**CSCI 400-01 Group 5**

**Cryptography Lab 02 Report**

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

**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 proper 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**

**Objective:** Using the tools provided, evilize, selfextract, and web version, find MD5 hash collisions of each type:

Executable files

Self-extracting archives

Strings

See the lab file for examples. Can you extend this to other hash functions? If so, how? If not, why not?

**4.2 Finding SHA-1 hash collisions**

Using the tools provided, find SHA-1 hash collisions for:

Two PDF documents

Both online and offline tools are provided. Examples of SHA-1 collisions are shown in the lab file. Start with the online tools to produce a collision between two images. Then extend to the offline tool for greater flexibility of the characteristics of the input documents and use two PDFs of your choice.

Prepare a report of your findings and for each tool include the 2 original images or PDFs, the hash values for each original file, the newly created PDFs, and the new common hash values.

**4.3 Man-in-the-middle attacks on SSL/TLS**

**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.

Find the MITM (man-in-the-middle) script at https://github.com/jrmdev/mitm\_relay. Using this script, implement an attack against Hotmail.com, GMail.com, or similar. You may have to filter out advanced authentication mechanisms, effectively forcing a downgrade to plaintext passwords, using the companion script for mitm shown here for hotmail.com, filtering out the SASL-IR mechanism:

You will need to create fake SSL/TLS site certificates to match the fully-qualified domain names (FQDNs) of the mail server, such as imap.google.com, imap.hotmail.com, or imap- mail.outlook.com. You may need to create a Certification Authority (CA) as well, in order to sign your SSL/TLS site certificates. The generated keys would look something like this (the ones shown here are not functional, so you must generate your own):

You will need to import those site certificates and CA certificates to the device(s) being deceived, such as your iPhone, iPad, Android phone, laptop, or similar.

Intercept the traffic between your IMAP email client and the purported email/IMAP server, and show that you managed to capture and read the network traffic between the client and the IMAP server, and in particular reveal the plaintext password for the email account created above. The interception is done with the script itself, which can dump it to stdout or to a log file.

After your experiment, please make sure to remove these certificates from any production devices, if you chose to add them there.

Note: You may use the Burp interception technique (described in connection with mitm script) if you wish, but it is not necessary or required.

**4.4 Web site SSL/TLS certificates**

**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, Intel Xeon CPU E5-2640 v3 @ 2.60GHz

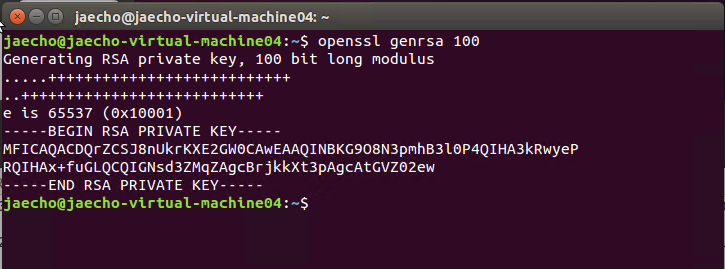
Memory: 16GB

Disk: 16.8GB

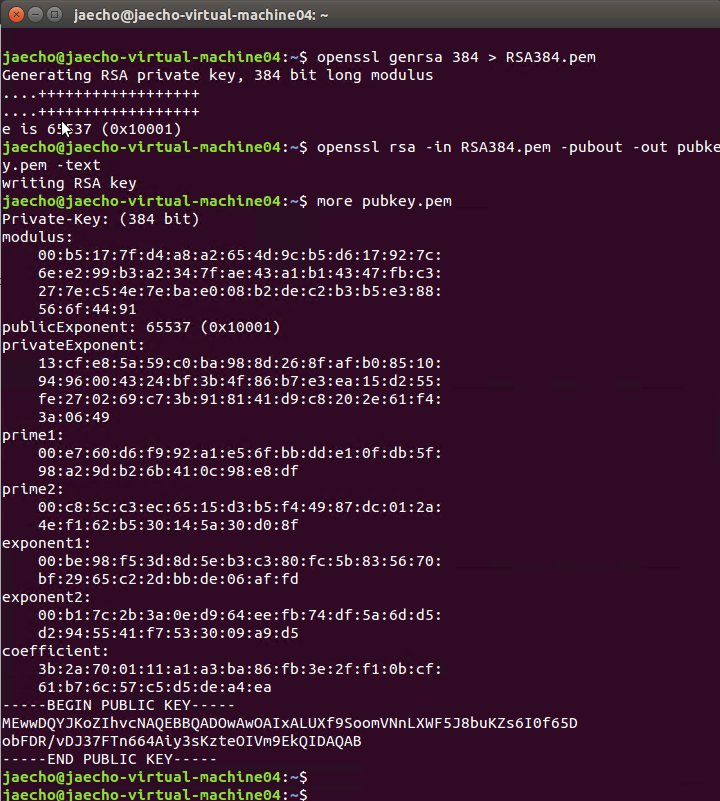
Tools used: yafu with gmp, gmp-ecm, msieve, ggnfs, Hex to decimal converter



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



**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

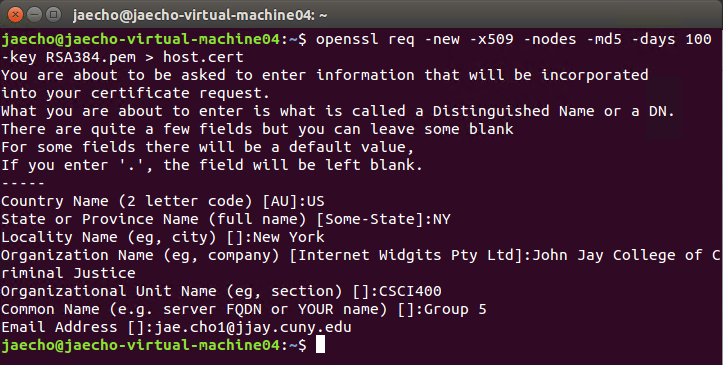
-----BEGIN PUBLIC KEY-----

MEwwDQYJKoZIhvcNAQEBBQADOwAwOAIxALUXf9SoomVNnLXWF5J8buKZs6I0f65D

obFDR/vDJ37FTn664Aiy3sKzteOIVm9EkQIDAQAB

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

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





**Step 4: Factoring of modulus**

Hex value of modulus is:

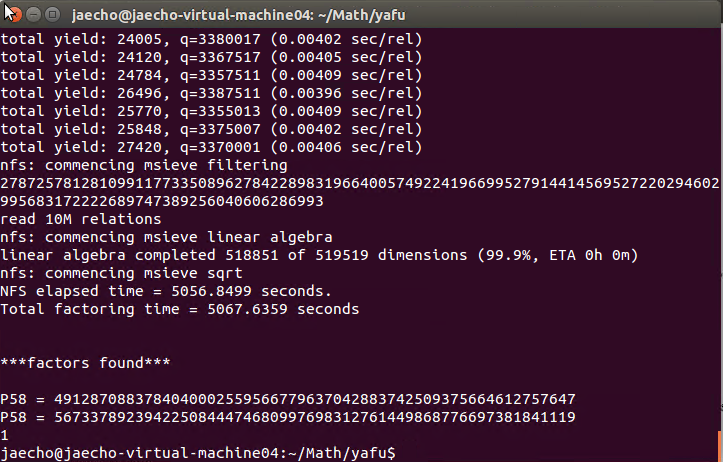
B5177FD4A8A2654D9CB5D617927C6EE299B3A2347FAE43A1B14347FBC3277EC54E7EBAE008B2DEC2B3B5E388566F4491

Convert Hex to decimal:

27872578128109911773350896278422898319664005749224196699527914414569527220294602995683172222689747389256040606286993

Factoring with yafu(yet another factoring utility)

./yafu\_"factor(27872578128109911773350896278422898319664005749224196699527914414569527220294602995683172222689747389256040606286993)"



**Step5: Verification of successful factorization of modulus**

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

Factored values match 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)

Decimal value of above 512-bit modulus is:

8550945691705148721284286590453630952790165134166374713718309143264924622622966370553572158983159460962724564288691603787407968988208541039153756343238001

Command used:

./yafu\_"factor(8550945691705148721284286590453630952790165134166374713718309143264924622622966370553572158983159460962724564288691603787407968988208541039153756343238001)"

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

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**

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)

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)

So, you are sitting there, staring at this assignment to factor a 1024-bit RSA modulus. You have been struggling with or contemplating factoring a 512-bit integer with the help of heavy- duty computing power, and now this? A friend of yours who has taken number theory swings by, looks at your computer screen, and you mumble something in frustration. She tells you something that sounds like “mind your own p’s and q’s,” but you haven’t said anything wrong. Or was it that she said, “mine your p’s and q’s?” Confused, yet intrigued, you start searching for an answer...

**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 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.

**5. Word Problems**

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

YAFU is used for factoring 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).

**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?**

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

Make sure p & q are generated randomly.

Authentication Certificates

Hackers will never go away, but one thing you can do is make it virtually impossible to penetrate your systems (e.g. Wi-Fi networks, email systems, internal networks) by implementing Certificate-Based Authentication for all employee machines and devices. This means only endpoints with properly configured certificates can access your systems and networks. Certificates are user-friendly (there is no additional hardware to manage or much user training needed) and deployments can be automated to make things simple for IT and make them hackers split their hair, as the cool kids would say.

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

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

Sharing prime numbers

Imagine what would happen if many Diffie-Hellman implementations used the same fixed prime number: an adversary could spend a lot of time and money doing the required computations for this prime number and subsequently use that to crack key exchanges as they happen in real time. Knowledge of the secret key allows an adversary to read all the supposedly encrypted traffic between millions of Alices and Bobs around the world.

And this is exactly what was (and to some extent still is) the case: the paper showed for example that one in six of the most used HTTPS servers shared the same prime number. It’s even worse when it comes to VPN servers, of which 66% shared the same prime number. Although the latter figure has been disputed, if this was indeed what the NSA used to read VPN traffic, it would explain some Snowden slides rather well.

It is worth noting that sharing the same prime number is not as stupid as it may seem. Using your own unique prime number may be safer against this kind of attack, there are many traps to avoid when doing so. Most importantly, a lot of prime numbers are unsuitable and make the protocol a lot weaker. The paper also showed that several implementations used such unsuitable primes (and, somewhat intriguingly, some implementations used numbers that weren’t prime at all). There is thus a lot to say for choosing known safe (though widely used) prime numbers, despite the downsides we now understand well.

It would actually be a good rule of thumb to choose the strength of your encryption algorithms such that it would be too expensive for the most powerful adversary to attack, even if they would automatically crack all other implementations of the same protocol. For Diffie-Hellman, using longer numbers of 2048 bits (more than 600 digits) will do just fine.

It is thus not true that the researchers have broken Diffie-Hellman. Nor is it true that choosing “non-random prime numbers” (as I’ve seen someone claim somewhere) is inherently wrong.

However, longer numbers makes the algorithm more expensive to run. There would thus be a good argument to use Elliptic Curve Diffie-Hellman (ECDH) instead, a similar protocol that uses a different kind of maths. Its most important benefit is that it provides the same level of security with much smaller numbers.

**6. Deliverables**

1. A report describing all your findings above.

2. A zip file containing:

* The source code of any tools you developed and data files, including the particular technique and keys used, if any.
* Answers to all word problems in Part 5.

3. Submit by the beginning of class, September 11, 2019.