Norwegian University of Science and Technology

TTM4135 Applied Cryptography and Network Security, Spring 2024

Web Security Lab
Group 25

Omdal, Tobias Stenersen   Peregrina, Oscar Peña   Ellingsund, Joachim
Gaalaas, Marius Berdal

27th September 2024

1 Introduction

In this report, we will explore crucial aspects of cybersecurity based on the course's syllabus and our group's personal experience during this lab. We will answer the questions related to our use of PGP and GPG to secure email messages, then the questions about configuring and establishing certificates of an Apache web server. After that we will go over our answers regarding our examination of TSL traffic, and lastly, we will discuss our experiences with the lab, and what it taught us.

2 Answers to the Questions

> **Q1:** Generating a symmetric key k just for encrypting that one message seems like an unnecessarily complicated step. Why does GPG do that, instead of just encrypting the message with p?

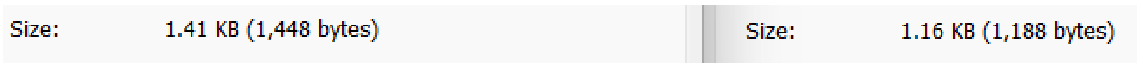
A:

Efficiency/Security tradeoff: Public Key cryptography depends on complex mathematical functions. According to Stallings, the complexity of calculating these functions becomes very high when we use large keys, as the time needed to complete the calculations grow faster than the key size does. At the same time, we also need to prevent brute force attacks[1]. Large key sizes are an effective counter to brute force attacks, but as we have mentioned, these large keys require so much computation that they are not usable for everyday communication. Therefore, we use RSA to generate and share a symmetric key k, as the symmetric algorithms allow faster computation times while still having large enough keys to prevent brute force attacks. Public key cryptography allows us to establish communication securely, but symmetric cryptography is needed to make fast communication secure.

> **Q2:** How many bytes does PGP use to store the private signing and public verification keys for each of the two signature types?

A:

By default, our previous keys were generated using RSA, but by changing the preference for personal keys using the command "–personal-pubkey-preferences", we were able to create a DSA key as well. We identified these files using the key IDs, and inspected the files to see their sizes, which are shown below.


Size: 1.41 KB (1,448 bytes)

Size: 1.16 KB (1,188 bytes)

Figure 1: The sizes of the files storing the RSA and DSA keys

For RSA, PGP used 1448 bytes, while for DSA, it used 1,188 bytes.

> **Q3:** How many bytes does PGP use to store the signatures in each of the four cases (both long and short messages)?

A:

We started by generating two files, kilo.txt at 1024 bytes (1 kB), and mega.txt at 1048576 bytes (1 MB). As there were multiple forms of signatures, we chose the detached version, as the resulting file only contains the signature, and not the compressed original files. As we can observe, they remain the same independent of file size.

- Kilo.txt: 1024 bytes
  - kilo_rsa.sig: 310 bytes
  - kilo_dsa.sig: 119 bytes
- mega.txt: 1,048,576 bytes
  - mega_rsa.sig: 310 bytes
  - mega_dsa.sig: 119 bytes

> **Q4:** How long, on average, does PGP uses for signature generation and verification in each of the four cases?

A:

Using the *time* command, we get three time values for each of the processes. *Real* is the actual time from starting to completing the command. This includes time when other processes are running the CPU, making it a poor metric for the efficiency of each of these tasks. *User* and *sys* are the time spent by the process in kernel and user mode, respectively. The sum of this time is the time actually spent by the process. Therefore, we end up with the following times:

- RSA signing 1 kB file: 2 ms
- RSA signing 1 MB file: 3 ms
- DSA signing 1 kB file: 2 ms
- DSA signing 1 MB file: 3 ms
- RSA verifying 1 kB file: 1 ms
- RSA verifiying 1 MB file: 4 ms
- DSA verifying 1 kb file: 4 ms
- DSA verifying 1 MB file: 7 ms

> **Q5:** Discuss your results for the above three measurements. In particular, how well do they correspond to what you expect from what was studied in the lectures? Include an explanation of how the different elements of the keys are stored (such as modulus, exponents, generators).

A:

Question 2: RSA saves private keys as (d, n) and public keys as (e, n), where $n = pq$, the product of two large primes [1]. The exponents d and e are chosen so that $ed \mod \phi(n) = 1$. As we use RSA-2048, n has 2048 bits. E is often a small number for higher efficiency, where $e = 2^{16} + 1$ is a common choice. Therefore, it has a low amount of bits. It is recommended to use a d of at least $\sqrt{n}$, which means at least 1024 bits. Adding it all up, we get 5136 bits, which is 642 bytes.

DSA uses p, a prime of 2048 bits, and q, a prime divisor of $p-1$ of 256 bits to generate the keys. It chooses a random x with $0 < x < q$, which has an upper bound of 256 bits. This is the private key. It calculates $y = g^x \mod p$ as the public key [1]. The upper bound for this is 2048 bits. In addition, q is used during signature generation, so that must be saved as well. Saving q, x and y would take 2560 bits, or 320 bytes. This is half of the storage we would need for a RSA keypair, so the difference in the files sizes in question 2 fits the theory.

The files we found were larger than the sizes we have calculated here. However, when listing the keys using gpg –list-keys, we see that each key also contains additional info, like creation date, algorithm used, and IDs for both key and user. This extra information is the reason for the larger files.

Question 3: The last step of signature generation in both RSA and DSA is calculating a value using mod. This places an upper bound on the size of the signatures. For RSA, the last step is mod n, which means that the signature cannot be larger than n, at 2048 bits. For DSA, the last step is mod q, at most 256 bits[1]. Therefore, it makes sense that the size of the DSA signature is much smaller than the size of the RSA signature.

When signing, the message is first run through a hash function, which has fixed length outputs. This means that we end up with signatures of the same size, regardless of the size of the original file.

Question 4: Our results show that RSA and DSA are equally fast when it comes to signing, which is somewhat unexpected. For signature generation, we would have expected DSA to perform better than RSA. This is because the first step, computing $r = g^k \mod p$, can be precalculated, as it does not involve the message m, nor the key x[1]. However, our method for meauring the time of the process only gave us one significant digit. Therefore, it is possible that we would have seen a difference between the two algorithms if we signed even larger files, or tried to sign many of them.

Even with a single significant digit, we can see that RSA is faster than DSA in our tests. This is to be expected, due to the mathematical operations involved in the verification algorithms. For RSA, the most computationally intensive part is calculating $\sigma^e \mod n$, in order to compare it to the hash of the message h'. Popular choices for e are numbers where the binary representation only have a single 1[1]. This minimizes the number of multiplications needed for the exponentiation. The s in DSA cannot be optimized in this manner, as the exponents u_1 and u_2 are not chosen, but calculated from r and s, both of which involve randomness. This makes it impossible to guarantee exponents that optimize the runtime[2].

> **Q6:** What assurances does someone receive who downloads your public key from the server where you have uploaded it? What is the difference between the role of the certification authority in X509 and the key server in PGP, with regard to the security guarantees they give?

A:

When someone downloads a key from the keyserver, they can be assured that any messages encrypted using that public key, will only be decrypted by the corresponding private key. By verifying messages with the downloaded public key, the user can be assured that the message was signed by the private key. However, the keyserver does not provide any assurances regarding the identity of the key holders, because it uses the web of trust model, as mentioned in the instructions. It is instead the responsibility of the user to validate the keys they download. For example, in GnuPGs official FAQ, they recommend meeting the owner face-to-face and verify their identity using identifications like passports or driver's licenses[3]. When doing task 3, we verified each others identities by sharing our keys fingerprints.

In contrast, a certificate authority in X509 is responsible for creating, storing and distributing Public Key Certificates, verifying that a certain user owns a certain private key. A user who downloads my key from a CA, gets all the assurances that the key-server provided, but can also rest assured that I am the actual owner of the key they downloaded, without taking additional steps to verify my identity.

> **Q7:** Who typically signs a software release like Apache? What do you gain by verifying such a signature?

A:

Normally, it is signed by a member of the *Project Management Committee* (PMC), referred to as release manager. It prepares and signs the proposed release materials and then the PMC votes to approve it. Verifying a signature on a software release provides several benefits:

- **Autenticity**: Ensures that the software package is genuine and has not been tampered with since it was signed.
- **Integrity**: You can verify that the software is the original that you want to install from the cryptographic signature.

> **Q8:** Why did you obtain a certificate from Let's Encrypt instead of generating a self-signed one yourself?

A: If we choose a self-signed certificate, we cannot prove to visiting users that our server is legitimate. Therefore we use the certificate from Let's encrypt, which is a widely known CA and is compatible with most browsers the browsers. This gives the users the credibility and assurance that we are looking for.

> **Q9:** What does Let's Encrypt have to verify, and how do they do it, before they issue a certificate?

A: B efore issuing a certificate, Let's Encrypt must verify that the requester has control over the domain names listed in the certificate request. This is done using "challenges" as defined by the ACME standard. These challenges ensure that the certificates are not issued to unauthorized parties and help maintain the security and integrity of web communications[4].

> **Q10:** What steps does your browser take when verifying the authenticity of a web page served over `https`? Give a high-level answer.

A: T he browser in order to verify the authenticity of a web page follows the next steps:

- **Certificate Retrieval**: The browser retrieves the server's SSL/TLS certificate, which includes the public key necessary for establishing a secure connection.
- **Certificate Authority Check**: The browser checks if the certificate is signed by a trusted Certificate Authority (CA).
- **Domain Verification**: The browser verifies that the certificate is valid for the domain to which it is connected.
- **Certificate Validity**: The browser checks the certificate's validity period to ensure it has not expired.
- **Revocation Status**: The browser may check if the certificate has been revoked by the CA.
- **Encryption Handshake**: If the certificate is valid, the browser and server perform a handshake to establish an encrypted session using the server's public key.
- **Data Transfer**: Once the secure channel is established, encrypted data can be transferred between the browser and the server.

> **Q11:** Have a look at the screenshot. What does the string `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` in the bottom left say about the encryption? Address all eight parts of the string.

A: T he string is the cipher suite used in the TLS protocol to secure communications.

- **TLS**: The communication is using the Transport Layer Security protocol[5].
- **DHE**: Stands for the Diffie-Heillman Ephemeral, which is a key exchange method for each session[1].
- **RSA**: Refers to the use of RSA public key cryptography for digital signatures, to ensure the key exchange method of above[1].
- **WITH**: This is a separator in the cipher suite notation, indicating that the following elements describe the encryption algorithm and hash function used.
- **AES 128**: Specifies the use of the Advanced Encryption Standard with a 128-bit key size for symmetric encryption[1].
- **CBC**: It provides a way to encrypt data of arbitrary length and ensures that each block of plain-text is XORed with the previous cipher-text block before being encrypted [1].
- **SHA**: Secure Hash Algorithm, used here to refer to the SHA-1 hash function[1].

> **Q12:** The screenshot is obviously from a few years ago. Do you think the encryption specified by this string is still secure right now? Motivate your answer.

A: Obviously it still gives you a basic level of security, but nowadays the encryption methods have improved as almost all the 8 components used are compromised or have other options that are more secure. For example the CBC has known vulnerabilities, such as the BEAST attack, that can compromise our connection. Other example is the SHA-1 is outdated because it has been compromised, but you have alternatives more stronger such as SHA-256. Otherwise the DHE or AES-128 is still considered both secure for encrypting and providing forward secrecy[6].

> **Q13:** What restrictions on server TLS versions and ciphersuites are necessary in order to obtain an A rating at the SSL Labs site? Why do the majority of popular web servers not implement these restrictions?

A:

In order to get an A rating in SSL Labs you have to have some features active:

- **Secure TLS version**: Only secure version of TLS should be enabled (TLS 1.2 and TLS 1.3). Older versions like TLS 1.0 and TLS 1.1 are considered insecure and should be disabled.
- **Cipher Suites**: Only strong cipher suites should be enabled. This means:
  - Ciphersuites that provide forward secrecy (like those using ECDHE or DHE for key exchange).
  - Ciphersuites that use strong encryption algorithms (like AES with 128-bit keys or higher).
  - Ciphersuites that use secure hash algorithms (like SHA-256 or SHA-384)
- **Certificate Strength**: The server's certificate should use a strong public key algorithm (like RSA with 2048-bit keys or better) and a secure signature algorithm.
- **Secure Renegotiation**: The server must support secure renegotiation to prevent attacks *like Man-In-The-Middle attacks*[1].
- **Protection Against Attacks**: The server must be configured to protect against known vulnerabilities like POODLE, BEAST, and Heartbleed.
- **HSTS**: *Implementing HTTP Strict Transport Security* (HSTS) with a long duration can also contribute to a higher rating.

> **Q14:** What are the values of the client and server nonces used in the handshake? How many bytes are they? Is this what you expect from the TLS 1.2 specification?
>
> What is the value of the encrypted (pre-)master secret in the client key exchange field sent to the server? Is this the size that you would expect given the public key of your server?

A:

Client Nonce: 461b60b153c76cea2ded9b7d385166754c3a1159decd3ad699ad7d05f01d0b53
This is 32 bytes, which is expected from TLS 1.2[7].

Server Nonce: dc554ab66469f88a6c5e1114c7dd15d5c34524f13fe100e018b83543a5773bd7
This is also 32 bytes, which is expected from TLS 1.2[7].

Encrypted PreMaster secret: 0afe1ab4211d377e1fa790eb88d7ca20749585261ecb0caea6c7678
cb0ae66ec36cac0e54af547f64144b22560a3c41981657bd2f5e552db179181916fae8cf998356e88
5b2c340acb8d38c1223140dad63336629707849a118172e7a1f21dd6b860a9a4321c4c98d23fce297

56945649e502d7560debf011da8563ede3fcf68b4ea1be6556aa6790fb5799385c11e3722f986bf6f
49fb3916105f6b98e6f29e5b9e4461f3c93dd83cac5c32dca81a1dfbe695186ffef87c7f57360a3c9
b3aadbb684336ef77f9af8bf1d965a8609d4414d2d85a8ea7f99526a99f20afc780d31ec1fc4d51d3
e13756a83e96dd96a47cfc0a554dcc1f601143b4f446196754e4 This is 256 bytes, which is the correct size according to[7].

> **Q15:** What was the value of the (pre-)master secret in the session that you captured? Is this the size (in bytes) that you expect from the TLS specification? Explain how Task 15 shows that this session does not have forward secrecy.

A: **(pre-)master secret:** 14 00 00 0c c8 d6 f7 4d f6 61 63 b9 6e 54 78 4c. As we use AES-128 the keys have 128 bits, which is 16 bytes, and also the expected size.

Forward secrecy: is a protocol that ensure every communication uses a different set of keys [8]. In task 15 we show that we use the same private key on every communication, and therefore the session does not have forward secrecy.

3 Discussion

While we are confident in our understanding of *Public Key Infrastructure* (PKI), applying it to practical scenarios is something we hope to develop further. When using GNUPG (GPG) during our first week of labs, we were able to use our knowledge of PKI to generate a set of public and private keys, which we sent to a key server. As this was our first time using keys directly without any interface, we found it challenging to export, download, sign and encrypt using these keys. For example, when generating our keys, we used the wrong email @ntnu instead of @stud.ntnu, resulting in failed connections to the key server *keys.openpgp.org*. Luckily, GPG was well documented[2], which gave us more confidence and energized us enough to complete the tasks. However, seeing the depth of knowledge required for the questions, left us feeling disheartened, but also eager to broaden our scope of understanding. Like *Souza, Moreira and Figueiredo* mentions, bridging this gap between theory and practice helps with understanding, but also in *learning satisfaction*[9]. It helped remind us that although we are able to understand the theory, applying it to real-world scenarios is much more difficult[10].

Our exploration of server environments extended to a deeper examination of web server hardening techniques. We initially thought that configuring the *Virtual Machine* (VM) and the corresponding web server was relatively easy, but that we lacked fundamental understanding of what was "beneath the hood". This corresponds with *Stallings*' opinion: "Although Web browsers are very easy to use, Web servers are relatively easy to configure and manage, and Web content is increasingly easy to develop, the underlying software is extraordinarily complex" [1]. Because of this, when we inevitably met a problem; that we couldn't enable the correct cipher-suite, we were unsure of how to tackle it. We were relieved to find other people had encountered the same problem, and that it had to do with the VM version (Ubuntu 22.04) and its handling of legacy ciphersuites. Even so, we had to redo the whole of part two again. Therefore, to improve our process, we should probably have investigated how the different environments would interact with each other. This also coincides with *Shaw's and Garlan's 1996*, cited by *Conradi and Westfechtel*, way of thinking, where creating an architectural overview is done before implementing the details[11].

The final leg of the lab involved diving into the world of TLS traffic analysis and RSA key exchange between our client and web server. Due to the previous problem of lacking the correct cipher-suites and TLS support, we were initially bewildered on how to proceed. TLS1.3 is moving away from RSA key exchange because of security problems and mandates the use of forward secrecy instead[5]. In our case, as explained earlier, this resulted in the RSA and SHA based cipher-suite not being supported by our VM. After reconfiguring our web server, all that remained was sniffing packets using Wireshark, something multiple members already had experience with. Although the final tasks weren't without trouble. Determining the size of the (pre-)master secret and adjusting the cipher-suites were difficult, but we realized they were great learning opportunities. We are interested in learning more about the differences between TLS1.3, TLS1.2 and more about why certain configurations are considered legacy.

In conclusion, our exploration of email encryption, web server configuration, and network traffic analysis has been enlightening. Each phase presented its own unique challenges, prompting us to draw upon both theoretical knowledge and practical experience. Through reflection and collaboration, we have gained a deeper appreciation for the multifaceted nature of cybersecurity and its real-world impact. In future work, for groups going through the same experience, we suggest three things. First, to not only focus on theory, but to also apply it in practical scenarios. Second, to gain an architectural overview before moving on to implementation details. And third, to encourage learning, both as individuals, but also inside the group.

Bibliography

[1] W. Stallings, *Cryptography and Network Security: Principles and Practices*. Pearson Education, 2023.

[2] "gpg manual." https://www.gnupg.org/documentation/manuals/gnupg.pdf, 2024. Accessed: 03/15/2024.

[3] R. Hansen, "Gnupg faq." https://www.gnupg.org/faq/gnupg-faq.html, 2017. Accessed: 17/04/2024.

[4] J. Aas, "Let's encrypt: An automated certificate authority to encrypt the entire web." https://doi.org/10.1145/3319535.3363192, 2019.

[5] K. Sako, S. Schneider, and P. Y. A. Ryan, *Computer Security – ESORICS 2019*. Springer Nature Switzerland AG, 2019.

[6] H. C. Rudolph, "Ciphersuiteinfo." https://ciphersuite.info/cs/TLS_DHE_RSA_WITH_AES_128_CBC_SHA/, 2019.

[7] M. Driscoll, "The illustrated tls connection." https://tls12.xargs.org/, 2023.

[8] I. Ristic, "Bulletproof tls guide." https://www.feistyduck.com/library/bulletproof-tls-guide/online/configuration/protocol-configuration/use-forward-secrecy.html, 2022.

[9] M. Souza, R. Moreira, and E. F. Figueiredo, "Students perception on the use of project-based learning in software engineering education," 2019.

[10] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, “Public key encryption with keyword search,” 2004.

[11] R. Conradi and B. Westfechtel, “Version models for software configuration management,” 1998.