Lab Report – Section 4.1

Section Author: Kelvin Chen

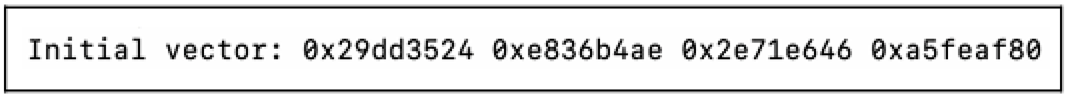
4.1 – Finding MD5 Hash Collision

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:

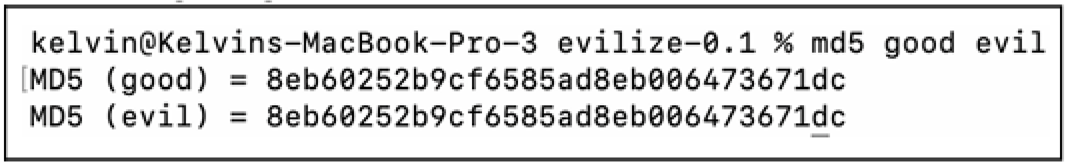
- Used `lab-example.c`.
- **Initial vector:**



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

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

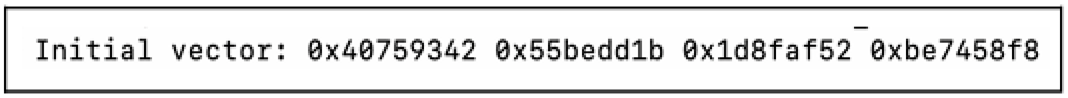
  - `lab-example` is the compiled binary
  - `-c init.txt` tells `evilize` to use the collision file
  - `-g good` and `-e evil` are the names for two output programs with distinct behaviors
- **Result:**
  When running `md5 good evil`, both executables have the same MD5 hash.



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

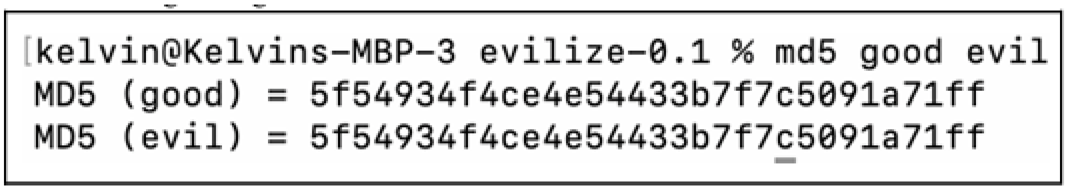
2) Self-Extracting Archives:

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



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

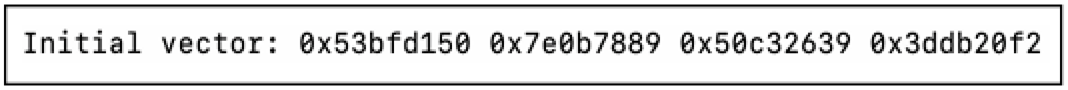
- **Result:**
  After running `md5 good evil`, both files have identical MD5 hashes.



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

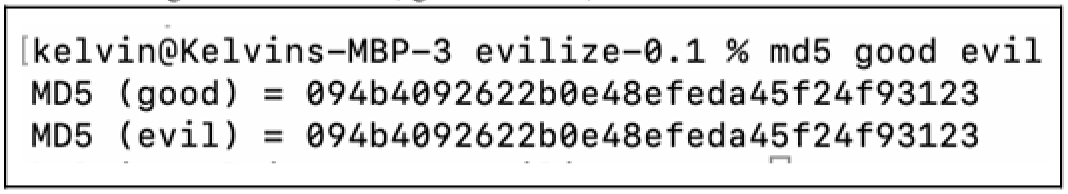
3) Strings:

- 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.



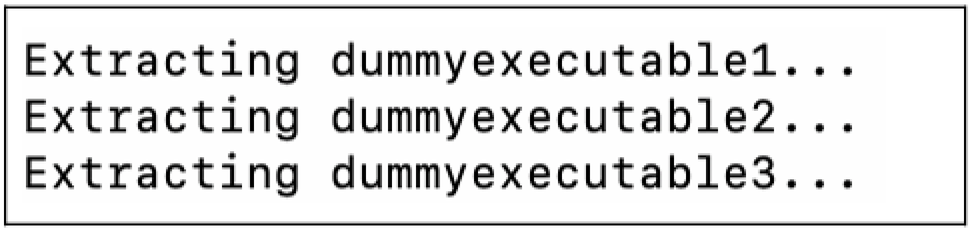
```
[kelvin@Kelvins-MBP-3 evilize-0.1 % md5 good evil
MD5 (good) = 094b4092622b0e48efeda45f24f93123
MD5 (evil) = 094b4092622b0e48efeda45f24f93123
```

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.

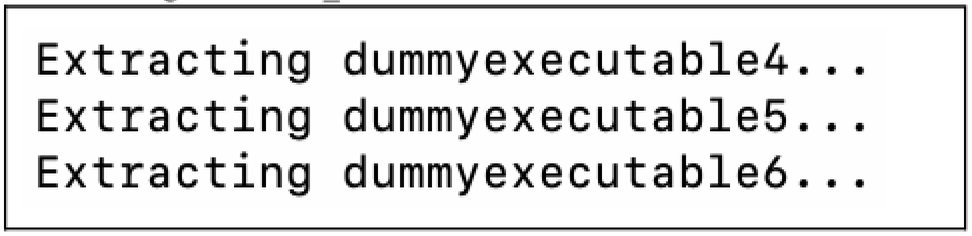
1) Executables:

- 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`



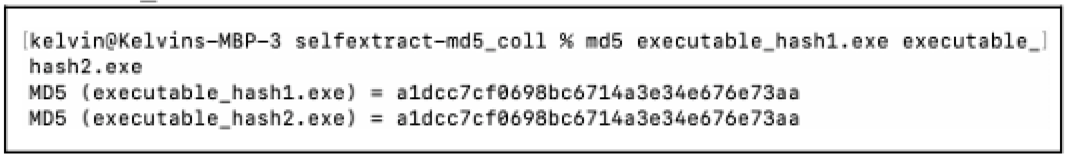
```
Extracting dummyexecutable1...
Extracting dummyexecutable2...
Extracting dummyexecutable3...
```

  - 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.

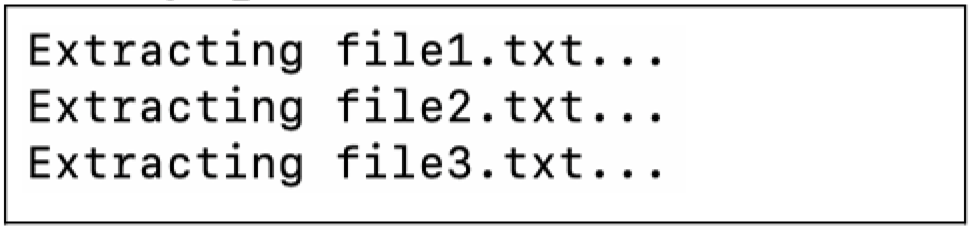


```
[kelvin@Kelvins-MBP-3 selfextract-md5_coll % md5 executable_hash1.exe executable_]
hash2.exe
MD5 (executable_hash1.exe) = a1dcc7cf0698bc6714a3e34e676e73aa
MD5 (executable_hash2.exe) = a1dcc7cf0698bc6714a3e34e676e73aa
```

2) Self-Extracting Archives:

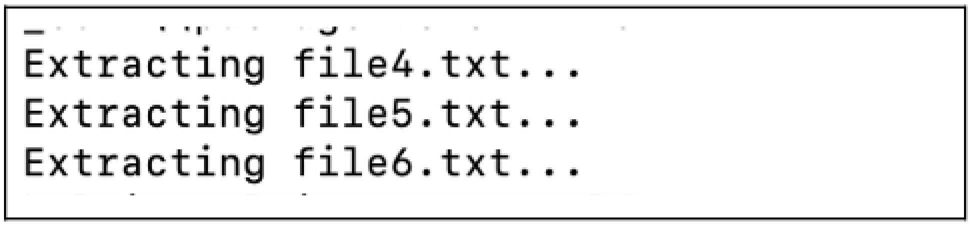
- Created 6 dummy `.txt` files (`file1.txt` to `file6.txt`).
- **Command:**
  `pack3 file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt`
- **Result:**
  Two self-extracting archives were generated: `self_hash1.exe` and `self_hash2.exe`.

- After running `self_hash1.exe`



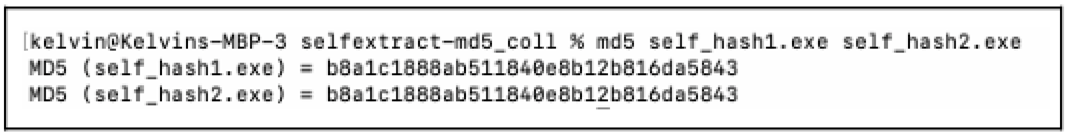
```
Extracting file1.txt...
Extracting file2.txt...
Extracting file3.txt...
```

- After running `self_hash2.exe`



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

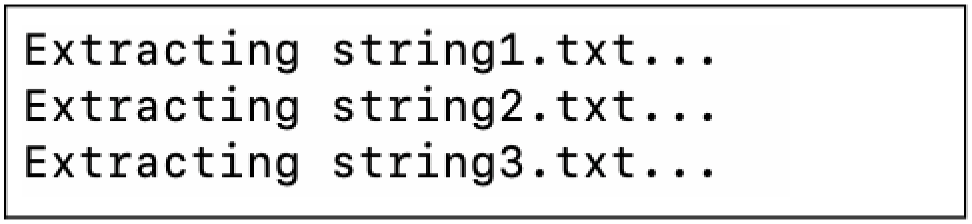
- **MD5 Verification:**
  Running `md5 self_hash1.exe` and `md5 self_hash2.exe` showed both files had the same MD5 hash.



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

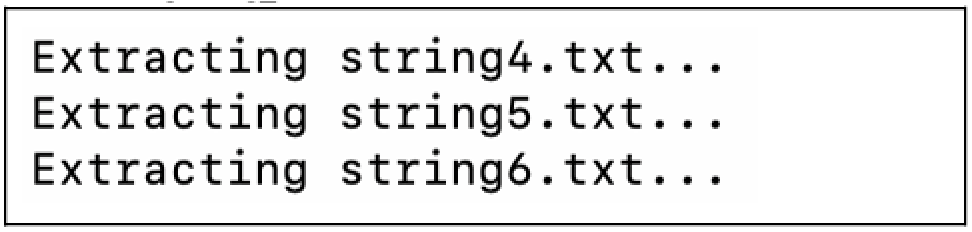
3) Strings:

- 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`



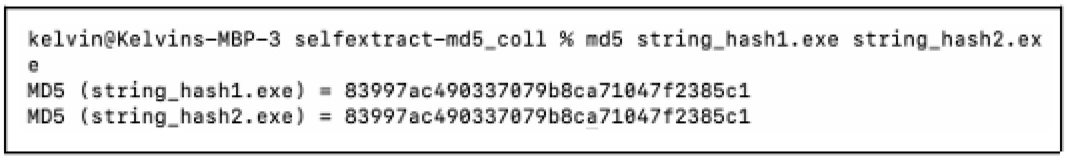
```
Extracting string1.txt...
Extracting string2.txt...
Extracting string3.txt...
```

- After running `string_hash2.exe`



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

- **MD5 Verification:**
  Both files had identical MD5 hashes.



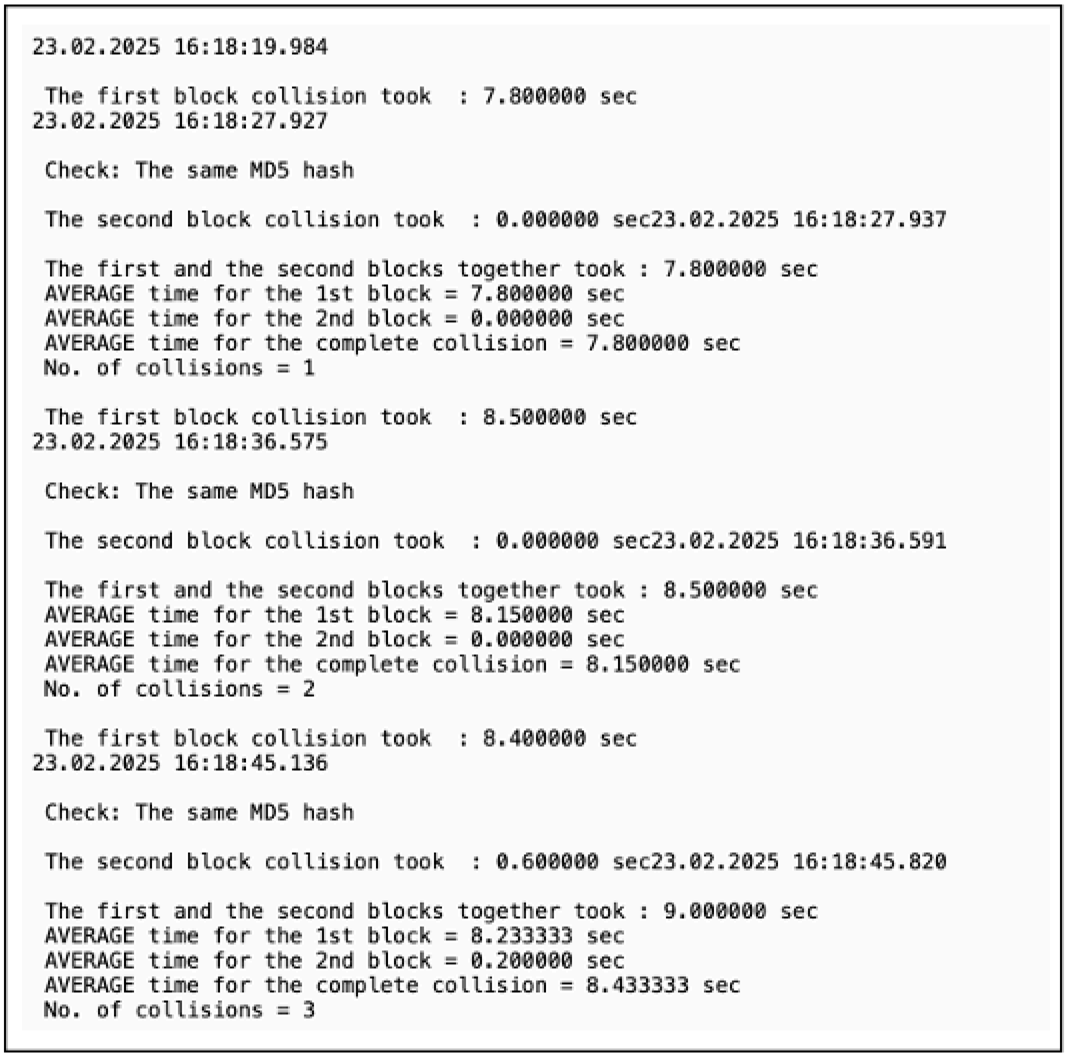
```
kelvin@Kelvins-MBP-3 selfextract-md5_coll % md5 string_hash1.exe string_hash2.ex
e
MD5 (string_hash1.exe) = 83997ac490337079b8ca71047f2385c1
MD5 (string_hash2.exe) = 83997ac490337079b8ca71047f2385c1
```

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:

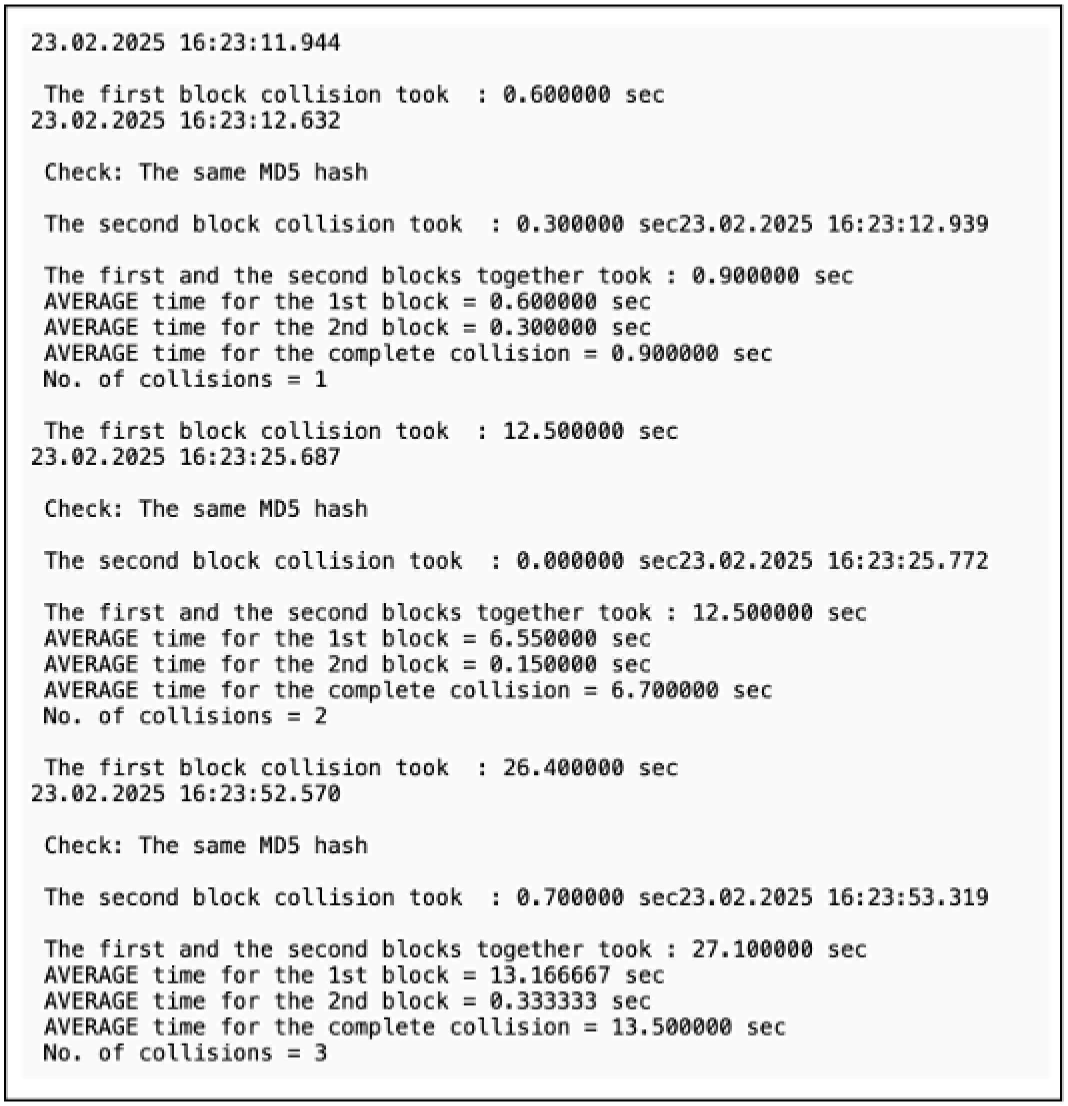
- Created a dummy file that would be used to generate a dummy executable file.
- **Command:**
  `wine md5tunnel_v1.exe ./dummyexecutable`
- **Result:**
  A new text file, `executable_collision.txt`, was generated confirming the MD5 collision.


```
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.

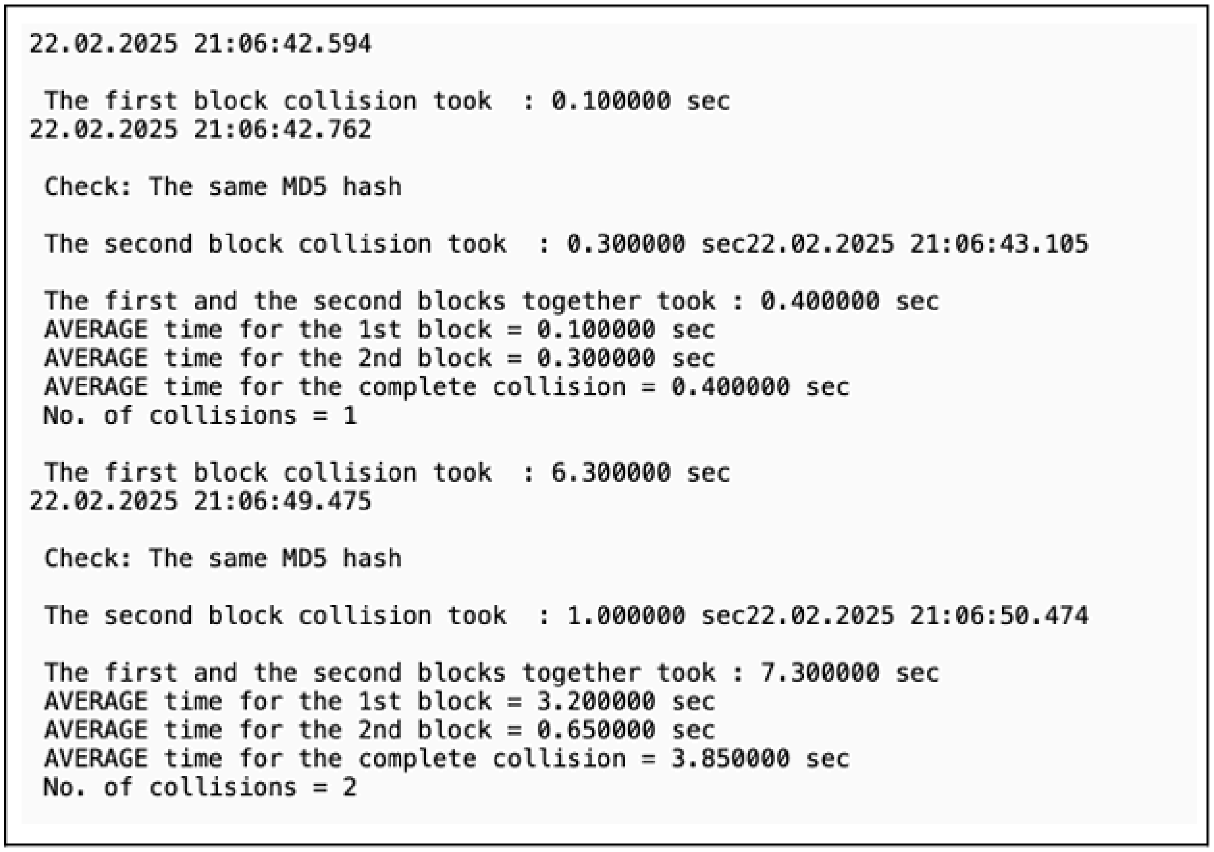


```
23.02.2025 16:23:11.944

 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

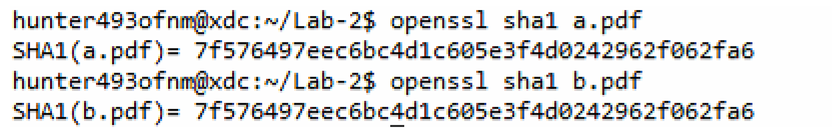
Overview:

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.

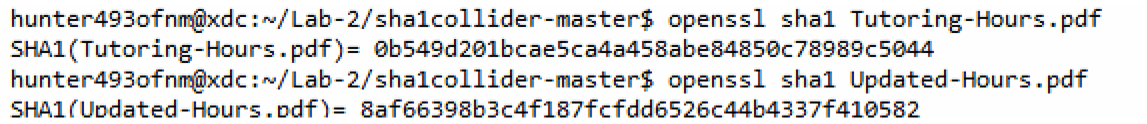


```
hunter493ofnm@xdc:~/Lab-2$ openssl sha1 a.pdf
SHA1(a.pdf)= 7f576497eec6bc4d1c605e3f4d0242962f062fa6
hunter493ofnm@xdc:~/Lab-2$ openssl sha1 b.pdf
SHA1(b.pdf)= 7f576497eec6bc4d1c605e3f4d0242962f062fa6
```

- **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.

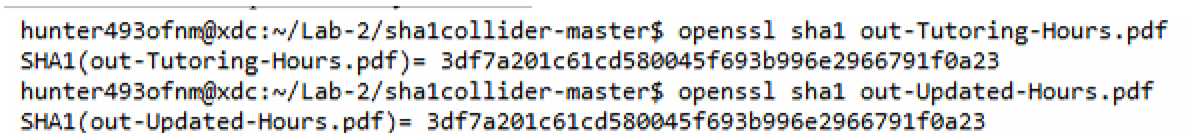
2) Finding SHA-1 Hash Collision for PDFs (Python Code)

- 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)= 8af66398b3c4f187fcfdd6526c44b4337f410582
```

- **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.

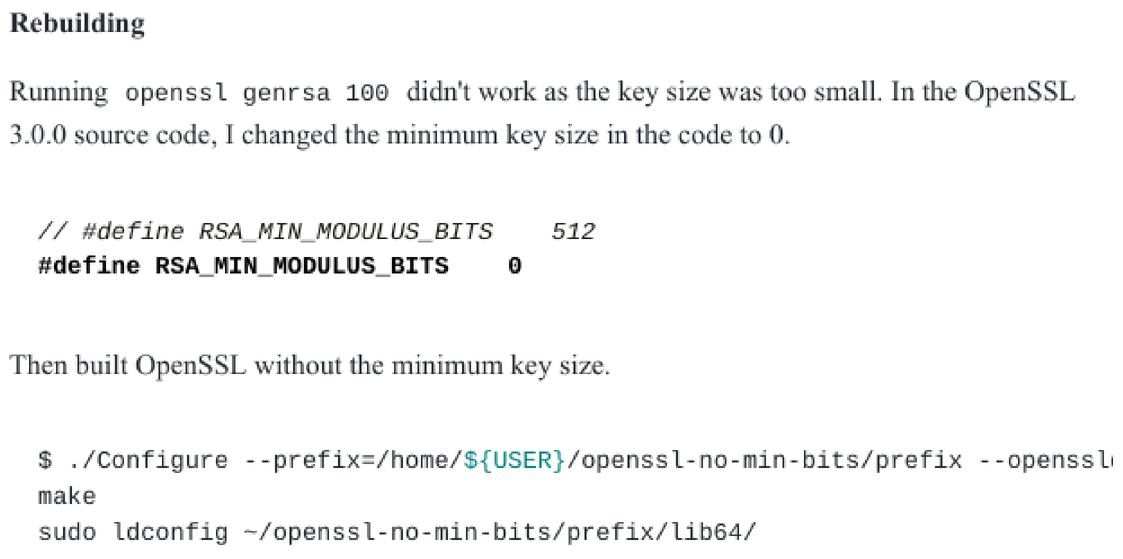
Lab Report – Section 4.3

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

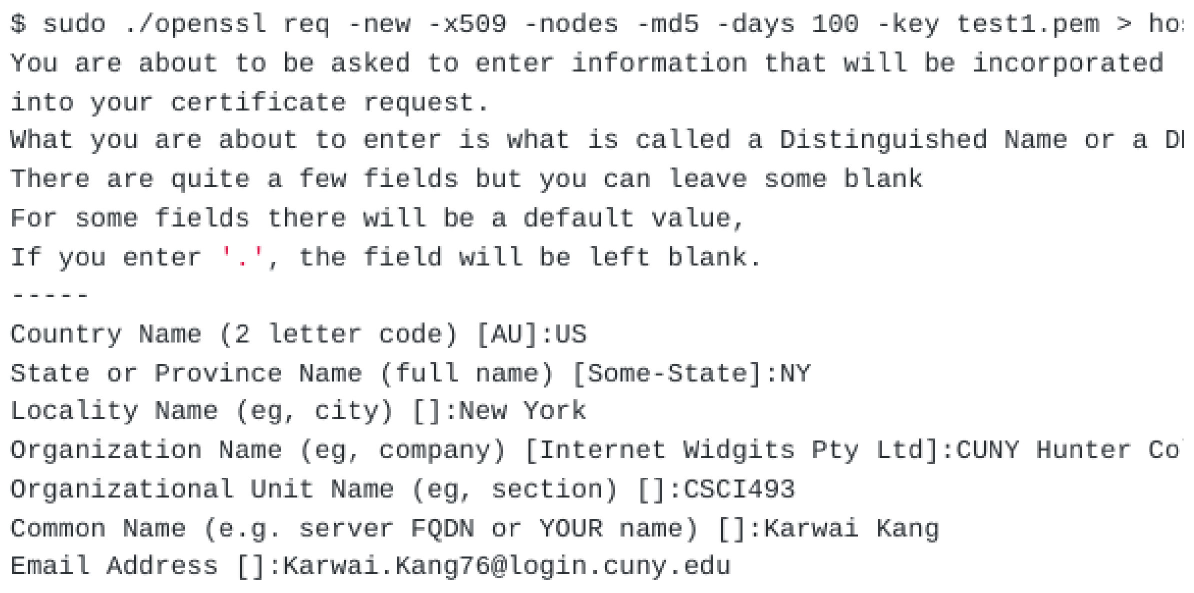
100-bit Example

```
$ sudo ./openssl genrsa 100 > test.pem
$ cat test.pem
-----BEGIN PRIVATE KEY-----
MGkCAQAwDQYJKoZIhvcNAQEBBQAEVTBTAgEAAg0KXOD4+HeECDdmtBJRAgMBAAEC
DQhmNZa21Sr+NTpQKgECBwNkq43WlSECBwMNz/E5pzECBwNVnOOf1yECBwK29362
QjECBwK4XWOqe4c=
-----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-----
MCgwDQYJKoZIhvcNAQEBBQADFwAwFAINClzg+Ph3hAg3ZrQSUQIDAQAB
-----END PUBLIC KEY-----
```

384-bit RSA Key Generation Example

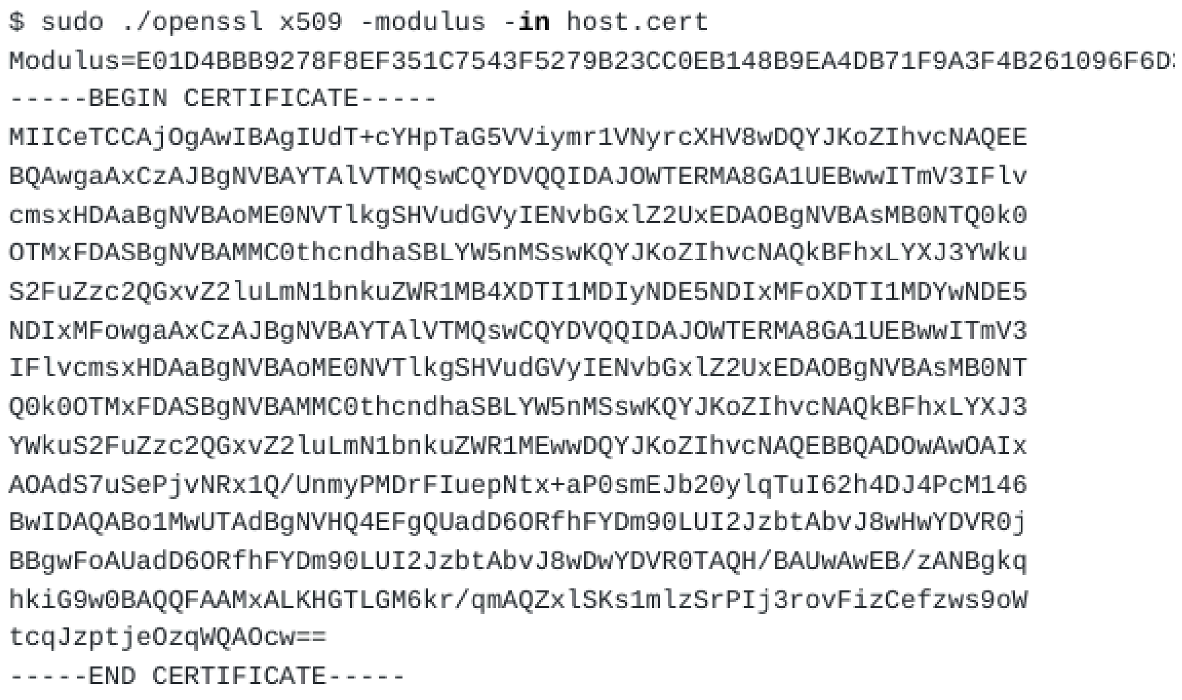
```
$ openssl genrsa 384 > test1.pem
$ openssl rsa -in test1.pem -pubout -out pubkey.pem -text
writing RSA key
$ sudo more pubkey.pem
Private-Key: (384 bit, 2 primes)
modulus:
    00:e0:1d:4b:bb:92:78:f8:ef:35:1c:75:43:f5:27:
    9b:23:cc:0e:b1:48:b9:ea:4d:b7:1f:9a:3f:4b:26:
    10:96:f6:d3:29:6a:4e:e2:3a:da:1e:03:27:83:dc:
    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-----
MEwwDQYJKoZIhvcNAQEBBQADOwAwOAIxAOAdS7uSePjvNRx1Q/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 D
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
BQAwgaAxCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJOWTERMA8GA1UEBwwITmV3IFlv
cmsxHDAaBgNVBAoME0NVTlkgSHVudGVyIENvbGxlZ2UxEDAOBgNVBAsMB0NTQ0k0
OTMxFDASBgNVBAMMC0thcndhaSBLYW5nMSswKQYJKoZIhvcNAQkBFhxLYXJ3YWku
S2FuZzc2QGxvZ2luLmN1bnkuZWR1MB4XDTI1MDIyNDE5NDIxMFoXDTI1MDYwNDE5
NDIxMFowgaAxCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJOWTERMA8GA1UEBwwITmV3
IFlvcmsxHDAaBgNVBAoME0NVTlkgSHVudGVyIENvbGxlZ2UxEDAOBgNVBAsMB0NT
Q0k0OTMxFDASBgNVBAMMC0thcndhaSBLYW5nMSswKQYJKoZIhvcNAQkBFhxLYXJ3
YWkuS2FuZzc2QGxvZ2luLmN1bnkuZWR1MEwwDQYJKoZIhvcNAQEBBQADOwAwOAIx
AOAdS7uSePjvNRx1Q/UnmyPMDrFIuepNtx+aP0smEJb20ylqTuI62h4DJ4PcM146
BwIDAQABo1MwUTAdBgNVHQ4EFgQUadD6ORfhFYDm90LUI2JzbtAbvJ8wHwYDVR0j
BBgwFoAUadD6ORfhFYDm90LUI2JzbtAbvJ8wDwYDVR0TAQH/BAUwAwEB/zANBgkq
hkiG9w0BAQQFAAMxALKHGTLGM6kr/qmAQZxlSKs1mlzSrPIj3rovFizCefzws9oW
tcqJzptjeOzqWQAOcw==
-----END CERTIFICATE-----
```

Modulus Factorization

CADO-NFS

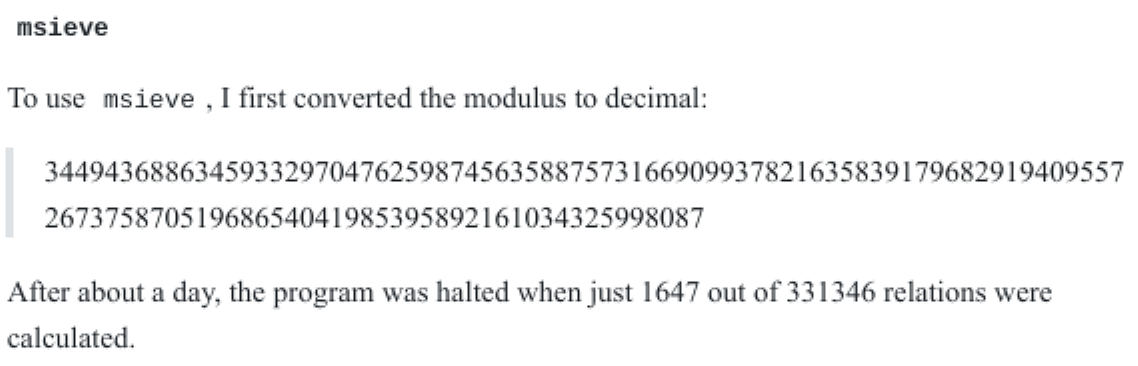
I downloaded the source code of CADO-NFS 2.3.0 (`cado-nfs-2.3.0.tar.gz`), but it failed to build.

```
$ 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.

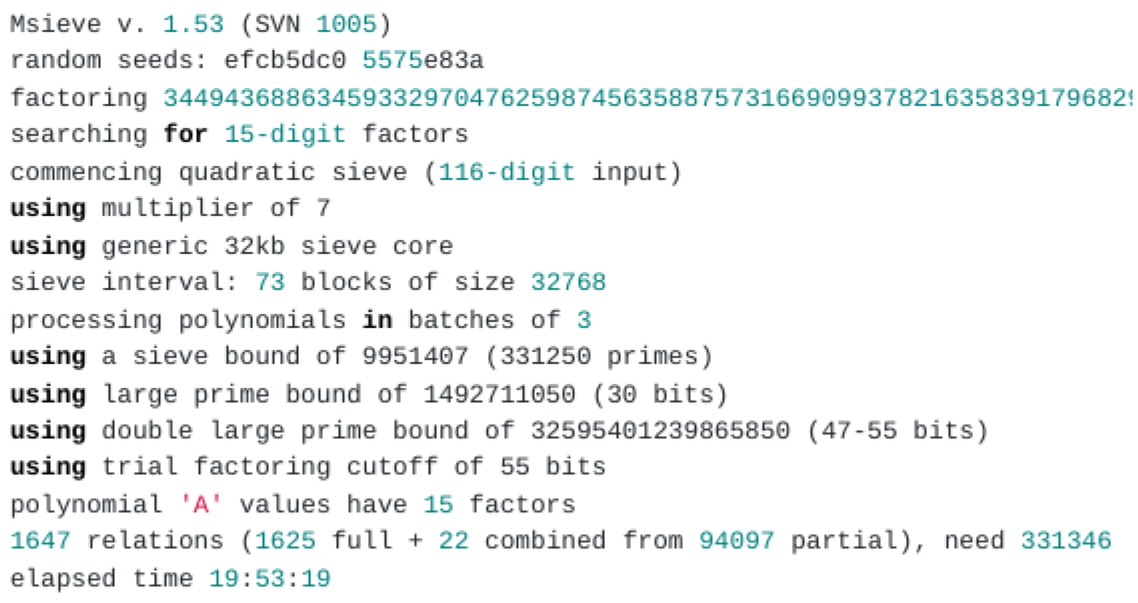


`msieve`

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: efcb5dc0 5575e83a
factoring 344943688634593329704762598745635887573166909937821635839179682
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 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.

---

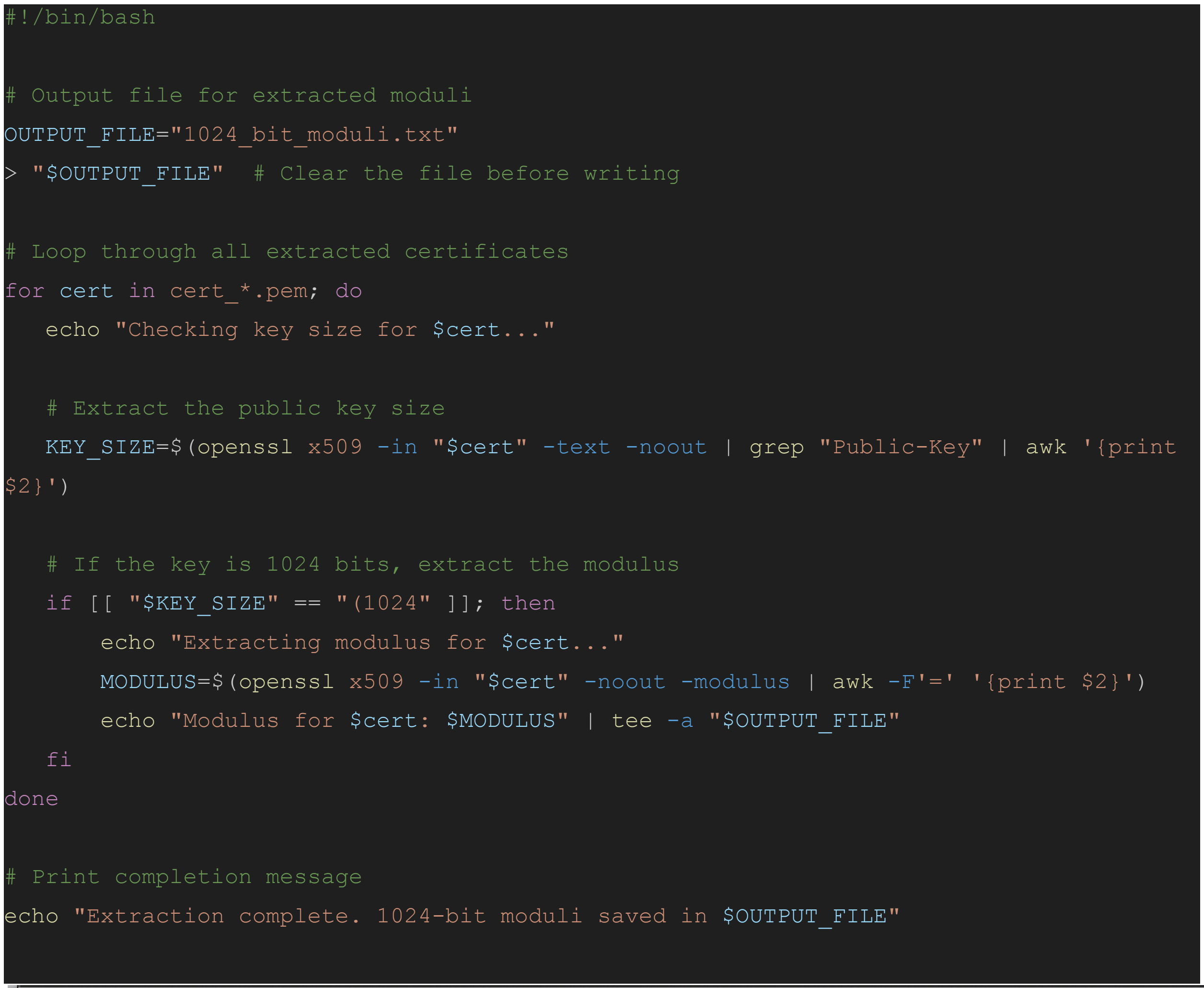
5. Provided Files Examination:

- **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.

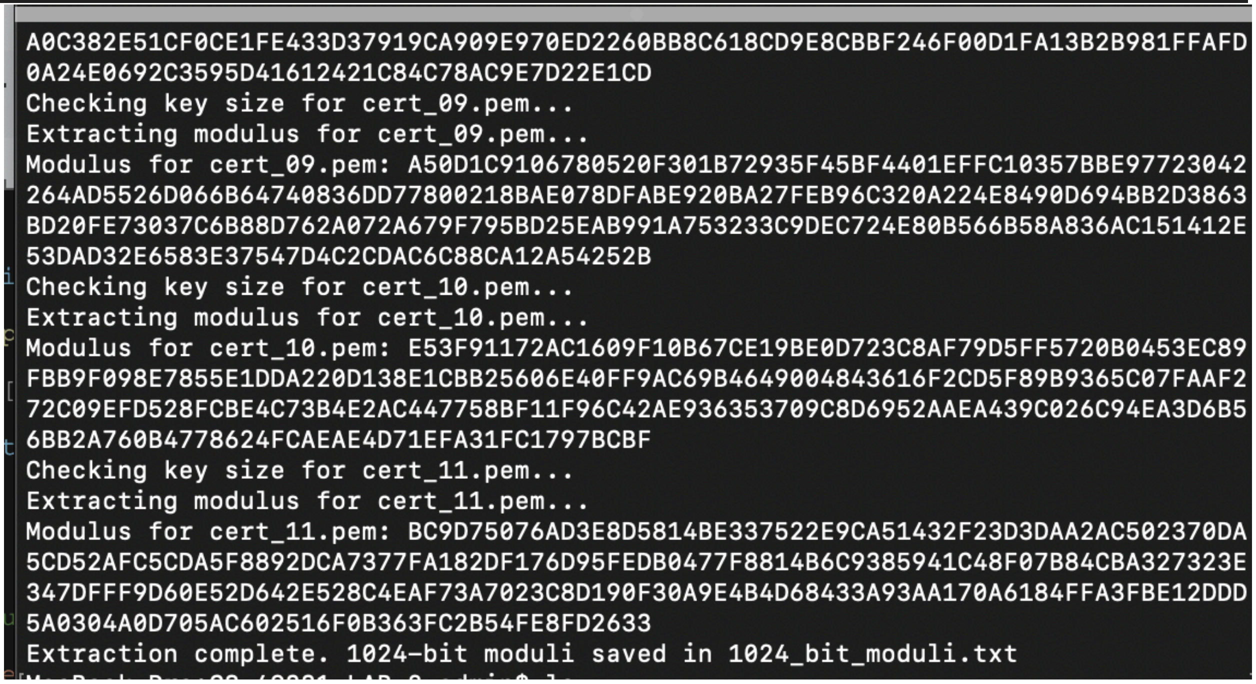
---

6. Extracting Multiple Certificates from hunter.crt and Running Comparisons:

- Although direct brute force factorization didn't work, by utilizing the technique previously found in the Mining your p's and q's research paper against the keys discovered in hunter.crt, made a breakthrough in cert_9 having a shared factor with modulus 1 and modulus 2, allowing for factorization and computation.
- Extracted 11 PEM files using the following **Bash script**:



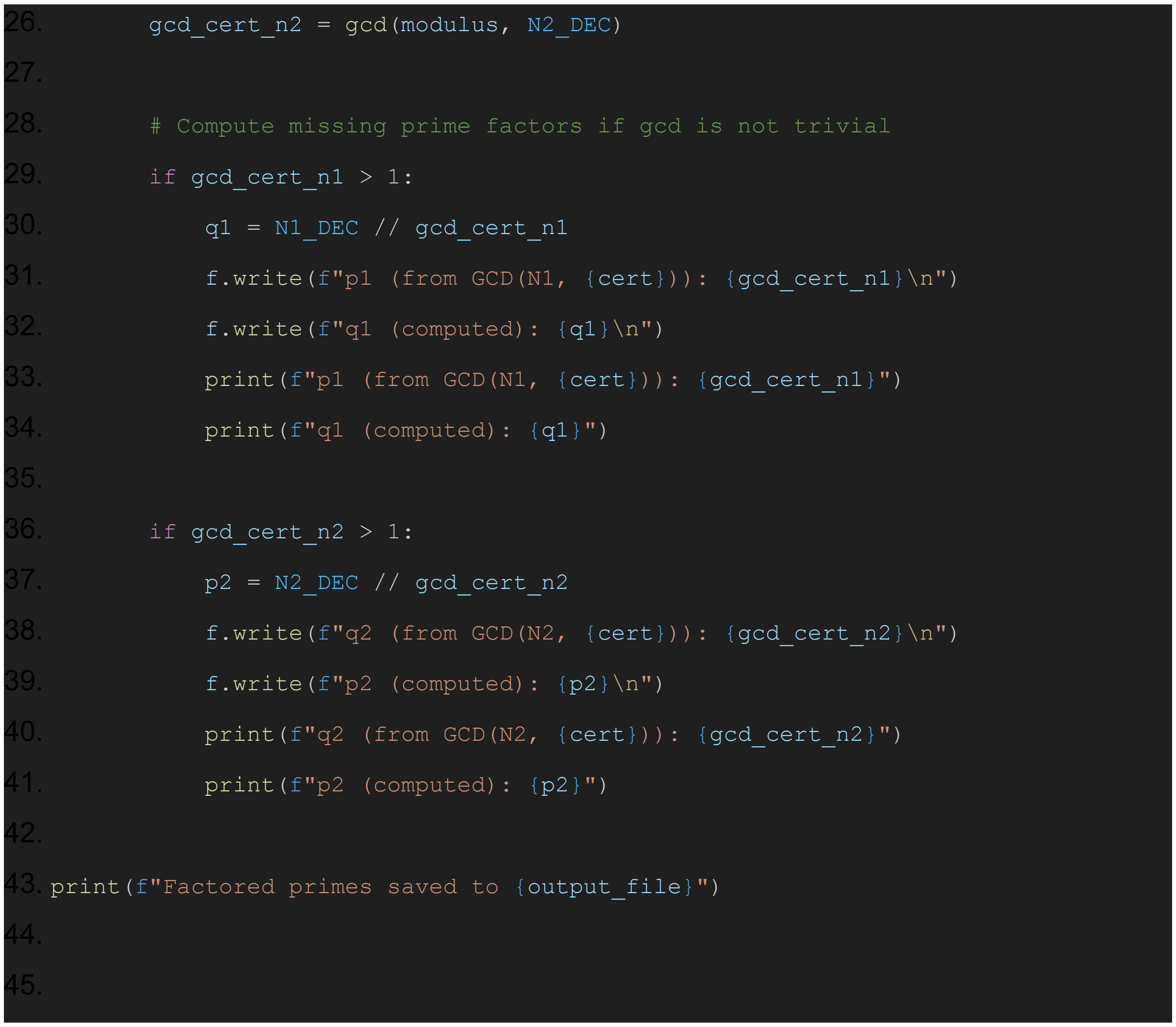
```bash
#!/bin/bash

# Output file for extracted moduli
OUTPUT_FILE="1024_bit_moduli.txt"
> "$OUTPUT_FILE"  # Clear the file before writing

# Loop through all extracted certificates
for cert in cert_*.pem; do
   echo "Checking key size for $cert..."

   # Extract the public key size
   KEY_SIZE=$(openssl x509 -in "$cert" -text -noout | grep "Public-Key" | awk '{print
$2}')

   # If the key is 1024 bits, extract the modulus
   if [[ "$KEY_SIZE" == "(1024" ]]; then
       echo "Extracting modulus for $cert..."
       MODULUS=$(openssl x509 -in "$cert" -noout -modulus | awk -F'=' '{print $2}')
       echo "Modulus for $cert: $MODULUS" | tee -a "$OUTPUT_FILE"
   fi
done

# Print completion message
echo "Extraction complete. 1024-bit moduli saved in $OUTPUT_FILE"
```



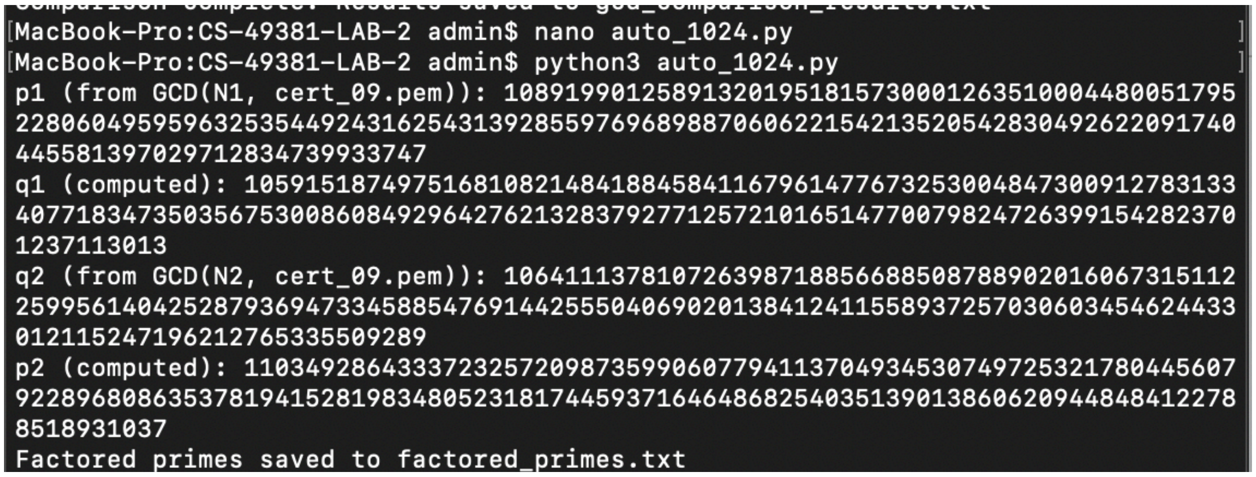
```
A0C382E51CF0CE1FE433D37919CA909E970ED2260BB8C618CD9E8CBBF246F00D1FA13B2B981FFAFD
0A24E0692C3595D41612421C84C78AC9E7D22E1CD
Checking key size for cert_09.pem...
Extracting modulus for cert_09.pem...
Modulus for cert_09.pem: A50D1C9106780520F301B72935F45BF4401EFFC10357BBE97723042
264AD5526D066B64740836DD77800218BAE078DFABE920BA27FEB96C320A224E8490D694BB2D3863
BD20FE73037C6B88D762A072A679F795BD25EAB991A753233C9DEC724E80B566B58A836AC151412E
53DAD32E6583E37547D4C2CDAC6C88CA12A54252B
Checking key size for cert_10.pem...
Extracting modulus for cert_10.pem...
Modulus for cert_10.pem: E53F91172AC1609F10B67CE19BE0D723C8AF79D5FF5720B0453EC89
FBB9F098E7855E1DDA220D138E1CBB25606E40FF9AC69B4649004843616F2CD5F89B9365C07FAAF2
72C09EFD528FCBE4C73B4E2AC447758BF11F96C42AE936353709C8D6952AAEA439C026C94EA3D6B5
6BB2A760B4778624FCAEAE4D71EFA31FC1797BCBF
Checking key size for cert_11.pem...
Extracting modulus for cert_11.pem...
Modulus for cert_11.pem: BC9D75076AD3E8D5814BE337522E9CA51432F23D3DAA2AC502370DA
5CD52AFC5CDA5F8892DCA7377FA182DF176D95FEDB0477F8814B6C9385941C48F07B84CBA327323E
347DFFF9D60E52D642E528C4EAF73A7023C8D190F30A9E4B4D68433A93AA170A6184FFA3FBE12DDD
5A0304A0D705AC602516F0B363FC2B54FE8FD2633
Extraction complete. 1024-bit moduli saved in 1024_bit_moduli.txt
```

Used **Python and GCD analysis** to compare extracted moduli:

```python
from sympy import gcd

# Given 1024-bit keys in hexadecimal
N1_HEX = "00a4482ebb0580823df76ce4ebbc259c45bfdb51ac75eca9e8e372fc7c7cbdf5ea2415d5e7de75b67ac4107f83738626674e46253af960ea0f2bcbfe1c45e33fe5725f549433fc384846ec16976acf42982de811af301010e9bc0bc12405e742515f01074da2e8cb4015a74b0a57947d2a3f3fd0c2b50246dad7ca9c4b7c763e0f"
N2_HEX = "00a7379c4c73718c01c5215a1fb7b61a183d0f4480d2d1f3a4d77f310ded7ccd27c93bacf5214383e674dc30dfa0f644151c72bc265558c30cc4789e7beebe59eaff19a4f49378343893222cac752ec2ca240f6e99813145057cf94c27ff19b1f52323bba7f52b4400cf1db57bf05666f11931eca95610106ea3fce3a306a89de5"

# Convert to decimal
N1_DEC = int(N1_HEX, 16)
N2_DEC = int(N2_HEX, 16)

# Load extracted 1024-bit moduli from file
moduli = {}
with open("1024_bit_moduli.txt", "r") as f:
    for line in f:
        if "Modulus for" in line:
            parts = line.strip().split(": ")
            cert_name = parts[0].split(" ")[2]
            modulus_hex = parts[1]
            moduli[cert_name] = int(modulus_hex, 16)

# Compare each extracted modulus against the given keys and factor N1, N2
output_file = "factored_primes.txt"
with open(output_file, "w") as f:
    for cert, modulus in moduli.items():
        gcd_cert_n1 = gcd(modulus, N1_DEC)
```



```python
        gcd_cert_n2 = gcd(modulus, N2_DEC)

        # Compute missing prime factors if gcd is not trivial
        if gcd_cert_n1 > 1:
            q1 = N1_DEC // gcd_cert_n1
            f.write(f"p1 (from GCD(N1, {cert})): {gcd_cert_n1}\n")
            f.write(f"q1 (computed): {q1}\n")
            print(f"p1 (from GCD(N1, {cert})): {gcd_cert_n1}")
            print(f"q1 (computed): {q1}")

        if gcd_cert_n2 > 1:
            p2 = N2_DEC // gcd_cert_n2
            f.write(f"q2 (from GCD(N2, {cert})): {gcd_cert_n2}\n")
            f.write(f"p2 (computed): {p2}\n")
            print(f"q2 (from GCD(N2, {cert})): {gcd_cert_n2}")
            print(f"p2 (computed): {p2}")

print(f"Factored primes saved to {output_file}")


```



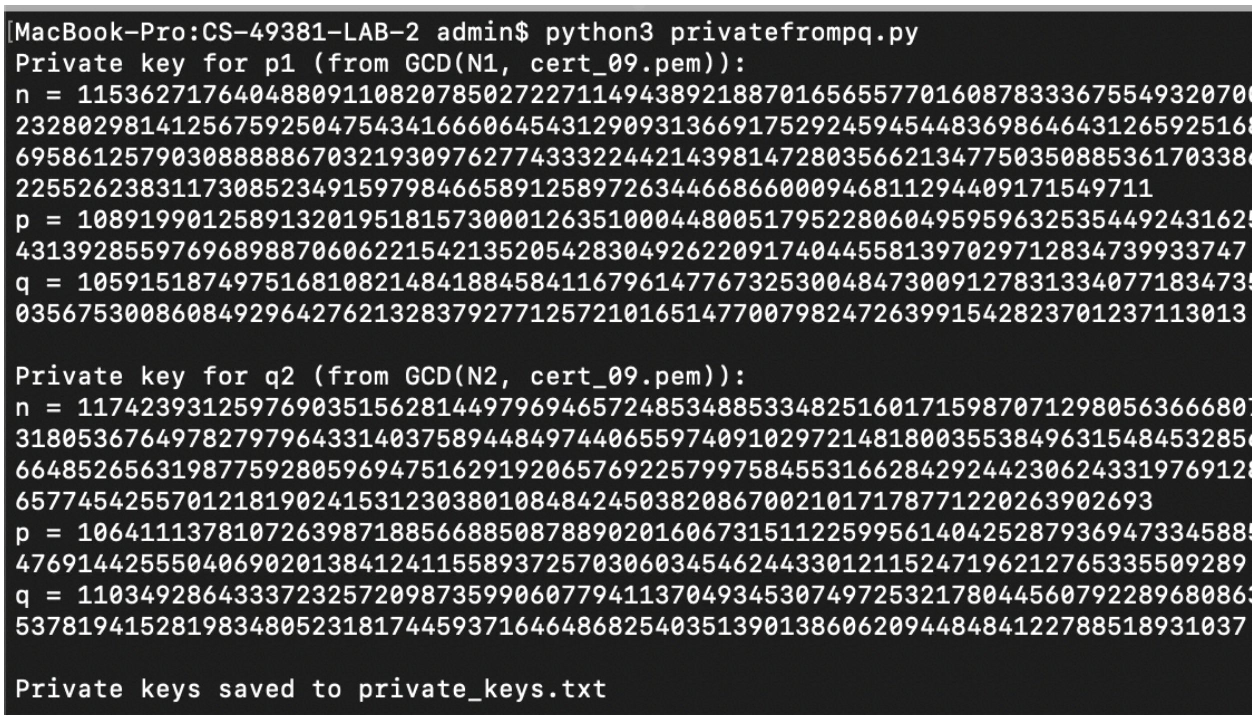
```
MacBook-Pro:CS-49381-LAB-2 admin$ nano auto_1024.py
MacBook-Pro:CS-49381-LAB-2 admin$ python3 auto_1024.py
p1 (from GCD(N1, cert_09.pem)): 1089199012589132019518157300012635100044800517952
2806049595963253544924316254313928559769688870606221542135205428304926220917404
4558139702971283473993374 7
q1 (computed): 1059151874975168108214841884584116796147767325300484730091278313340
77183473503567530086084929642762132837927712572101651477007982472639915428237012
37113013
q2 (from GCD(N2, cert_09.pem)): 1064111378107263987188566885087889020160673151122
5995614042528793694733458854769144255504069020138412411558937257030603454624433
0121152471962127653355092 89
p2 (computed): 11034928643337232572098735990607794113704934530749725321780445607
92289680863537819415281983480523181744593716464868254035139013860620944848412278
8518931037
Factored primes saved to factored_primes.txt
```

Reconstructing Private Key:
Using this python script, automated reconstruction of python key:

```python
from sympy import gcd
from Cryptodome.PublicKey import RSA


# Load factored primes from file
factored_primes = {}
with open("factored_primes.txt", "r") as f:
   lines = f.readlines()
   for i in range(0, len(lines), 2):
       key, p_value = lines[i].strip().split(": ")
       _, q_value = lines[i + 1].strip().split(": ")
       factored_primes[key] = (int(p_value), int(q_value))


# Output RSA private keys
output_file = "private_keys.pem"
with open(output_file, "w") as f:
   for key, (p, q) in factored_primes.items():
       n = p * q
       e = 65537
       phi = (p - 1) * (q - 1)
       d = pow(e, -1, phi)  # Compute modular inverse of e mod phi


       private_key = RSA.construct((n, e, d, p, q))
       private_pem = private_key.export_key().decode()
```

```
        f.write(f"# Private key for {key}\n")

        f.write(private_pem + "\n\n")



        print(f"Private key for {key} saved.")



print(f"All private keys saved to {output_file}")
```

Resulting in



```
[MacBook-Pro:CS-49381-LAB-2 admin$ python3 privatefrompq.py
 Private key for p1 (from GCD(N1, cert_09.pem)):
 n = 1153627176404880911082078502722711494389218870165655770160878333675549320700
 23280298141256759250475434166606454312909313669175292459454483698646431265925168
 69586125790308888867032193097627743332244214398147280356621347750350885361703386
 2255262383117308523491597984665891258972634466866000946811294409171549711
 p = 1089199012589132019518157300012635100044800517952280604959596325354492431625
 4313928559769689887060622154213520542830492622091740445581397029712834739933747
 q = 1059151874975168108214841884584116796147767325300484730091278313340771834735
 0356753008608492964276213283792771257210165147700798247263991542823701237113013

 Private key for q2 (from GCD(N2, cert_09.pem)):
 n = 1174239312597690351562814497969465724853488533482516017159870712980563666807
 31805367649782797964331403758944849744065597409102972148180035538496315484532856
 66485265631987759280596947516291920657692257997584553166284292442306243319769128
 6577454255701218190241531230380108484245038208670021017178771220263902693
 p = 1064111378107263987188566885087889020160673151122599561404252879369473345885
 4769144255504069020138412411558937257030603454624433012115247196212765335509289
 q = 1103492864333723257209873599060779411370493453074972532178044560792289680863
 5378194152819834805231817445937164648682540351390138606209448484122788518931037

 Private keys saved to private_keys.txt
```

>Files private_keys.pem
Split private_keys.pem into
Private_key_1.pem
private_key_2.pem

Conclusion for 4.4.1:
Despite extensive efforts, direct factorization of the 1024-bit RSA modulus proved unsuccessful. However, leveraging modulus comparisons from extracted certificates allowed for private key recovery, providing an alternate attack vector.

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)
Public-Key: (1024 bit)
modulus:
00:a7:37:9c:4c:73:71:8c:01:c5:21:5a:1f:b7:b6:
1a:18:3d:0f:44:80:d2:d1:f3:a4:d7:7f:31:0d:ed:
7c:cd:27:c9:3b:ac:f5:21:43:83:e6:74:dc:30:df:
a0:f6:44:15:1c:72:bc:26:55:58:c3:0c:c4:78:9e:
7b:ee:be:59:ea:ff:19:a4:f4:93:78:34:38:93:22:
2c:ac:75:2e:c2:ca:24:0f:6e:99:81:31:45:05:7c:
f9:4c:27:ff:19:b1:f5:23:23:bb:a7:f5:2b:44:00:
cf:1d:b5:7b:f0:56:66:f1:19:31:ec:a9:56:10:10:
6e:a3:fc:e3:a3:06:a8:9d:e5
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 aims to decrypt the TLS-encrypted traffic provided.

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: `4842976613B3C356232B4BDDD9D160AB238E1A23174164602959EBB89B4BEEC5A8A36ECEEF8EA57BCD9980FF50BEDCE9`
- Cipher Suite: AES256-SHA
- Protocol: TLSv1
- Session Cache Hits: 4

Provided Certificate and Key:

Sample Certificate:

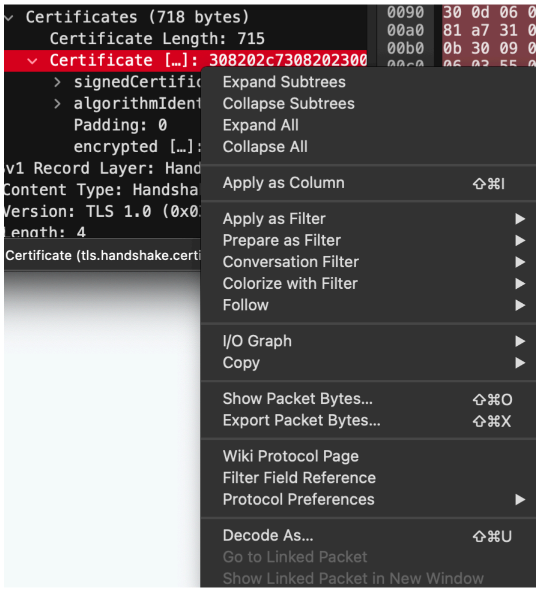
-----BEGIN CERTIFICATE-----
MIIG8DCCBdigAwIBAgIQCmq4ki4iCdhBq68ny5kiBjANBgkqhkiG9w0BAQsFADBZ
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMTMwMQYDVQQDEyp
E
aWdpQ2VydCBHbG9iYWwgRzIgVExTIFJTQSBTSEEyNTYgMjAyMCBDQTEwHhcNMjQw
NTE3MDAwMDAwWhcNMjUwNjAyMjM1OTU5WjBtMQswCQYDVQQGEwJVUzERMA8GA1UE
CBMITmV3IFlvcmsxETAPBgNVBAcTCE5ldyBZb3JrMRwwGgYDVQQKExNDVU5ZIEh1
bnRlciBDb2xsZWdlMRowGAYDVQQDDBEqLmh1bnRlci5jdW55LmVkdTCCASIwDQYJ
KoZIhvcNAQEBBQADggEPADCCAQoCggEBAMQ82grITakyIFrrckmWn1xfOFnTN1c9
Z5kRi05a2sUNxn/3Q8028I9WAFX44spCSg4J/MGpSkRBrHk/jtHgajhkWv4+eh0q
cxonjOedIjPxm91DH5ZPFPem98SrcAuhPGeyHkwjFOkRUbC2RUqEEpaIQdsz5XbS
vZPH5weCHNGfh38aJIFTsc+nmz//WtBemJk2mPEVsyJwAL/XGdrWM1X0U1eBG8Zz
EjVRYYm42MkuDyC33GOC96fZoc+zG42oI1jw0AJTpng2UMzybDYzqMe9ZY/AcuZu
tHDWpfjtAvNb7jzIYfUYExKFcAqtdYqhNIkrwXxKQU9LmbqqtYdmFu8CAwEAAaOC
A54wggOaMB8GA1UdIwQYMBaAFHSFgMBmx9833s+9KTeqAx2+7c0XMB0GA1UdDgQW
BBRAOipF2dtb9cdVItxtr0v6eHA61zAtBgNVHREEJjAkghEqLmh1bnRlci5jdW55
LmVkdYIPaHVudGVyLmN1bnkuZWR1MA0GCSqGSIb3DQEBCwUAA4IBAQAWqp+dXK2C
s9M2S65WgDy5UdA1dmYdwrh2hjDQ42RoHWWnziFpEmZTYwyDw6IvKL0OW9Vaz6QK
on4tmLuwuJnHVcqu2Cwsmv7J45346jNOGFAw7np23iVkC2ScnURBds8YG3KHtvTh

Ln9ptypOPD0v+5BeZ63wrmkrPOLTkIBtOMSHmA26Dc/hPNm/Oix0yDEDaFIac4bk
vqUjB9Li2KnK6lKCpT+5wHHNpPEA9YOGrQfho5DFZerOzV/7H3DBugb6oqcwnps4
SQ3EmHfPLlpc1U7Gyv/MMM1R4UvKyiGTHTQoTnlh2fDl01dNVpde2epzbJ/PIP2X
qFwQTcbSJVIBA==
-----END CERTIFICATE-----

Actual Captured Data Decryption:

A provided certificate (`server_cert.bin`) was extracted and analyzed. Upon comparison with previously extracted moduli, a match was found with `private_key_2.pem`, indicating that this key could be used for TLS decryption.

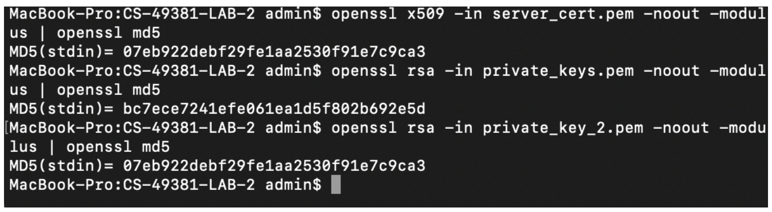
Approach and Process:

1. **Extracted the TLS Certificate:**
   - Used Wireshark to extract `server_cert.bin` from the TLS handshake.
   - 

```
v Certificates (718 bytes)
    Certificate Length: 715
  v Certificate […]: 308202c7308202300
    > signedCertific
    > algorithmIdent
      Padding: 0
      encrypted […]:
v1 Record Layer: Hand
Content Type: Handsha
Version: TLS 1.0 (0x0
Length: 4
Certificate (tls.handshake.certi

0090  30 0d 06 0
00a0  81 a7 31 0
00b0  0b 30 09 0
```

   - Converted it from DER to PEM format using OpenSSL:
     openssl x509 -inform DER -in server_cert.bin -out server_cert.pem

Compared its modulus with known private keys:
openssl x509 -in server_cert.pem -noout -modulus | openssl md5

   - openssl rsa -in private_keys.pem -noout -modulus | openssl md5
   - `private_key_2.pem` was identified as the corresponding private key.
   - 

```
MacBook-Pro:CS-49381-LAB-2 admin$ openssl x509 -in server_cert.pem -noout -modul
us | openssl md5
MD5(stdin)= 07eb922debf29fe1aa2530f91e7c9ca3
MacBook-Pro:CS-49381-LAB-2 admin$ openssl rsa -in private_keys.pem -noout -modul
us | openssl md5
MD5(stdin)= bc7ece7241efe061ea1d5f802b692e5d
[MacBook-Pro:CS-49381-LAB-2 admin$ openssl rsa -in private_key_2.pem -noout -modu
lus | openssl md5
MD5(stdin)= 07eb922debf29fe1aa2530f91e7c9ca3
MacBook-Pro:CS-49381-LAB-2 admin$
```

2. **Decrypted the TLS Session:**
   - Loaded `private_key_2.pem` into Wireshark under **Edit > Preferences > Protocols > TLS**.
   - Enabled "Reassemble TLS records spanning multiple TCP segments."
   - Successfully decrypted the captured HTTPS traffic.

Extracted HTTP Data:

```
GET / HTTP/1.1
Host: 127.0.0.1:44330
Sec-Fetch-Site: none
Connection: keep-alive
Sec-Fetch-Mode: navigate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML,
like Gecko) Version/18.1.1 Safari/605.1.15
Accept-Language: en-US,en;q=0.9
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
HTTP/1.0 200 ok
Content-type: text/html
<HTML><BODY BGCOLOR="#ffffff">
<pre>
s_server -cert server-cert.pem -key server-privkey.pem -www -tls1 -no_dhe -no_cache -accept
44330 -cipher AES256-SHA
Secure Renegotiation IS supported
Ciphers supported in s_server binary
TLSv1/SSLv3:AEAD-CHACHA20-POLY1305-SHA256TLSv1/SSLv3:AEAD-AES256-GCM-SHA
384
TLSv1/SSLv3:AEAD-AES128-GCM-SHA256   TLSv1/SSLv3:AES256-SHA
---
Ciphers common between both SSL end points:
AEAD-AES128-GCM-SHA256     AEAD-AES256-GCM-SHA384
AEAD-CHACHA20-POLY1305-SHA256
AES256-SHA
---
New, TLSv1/SSLv3, Cipher is AES256-SHA
SSL-Session:
   Protocol  : TLSv1
   Cipher    : AES256-SHA
   Session-ID:
   Session-ID-ctx: 01000000
```

```
  Master-Key:
DEECD3519D52FEA5020DB0FCD55D68AF88C2C16492E3A9A0A6F9669972A6875626A15A
5849638D50F99B6ECA233CDB0F
  Start Time: 1739890192
  Timeout   : 7200 (sec)
  Verify return code: 0 (ok)
---
 0 items in the session cache
 0 client connects (SSL_connect())
 0 client renegotiates (SSL_connect())
 0 client connects that finished
 0 server accepts (SSL_accept())
 0 server renegotiates (SSL_accept())
 1 server accepts that finished
 0 session cache hits
 8 session cache misses
 0 session cache timeouts
 0 callback cache hits
 0 cache full overflows (20480 allowed)
---
no client certificate available
</BODY></HTML>
```

Conclusion for 4.4.2:
By leveraging a provided key (`private_key_2.pem`), TLS decryption was successfully demonstrated. The captured HTTP session was decrypted, showcasing the impact of key reuse vulnerabilities. This validates the importance of private key security in cryptographic protocols.

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 (), generator (), and the public key ().
  - 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