJpc29uIHVwb24gaGlzICAgY29udmljdGlvbiBmb3Igc29kb215LiIiIgpibG9iMSA9IC
AHbqodNmep39Mq8GL5oj7qRC2n7fqgh3tBAagtmtKysBUtX+4lrFGXPn58oD+1UAwDHTvzFHT03s/0PU6WhjwNl
AMXakInqAhvaKrVHNSClrObCiIiIgoKYmxvYjIgPSAiIiIKCgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
X+4lrFGXPn58oD+9X/vzHTvzFHT03s/0PUaWhjwNlLyH8ur/jbEVpM075OzgOc7/9OSetHiapsdc9XSx1DQXH8r
uYXJndlswXQoKaWYgc2NyaXB0ID09ICJtYWxpY2lvdXMucHkiOgogICAgcHJpbnQoIlJ1biBtb2RlOiBNQUxJQ0
IE5PTi1NQUxJQ0lPVVMiKQogICAgcHJpbnQoUjEpCg==uurtjjxz@rsa:~/md5_evilize$ ^C
uurtjjxz@rsa:~/md5_evilize$ ^C
```

Figure 3.34: base64 values

3.8.3 Proposed solutions

One should not trust MD5 hashes blindly when checking the integrity of files especially if the hash is obtained from the same channel as the file. This assignment is closely related to 3.7, for more information on proposed solutions please refer 3.7.3.

3.9 DSA file-message decryption

3.9.1 General information

This was i quite an interesting (while also quite infuriating) exercise. The exercise asks to decryption using DSA, but I believe this is impossible. I tried to scourge the internet for an answer or a hint how to approach this, since there are sources that indicate that encrypting using DSA would be possible. But after so much searching on information I have concluded this to be impossible and I have gathered here few of the best sources that actually explain this case.

DSA stands for Digital Signature Algorithm. When a person sends any data through digital means it is very important to check their authenticity for security reasons. For this reasons digital signatures are used and DSA is a public-key encryption algorithm used to generate an electronic signatures. I can not be used for encryption by it self, for this reason DSA is often used hand-to-hand with Diffie-Hellman as DSA keys can be used to authenticate Diffie-Hellman parameters and keys. This way the DSA will provide the authentication and privacy, and the shared secret from the Diffie-Hellman exchange can be used for asymmetric cipher. This can be then used for encryption. (Administration, n.d.)

As the name states DSA is a signing algorithim and is specially created to create signatures and not perform encryption. DSA, like RSA, also deals with public and private keys but both have different purposes for them. In RSA the public keys are used so anyone can encrypt and in DSA the public keys are needed so anyone can verify. In same manner, in RSA private keys allow decryption and in DSA private key allows signature creation (user46 2012).

Bibliography

Administration, Linux. n.d. “Public Key Cryptography with DSA,” http://www.linux-admins.net/2010/09/public-key-cryptography-with-dsa.html.

concise. 2017. “John the Ripper Password Cracking Tool,” https://web.archive.org/web/20170404031459/https://www.concise-courses.com/hacking-tools/password-crackers/john-the-ripper/.

Cook, John D. 2019. “Salting and stretching a password,” https://www.johndcook.com/blog/2019/01/25/salt-and-stretching/.

Doyle, Ray. 2019. “Cracking 256-bit RSA – Introduction,” https://www.doyler.net/security-not-included/cracking-256-bit-rsa-keys.

Jancis, Mindaugas. 2022. “Most popular password cracking techniques: learn how to protect your privacy,” https://cybernews.com/best-password-managers/password-cracking-techniques/.

jfoug. 2016. https://github.com/openwall/john/issues/2096#issuecomment-200683179.

Lenstra, Arjen K, James P Hughes, Maxime Augier, Joppe W Bos, Thorsten Kleinjung, and Christophe Wachter. 2012. “Ron was wrong, Whit is right.” *Cryptology EPrint Archive.*

Sotirov, Alexander, Jacob Appelbaum Marc Stevens, David Molnar Arjen Lenstra, and Benne de Weger Dag Arne Osvik. n.d. “MD5 considered harmful today, Creating a rogue CA certificate,” https://www.win.tue.nl/hashclash/rogue-ca/.

Thompson, Eric. 2005. “MD5 collisions and the impact on computer forensics.” *Digital investigation* 2 (1): 36–40.

user46. 2012. “Why can’t DSA be used for encryption?,” https://crypto.stackexchange.com/a/2586.