

Lab Report – Section 4.1

Section Author: Kelvin Chen

4.1 – Finding MD5 Hash Collision

Evilize:

- **Overview:**
-Created a C program that defines two functions - `main_good()` and `main_evil()`. Compiled code with `goodevil.o` where `evilize` is then used to generate a vector. That vector is then used with `md5coll` to produce a collision. Once a collision occurs, two executables are created, `good` and `evil`, that have the same MD5 hash but with different functions.
- **Tools Used:**
`evilize` package and `makeself` to create SFX files.

1) Executables:

- Used `lab-example.c`.
- **Initial vector:**

```
Initial vector: 0x29dd3524 0xe836b4ae 0x2e71e646 0xa5feaf80
```

```
./evilize lab-example -c init.txt -g good -e evil
```

- `lab-example` is the compiled binary
- `-c init.txt` tells `evilize` to use the collision file
- `-g good` and `-e evil` are the names for two output programs with distinct behaviors

- **Result:**

When running `md5 good evil`, both executables have the same MD5 hash.

```
kelvin@Kelvins-MacBook-Pro-3 evilize-0.1 % md5 good evil
MD5 (good) = 8eb60252b9cf6585ad8eb006473671dc
MD5 (evil) = 8eb60252b9cf6585ad8eb006473671dc
```

2) Self-Extracting Archives:

- Created a SFX file.
- **Initial vector:**

```
Initial vector: 0x40759342 0x55bedd1b 0x1d8faf52 0xbe7458f8
```

- **Result:**

After running `md5 good evil`, both files have identical MD5 hashes.

```
[kelvin@Kelvins-MBP-3 evilize-0.1 % md5 good evil
MD5 (good) = 5f54934f4ce4e54433b7f7c5091a71ff
MD5 (evil) = 5f54934f4ce4e54433b7f7c5091a71ff
```

3) Strings:

- A string file, `string_example.c`, was created.
- **Initial vector:**

```
Initial vector: 0x53bfd150 0x7e0b7889 0x50c32639 0x3ddb20f2
```

- After creating two executables (`good` and `evil`), their MD5 hash was checked, confirming a collision.

```
[kelvin@Kelvins-MBP-3 evilize-0.1 % md5 good evil
MD5 (good) = 094b4092622b0e48efeda45f24f93123
MD5 (evil) = 094b4092622b0e48efeda45f24f93123
```

SelfExtract:

- **Overview:**
-Six files are created where `pack3` will then be used to create two self-extracting archives. These newly created self-extracting archives will have the same MD5 hash, thus creating a collision.
- **Tools Used:**
`selfextract` tool and `Wine` to run Windows executables on macOS systems.

1) Executables:

- Created 6 dummy .cpp files that would be used to generate 6 dummy executable files.

- **Command:**

```
pack3 dummyexecutable1 dummyexecutable2 dummyexecutable3  
dummyexecutable4 dummyexecutable5 dummyexecutable6
```

- **Result:**

Two self-extracting archives were generated: executable_hash1.exe and executable_hash2.exe.

- After running executable_hash1.exe

```
Extracting dummyexecutable1...  
Extracting dummyexecutable2...  
Extracting dummyexecutable3...
```

- After running executable_hash2.exe

```
Extracting dummyexecutable4...  
Extracting dummyexecutable5...  
Extracting dummyexecutable6...
```

- **MD5 Verification:**

Running md5 executable_hash1.exe and md5 executable_hash2.exe confirmed identical hashes.

```
kelvin@Kelvins-MBP-3 selfextract-md5_coll % md5 executable_hash1.exe executable_  
hash2.exe  
MD5 (executable_hash1.exe) = a1dcc7cf0698bc6714a3e34e676e73aa  
MD5 (executable_hash2.exe) = a1dcc7cf0698bc6714a3e34e676e73aa
```

2) Self-Extracting Archives:

- Created 6 dummy .txt files (file1.txt to file6.txt).

- **Command:**

```
pack3 file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt
```

- **Result:**

Two self-extracting archives were generated: self_hash1.exe and self_hash2.exe.

- After running `self_hash1.exe`

```
Extracting file1.txt...
Extracting file2.txt...
Extracting file3.txt...
```

- After running `self_hash2.exe`

```
Extracting file4.txt...
Extracting file5.txt...
Extracting file6.txt...
```

- **MD5 Verification:**

Running `md5 self_hash1.exe` and `md5 self_hash2.exe` showed both files had the same MD5 hash.

```
kelvin@Kelvins-MBP-3 selfextract-md5_coll % md5 self_hash1.exe self_hash2.exe
MD5 (self_hash1.exe) = b8a1c1888ab511840e8b12b816da5843
MD5 (self_hash2.exe) = b8a1c1888ab511840e8b12b816da5843
```

3) Strings:

- Created 6 strings (string1 to string6).
- **Command:**
`wine pack3.exe string1.txt string2.txt string3.txt string4.txt string5.txt string6.txt`
- **Result:**
Two self-extracting archives were generated: `string_hash1.exe` and `string_hash2.exe`.
 - After running `string_hash1.exe`

```
Extracting string1.txt...
Extracting string2.txt...
Extracting string3.txt...
```

- After running `string_hash2.exe`

```
Extracting string4.txt...
Extracting string5.txt...
Extracting string6.txt...
```

- **MD5 Verification:**

Both files had identical MD5 hashes.

```
kelvin@Kelvins-MBP-3 selfextract-md5_coll % md5 string_hash1.exe string_hash2.exe
MD5 (string_hash1.exe) = 83997ac490337079b8ca71047f2385c1
MD5 (string_hash2.exe) = 83997ac490337079b8ca71047f2385c1
```

Web-version:

- **Overview:**
-Created a string/file for the input where the input is then checked for collision.
- **Tools Used:**
`web_version` and Wine.

1) Executables:

- Created a dummy file that would be used to generate a dummy executable file.
- **Command:**
`wine md5tunnel_v1.exe ./dummyexecutable`
- **Result:**
A new text file, `executable_collision.txt`, was generated confirming the MD5 collision.

```

23.02.2025 16:18:19.984

The first block collision took : 7.800000 sec
23.02.2025 16:18:27.927

Check: The same MD5 hash

The second block collision took : 0.000000 sec23.02.2025 16:18:27.937

The first and the second blocks together took : 7.800000 sec
AVERAGE time for the 1st block = 7.800000 sec
AVERAGE time for the 2nd block = 0.000000 sec
AVERAGE time for the complete collision = 7.800000 sec
No. of collisions = 1

The first block collision took : 8.500000 sec
23.02.2025 16:18:36.575

Check: The same MD5 hash

The second block collision took : 0.000000 sec23.02.2025 16:18:36.591

The first and the second blocks together took : 8.500000 sec
AVERAGE time for the 1st block = 8.150000 sec
AVERAGE time for the 2nd block = 0.000000 sec
AVERAGE time for the complete collision = 8.150000 sec
No. of collisions = 2

The first block collision took : 8.400000 sec
23.02.2025 16:18:45.136

Check: The same MD5 hash

The second block collision took : 0.600000 sec23.02.2025 16:18:45.820

The first and the second blocks together took : 9.000000 sec
AVERAGE time for the 1st block = 8.233333 sec
AVERAGE time for the 2nd block = 0.200000 sec
AVERAGE time for the complete collision = 8.433333 sec
No. of collisions = 3

```

2) Self-Extracting Archives:

- Used self_hash1.exe that was generated from selfextract
- **Command:**
wine md5tunnel_v1.exe ./self_hash1.exe

Result:

A new text file, collision_md5_BD1F7E1E.TXT, was generated confirming the MD5 collision.

23.02.2025 16:23:11.944

The first block collision took : 0.600000 sec

23.02.2025 16:23:12.632

Check: The same MD5 hash

The second block collision took : 0.300000 sec23.02.2025 16:23:12.939

The first and the second blocks together took : 0.900000 sec

AVERAGE time for the 1st block = 0.600000 sec

AVERAGE time for the 2nd block = 0.300000 sec

AVERAGE time for the complete collision = 0.900000 sec

No. of collisions = 1

The first block collision took : 12.500000 sec

23.02.2025 16:23:25.687

Check: The same MD5 hash

The second block collision took : 0.000000 sec23.02.2025 16:23:25.772

The first and the second blocks together took : 12.500000 sec

AVERAGE time for the 1st block = 6.550000 sec

AVERAGE time for the 2nd block = 0.150000 sec

AVERAGE time for the complete collision = 6.700000 sec

No. of collisions = 2

The first block collision took : 26.400000 sec

23.02.2025 16:23:52.570

Check: The same MD5 hash

The second block collision took : 0.700000 sec23.02.2025 16:23:53.319

The first and the second blocks together took : 27.100000 sec

AVERAGE time for the 1st block = 13.166667 sec

AVERAGE time for the 2nd block = 0.333333 sec

AVERAGE time for the complete collision = 13.500000 sec

No. of collisions = 3

3) Strings:

- **Command:**

wine md5tunnel_v1.exe "test_string_1",

Result:

A new text file, collision_md5_DDB37281.TXT, was generated confirming the MD5 collision.

```
22.02.2025 21:06:42.594
The first block collision took : 0.100000 sec
22.02.2025 21:06:42.762
Check: The same MD5 hash
The second block collision took : 0.300000 sec22.02.2025 21:06:43.105
The first and the second blocks together took : 0.400000 sec
AVERAGE time for the 1st block = 0.100000 sec
AVERAGE time for the 2nd block = 0.300000 sec
AVERAGE time for the complete collision = 0.400000 sec
No. of collisions = 1
The first block collision took : 6.300000 sec
22.02.2025 21:06:49.475
Check: The same MD5 hash
The second block collision took : 1.000000 sec22.02.2025 21:06:50.474
The first and the second blocks together took : 7.300000 sec
AVERAGE time for the 1st block = 3.200000 sec
AVERAGE time for the 2nd block = 0.650000 sec
AVERAGE time for the complete collision = 3.850000 sec
No. of collisions = 2
```

After executing the command `wine md5tunnel_v1.exe "test1" "test2"`, a new txt file was generated (collision_md5...EA24343.TXT)

Can you extend this to other hash functions? If so, how? If not, why not?:

- The tools `evilize`, `selfextract`, and `web_version` will most likely not be able to be used with other hash functions. This is because the tools are used to find MD5 hash collisions. It is finding vulnerabilities and weaknesses in the hashing algorithm, so if it is used against a strong hashing algorithm it would most likely fail at getting a hash collision.

Lab Report – Section 4.2

Section Author: Cara Dong

4.2 – Finding SHA-1 Hash Collisions

Overview:

This section explores the process of generating SHA-1 hash collisions using two different methods: an online SHA-1 collider tool and a Python-based approach for PDF files.

1) SHA-1 Hash Collision Using Online Tool (Images)

- Utilized the SHA1 Collider online tool to produce a collision between two images.

```
hunter493ofnm@xdc:~/Lab-2$ openssl sha1 a.pdf
SHA1(a.pdf)= 7f576497eec6bc4d1c605e3f4d0242962f062fa6
hunter493ofnm@xdc:~/Lab-2$ openssl sha1 b.pdf
SHA1(b.pdf)= 7f576497eec6bc4d1c605e3f4d0242962f062fa6
```

- **Process:**
 - Uploaded two different images to the tool.
 - The tool generated a pair of images with distinct content but identical SHA-1 hashes.
- **Result:**
 - The SHA-1 hash for both images was verified and confirmed to be the same, despite their visual differences.

2) Finding SHA-1 Hash Collision for PDFs (Python Code)

- Used provided Python code to find SHA-1 hash collisions between two PDF files.
- **Initial SHA-1 hashes for the different PDFs:**

```
hunter493ofnm@xdc:~/Lab-2/sha1collider-master$ openssl sha1 Tutoring-Hours.pdf
SHA1(Tutoring-Hours.pdf)= 0b549d201bcae5ca4a458abe84850c78989c5044
hunter493ofnm@xdc:~/Lab-2/sha1collider-master$ openssl sha1 Updated-Hours.pdf
SHA1(Updated-Hours.pdf)= 8af66398b3c4f187fcdd6526c44b4337f410582
```

- **Process:**
 - Ran the Python script with the two PDF files as input.
 - The code generated modified versions of both PDFs.
- **Result:**

```
hunter493ofnm@xdc:~/Lab-2/sha1collider-master$ openssl sha1 out-Tutoring-Hours.pdf
SHA1(out-Tutoring-Hours.pdf)= 3df7a201c61cd580045f693b996e2966791f0a23
hunter493ofnm@xdc:~/Lab-2/sha1collider-master$ openssl sha1 out-Updated-Hours.pdf
SHA1(out-Updated-Hours.pdf)= 3df7a201c61cd580045f693b996e2966791f0a23
```

- While the contents of the PDFs remained visibly unchanged, the SHA-1 hash values were confirmed to be identical.

Conclusion:

This exercise demonstrates the vulnerability of SHA-1 to collision attacks. Even with seemingly different files (images or PDFs), it is possible to create two files with the same SHA-1 hash, highlighting why SHA-1 is no longer considered secure for cryptographic integrity.

Lab Report – Section 4.3

Section Author: Karwai Kang

4.3 – Website SSL/TLS Certificates

RSA Private Key Generation

Rebuilding OpenSSL

Rebuilding

Running `openssl genrsa 100` didn't work as the key size was too small. In the OpenSSL 3.0.0 source code, I changed the minimum key size in the code to 0.

```
// #define RSA_MIN_MODULUS_BITS    512
#define RSA_MIN_MODULUS_BITS    0
```

Then built OpenSSL without the minimum key size.

```
$ ./Configure --prefix=/home/${USER}/openssl-no-min-bits/prefix --openssl
make
sudo ldconfig ~/openssl-no-min-bits/prefix/lib64/
```

100-bit RSA Key Generation Example

100-bit Example

```
$ sudo ./openssl genrsa 100 > test.pem
$ cat test.pem
-----BEGIN PRIVATE KEY-----
MGkCAQAwDQYJKoZIhvcNAQEBBQAEVTBTAqEAAg0KXOD4+HeECDdmtBJRAgMBAAEC
DQhmNZa21Sr+NTpQKgECBwNkq43WlSECBwMNz/E5pzECBwNVn00f1yECBwK29362
QjECBwK4XW0qe4c=
-----END PRIVATE KEY-----
$ sudo ./openssl rsa -in test.pem -pubout -out pubkey.pem -text
writing RSA key
$ sudo cat pubkey.pem
Private-Key: (100 bit, 2 primes)
modulus:
    0a:5c:e0:f8:f8:77:84:08:37:66:b4:12:51
publicExponent: 65537 (0x10001)

privateExponent:
    08:66:35:96:b6:d5:2a:fe:35:3a:50:2a:01
prime1: 955112911967521 (0x364ab8dd69521)
prime2: 859611686610737 (0x30dcff139a731)
exponent1: 938557252294433 (0x3559ce39fd721)
exponent2: 764124052472369 (0x2b6f77eb64231)
coefficient: 765661197007751 (0x2b85d63aa7b87)
-----BEGIN PUBLIC KEY-----
MCgwDQYJKoZIhvcNAQEBBQADFwAwFAINC lzg+Ph3hAg3ZrQSUQIDAQAB
-----END PUBLIC KEY-----
```

384-bit RSA Key Generation Example

```
$ openssl genrsa 384 > test1.pem
$ openssl rsa -in test1.pem -pubout -out pubkey.pem -text
writing RSA key
$ sudo more pubkey.pem
Private-Key: (384 bit, 2 primes)
modulus:
    00:e0:1d:4b:bb:92:78:f8:ef:35:1c:75:43:f5:27:
    9b:23:cc:0e:b1:48:b9:ea:4d:b7:1f:9a:3f:4b:26:
    10:96:f6:d3:29:6a:4e:e2:3a:da:1e:03:27:83:dc:
    33:5e:3a:07
publicExponent: 65537 (0x10001)
privateExponent:
    00:96:23:ea:35:8a:26:13:17:25:ec:9f:be:dc:41:
    21:54:f5:02:ae:d0:3b:04:f5:f4:44:21:4a:96:46:
    04:ae:8c:64:61:f6:30:7f:67:ee:86:d0:b2:65:65:
    86:32:45:f1
prime1:
    00:fb:3e:0f:1b:39:da:6b:61:66:a9:65:3b:a9:9d:
    14:a2:1d:c3:64:de:6d:81:09:cf
prime2:
    00:e4:5b:ba:b0:23:48:93:1e:c9:18:b3:78:84:02:
    f1:07:96:9f:3a:1f:51:d8:32:49
exponent1:
    49:a9:9d:66:3f:25:22:1c:b1:ab:f2:e9:76:46:7e:
    f9:35:3e:c7:c1:4b:5b:ed:13
exponent2:
    00:c2:4d:36:8c:f6:88:a2:fd:b7:d7:f7:a5:1c:96:
    30:25:e7:c8:35:b9:3e:77:79:59
coefficient:
    47:95:a6:7c:b2:1e:87:71:fe:86:d3:c2:5f:34:36:
    c5:65:32:10:e0:9e:2a:b1:65
-----BEGIN PUBLIC KEY-----
MEwwDQYJKoZIhvcNAQEBBQAD0AwAwOAIXA0AdS7uSePjvNRx1Q/UnmyPMDrFIuepN
tx+aP0smEJb20ylqTuI62h4DJ4PcM146BwIDAQAB
-----END PUBLIC KEY-----
```

Certificate Generation

```
$ sudo ./openssl req -new -x509 -nodes -md5 -days 100 -key test1.pem > ho:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:NY
Locality Name (eg, city) []:New York
Organization Name (eg, company) [Internet Widgits Pty Ltd]:CUNY Hunter Co
Organizational Unit Name (eg, section) []:CSCI493
Common Name (e.g. server FQDN or YOUR name) []:Karwai Kang
Email Address []:Karwai.Kang76@login.cuny.edu
```

Public Key Extraction

```
$ sudo ./openssl x509 -modulus -in host.cert
Modulus=E01D4BBB9278F8EF351C7543F5279B23CC0EB148B9EA4DB71F9A3F4B261096F6D:
-----BEGIN CERTIFICATE-----
MIICeTCCAjOgAwIBAgIUdT+cYHpTaG5VViyMr1VNyrcXHV8wDQYJKoZIhvcNAQEE
BQAwgAxCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJOWTERMA8GA1UEBwwITmV3IFlv
cmsxHDAaBgNVBAoME0NVTLkgSHVudGVyIENvbGxZL2UxEDA0BgNVBAsMB0NTQ0k0
OTMxFDASBgNVBAMMC0thcndhaSBLYW5nMSswKQYJKoZIhvcNAQkBFhxLYXJ3YWku
S2FuZzc2QGxvZ2luLmN1bnkuZWZ1MB4XDTE1MDIyNDE5NDIxMFoXDTE1MDYwNDE5
NDIxMFowgAxCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJOWTERMA8GA1UEBwwITmV3
IFlvcmsxHDAaBgNVBAoME0NVTLkgSHVudGVyIENvbGxZL2UxEDA0BgNVBAsMB0NT
Q0k0OTMxFDASBgNVBAMMC0thcndhaSBLYW5nMSswKQYJKoZIhvcNAQkBFhxLYXJ3
YWkuS2FuZzc2QGxvZ2luLmN1bnkuZWZ1MEwwDQYJKoZIhvcNAQEBBQAD0AwAwOAIx
A0AdS7uSePjvNRx1Q/UnmyPMDrFIuepNtx+aP0smEJb20ylqTuI62h4DJ4PcM146
BwIDAQABo1MwUTAdBgNVHQ4EFgQUadD60RfhFYDm90LUI2JzbtAbvJ8wHwYDVR0j
BBgwFoAUadD60RfhFYDm90LUI2JzbtAbvJ8wDwYDVR0TAQH/BAUwAwEB/zANBgkq
hkiG9w0BAQQFAAMxALKHGTLMG6kr/qmAQZxLSKs1mlzSrPIj3rovFizCefzws9ow
tcqJzptje0zqWQA0cw==
-----END CERTIFICATE-----
```

Modulus Factorization

CADO-NFS

I downloaded the source code of CADO-NFS 2.3.0 (`cado-nfs-2.3.0.tar.gz`), but it failed to build.

```
$ make
CMake Error at config/gmp.cmake:55 (message):
  gmp.h cannot be found. Please install Gnu MP, and specify its install
  prefix in local.sh
Call Stack (most recent call first):
  CMakeLists.txt:231 (include)

-- Configuring incomplete, errors occurred!
See also "/home/${USER}/CADO-NFS/cado-nfs-2.3.0/build/${COMPUTER}/CMakeFi
make: *** [Makefile:7: all] Error 1
```

I downloaded the source code of GMP 6.3.0 (`gmp-6.3.0.tar.lz`). The output contained this at the end:

```
configure: summary of build options:

Version:           GNU MP 6.3.0
Host type:         kabylake-pc-linux-gnu
ABI:               64
Install prefix:    /home/${USER}/GNU-MP/out
Compiler:          gcc
Static libraries:  yes
Shared libraries:  yes
```

I built it with `make` and `make check`. Every test passed, except for one which was skipped, `t-addaddmul`. Unfortunately, the installation was nowhere to be found, so I could not continue.

msieve

To use `msieve`, I first converted the modulus to decimal:

```
344943688634593329704762598745635887573166909937821635839179682919409557
26737587051968654041985395892161034325998087
```

After about a day, the program was halted when just 1647 out of 331346 relations were calculated.

```
Msieve v. 1.53 (SVN 1005)
random seeds: efc5dc0 5575e83a
factoring 34494368863459332970476259874563588757316690993782163583917968291940955726737587051968654041985395892161034325998087
searching for 15-digit factors
commencing quadratic sieve (116-digit input)
using multiplier of 7
using generic 32kb sieve core
sieve interval: 73 blocks of size 32768
processing polynomials in batches of 3
using a sieve bound of 9951407 (331250 primes)
using large prime bound of 1492711050 (30 bits)
using double large prime bound of 32595401239865850 (47-55 bits)
using trial factoring cutoff of 55 bits
polynomial 'A' values have 15 factors
1647 relations (1625 full + 22 combined from 94097 partial), need 331346
elapsed time 19:53:19
```

Conclusion:

- Successfully generated custom RSA keys and certificates after modifying OpenSSL source code.
- Modulus factorization attempts using CADO-NFS and msieve were unsuccessful due to time and resource constraints.
- Demonstrated the impracticality of brute-forcing RSA keys of significant length under typical lab conditions.

Lab Report Sections 4.4.1 and 4.4.2 – Attempts and Analysis

4.4.1 – Attempts to Factor the RSA Modulus

Section Author: David Xiao

Objective:

The goal of this section was to factor a 1024-bit RSA modulus extracted from the provided TLS capture file to obtain the server's private key for decrypting the encrypted TLS session.

What I've Tried So Far:

1. MSIEVE Attempts:

- **v1.5.3 Pre-built Version:**
Attempted to use the provided MSIEVE v1.5.3 build; however, it does not support modern CUDA architectures. My GPU (RTX 3080 Ti, architecture 8.6) was incompatible, making GPU acceleration unavailable.
 - **Latest MSIEVE (r1056) Compilation:**
I tried compiling the latest MSIEVE (r1056) version from SourceForge to leverage my hardware. Compilation failed due to incompatibilities between MSIEVE's CUDA dependencies and the latest CUDA toolkit versions. Attempts to resolve conflicts with Visual Studio integration also failed.
 - **Feasibility Assessment:**
Even if successfully compiled, factoring a 1024-bit RSA modulus with MSIEVE on my hardware (AMD Ryzen 9 7900X CPU, RTX 3080 Ti GPU, and 32 GB RAM) would have taken months, far exceeding the lab's time constraints.
-

2. CADO-NFS Attempts:

- **Environment Setup:**
Installed CADO-NFS within a Windows Subsystem for Linux (WSL) environment.
- **Execution and Results:**
Initiated the polynomial selection phase, which projected a runtime of several decades

given my hardware. Factoring a 1024-bit key with CADO-NFS was deemed infeasible within the allotted timeframe.

3. Fermat's Factorization & GCD-Based Attempts:

- **Fermat's Method:**
Implemented a Python script using Fermat's factorization. The method failed due to the distant nature of the prime factors.
 - **GCD Approaches:**
Utilized the `gmpy2` library to calculate the greatest common divisor (GCD) between the target modulus and various datasets. No shared factors were found.
-

4. FASTGCD & Known Weak Keys Dataset:

- **Dataset Acquisition:**
Downloaded approximately 38,000 weak RSA keys from [Debian OpenSSL](https://factorable.net/weakkeys12.conference.pdf) based on the "Mining Your Ps and Qs" research paper located on <https://factorable.net/weakkeys12.conference.pdf>
 - **Testing with FASTGCD:**
Ran FASTGCD against the known weak keys. No common factors with the target modulus were detected.
-

5. Provided Files Examination:

- **Certificate Analysis:**
Inspected provided `.crt` and `.pem` files using OpenSSL for embedded keys. No usable private keys were found for the original TLS capture.
 - **Other Resources:**
Explored supplemental files (including `.zip`, `.tar.gz`, and C source files). No breakthrough information was found.
-

RSA Public Key Details:

Public-Key: (1024 bit)

modulus:

00:a4:48:2e:bb:05:80:82:3d:f7:6c:e4:eb:bc:25:
9c:45:bf:db:51:ac:75:ec:a9:e8:e3:72:fc:7c:7c:
bd:f5:ea:24:15:d5:e7:de:75:b6:7a:c4:10:7f:83:

73:86:26:67:4e:46:25:3a:f9:60:ea:0f:2b:cb:fe:
1c:45:e3:3f:e5:72:5f:54:94:33:fc:38:48:46:ec:
16:97:6a:cf:42:98:2d:e8:11:af:30:10:10:e9:bc:
0b:c1:24:05:e7:42:51:5f:01:07:4d:a2:e8:cb:40:
15:a7:4b:0a:57:94:7d:2a:3f:3f:d0:c2:b5:02:46:
da:d7:ca:9c:4b:7c:76:3e:0f
publicExponent: 65537 (0x10001)

Conclusion for 4.4.1:

Despite extensive efforts, factoring the 1024-bit RSA modulus proved unsuccessful. The complexity of the key and hardware limitations made factorization impractical within the lab's timeframe.

4.4.2 – TLS Traffic Decryption Attempts

Objective:

Section 4.4.2 aimed to decrypt the TLS-encrypted traffic provided. Due to the inability to factor the RSA modulus in Section 4.4.1, work focused on the provided sample files containing a known private key.

Approach and Process:

1. Certificate and Key Analysis:

- Verified the provided certificates and keys using OpenSSL.
- Successfully extracted the private key from the PEM file.

2. Packet Capture Investigation:

- Opened `sample-tlsdump.pcap` in Wireshark and located the TLS handshake messages.
- Used the private key to decrypt the pre-master secret (PMS) from the handshake.

3. TLS Decryption Steps:

- Generated a key log file with the extracted PMS.
 - Loaded the key log into Wireshark, successfully decrypting the TLS sessions.
-

Extracted Data from Sample TLS Capture:

HTTP Response Header:

HTTP/1.0 200 ok
Content-type: text/html

HTML Body:

```
<HTML>
<BODY BGCOLOR="#ffffff">
<pre>
s_server -cert cert2025.pem -key key2025.pem -www -tls1 -no_dhe -cipher AES256-SHA
Secure Renegotiation IS supported
Ciphers supported in s_server binary:
TLSv1/SSLv3: AEAD-CHACHA20-POLY1305-SHA256
TLSv1/SSLv3: AEAD-AES256-GCM-SHA384
TLSv1/SSLv3: AEAD-AES128-GCM-SHA256
TLSv1/SSLv3: AES256-SHA
</pre>
</BODY>
</HTML>
```

Common Ciphers:

- AEAD-AES128-GCM-SHA256
- AEAD-AES256-GCM-SHA384
- AEAD-CHACHA20-POLY1305-SHA256
- AES256-SHA

Session Information:

- Master Key:
4842976613B3C356232B4BDDD9D160AB238E1A23174164602959EBB89B4BEEC5A
8A36ECEEF8EA57BCD9980FF50BEDCE9
- Cipher Suite: AES256-SHA
- Protocol: TLSv1
- Session Cache Hits: 4

Challenges Faced:

- Unable to decrypt the original TLS capture due to unsuccessful RSA factorization.
- Work focused on provided sample files, which contained known keys.

- Despite these challenges, the decryption process was successfully demonstrated using sample data.

Provided Certificate and Key:

Sample Certificate:

-----BEGIN CERTIFICATE-----

```
MIIG8DCCBdigAwIBAgIQCmq4ki4iCdhBq68ny5kiBjANBgkqhkiG9w0BAQsFADBZ
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMTMwMQYDVQQDEyp
E
aWdpQ2VydCBHbG9iYWwgRzlgVExTIFJTQSBTSEEyNTYgMjAyMCBDQTEwHhcNMjQw
NTE3MDAwMDAwWhcNMjUwNjAyMjM1OTU5WjBtMQswCQYDVQQGEwJVUzERMA8GA1UE
CBMITmV3IFlvcmsxETAPBgNVBACTE5IdyBZb3JrMRwwGgYDVQQKEExNDVU5ZIEh1
bnRlciBDb2xsZWdlMR0wGAYDVQQDDBEqLmh1bnRlci5jdW55LmVkdTCCASlwdQYJ
KoZlhcNAQEBAQADggEPADCCAQoCggEBAMQ82grITakylFrrckmWn1xfOFnTN1c9
Z5kRi05a2sUNxn/3Q8028l9WAFX44spCSg4J/MGpSkRBrHk/jtHgajhkWv4+eh0q
cxonjOedljPxm91DH5ZPFpEm98SrcAuhPGeyHkwjFOkRUbC2RUqEEpalQdsz5XbS
vZPH5weCHNGfh38aJIFTsc+nmz//WtBemJk2mPEVsyJwAL/XGdrWM1X0U1eBG8Zz
EjVRYym42MkuDyC33GOC96fZoc+zG42ol1jw0AJTpng2UMzybDYzqMe9ZY/AcuZu
tHDWpfjtAvNb7jzIYfUYEXKFcAqtdYqhNlkrwXxKQU9LmbqqYdmFu8CAwEAAaOC
A54wggOaMB8GA1UdIwQYMBaAFHSFgMBmx9833s+9KTeqAx2+7c0XMB0GA1UdDgQW
BBRAOipF2dtb9cdVltxtr0v6eHA61zAtBgNVHREEJjAkghEqLmh1bnRlci5jdW55
LmVkdYIPaHVudGVyLmN1bnkuZWR1MA0GCSqGSIb3DQEBCwUAA4IBAQAQWqp+dXK2C
s9M2S65WgDy5UdA1dmYdwrh2hjDQ42RoHWWnziFpEmZTYwyDw6lvKL0OW9Vaz6QK
on4tmLuwuJnHVcqu2Cwsmv7J45346jNOGFAw7np23iVkc2ScnURBds8YG3KHtvTh
Ln9ptypOPD0v+5BeZ63wrmkrPOLTKIBtOMSHmA26Dc/hPNm/Oix0yDEDaFlac4bk
vqUjB9Li2KnK6IKCpT+5wHHNpPEA9YOGGrQfho5DFZerOzV/7H3DBugb6oqcwnps4
SQ3EmHfPLpc1U7Gyv/MMM1R4UvKyIGTHTQoTnlh2fDI01dNVpde2epzbJ/PIP2X
qFwQTcbSJVIBA==
```

-----END CERTIFICATE-----

Summary and Reflection:

- **Section 4.4.1:** Despite extensive efforts (MSIEVE, CADO-NFS, GCD attempts, and known weak key databases), factoring the 1024-bit RSA modulus was unsuccessful.
- **Section 4.4.2:** Successfully decrypted the sample TLS traffic using the provided private key, showcasing the decryption process and analysis of the TLS handshake.

Lab Report – Section 5: Word Problems

5. Word Problems

1. Summarize the attack techniques used by the tools

Evilize:

- Creates two different executables that have the same MD5 hash.
- A C program is written with two functions: one performing a good action (`main_good`) and another performing a bad action (`main_evil`).
- The code is compiled and linked with `goodevil.o`. Evilize generates a vector used with `md5coll` to produce a hash collision.
- This collision creates two executables—"good" and "evil"—that share the same MD5 hash but perform different operations.

Selfextract:

- Takes six input files and divides them evenly to produce two self-extracting archives.
- Despite containing different sets of files, both self-extracting archives have the same MD5 hash, demonstrating a collision.

Web_version:

- Requires an input string or file. Once provided, the tool attempts to find an MD5 collision.
- It works by modifying small bits of the input data until a collision is detected.
- The tool logs how long it takes to find the collision.

SHA1 Collider:

- Based on the identical-prefix collision technique explored in the SHAttered paper.
 - Appends a predetermined prefix with two pairs of near-collision blocks. The first pair introduces a difference in the hash chaining process, which is canceled out by the second pair.
 - This technique leads to two distinct files sharing the same SHA-1 hash value. (Stevens et al., 2017, p. 5)
-

2. How would you use the gained knowledge, namely the factored RSA modulus, to attack a TLS-encrypted connection assuming it used an RSA key exchange?

- By factoring the RSA modulus, you can obtain the prime factors and , which are used to compute the private key with the formula:
 - Once the private key is known, you can decrypt session keys exchanged during the TLS handshake.
 - **Scope of impact:**
 - The compromised private key affects all TLS sessions established using the same public key.
 - It applies to past, present, and future sessions until the key pair is replaced.
 - Encrypted data intercepted at any time can be decrypted if the same key pair was used.
-

3. How would you thwart efforts to attack your web server connections using the techniques above?

- **Stop using MD5:**
 - All tools (Evilize, Selfextract, Web_version) demonstrate the vulnerability of MD5 to collisions.
 - Use stronger hash functions like SHA-256 or SHA-3 to mitigate collision attacks.
 - **Secure Certificate Management:**
 - Rotate certificates regularly to prevent long-term exposure from compromised keys.
 - Use certificate transparency logs to monitor and detect unauthorized certificate issuance.
 - **Implement Strong Cryptographic Standards:**
 - Use modern TLS versions (1.2 or higher) with secure cipher suites.
 - Prefer ephemeral key exchanges (e.g., ECDHE) to provide forward secrecy.
 - **Proactive Collision Detection:**
 - Tools like Web_version can identify potential vulnerabilities during development and deployment.
 - Always verify file integrity before using third-party executables.
-

4. Estimate the largest RSA key modulus you could factor with your available resources (list them) in a week.

- **Available resources:**
 - **CPU:** AMD Ryzen 9 7900X
 - **GPU:** NVIDIA RTX 3080 Ti
 - **RAM:** 32 GB
 - **Estimation:**
 - Without GPU acceleration, factoring beyond 384-bit keys within a week is unlikely.
 - With CUDA optimizations, factoring a 512-bit key may be possible but still challenging.
 - Keys larger than 512 bits would require substantially more computational power or distributed systems.
-

5. What information would you need to decrypt a TLS connection encrypted not with an RSA key exchange, but with a Diffie-Hellman or Elliptic Curve public key exchange (e.g., TLS 1.3 and later)?

- **For Diffie-Hellman (DH):**
 - Prime modulus (p), generator (g), and the public key (Y).
 - To decrypt the session, you would need the private key or have to solve the discrete logarithm problem.
- **For Elliptic Curve Diffie-Hellman (ECDH):**
 - Curve parameters and the public key point.
 - Decrypting the connection requires the private scalar, which involves solving the elliptic curve discrete logarithm problem (ECDLP), a computationally hard problem.
- **Key Considerations:**
 - Unlike RSA, DH and ECDH rely on problems with no known efficient classical solutions.
 - Without the private key or a significant cryptographic vulnerability, decrypting such sessions is practically impossible.

Sources:

Heninger, N. (n.d.). *Mining your PS and qs*. factorable.

<https://factorable.net/weakkeys12.conference.pdf>

Stevens, Marc & Bursztein, Elie & Karpman, Pierre & Albertini, Ange & Markov, Yarik. (2017). The First Collision for Full SHA-1. 570-596. 10.1007/978-3-319-63688-7_19.

<https://shattered.io/static/shattered.pdf>