**CSCI 400-01 Group 5**

Cryptography Lab 02 Report

**Topic:** Cryptography, Collisions, MITM, and High Factors

Prof. Dietrich

September 5, 2019

Lab members and contributions:

**Victoria Lau: 4.1** Finding MD5 hash collisions **& 4.2** Finding SHA-1 hash collisions & **5**

**James Stefanik: 4.3** Man-in-the-middle attacks on SSL/TLS & **5**

**Jae Cho: 4.4** Web site SSL/TLS certificates & **5**

**Aaron Delgado: 4.5** Attacking the RSA modulus from a different angle & Decrypting SSL/TLS traffic & **5**

1. General description

This lab explores hash collisions, man-in-the-middle attacks (MITM, only cryptographic), and website certificates.

For hash collisions: contrary to dictionary attacks, where we are trying to find M: H(M)

= C, here given M, we are looking for M’: H(M) = H(M’).

For website certificates: we want to challenge the security of a certificate. With a weak key, we can attack the connection and decrypt network traffic.

Except maybe for the first item, these exercises require proper planning, computational power, and time management.

2. File with samples and results

The files are in lab-crypto1.zip. It contains most files to get you started. In some cases, you still have to develop some scripts or write your own code to improve the results.

**3. Attacking systems**

We attack systems at multiple levels: from challenging the trust in integrity primitives such as MD5, we move to question the network itself and its ability to provide a secure channel since there may be an attacker in the middle. Further, the trust instilled in us by a certificate needs to be properly constructed: picking a low-strength key could allow for an attacker to extract enough information to eavesdrop on the connection secured by SSL/TLS.

**Tasks**

4.1 Finding MD5 hash collisions - by Victoria Lau

**Lab Setup:**

OS: iMac macOS Mojave

Processor: 1.6 GHz Intel Core i5

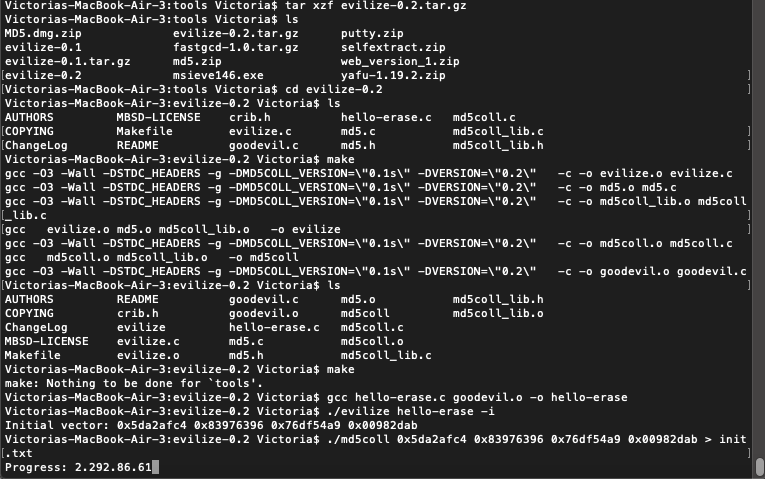
Memory: 8 GB 1867 MHz DDR3

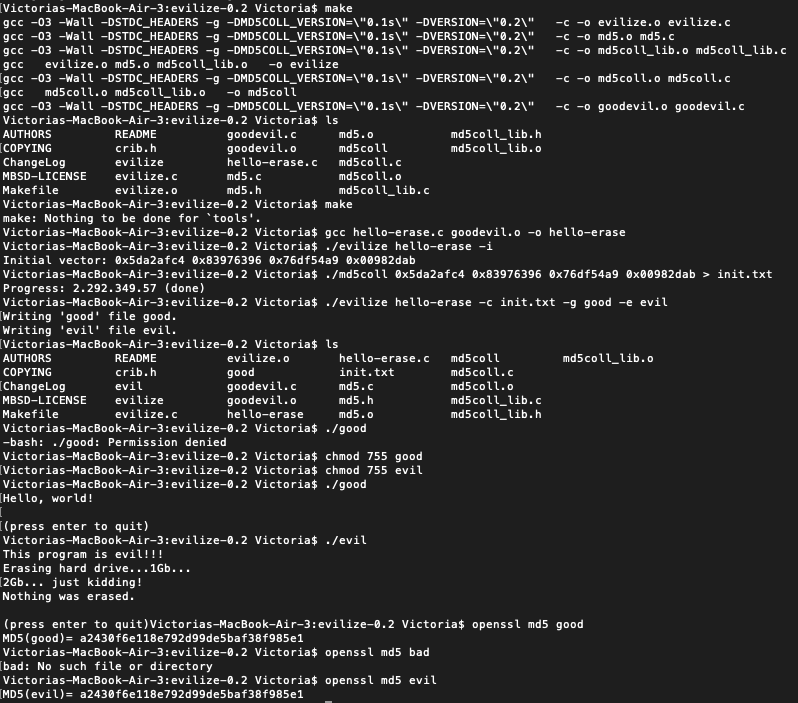
Tools used: Evilize, Selfextract, Webversion

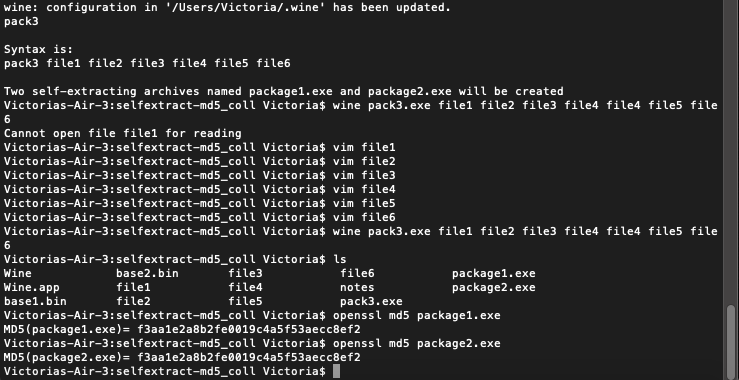
To complete this portion of the lab, I used the built-in terminal on OSX. After unzipping the provided tools for evilize, selfextract and webversion, I referred to index.htm to guide me through the tasks.

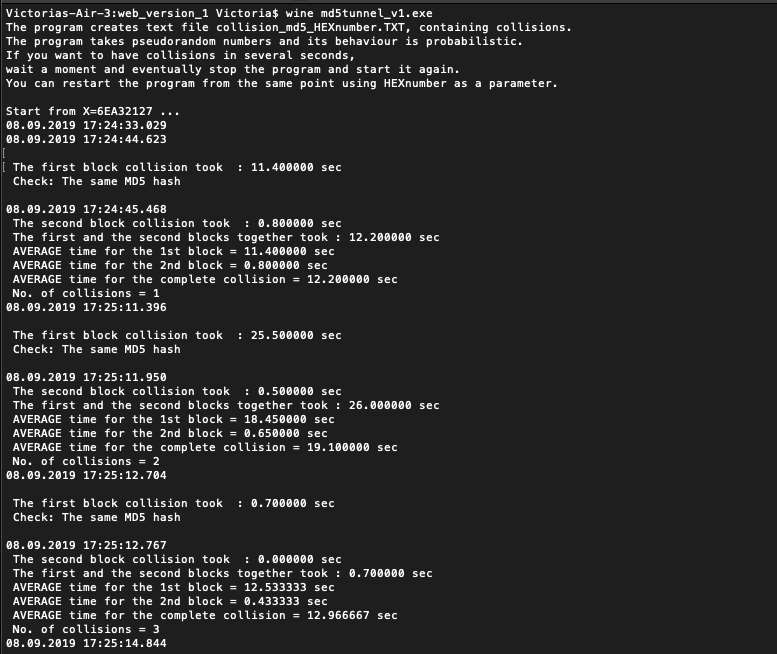
For my executable files, I had OpenSSL already downloaded and first compiled the evilize tool and found that it did not work as the evilize 0.1 tool was outdated. I found evilize 0.2\* online and downloaded that zip file instead and was able to create my executable files with the same MD5 hash.

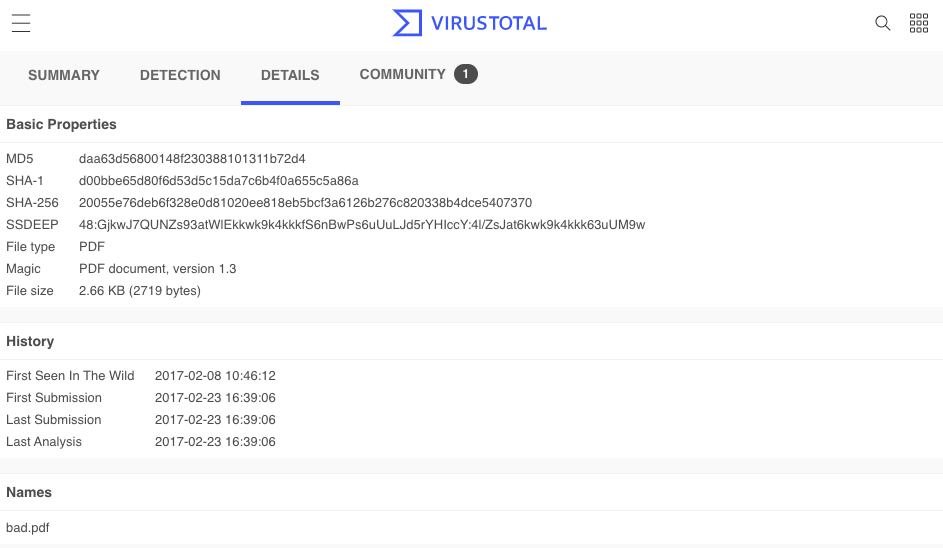
\*<https://www.mscs.dal.ca/~selinger/md5collision/>

The screenshot below shows the compilation of the evilize 0.2 tool and then the implementation to create the MD5 collision for the good and bad executable files. It took several minutes for the MD5 collision to be created. 

I then ran both the good and bad files to compare the MD5 hash code and indeed found it to be the same for both, but with different behaviors. As shown, the good executable file printed “Hello, world!” while the bad executable file printed “Erasing hard drive”. 

To find the MD5 hash for the self-extracting archives, I used the provided selfextract tool in order to generate two self-extracting archives with the same MD5 hash code. In order to run .exe files on an OSX interface, I had to download Winebottler and Homebrewer which would allow me to run package1.exe and package2.exe on my Mac terminal. I found the MD5 hash code to be the same.

I used the provided Webversion tool to find the MD5 hash collision for strings. The commands took a few seconds to run. For each collision, the first block collision took more time than the second block collision. The MD5 hash was the same. 

Depending on the hash function, different commands can be extended for different hash functions such as SHA256, SHA384, and SHA512. There are also online tools provided where the hash code can be looked up. VirusTotal is a tool I’ve worked with that allowed me to view the MD5, SHA1, and SHA256 hash code. The screenshot below is an example of what pops up on VirusTotal using the provided bad.pdf. 

4.2 Finding SHA-1 hash collisions - by Victoria Lau

**Lab Setup:**

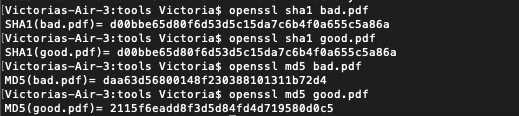
OS: iMac macOS Mojave

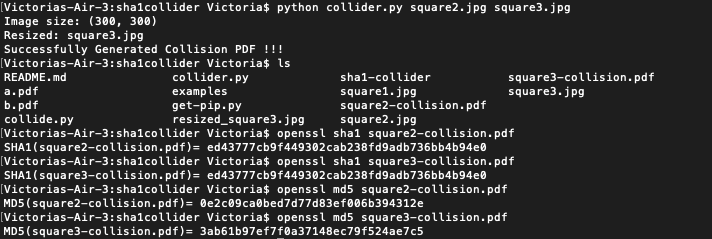
Processor: 1.6 GHz Intel Core i5

Memory: 8 GB 1867 MHz DDR3

Tools used: Offline SHA1 Collider, Online SHA1 Collider (alf.nu/SHA1), SHA1 Collider Python

To complete this portion of the lab, I used the built-in terminal on OSX. After unzipping the SHA1 collider tool, I referred to index.htm to guide me through the tasks.

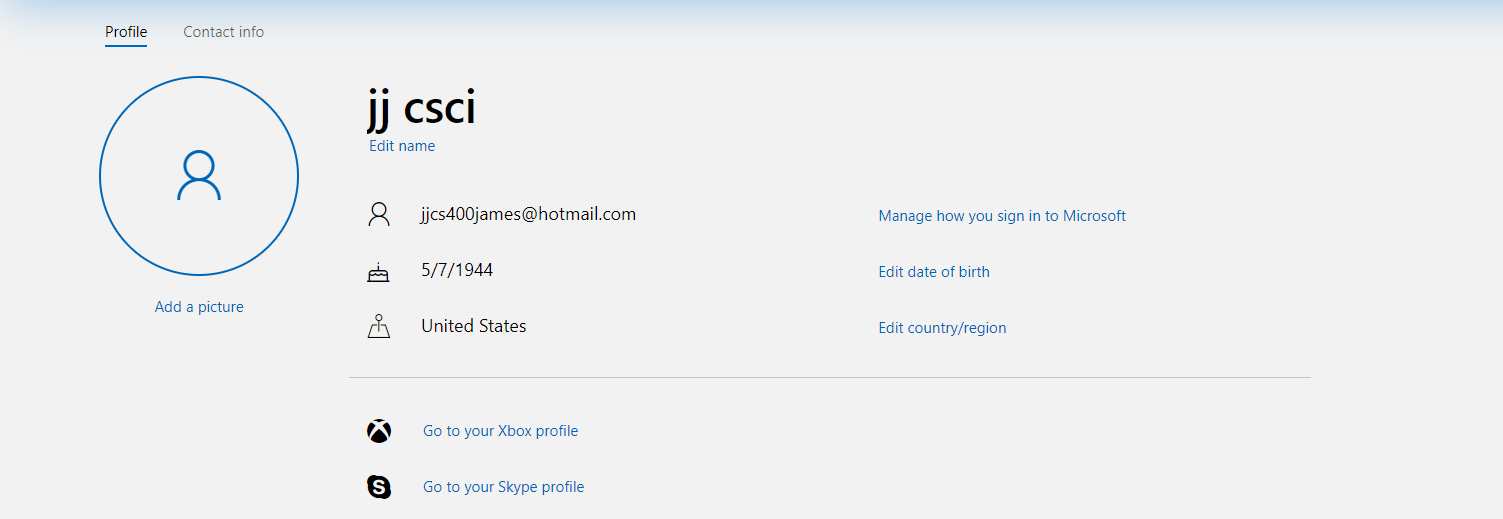
Using the provided SHA1 collider tool, after unzipping the file and compiling it, I was able to find that the SHA1 hash for the provided good and bad pdf to be the same. When comparing the MD5 hash, however, the good and bad pdf was different. 

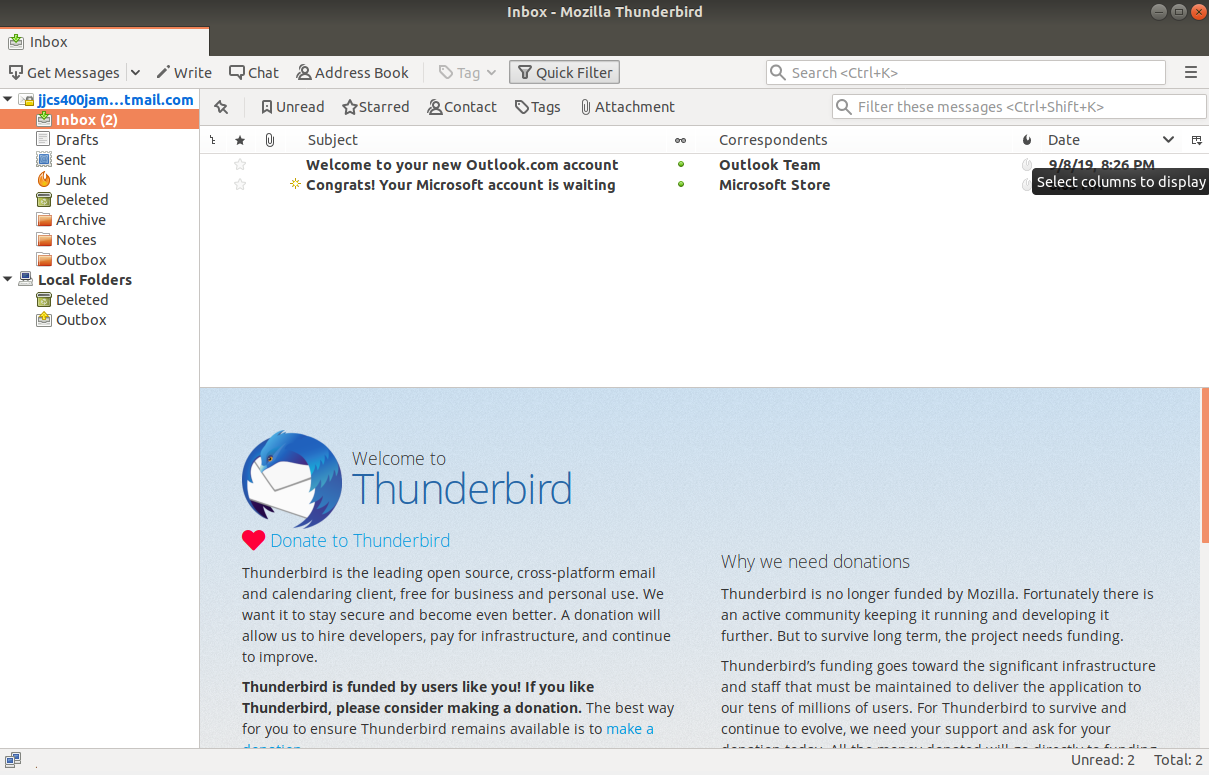
I then did a quick google image search of squares to use as my two .jpg files and saved them. Using the online tool from alf.nu/SHA1, I created two new pdfs of square2 and square3. I also used the python code with the newly created square2-collision and square3-collision pdf files. It was found that the SHA1 hash code matched for both squares while the MD5 hash was different. 

4.3 Man-in-the-middle attacks on SSL/TLS - by James Stefanik

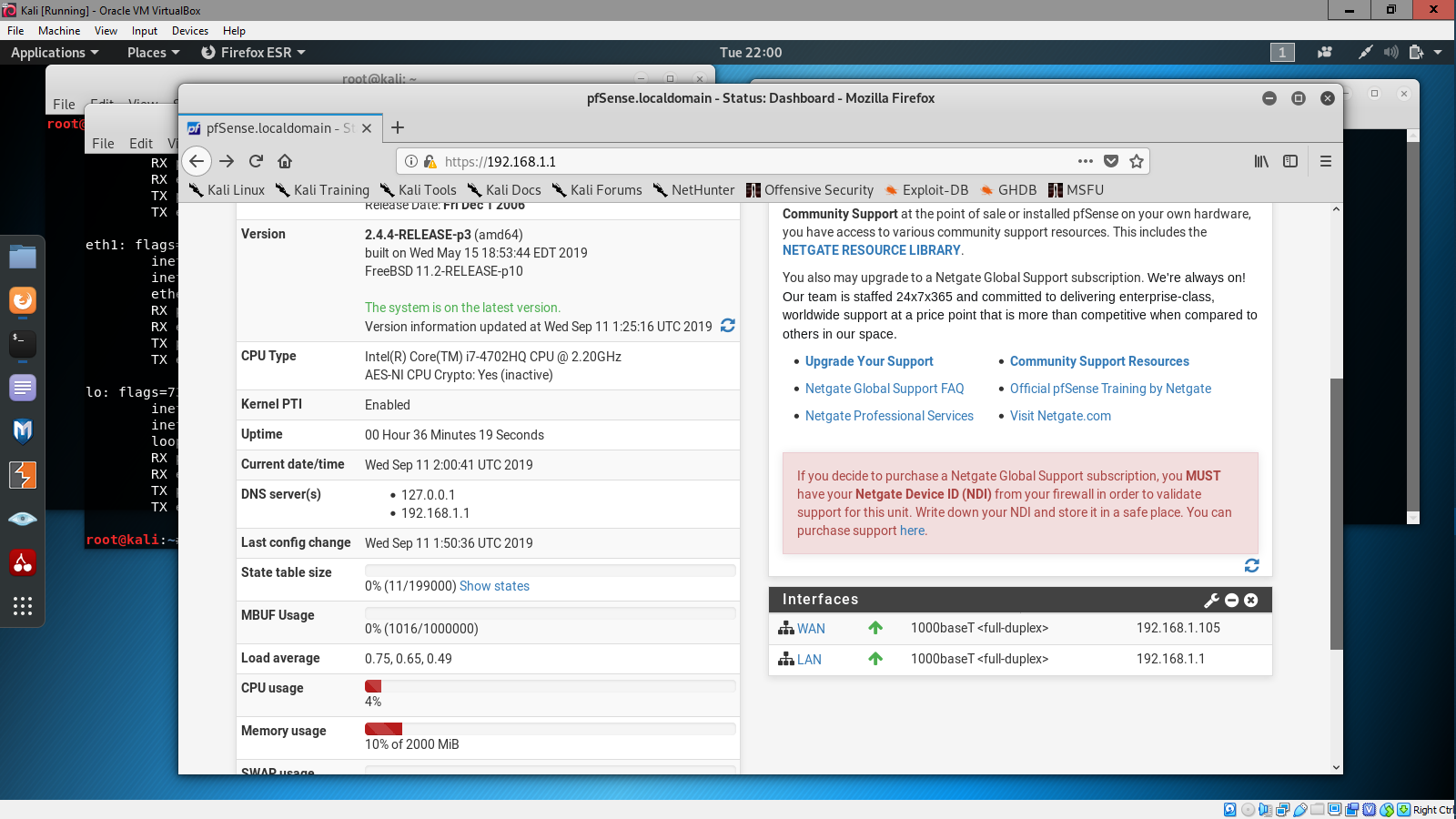
**Objective:** Perform some man-in-the-middle attacks (MITM) on SSL/TLS- protected IMAP traffic, e.g. checking on email. This can be done with a mobile device and a broadband router that you control, or with a set of two virtual machine guests implementing the email client and router respectively. The router should provide access to the Internet (facing Hotmail, Gmail, etc.) for your email client. All traffic for the email client must go through the router for this to work.

**Created Hotmail account**

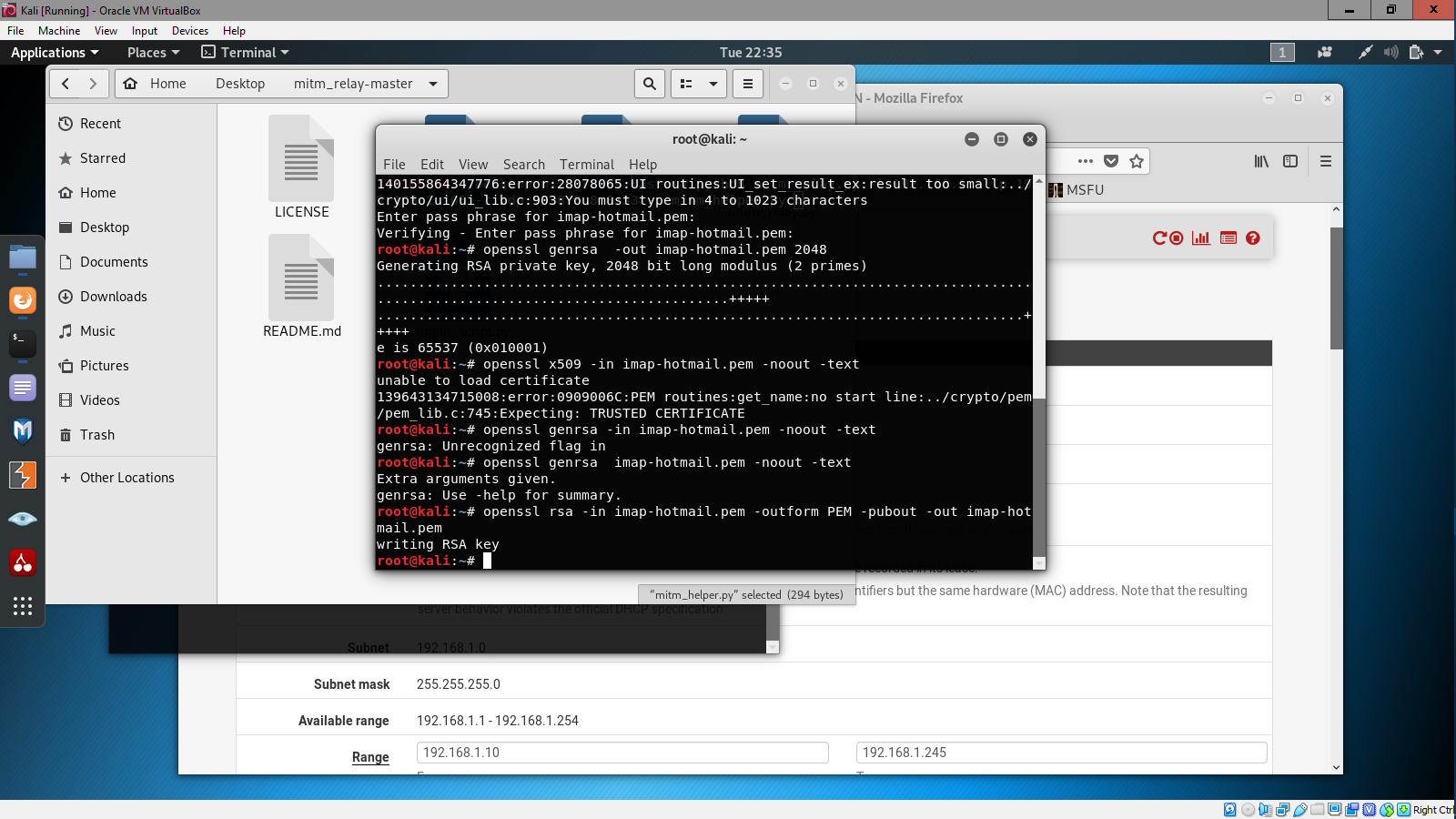


**Choose thunderbird to use for this in ubuntu**

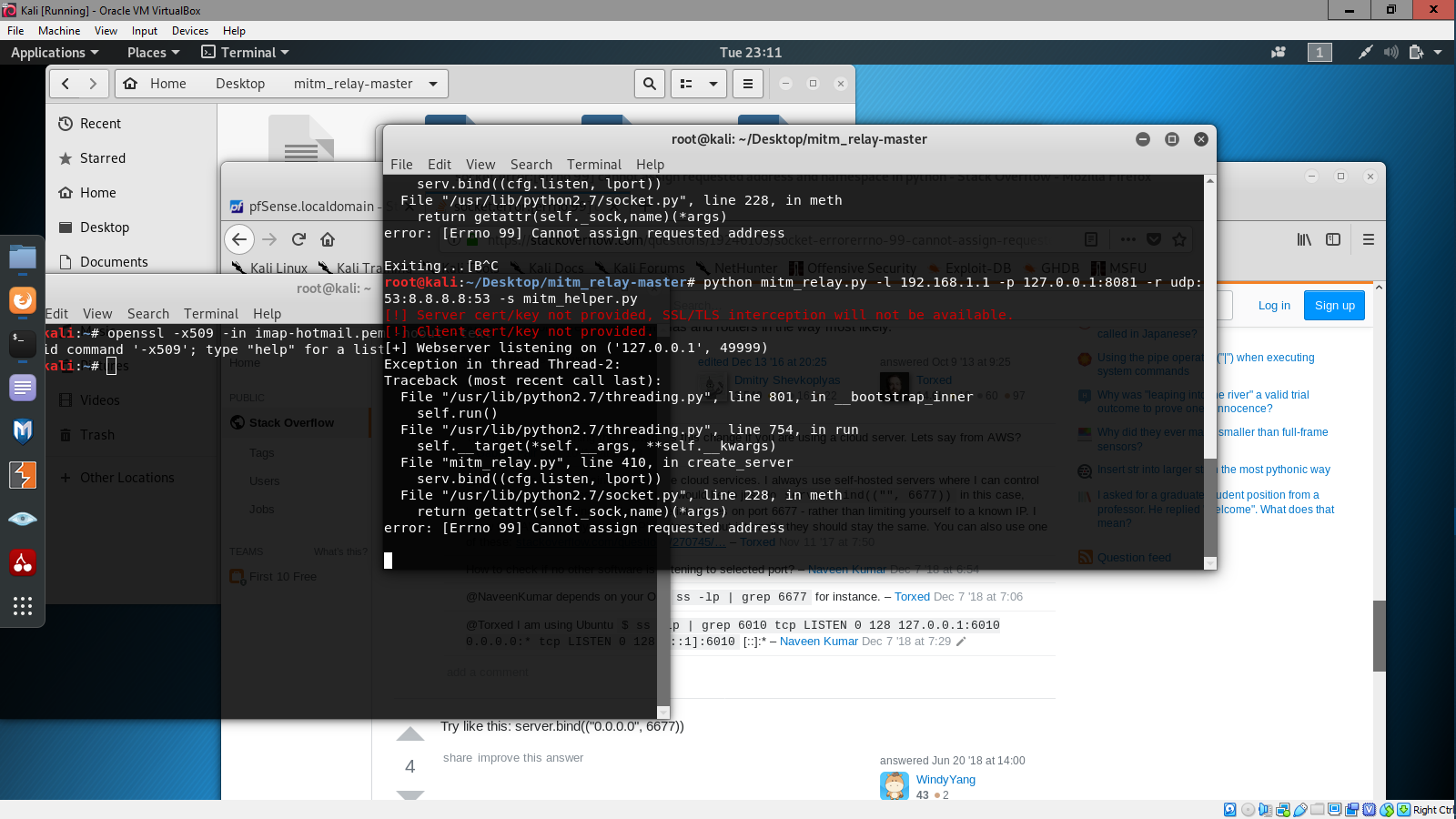
**I managed to set up pfsense and after some back and forth got it talking with Kali Linux and ubuntu. After setting it up**

****

**I attempted to create the SSL certs but they were still of no use because when trying to run the mitm\_relay I ran into a socket error**

****

**This was as far as I got after trying to run the -r a few ways but the helper file was created at least.**

****

**The issues I ran into doing this question was mainly there being small bits of information giving me hopeful leads but ultimately still leading me to dead ends.**

**Over the course of this question, I had to reinstall kali and ubuntu 2 times due to corrupt os issues and only learning after the fact.**

**I'm sure that the last few steps to getting this running would be considered easy but I can't see them clearly.**

**4.4 Web site SSL/TLS certificates - by Jae Cho**

**Objective:** Create a small RSA-based x509 certificate (384-bit modulus, about 116 decimal digits), extract the modulus n, factor n into primes p and q (use openssl to create the certificate).

**Lab Setup:**

OS: ubuntu 14.04 LTS

Processors: 16 x Intel Xeon CPU E5-2640 v3 @ 2.60GHz

Memory: 16GB

Disk: 16.8GB

Tools used: yafu with gmp, gmp-ecm, msieve, ggnfs, Hex to decimal converter

\*\*\*Used settings from <https://www.mersenneforum.org/showthread.php?t=23087> for yafu

**YAFU ini setting:**

B1pm1=100000

B1pp1=20000

B1ecm=11000

rhomax=1000

threads=16

pretest\_ratio=0.25

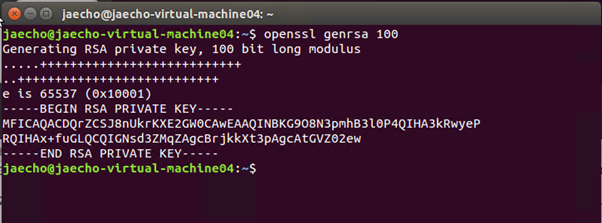
ggnfs\_dir=/home/jaecho/Math/ggnfs/bin/

ecm\_path=../ecm/current/ecm

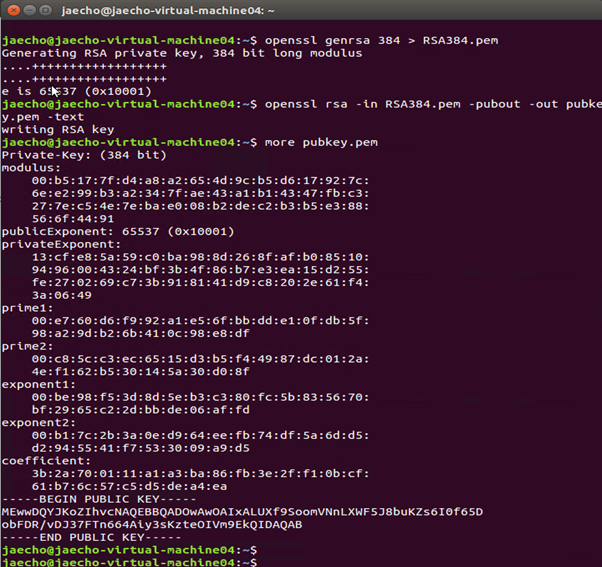
tune\_info=Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz,LINUX64,2.54518e-05,0.195996,0.458379,0.0983109,100.308,2600

**Step 1: Testing of RSA key generation for 100 bits:**

I have initially used ubuntu 18.04 LTS version but due to security restriction on openssl which does not allow generation of RSA key smaller than 512 bit, I have opted to use Ubuntu 14.04 LTS version.

****

**Step 2: RSA key generation for 384 bits:**

****

Private-Key: (384 bit)

modulus:

00:b5:17:7f:d4:a8:a2:65:4d:9c:b5:d6:17:92:7c:

6e:e2:99:b3:a2:34:7f:ae:43:a1:b1:43:47:fb:c3:

27:7e:c5:4e:7e:ba:e0:08:b2:de:c2:b3:b5:e3:88:

56:6f:44:91

publicExponent: 65537 (0x10001)

privateExponent:

13:cf:e8:5a:59:c0:ba:98:8d:26:8f:af:b0:85:10:

94:96:00:43:24:bf:3b:4f:86:b7:e3:ea:15:d2:55:

fe:27:02:69:c7:3b:91:81:41:d9:c8:20:2e:61:f4:

3a:06:49

prime1:

00:e7:60:d6:f9:92:a1:e5:6f:bb:dd:e1:0f:db:5f:

98:a2:9d:b2:6b:41:0c:98:e8:df

prime2:

00:c8:5c:c3:ec:65:15:d3:b5:f4:49:87:dc:01:2a:

4e:f1:62:b5:30:14:5a:30:d0:8f

exponent1:

00:be:98:f5:3d:8d:5e:b3:c3:80:fc:5b:83:56:70:

bf:29:65:c2:2d:bb:de:06:af:fd

exponent2:

00:b1:7c:2b:3a:0e:d9:64:ee:fb:74:df:5a:6d:d5:

d2:94:55:41:f7:53:30:09:a9:d5

coefficient:

3b:2a:70:01:11:a1:a3:ba:86:fb:3e:2f:f1:0b:cf:

61:b7:6c:57:c5:d5:de:a4:ea

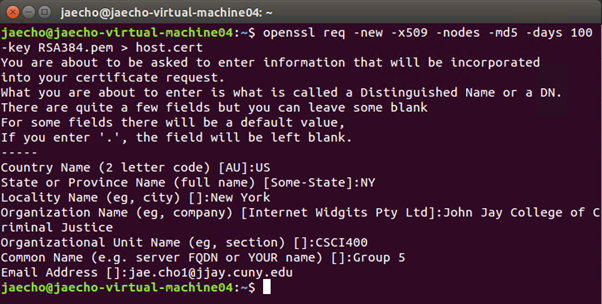
-----BEGIN PUBLIC KEY-----

MEwwDQYJKoZIhvcNAQEBBQADOwAwOAIxALUXf9SoomVNnLXWF5J8buKZs6I0f65D

obFDR/vDJ37FTn664Aiy3sKzteOIVm9EkQIDAQAB

-----END PUBLIC KEY-----

**Step 3: Certificate generation using openssl:**

****

****

**Step 4: Factoring of modulus**

Hex value of modulus:

B5177FD4A8A2654D9CB5D617927C6EE299B3A2347FAE43A1B14347FBC3277EC54E7EBAE008B2DEC2B3B5E388566F4491

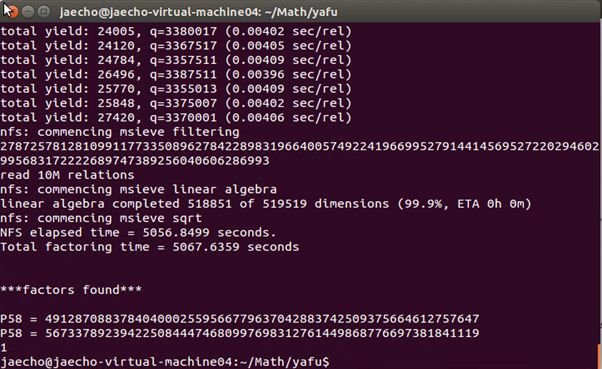
Convert Hex to decimal:

27872578128109911773350896278422898319664005749224196699527914414569527220294602995683172222689747389256040606286993

**Factoring with yafu(yet another factoring utility)**

./yafu\_"factor(27872578128109911773350896278422898319664005749224196699527914414569527220294602995683172222689747389256040606286993)"

**The whole factorization process took 85 minutes with 16 virtual cores Intel Xeon CPU E5-2640 v3 @ 2.60GHz.**



**Step 5: Verification of successful factorization of the modulus**

Compare the Prime 1 & 2 value to factors found by the factorization process.

Result: Factored values matched the values from the private key info.

prime1 Hex from private key info:

00:e7:60:d6:f9:92:a1:e5:6f:bb:dd:e1:0f:db:5f:

98:a2:9d:b2:6b:41:0c:98:e8:df

prime1 Decimal from factorization: 5673378923942250844474680997698312761449868776697381841119

prime2 Hex from private key info:

00:c8:5c:c3:ec:65:15:d3:b5:f4:49:87:dc:01:2a:

4e:f1:62:b5:30:14:5a:30:d0:8f

prime2 Decimal from factorization:

4912870883784040002559566779637042883742509375664612757647

**Bonus to try:**

**Factor the modulus of a 512-bit RSA key. The key in question is:**

Public-Key: (512 bit) modulus:

00:a3:44:26:e2:c0:2b:79:71:28:f1:25:e7:c9:7e:

8c:5a:bd:7f:66:f3:24:c7:1d:40:fd:ea:ae:a1:36:

a4:f8:c2:90:10:15:41:9c:dd:5f:03:bd:32:db:d9:

d6:d5:2f:19:d0:00:e8:38:b9:bb:8c:21:3a:d4:75:

50:46:c9:fd:71

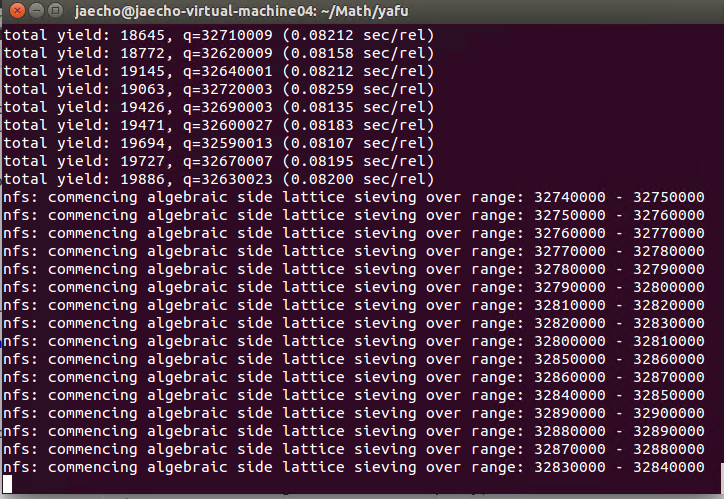
publicExponent: 65537 (0x10001)

The decimal value of above 512-bit modulus is:

8550945691705148721284286590453630952790165134166374713718309143264924622622966370553572158983159460962724564288691603787407968988208541039153756343238001

Command used:

./yafu\_"factor(8550945691705148721284286590453630952790165134166374713718309143264924622622966370553572158983159460962724564288691603787407968988208541039153756343238001)"



**Note: factoring for 512-bit RSA key is still running after 3 days.**

**4.5 Kicking it up a notch: 1024-bit RSA keys - by Aaron Delgado**

In this section, we push the limits of computation and technique. 1024-bit RSA keys are still in common use, despite warnings to gradually upgrade to 2048-bit keys in the long run. Let’s see what can be done, given some special conditions.

4.5.1 Attacking the RSA modulus from a different angle (Aaron Delgado)

Factor the moduli for the following RSA public keys:

Public-Key: (1024 bit) modulus:

00:d9:57:af:3a:15:5e:15:a8:1f:9f:fc:ef:85:de:

f8:b9:dc:2d:f8:d0:d4:03:5d:63:fc:6c:ed:a6:38:

e1:50:07:ca:c3:dd:8d:3f:16:f4:3a:33:a8:1a:18:

92:86:25:ea:1f:9a:62:9c:1e:6c:49:81:74:8d:68:

38:15:5e:e4:7a:5f:21:9e:a4:5c:d0:48:0f:20:61:

58:69:60:cf:aa:08:b4:ef:68:ea:ce:f6:dd:27:f9:

23:39:51:df:af:73:bc:3b:77:f8:48:3d:52:0a:01:

61:2f:49:a0:de:94:b3:1d:d0:f4:a5:ae:fb:65:ba:

04:dd:f3:f4:56:d8:64:5d:d7

publicExponent: 65537 (0x10001)

The modulus N1 in decimal form is

152623106102998859355071417176551223629547132342475055921113655030598105103295962158619774832211755537121217985142194043382936112115788261691421191777978074316019608144843975141961739666317964113156874048882931598523622294034736041234900051121641913634966224247780499751205718318247665396814452113278578744791.

N1 = P \* Q1. 152623106102998859355071417176551223629547132342475055921113655030598105103295962158619774832211755537121217985142194043382936112115788261691421191777978074316019608144843975141961739666317964113156874048882931598523622294034736041234900051121641913634966224247780499751205718318247665396814452113278578744791 = 19,958,403,095,347,198,116,563,727,130,368,385,660,674,512,604,354,575,415,025,472,424,372,118,918,689,640,657,849,579,654,926,357,010,893,424,468,441,924,952,439,724,379,883,935,936,607,391,717,982,848,314,203,200,056,729,510,856,765,175,377,214,443,629,871,826,533,567,445,439,239,933,308,104,551,208,703,888,888,552,684,480,441,575,071,209,068,757,560,416,423,584,952,303,440,099,278,848Q.

Q1 ≅ 7,647,060,006,448,067.

ɸ(N1) = (19,958,403,095,347,198,116,563,727,130,368,385,660,674,512,604,354,575,415,025,472,424,372,118,918,689,640,657,849,579,654,926,357,010,893,424,468,441,924,952,439,724,379,883,935,936,607,391,717,982,848,314,203,200,056,729,510,856,765,175,377,214,443,629,871,826,533,567,445,439,239,933,308,104,551,208,703,888,888,552,684,480,441,575,071,209,068,757,560,416,423,584,952,303,440,099,278,847)(7,647,060,006,448,067) =

152,623,106,102,998,865,201,055,936,907,431,589,556,650,723,778,124,221,938,447,483,190,156,108,597,358,659,208,693,561,763,739,313,180,524,700,587,804,037,458,756,636,418,738,407,406,417,992,053,870,134,522,409,233,806,633,258,872,530,388,901,730,737,403,276,655,704,919,008,221,414,761,614,137,508,412,220,949,762,749,350,096,948,793,623,787,486,977,875,804,372,495,867,008,720,798,235,685,830,456,657,138,749.

e = 65537. gcd(65537, ɸN1) = 1.

d1 = e^-1 mod N1= 121999277651064307967830650880718047547252108985223516711009785340841174620288204830337513467461302769247104531688818687945495095418602450524426655570055651394670665854349948201008337198348113973086384997666717964739256905590020767183350186157491628491563076660211941037332528815554356416332344684683963219672 mod N1 = 121999277651064307967830650880718047547252108985223516711009785340841174620288204830337513467461302769247104531688818687945495095418602450524426655570055651394670665854349948201008337198348113973086384997666717964739256905590020767183350186157491628491563076660211941037332528815554356416332344684683963219672.

Public-Key: (1024 bit) modulus:

00:a6:8e:a1:94:b9:fd:c8:62:ad:e8:d3:96:f1:b1:

ed:8d:5b:78:32:a8:5e:00:bb:de:75:4a:53:aa:03:

30:5a:24:75:f7:82:f7:4f:0a:ef:47:3d:41:99:ae:

4f:52:04:1e:8f:8d:98:94:b5:c9:dd:be:9d:32:2f:

60:96:6d:39:73:79:05:4f:3f:76:fc:20:7a:58:61:

af:95:2e:0a:de:5a:ed:f3:20:d6:f2:0a:8a:3f:22:

ad:5d:dc:00:d3:31:39:df:a7:59:2d:c0:d7:92:f6:

d6:79:8e:54:f6:2a:ff:4c:0e:fa:8f:31:60:52:fe:

0b:ae:35:0b:75:b9:46:7d:71

publicExponent: 65537 (0x10001)

The modulus in N2 decimal form is

116960410376538796436586976720948821611672454495183638944999105642425335107298642427290299095547711533650561189299925421861921679870758372620886759643453123314517678884015959296081753004079171849136211214393988944287474900393176025413873079837842577551328430047515246344177615184104529106722233164211788610929.

P = gcd(N1, N2) = 19,958,403,095,347,198,116,563,727,130,368,385,660,674,512,604,354,575,415,025,472,424,372,118,918,689,640,657,849,579,654,926,357,010,893,424,468,441,924,952,439,724,379,883,935,936,607,391,717,982,848,314,203,200,056,729,510,856,765,175,377,214,443,629,871,826,533,567,445,439,239,933,308,104,551,208,703,888,888,552,684,480,441,575,071,209,068,757,560,416,423,584,952,303,440,099,278,848.

N2 = P \* Q2. N2 = 19,958,403,095,347,198,116,563,727,130,368,385,660,674,512,604,354,575,415,025,472,424,372,118,918,689,640,657,849,579,654,926,357,010,893,424,468,441,924,952,439,724,379,883,935,936,607,391,717,982,848,314,203,200,056,729,510,856,765,175,377,214,443,629,871,826,533,567,445,439,239,933,308,104,551,208,703,888,888,552,684,480,441,575,071,209,068,757,560,416,423,584,952,303,440,099,278,848Q.

Q2 ≅ 5,860,208,846,258,105.

ɸ(N2) = (19,958,403,095,347,198,116,563,727,130,368,385,660,674,512,604,354,575,415,025,472,424,372,118,918,689,640,657,849,579,654,926,357,010,893,424,468,441,924,952,439,724,379,883,935,936,607,391,717,982,848,314,203,200,056,729,510,856,765,175,377,214,443,629,871,826,533,567,445,439,239,933,308,104,551,208,703,888,888,552,684,480,441,575,071,209,068,757,560,416,423,584,952,303,440,099,278,847)(5,860,208,846,258,104) =

116,960,410,376,538,775,516,522,786,074,792,573,114,680,934,903,339,915,368,940,728,149,554,291,817,689,128,293,080,837,043,534,672,741,046,876,883,135,852,872,558,136,690,722,208,450,349,495,575,692,339,199,148,025,495,243,084,099,510,433,360,467,988,224,303,784,456,833,897,616,276,104,626,253,918,232,661,538,944,468,858,267,021,634,293,612,356,940,657,127,187,537,594,241,700,716,554,344,513,464,829,526,088.

e = 65537. gcd(65537, ɸN2) = 1.

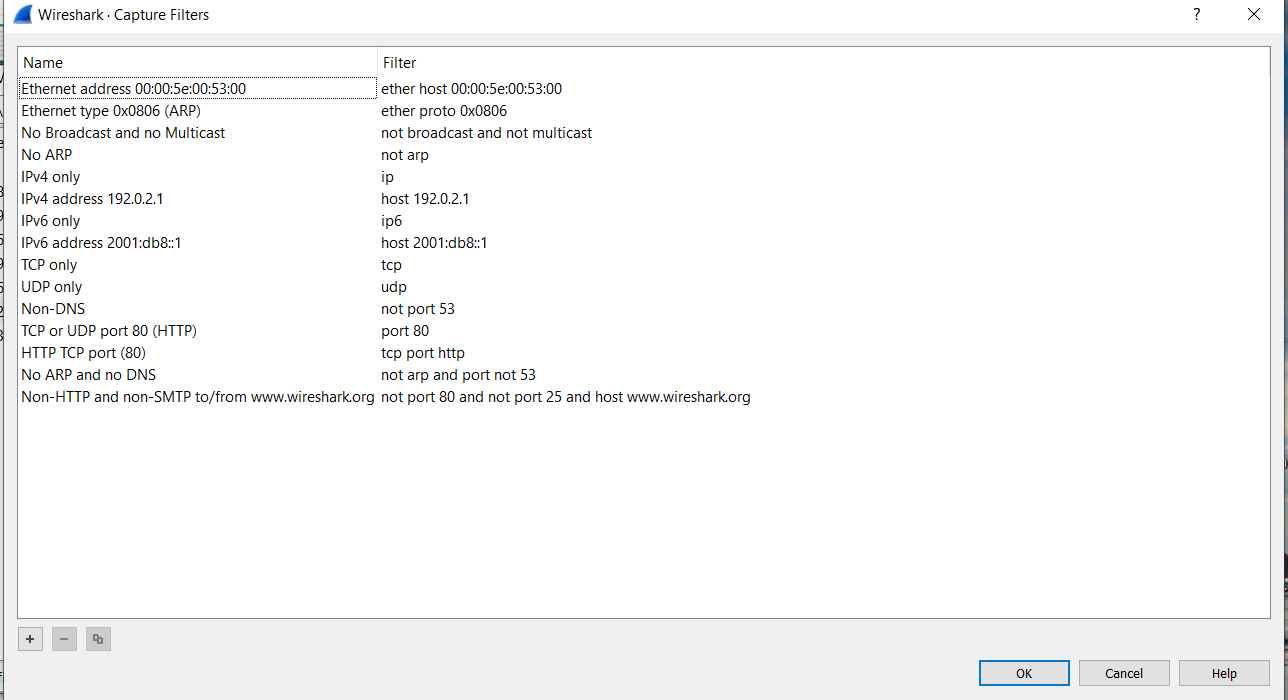
d2 = e^-1 mod N2 = 109527355625508323769873749718755376046379325835457989980201796107342839406825644796796622306360592523966648478107252742580677439263746934471963352195523217175924729366156941177031743066761477252928689971310677777267969385033339930599512010861163536162475066128692276098003701711687491956265268378992109657597 mod N2 = 109527355625508323769873749718755376046379325835457989980201796107342839406825644796796622306360592523966648478107252742580677439263746934471963352195523217175924729366156941177031743066761477252928689971310677777267969385033339930599512010861163536162475066128692276098003701711687491956265268378992109657597.

4.5.2 Decrypting SSL/TLS traffic

Having successfully factored the RSA moduli, apply this newfound knowledge and move on to the next task, namely decrypting SSL/TLS traffic based on one of these two 1024-bit RSA private keys. Look inside the traffic to determine which private key is a need, i.e. look for the public key matching one of the keys in the previous section. You may have to (re)construct the RSA private key from the two prime factors of the modulus and the given exponent by writing some code or

manipulating the correct tools. Using Wireshark and the guidelines from the lab file, decrypt the SSL/TLS traffic found in the file ssldump.pcap. The SSL/TLS traffic in question is on TCP port 44330, which you can narrow down using a filter in Wireshark: TCP port 44330. There is no other relevant traffic in the pcap file, so the filter should not be necessary.

Supply the prime factors of the RSA moduli and decrypted SSL/TLS traffic (the conversation should be 2671 bytes long), both in text form, as part of the report.



I used several online converters and calculators to get the large results I have, as I did not think to put Kali Linux or UNIX on one of the several other Flash Drives I have (I have more Flash Drives I could use for Labs like this than I thought).

5. Word Problems

**1. Summarize the attack techniques used by the tools.**

**(James Stefanik)** - For mitm\_relay with the use of pfsense to create a dummy router, openssl to create fake certs, and burp you set yourself up as being the trusted and sniff data and try to find useful information from doing so.

**(Jae Cho)** -For breaking the RSA key, YAFU is used for factoring the RSA key which is primarily a command-line driven tool. You provide the number to factor and, via screen output and log files and it uses the most powerful modern algorithms (and implementations of them) to factor input integers in a completely automated way. Most algorithm implementations are multi-threaded, allowing YAFU to fully utilize multi- or many-core processors (including SNFS, GNFS, SIQS, and ECM).

**(Victoria Lau)** - The evilize tool allowed for the creation of executable files with the same MD5 hash code. The selfextract tool allowed for the generation of self-extracting archives with the same MD5 hash code. The webversion tool allowed the viewing of log files to check the MD5 hash code as well as the time it took for the block collisions. Each tool allows for the checking of hash-based on the file type and hash function.

**2. How would you use the gained knowledge, namely the factored modulus, to attack an encrypted connection in general? Is it localized to a single session where you intercepted the public key in transit? Or does it apply to all encrypted sessions with the server hosting that specific public key, in the past, present, and future?**

**(Aaron Delgado) -** I would use the gained knowledge to attack an encrypted system via a Man-in-the-Middle Attack, and it is localized to a single session but can still have lasting consequences. I could get private information from one of the users and rewrite it as I see fit, for example.

**3. How would you thwart the other groups’ efforts, i.e. from attacking your systems using the techniques above?**

I would thwart the other groups’ efforts via a botnet so that the other groups have no idea which computer is the one they’re supposed to focus on blocking.

**(Jae Cho)** -In case of breaking RSA key and mitm, one way to prevent factoring of the 1024-bit key is to make sure **p** & **q** are generated randomly. Also, implementing Certificate-Based Authentication for communication which allows only endpoints with properly configured certificates can access systems and networks, can block hackers.

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

**(Jae Cho)** - Estimated time to factor RSA key modulus depends on a couple of factors such as CPU, GPU and amount of RAM used. Also factoring algorithms such as ECM, SNFS, NFS GGNFS also can contribute to factoring time. My lab was set up on OS: ubuntu 14.04 LTS

Processors: 16 x Intel Xeon CPU E5-2640 v3 @ 2.60GHz, Memory: 16GB, Disk: 16.8GB and it took 85 minutes to factor 384-bit RSA key however factoring for the 512-bit key has been running for almost 3 days and it is only thru running for 33,000,000 NFS range. If a full week has been given I am confident to say 64 bit ~512-bit keys can be factored within a week.

**5. How would you approach breaking a Diffie-Hellman public key?**

**(Victoria Lau) -** In Diffie-Hellman, Alice and Bob both choose a large number, *a* and *b*. Alice and Bob then exchange their chosen numbers *a* and *b* and then compute a second number *na and nb*. Their shared key can now be computed with their original chosen numbers *a* and *b* to find (*nb)a* and *(na)b*. The attacker listening only knows the computed numbers *na and nb*. To find the public key, it would cost millions of dollars and take a tremendous amount of computing resources as many approaches use 1024 bits. To approach breaking a Diffie-Hellman public key, given that small sets of prime numbers were used and enough computing resources, it is not easy, but possible to try every possible key much like a brute-force attack to calculate the public key.

6. Deliverables

A zip file containing:

The source code of any tools you developed and data files, including the particular technique and keys used, if any.

Task-4.1-4.2-Victoria Lau.zip (SHA1 files)

Task-4.4-4.5-Jae Cho.zip (contains RSA key files, certificate, Python script to calculate GCD of N1 and N2, etc…)

Task-4.5-Aaron Delgado.zip (RSA Keys)