Creating Capture-the-Flag Challenges Inspired by Common Crypto Mistakes

Simen Karlsen Lone

Dissertation for the degree of master
at the University of Bergen, Norway

June 2020

Year: June 2020
Title: CTF
Author: Simen Karlsen Lone

Acknowledgements

First and foremost I wish to thank my supervisors Håvard Raddum and Chris Dale, for the opportunity to do this fun thesis. Your knowledge, guidance and patience has been invaluable. Also thank you for helpful comments on the text.

Secondly i want to thank family and friends and my girlfriend Hege for support and encouragement.

-Simen Lone

Abstract

Security has become an increasingly important aspect of computer science as our lives become more interconnected with the internet. In modern times it has become near impossible to live a disconnected life. Most of us use multiple services like banking, welfare services (NAV) and social networks on a daily basis. As all these services are always online they face a constant threat of being attacked. Creating software is hard and a tiny bug can have severe results leading to a data breach. This has resulted in an increased need for personnel with knowledge of security and how to write secure code.

This thesis aims to look at some common implementation flaws with the theme of cryptography and teach how to prevent them. This was achieved by creating gamified versions of the cryptographic flaws , where the intent is to learn how to exploit the flaws. This was done by crating the challenges in a form suited for CTF competitions. Through the understanding gained by performing an attack one should also learn how to prevent the flaw in future.

The thesis is written in four parts. The first part gives a summary of the cryptographic primitives and tools needed to understand the flaws. The second describes the cryptographic flaws, and how they occurred and can be abused. A solution or fix is suggested if appropriate. The third describes the implementation process of the CTF challenges with the flaws and provides the intended solution to them (here meaning how to exploit them). The final part is a summery of the process of making the scenarios, implementing them and lessons learned.

Contents

Chapter 1

Introduction

1.1 Motivation

Internet and the ever growing online presences has come to stay. Most of us do many forms of online activity on a daily basis. It can be a personal chat with a partner or friend. A checkup on personal finance. Consuming news from an online tabloid or other news source or interacting in other online communities. This in turn has resulted in vast amounts of information about us that are stored on different servers owned by the service providers, that needs to be protected. Personal finance probably being one of the most important.

Despite companies today starting to focus on solving security issues, this has not always been the case. In the early stages of the Internet people were naive and traffic was sent as plaintext. One could simply climb up a lamp post and read what was sent over the wire with the help of a computer. The early lack of verification made it easy to set up lookalikes of web pages and harvest user credentials from victims landing on the page and then redirecting to the real site without the them knowing. Another problem is/was passwords stored in plaintext on the server. If a data breach were to happen the passwords gained by the attacker could be tried on other pages if the same password was used in multiple places.

Luckily we have learned form our mistakes and there are solutions for a few of these problems. When chatting with a partner or friend, cryptography can be used to establish secure communications between the two of you. The messages will only be readable by the sender and receiver and no one in between, not even the maker of the app. Cryptography allows us to validate identity. We can check if the website we visit actually is the site we tried to visit and not someone serving a fake version of it. We have invented better ways of authenticating ourselves with using more than a simple password, by introducing a second factor like a code generator or using sim card technology. The server does not need to know your password any more. It only knows a hashed version that can not be used to find the actual password, thus stopping the attacker from trying your password other places. Even in the case of a leak the data can be encrypted such that it is useless without a key. But alas, cryptography is hard. Mistakes have been made and will be made.

The motivation for this thesis is to showcase common mistakes done by developers when implementing cryptography in applications, hopefully in a fun and engaging way. The purpose is that the showcased mistakes will not be repeated. This will be done by creating applications with the flaws built in to them in the form of capture the flag challenges. Capture the flag, often abbreviated to CTF, has its name from the child game with the same name.

The objective of a CTF competition is the same as in the child game: Capture the flag. In a CTF competition the flag is usually a string on the format ”flag{some text}”. The flag can be turned in and the player(s) receives points. CTFs come in many shapes. There are big online events like Google’s CTF. There are smaller local CTFs where the contestants physically meet up and there are always online challenges where one can play at ones own pace.

Now to the fun and engaging part. In this thesis i will create a number of applications with common flaws that can be found in real life. Each of the applications contain a hidden flag which can be turned in to the game server for points. The flag is hidden in such a way that it can (hopefully) only be found by exploiting the intended flaw in the application. To be able to exploit the flaw the player needs to have an understanding on how the flaw came to be and how to leverage it to do something malicious.

An example of such a flaw could be a bug that causes the server application to crash if malformed data is sent. A crash is bad, but the client and server code are both written by the same developer, therefore the developer assumes that malformed data will never be sent from the client. This however does not hinder a malicious attacker from sending malformed data to the server and making the services provided by the server unavailable to all users, commonly called a denial of service (DoS) attack.

By exploiting these vulnerabilities oneself, one learns how they work, and how something that seems innocent can be exploited to cause big trouble. With the attacker’s perspective it is easier to understand how one will be attacked, and I believe this knowledge makes it is easier to figure out how to defend against attacks.

> If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle. — Sun Tzu, The Art of War

1.2 NetSecurity

1.2.1 Collaboration

This thesis is done in collaboration with Netsecurity. The company holds multiple CTF competitions a year with the purpose of training people in security and raise awareness around the topic. The intent with this collaboration is to make more CTF challenges for the platform, making it more complete. A special focus was put on challenges with cryptography as a theme, as Netsecurity wants more challenges of that type. Hopefully this thesis and challenges it contains can provide insight to a few common common attack vectors of crypto systems, and how to prevent them. The goals of this collaboration was met and the challenges in this thesis have been added to Netsecurity’s CTF platform.

1.2.2 About

Netsecurity AS is a Norwegian company established in 2009. They are located in Oslo, Bergen, Kristiansand, Grimstad and Stavanger. They deliver services and solutions that allows businesses and organisations to conduct their online work in safety, allowing cyber

attacks to be detected and stopped in an early phase, preventing severe consequences for their customers.

Their employees are the customers' most important resource. They especially emphasize competence, integrity and customer proximity. They employ the best and motivate their employees for continuous learning and improvement.

They claim to live by their reputation. They strive to be close and accessible to their customers. They want customers where they can actively support and assist their customers' business. They claim that if one choose Netsecurity, one should know that one can trust them and that one will have access to the best expertise, services and solutions on the market.

They have customers throughout the country within the public and private sectors. The common denominator for their customers is that their network and security are important to their business.

While most companies and consultants in the industry provide a wide range of services, Netsecurity works exclusively with network and security solutions, and provides solutions and services like:

- Firewall Audit
- Penetration testing
- Security assessment
- Expert consultancy within cyber security
- Netsecurity Security Operations Centre
- Incident Response
- Backup
- Disaster Recovery
- Managed IT-services

Chapter 2

Cryptographic primitives

2.1 Symmetric cryptography

Symmetric crypto is what one usually thinks when one thinks of crypto. It is called symmetric due to the fact that the key is shared by both parties. The key is used both for encryption and decryption of the message.

2.1.1 The Advanced Encryption Standard

The Advanced Encryption Standard, often abbreviated AES, is a NIST (National Institute of Standards and Technology) standard for encryption of electronic data. AES is a subset of the Rijndael family, named after its creators Vincent Rijmen and Joan Daemen, with 128-bit blocks and supporting key lengths of 128, 192 and 256 bit keys.

DES which was the old standard for encryption of electronic data had proven to be breakable in a short amount of time in the nineties. Using the DES algorithm three times with different keys resulted in a longer total key length and was used as a temporary solution. NSA had modified the algorithm before it was made public by changing the numbers making up its S-boxes[31]. There was never released any reasoning behind the chosen values for the S-boxes by the NSA, this lead to speculation that a back door may have been planted.

These two factors led to NIST announcing the open competition for a new standard. After three rounds of cryptanalysis by the worlds leading experts, Rijndael won. The Rijndael met the criteria for a strong algorithm that could run fast both in software and hardware implementations.

AES is a block chiper that works on 128-bit blocks. The algorithm as described below is run on each block. The last block is always padded such that the total length of the plaintext is a multiple of 128. In the edge case of a plaintext being a multiple of 128, a block containing only padding is added. AES consists of a loop run multiple times over the plaintext blocks. Each block is arranged in a four by four matrix of bytes called the state. The number of rounds is decided by the key length such that 128, 192 and 256 bit keys give 10, 12 and 14 rounds, respectively.

Before the first round is performed a key scheduling algorithm is run on the key producing $r + 1$ round keys, with r being the number of rounds. The first key is xor'ed with the state. Each round up to but excluding the last consists of four steps. First a byte substitution where each byte is replaced by a byte from a lookup table. This is followed by row shifting where

all the rows are left-shifted by a number of steps equal to their zero-indexed row index. The third step is the mixing of columns. Each column is multiplied by a predefined matrix in $GF(2^8)$ using $x^8+x^4+x^3+x+1$ as the irreducible polynomial. The resulting vector is used as the new column. Lastly, the next round key is xor'ed with state. For the last round only the byte substitution, row shifting and round key xor'ing is preformed on the state, and the mix columns is left out.

2.2 Asymmetric cryptography

An asymmetric crypto system is a crypto system where the key used to encrypt a message and the key used to decrypt it differs. This allows the receiver to share the encryption key without concern as it can not be used to decrypt the message after it has been encrypted.

2.2.1 RSA

RSA is an acronym made of the initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman. They first publicly described the algorithm in 1977. Clifford Cocks, an English mathematician working for the British intelligence agency Government Communications Headquarters (GCHQ), had developed an equivalent system in 1973, but this was not declassified until 1997.

RSA is one of the first asymmetric crypto systems, also called public key crypto system. RSA is based on the problem of factoring a number into primes. The problem can be framed as follows: given a big random number n find all the prime factors of the number. There is no known algorithm efficient enough to break today's key sizes [23] of at least 2048 bits. The hardness of this problem is the foundation of the security of RSA.

RSA can be described with the three steps of key generation, encryption and decryption.

Key generation: For key generation, two random primes p and q are chosen. They have to be sufficiently big such that the task of factoring the product is not feasible in reasonable time on today's hardware.

- n is computed as $p \times q$
- $\lambda(n)$ is computed. This can be computed as $lcm(p-1, q-1)$
- e is chosen, it needs to be $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$. One wants e to be as small as possible as it yields faster computation. The number 3 is the smallest possible candidate but it is not used as it has been shown that small public exponents are insecure [21]. The standard public exponent used today is 65537. This is because it is big enough to be safe from attacks working on smaller primes and gives a good trade off between security and computational speed as it is a 2^n+1 prime making exponentiation efficient to compute.
- d is the last thing computed and is the inverse of e mod $\lambda(n)$. This can be computed using the extended euclidean algorithm.
- e and n are exposed as the public key. The values of p, q, d and $\lambda(n)$ are the private key, and are to be kept secret.

Encryption: The message m is encrypted into the cipher text c as follows $c \equiv m^e \mod n$

Decryption: To decrypt the ciphertext, c is simply raised to the power d: $c^d = (m^e)^d = m^{(d*e)} = m^1 = m \mod n$. This follows from the definition above, where d is e's multiplicative inverse modulo $\lambda(n)$.

2.3 Diffie–Hellman key exchange

Diffie–Hellman is one of the first practical implementations of Ralph Merkle's public key cryptography. Whitfield Diffie and Martin Hellman implemented the scheme using the hardness of the discrete logarithm problem.

The Diffie–Hellman key exchange allows a pair to establish a common secret over an open channel without letting any eavesdroppers on the channel know the secret. Even though a secret is established between the parties, the scheme has no verification of the parties involved, thus the scheme is vulnerable to MITM attacks as the eavesdroppers can simply establish a key with both parties and forward traffic. In Figure 2.1 the idea of Diffie-Hellman is shown using paint as a metaphor. In practise this is achieved using mathematics and the discrete logarithm problem.

First the parties agree on a multiplicative group of a large prime p and a primitive root g in that group over the insecure channel. Then both parties choose a random integer and raises g to the power of it. The elements g^{s_1} and g^{s_2} are exchanged over the insecure channel. Due to the hardness of discrete logarithm problem s_1 and s_2 can not be computed from g^{s_1} and g^{s_2} by an eavesdropper. The sender of g^{s_1} receives g^{s_2}, and knowing the value s_1 raises g^{s_2} to the power of s_1 resulting in $g^{s_2*s_1}$. The other party does that same with its secret, leaving both parties with $g^{s_2*s_1}$, which becomes a common key. The key can be used to for example send traffic using a traditional block cipher like AES. Since s_1 and s_2 were never sent over the insecure channel the eavesdropper can not compute $g^{s_2*s_1}$.

2.4 Hashing

A hash function is a function taking an input of arbitrary length and returning a value of fixed length. The value returned is called a hash or digest. A hashing function is a deterministic function and that means that for a given input a the resulting value b will always be the same. From the definition of hashing given above one can observe that the domain of the function is infinite but the co-domain is finite. This means that for each value a there will exist a value a' such that $H(a) = H(a')$. This is called a collision.

Hash functions are used in a common programming structure called a hashmap. This is a list structure where fetching an element given a key can be done in constant time. It is implemented where the key is hashed and the resulting hash is used as the index in to a list where the data is stored. This allows for arbitrary length keys and not needing to know a specific index to retrieve an item. In this scenario a collision would be bad as it results in two items being mapped to the same spot in the list. Therefore one wants hashing functions where collisions do not happen in practical use.

Hashing is a primitive often used in cryptography, but for a hash function to be cryptographically secure it needs to have a few extra properties namely :

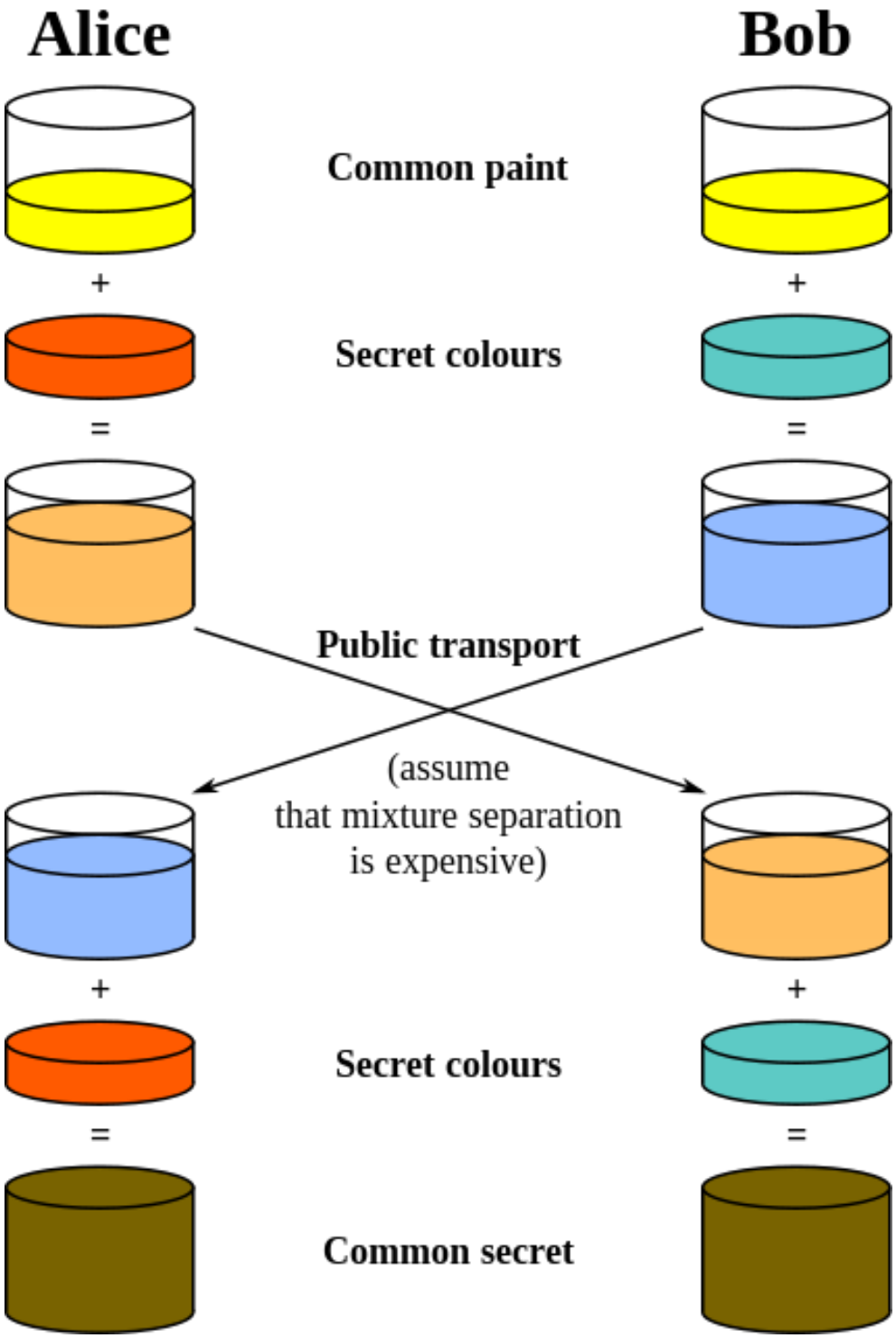


Figure 2.1: Diffe-Hellman visualised using paint mixing

- it is a one way function
- it is computationally infeasible to find a collision

A one way function is a function where one can easily compute the output of the function given the input a, but if given the output b it is computationally infeasible to compute a.

A categorically safe hash function has a strict requirement to collisions. For a hashmap, as long as collisions rarely happens, there is a speed vs collision rate trade-off that is not acceptable for cryptographic uses. If a hash is used as message check sum collisions can be fatal. Let us assume a scenario where a hash is used to provide a checksum. If a collision were to be found an attacker could change the message to the colliding input and thus altering the entire message sent to the receiver without the receiver noticing.

2.4.1 Bcrypt

In the case of a hashmap one wants a fast hashing function to minimize the time it takes to retrieve an element in the list. There are many fast hashing algorithms, like SHA-256. These algorithms should not be used to store password.

The average human does not like to remember long and complicated things, like passwords. This often results in short and common passwords. So when password hashes are leaked, instead of trying to reverse the hashing function with is be design hard to do, the attacker tries many variants of short passwords. The attacker is able to test a given amount of passwords per second. So the faster the hashing algorithm is to compute, the more passwords the attacker gets to test per second.

Bcrypt is a hashing algorithm specialized for hashing passwords. The algorithm allows a second argument to be passed to it, determining the hardness of the hash computation. When a user logs on the server has to compute a single hash to validate the users login request. The attacker on the other hand has to try to guess the user's password and try a lot of possible passwords. So a half second wait on the login page would not pose a big annoyance to the user, but would cripple the attacker, as for Ethash hashes with a high end graphics card like AMD Radeon VII 16 GB one can compute $95.3Mh/s$, compared to only a few bcrypt hashes per second [6].

Bcrypt also comes with a third parameter which is the salt. A salt is just a random number. The idea behind a salt is that users tends to use easy passwords and this often results in multiple users having the same password. This is bad as this results in two users having the same password hash, when the attacker finds the password of one user the attacker gets the second user for free. This can be prevented by giving each user a unique salt. This is simply appended to the password and the combined string is hashed. In the case of Bcrypt it is passed as the third argument. Since each user has a unique salt, this makes the string passed for hashing unique pr user even if the password part of the string is the same. This results in unique hashes, thus eliminating the attacker's one for free opportunities.

Chapter 3

Applied Cryptography

3.1 Examples of crypto in software

3.1.1 An IoT device example

When you buy a smart fridge there is probably a micro controller somewhere, controlling the temperature. This micro controller runs code, and the company selling you the fridge would wish to update the software of the fridge if there is something wrong with it and it is not discovered during production. If the fridge is of the more advanced/expensive models, it probably comes with a WiFi chip and an app allowing the owner to monitor and change the temperature while not home or some other WiFi controlled gimmick. You as the owner of the fridge do not wish other people to be able to change settings on your fridge.

With this scenario in mind we will now go over how and where cryptography is used and implemented. Let us start with the chip software. To avoid unauthorized software changes to the fridge the company would typically deploy asymmetric encryption. The company signs the update with a private key and embeds the public key as a part of the original installed software of the fridge. When someone tries to install a new software update, the fridge can validate the signature on the update using its public key to confirm that it came from the company and not some malicious third party.

As this fridge uses WiFi it also has to implement the required standards for WiFi connection, 802.11[2]. This standard requires quite a few cryptographic primitives as it has to implement Password-Based Key Derivation Function 2 (PBKDF2). This introduces HMAC using SHA-1 as the hashing method. For the actual sending of messages the Counter Mode Cipher Block Chaining Message Authentication Code Protocol (CCMP) is used. This uses AES in Counter mode (CTR) and combines it with CBC-MAC to provide message integrity and authenticity. There also needs to be a safe way to disclose the pre-shared key of the WiFi network to the fridge.

Then for the fridge to talk to the app running on a phone outside the WiFi the fridge needs a way to establish a connection between the phone and itself. Neither the phone nor a WiFi router typically allow incoming traffic. In the phone's case it is not allowing port binds and in the fridge's case it is typically behind a router requiring port forwarding to make a bound port accessible outside the network. To solve this a mediator, often provided by the company, is used. The fridge will connect to this mediator. The WiFi will provide security to the packet sent from the fridge to the router but, when it crosses from the router to the mediator it is

unwrapped from the WiFi wrappings and sent as plain text by default. So a new layer of encryption is deployed to ensure that the traffic gets safely to the mediator. This is typically http traffic combined with TLS (sent over a TLS connection). The phone then connects to the mediator over TLS as the fridge did.

A new problem arises as there are thousands or millions of fridges and owners with apps on their phones. Who should be connected to witch fridge? A new crypto concept needs to be introduced: the identity. We need a way to identify a fridge, a user, and validate a user's claim to ownership over said fridge. The fridge is usually identified by sending a serial number over the secure channel in such a way that the identifying packet cannot be replayed or spoofed. Timestamps and Message Authentication Codes (MACs) can be employed to ensure this. The user typically creates an account and enters the serial number of the fridge, found in a manual or on the device itself. So when the fridge and the user claim the same serial number the mediator knows how to pair them up.

In this scenario the user also needs to be identified and this is usually done with a password and hopefully a second factor like a verification code on SMS or another second factor. The mediator stores the user's password so it can compare it with the user's entered password. Here hashing is used so that the user's actual password is not stored on the server in case of a breach of the server. The code used as the second factor must not be guessable and this introduces the need for a pseudo random number generator.

As we can see from this small example, there is a lot of crypto all around us at all times. In this small example we have seen use of AES in multiple block modes, MAC algorithms, the SHA-1 hashing algorithm and the use public key cryptography used both for integrity and authenticity. Cellular networks were also briefly named and that also bring with it a few extra primitives.

3.2 Identity

3.2.1 Certificates

A digital certificate functions much like its real-life counter part in that it is a document proving something about the owner of the document. It is issued by an authority and singed by this authority. This signing of the backing authority is what gives the certificates their trust and validity.

In the case of a public key certificate it proves the public key on the certificate belongs to the owner. By being able to verify the owner's public key one can establish a secure connection with the owner as only the owner has the corresponding private key, thus rendering man in the middle attacks impossible. This is the main point of digital certificates: being able to identify and establish secure connections between two parties.

Identity

As mentioned, certificates online can be used to establish the identity of the owner. The most common standard for certificates is the X.509, described in RFC 5280 [18]. X.509 certificates contain three parts. A tbsCertificate, signatureAlgorithm and signatureValue. The identity is provided by the tbsCertifcate as this is the part of the certificate that gets signed. This part

contains the fields version, serialNumber, signature, issuer, validity, subject, subjectPublicKeyInfo. If the version of the certificate is greater than one, issuerUniqueID, subjectUniqueID and extensions may be included. The first two extra fields requires version two and extensions require version three.

Version

The version field gives the version of the certificate and allowed values are v1, v2, v3. Version 3 is what is commonly used.

serialNumber

is the certificate's serial number and is an integer chosen by the certificate authority and must be unique for all certificates issued by that certificate authority. The integer must be in the range 0 to 2^{160}.

signature

The signature field contains the signing algorithm of the certificate and must have the same value as the signatureAlgorithm field. As to why there is a duplicate field there is no good explanation, according to Peter Gutmann's X.509 style guide [20]:

> There doesn't seem to be much use for this field, although you should check that the algorithm identifier matches the one of the signature on the cert.

The signature is of the type AlgorithmIdentifier containing the fields algorithm and parameters. The algorithm field being a string identifying the algorithm used for signing and parameter being an optional field used to specify any parameters to be used with the algorithm.

Issuer

Issuer is a field detailing the certificate issuer, also known as the certificate authority. This field contains a distinguished name, DN for short. The issuer field is defined in X.501 as Name. A Name is defined as a RelativeDistinguishedName. The name contains the following fields: Country, State/Province, Locality, Organization, Organizational Unit, Common Name. Example string C=US, ST=Arizona, L=Scottsdale, O=GoDaddy.com, Inc., OU=http://certs.godaddy.com/repository/, CN=Go Daddy Secure Certificate Authority - G2 Validity.

Validity

Validity contains two subfields, Not Before and Not After. These fields contain dates and give the validity period of the certificate.

Subject

The subject field is of type Name and is the same as described for the issuer. This details the identity of the owner of the certificate. For web certificates the common name would be the domain.

subjectPublicKeyInfo

The field subjectPublicKeyInfo has two subfields, namely algorithm and subjectPublicKey. Algorithm is of the same type as signature and has an algorithm string and parameters. This provides information on which public key algorithm is to be used with the public key and corresponding parameters. SubjectPublicKey is a byte array representation of the public key.

subjectUniqueID and issuerUniqueID

These fields are unique identifiers that can be added to the certificate. To uses these identifiers one must use at least version 2.

Extentions

This field is a list of optional extentions. Extensions require version three and allows users to define private extensions to carry user defined information in the certificate. In the extension list they are listed with the following properties. extnID, critical, extnValue. ExtnID is the id of the extension. Critical is a boolean that determines if the field is critical to the certificate. If a validator does not support the extension type used in the certificate and it is marked as critical, the certificate must be treated as invalid. The last field is a value and contains the actual data of the extension type.

Subject Alternative Name is a an example of a commonly used extension. This allows a certificate to specify multiple subject names. This can for example be used to add multiple domains to a certificate, which is a common practise. Examples of this can be adding the www sub domain to the certificate. Other subjects that can be added to the alternative name is IP addresses, mail addresses, or a Uniform Resource Identifier (URI)

signatureAlgorithm should be the same as signature, and **signatureValue** is the digital signature computed using the signature algorithm.

3.2.2 Certificate Authority

A certificate authority is someone that issues certificates. The certificate authority is primarily tasked with two things. Signing certificate on requests and validating if the claims on the certificate are true. A company certificate authority may have a unique certificate for every employee. The HR department checks if the employee is who the employee claims, through ID check or similar, and if the ID is valid a certificate can be produced. This certificate can then be used to produce id cards capable of signing and can be used for authentication. This is done by passing the certificate to the authenticator which checks if the certificate is valid and if that is confirmed, sends a challenge to the ID card encrypted with the public key in the certificate. The ID card has the corresponding private key and thus can send a correct response to the challenge.

Let's Encrypt [1] is a popular certificate authority used for generating certificates for domains. A request can be sent to Let's Encrypt and Let's Encrypt will connect back to the requested domain using DNS and send a challenge response to a bot that needs to be running on the server pointed to by the DNS. If the bot sends the proper response Let's Encrypt will create a valid certificate for said domain. This validations of the validity of a certificate request assumes that only the owner of the domain can alter its DNS.

In both of theses examples a certificate was signed by a certificate authority. The signing process works by sending a signing request to the certificate server, a common format that is used for this is PKCS #10. This request closely resembles the subject part in the tbsCertifcate section of a certificate, with the addition of the public key and its corresponding algorithm, and signing algorithm. The certificate authority takes the request and generates the tbsCertifcate part of the final certificate and hashes it using the algorithm specified in the signing request. The hash is then encrypted using the certificate authority's private key. The resulting value is the signatureValue. The signing algorithm is the signatureAlgorithm. tbsCertificate, signatureAlgorithm and signatureValue is appended to a file in that order which becomes the complete certificate.

Validation

Validating a certificate is done by checking if the current date is in the interval set by the validity section of the certificate. Then one checks if the algorithms that are used by the certificate is supported. Certificates with old schemes will be refused. If the certificate is version three all critical extensions if present are checked for support. One checks if the certificate has been revoked by checking the revocation list provided by the certificate authority. Certificates can be revoked if the cryptographic algorithms used are shown to be weak or if a private key is reported leaked by the certificate owner. When all these checks are passed one takes the tbsCertificate and hashes it using the algorithm specified. Then one decrypts the signatureValue and compares the newly generated hash with the decrypted one. If they match the certificate is valid.

Trust Chain

From the description given above, the certificate authority's public key was simply trusted and it was never explained where the trust came from. The answer is that the certificate authority has its own certificate. This certificate was used to get the server's public key. This creates a recursion as the certificate authority's certificate must be signed by some entity as well. This chain of recursive lookups and validations of certificates is called the trust chain and ends in a root certificate authority. The root certificates are simply self signed. One does not usually expect self-signed certificates as anyone can make one. In the case of the root certificate authorities their certificates are pre-installed, either in the OS or in a browser. So when the chain reaches a root certificate authority if it matches the one on the machine, it is just trusted.

Certificates in nature have expiry dates and so do those of the root certificates. The problem of getting new certificates from them is solved with updates to the OS or browser.

3.2.3 Authentication

Authentication is the process of validating an identity claim. When a user logs on to a web page the user provides an identity, like the username or email, and a proof of the identity claim, for example a password. There are three classes of proofs the user can give: knowledge, ownership, inheritance.

Knowledge is the typical proof used, usually in the form of a password. An example where knowledge is not a password is when the user answers a CAPTCHA. Knowledge

is demonstrated in the form of being able to transform an image to text. This proves that whoever is using the site is not a bot/script but a human based on the assumption the bot would not be capable of doing the same. The main problem with passwords are that they are easy to forget, resulting in short or guessable passwords as they are easier to remember.

Ownership is a proof factor that is possessed by the user. This can be a mobile phone, yubikey, code generator like the BankID device, or a smart card. These are typically validated by either getting a code from said device and getting it validated on the server end or, in the case of a locked door, directly on the identifying interface. The weakness of this factor is that you need to have something on your person to gain access and as it is a physical item, it can be lost or stolen.

Inheritance is the last type of proof and is something the user does or is. The most common examples of this are fingerprint or face scan, which are the standard on phones besides the 4 digit pins. There are other inheritance proofs like iris scan, voice recognition and DNA scans. The problem with inheritance is that fingerprint sensors and the like are often inaccurate. The sensor often fails to read on first attempt and physical factors like a dirty wet thumb or a wet sensor may increase the sensors inaccuracy. This is called a false negative. Fingerprint sensors have also been shown to be prone to false positives if attacked by a so called masterprint [30]. A masterprint a artificially fingerprint created to be the average of common fingerprint features.

3.2.4 Multi factor Authentication

Multi factor authentication is slowly becoming the norm for sing-in's and is the standard on web pages granting access to vital services like banking and other government systems like tax. It has also become quite common for email services. This is good as one can typically reset the password of other services through the mail and a forgot-my-password button.

So what is multi factor? In a typical login one provides either a user name or email address together with a password. Password is the single proof, or factor, for the claimed identity. The user's username or email is looked up in the database and a corresponding password hash is found. The user's entered password is hashed and matched with that in the database. If they match (are equal) the user is granted accesses to the resource.

In the case of a multi factor the user is asked for a second, or more proofs/factors, password being the first proof, for the authentication. This can be a time based code generated from a known seed on a small handheld device, an ownership based proof. In this case the server has all the users' seeds and can generate the matching codes. Other examples can be pre-printed sheets with numbers on them and the server requests one at random. SMS is also commonly used, one receives a code on a text message and uses it as the second factor. There are also versions of this where a simple yes or no dialog pops up on the user's cell phone. The idea behind multi factor is to make sure that even if a password is guessed or leaked, one still needs the second factor to gain access, and this is not stored with the password. One note is that inheritance factors are for some reason rarely used as a second factor, only as primary factor as an alternative to the password. This could be because inheritance proof often has a high false negative rate.

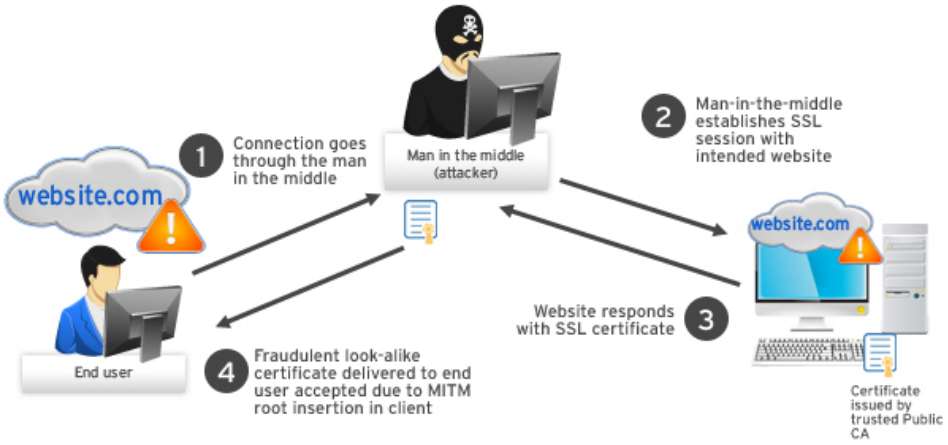


Figure 3.1: Illustration of a man in the middle attack [17]

3.3 Attacks on identity

3.3.1 Man in the middle

Man in the middle, often abbreviated to MITM, is an attack where the user communicates with a server and the attacker ”sits” somewhere between the server and user in the network chain. From this position the attacker can view and edit traffic sent between the server and the user. See Figure 3.1.

If the traffic is unencrypted the attacker can get sensitive information or login credentials. Using the found credentials the attacker can use these to forge a request on behalf of the user. The attacker can also misinform the client by changing answers from the server. These are a common attack vectors that can be used after a MITM position is achieved by the attacker.

3.3.2 SSL Stripping

SSL stripping is a technique done from a man in the middle position. When a user connects to a http server on port 80 to perform a login he will typically be redirected to port 443 and establish a TLS connection to the server and send the credentials over the secure channel.

The attack works as follows. The attacker observes the user’s GET request to the login site. The attacker stops this request from being sent through, and the attacker establishes a connection to the server requesting the login. The attacker gets redirected to the 443 port and creates a proper TLS connection and sends back the login site over http to the user, pretending to be the server. The user, assuming the site has no https/TLS support or not noticing it is lacking, conducts his login over http and the attacker receives the users credentials and makes a copy. The attacker then forwards the credentials over his TLS connection and forwards the response to and from the server to the user. SSL stripping can also be applied to whole sessions, letting the attacker see all of the user’s traffic despite the server enforcing end to end encryption.

3.3.3 HTTP Strict Transport Security

HTTP Strict Transport Security, HSTS for short, is a mechanism implemented in browsers to prevent SSL stripping and other downgrade attacks. It works in the following way. When a user establishes a secure TLS connection to the web server for the first time, a HSTS policy is sent to the user's browser stating the allowed connection types to the server. This policy is kept by the browser for future connections. So in the scenario where a malicious attacker tries to downgrade the connection type and the user has connected to the site prior to the attack, the browser will check its HTST policy for the site and only accept a secure connection. When the browser then asks for a secure connection from the attacker, the attack will fail as it does not possess the private key for the certificate used by the original site and can thus not establish a valid TLS connection. Also, browsers like Google Chrome comes with preinstalled HTST polices for certain domains, like Google's own for example.

3.4 Reverse engineering

Reverse engineering is the art of taking an application and tearing it apart piece by piece to discover its inner workings. This is typically done in scenarios where source code is unavailable to the examiner. This is a common technique used by hackers to look for client secrets.

3.4.1 Client Security

There is no such thing as client security in the sense that when an application runs on an attacker controlled machine one should always assume that he knows all the secrets the client software knows. A typical example of this are hard coded keys [29]. If a key is hard coded, some one will find it and abuse it. An example of this could be an app with hard coded credentials to an URL "secret" API in the app. With this the attacker can now use the credentials and API for malicious purposes and the developer might have assumed that the API and key would never be known and have thus not implemented proper access restrictions. Also if all the clients use the same hard coded credentials that means that the attacker can potentially pollute the information used by other clients as they all uses the same credentials to access the API. This only applies to API's with create or update interfaces.

A common setting with client secrets is online gaming. The game designers want the server load to be as low as possible to support as many users as possible, and thus offloading as much as possible to the client side. This has resulted in clients doing collision calculations, resulting in attackers being allowed to walk though walls and be invulnerable. In an online game the game client sends a constant stream of the player position to the server and the server sends back the position of the other (visible) players to the client. Collision detection simply calculates if the current position is a valid position or not. A player walking up to a wall is a valid position but walking through a solid wall makes no sense. So the collision detection will validate the position as invalid and restrict the players movement. The problem arises when this detection is done on the client side as nothing is stopping the attacker to change the game client in such a manner that it will not restrict the players movement allowing the player to report a position inside the wall, to the server, the perfect hiding spot. Invulnerability is the same principle but opposite. If we assume in a first person shooter game

one simply patches the ”did the bullet hit” function to always return false, it will result in the game client never sending a hit to the server.

3.4.2 Android

Mobile platforms are a common place to find client security flaws as the apps are isolated from each other with limited communication between them, and the user is considerably more restricted as the user does not have the same privileges as one typically has on ones own computer. This has fooled developers into a false sense of security, but as we shall see this is not the case. Android apps are typically written in Java or Kotlin, with both languages compiling to java byte code and run on a Android device. Due to the nature of the runtime environment there exist decompilers for dex files back to java. These decompilers typically produce a more accurate representation of the original source code than assembly decompilers, as less information is removed in the translation from java to dex than from C to assembly. If a binary is not stripped, function names and variable names are also kept. This makes it easier to reverse engineer Android apps than their assembly counterparts. Hard-coded strings can be found with a simple regex expression. So it becomes very hard to hide client side secrets.

3.4.3 Rooting

As a user, the interactions with the phone is done through its UI and apps. The apps have restricted access to the phone. This leaves the user only able to perform certain actions, but as the Android operating system is built on Linux there always exists a root/superuser which has full access to the phone. This superuser is hidden from the user and is not accessible from the UI. The process of obtaining the root user is called rooting. This is done by first unlocking the android phone’s bootloader. The bootloader on android phones act as both the bios and a traditional bootloader like GRUB. It is placed as the first partition on the phones internal flash memory and is where the devices code execution starts. The bootloader under normal use is responsible for booting the Android OS but it also manages other things like installing updates to the android system partitions and allow one to boot in the recovery partition of the device. This partition is a small ”os” that allows to restore the android system partition if it where to be damaged.

By unlocking the bootloader on can use it to write to the partitions of the internal flash. This is used to install a custom recovery to the recovery partition. By booting to recovery on can now use the newly installed custom recovery to modify the android system partition. This can be done directly form the recovery but the custom recovery is a convenience. This can be done by downloading zip files to the phones sdcard. These can then be read and installed by the phones custom recovery.

To root the phone one installs a zip file with a root patch from the custom recovery. This will modify the android system partition in such a way that apps can request to execute as the root/superuser of the system. This allows apps to circumvent all restrictions placed by the android system and lows for expanding the original functionality of the Android OS. An example of this is one can use this to disable vibration for certain apps like the finger print sensor.

3.4.4 Reverse Engineering Tools

Magisk

One of the problems with rooting a phone is that bank apps and similar apps needing a high level of security consider rooting unsafe as other apps can gain the opportunity to affect the them. This has lead to root checks where the apps check if they are allowed to request root access, and simply refusing to run on the phone if this is case. Google has even made an API allowing one to check if the file system of Android has been modified. This makes rooting a phone problematic as rooting would be detected as such a change. Magisk [8] was developed as a solution to this. Magisk is a modification of system but it will not change the Android system partition and instead modifies the boot image. This allows it to be undetectable by any root check but still allowing root access to the Android system. Magisk is installed as a zip file through the custom recovery as traditional root modifications.

Frida

With root access to the phone, tools like Frida [7] can be used. Frida allows dynamic analysis of Android apps. What this means is that one can run the app and examine its behavior instead of simply studying the source code. Frida comes with two components: A server installed on the phone and run as root, and a client run on a computer.

Frida lets one hook into the application's functions. One can overwrite functions, get custom functions called before the actual function is called and do a custom function after a function is done, but before it returns it value to the caller. One can also interact with functions already present in the app. All this is done from a simple JavaScript api. It also has supports for python scripting.

Wireshark

Wireshark [14] is an open-source network capture and analysis program with a graphical user interface. Wireshark allows the user to capture a copy of all network traffic going to and from a network interface. It will even allow one to capture traffic not intended for your interface if the interface itself supports promiscuous mode. Promiscuous mode is best explained with an example. In the case of WiFi there is nothing stopping a third party from picking up the signal sent between two devices. A WiFi card used in none-promiscuous mode will sort the traffic it receives and only forward traffic which matches its mac address, in other words traffic that was intended for it. In promiscuous all traffic is forwarded.

Wireshark also lets one analyze the traffic captured. It provides support for hundreds of protocols. It comes with many filters allowing one to filter on protocols, network layers, ports, ip, mac addresses, sender, receiver to name a few. It also has support for following specific streams between two parties.

Burp suite

Burp suite [12] is collection of tools created by PortSwigger. The main part of the Burp suite is its web proxy used for MITM'ing. This allows the user to inspect and edit http traffic going through it. It has support for analysing https by setting up its own certificate authority. Then by installing its certificate on the client the burp proxy will generate a certificate for every

requested site by the client, decrypt the requests from the client and establish its own connection to the server, forwarding the clients request. As Burp generated the client's certificate it can inspect and edit all traffic coming from the client and since it was Burp that established the connection with the server all traffic coming from the server can also be inspected and edited by Burp.

Chapter 4

Capure The Flag

CTF, short for 'capture the flag' and sometimes also referred to as a war game is a broad term encompassing many forms for security challenges and competitions. The core concept is that there is some form for vulnerability in hardware or software which that must be exploited to retrieve a flag, this is where the name capture the flag comes form. The flag is usually a token, often a string, which only can be retrieved by exploiting the vulnerability. The token is often in the form flag{something} or in another known format so that the player easily can confirm that the challenge is completed. The flag can be turned in to prove that one completed the challenge. In some CTFs this results in points on a scoreboard and in other CTFs this is grants access to the next level.

CTF challenges are often separated into different categories. Common categories are reversing, pwn, steno, web, forensics, crypto and "boxes".

Reversing refers to challenges tied to reverse engineering. In these challenges one usually gets a binary or executable and the goal is to figure out how the binary works so that the flag can be extracted from it. A common example of this is a binary that wants a password and the flag itself is the password.

Pwn is all about binary exploitation and how to get remote code execution. The player usually gets a binary with a flaw in it and a copy is running on a server containing the flag. This can be a binary that has a format string vulnerability, or a buffer or heap overflow. This can be as simple as overwriting a return pointer on the stack by performing a buffer overflow on a binary hosted on the server or something more convoluted as performing a ROP chain attack as the stack is not executable.

Steno is short for steganography and is the art of hiding messages in plain sight. A common example of this can be hiding the flag message in an image by appending the message as bytes trailing the image data, or hiding morse code in a sound clip as high frequency pitch outside of the human hearing range.

Web is short for web application and revolves around exploiting web applications. This can be exploiting a plugin on a wordpress site, SQL injections, enumerating a domain for unprotected sites, exploiting JWT tokens not done properly and any other web related vulnerability.

Forensics revolves around finding information. The player gets a log, a network traffic dump, a USB dump or some other big collection of data. Then the challenge typically is to sort the information and extract the valuable information from the noise. One example could be to get a USB dump where the users at some point entered a password on a USB keyboard.

Crypto challenges typically revolves around getting a file decrypted or to exploit an im-

plementation flaw in a crypto system. Examples can be as simple as brute forcing a password from an encrypted file to exploiting a padding oracle flaw in a web server.

Boxes is usually a multi-stage challenge. A box is a server with one or more services running on it. If only one service is running this can be exploited, with a known or a custom vulnerability to gain some form of remote code execution on the server. In the case of multiple services one often has to use one weakness in a service to gain information like a username or password to another service and from there exploit a service as a logged in user. Once remote code execution is established, one wants to spawn a reverse shell on a server so that the system of the server can be traversed. It is a common practice to leave the flag as a text file in the user's home directory on linux and freeBSD, or on the user's desktop on hosts running Windows. Once a user is done there is a way to gain admin or root on the machine. This is done by exploiting some flaw in the machine only accessible by the user like a program running with elevated privileges or a kernel exploit. The flag on linux resides in the /root folder which requires root user access to be read. On Windows it can be found in the administrator's desktop.

There are multiple forms of CTFs. There are always the online challenges like hackthebox, overthewire, bandit, hack this site and a lot more. Then there are the annual competitions like Google's CTF, pico CTF, DEF CON, TG Hack and so on.

These CTFs come in different formats. There is the most common, the 'Jeopardy style', but there are also 'attack/defence' and 'king of the hill'. The jeopardy style has its name from the American TV show because of the way the challenges are chosen. They are hosted on a web page and divided into categories, typically the ones listed above. The player(s) can freely pick which challenge to start with. Some competitions operate with challenges being locked so that one has to complete one of the easy challenges in a category to unlock the next. The challenges reward points and are usually weighted on difficulty. So challenges gives less points when the number of solves increases, sometimes this decrease in value is applied to all players scores other times the players who have already solved it keep their points.

Attack/defence is a CTF style where each team gets their own server which has a number of vulnerable services on them. Both run the same services. The goal is twofold. Exploit the opposition's services and protect your own. The protection is done by setting up firewall rules to stop certain types of incoming traffic. It is of course not allowed to block all traffic as the services are expected to work normally. This can be tested by a third party bot that uses the services and reports back if they do not work as intended. One does also want to monitor the opposition attempts as their exploits can potentially be used against them or other teams that has not patched that exploit yet. The attacking is done in much the same fashion as attacking a box with the added complexity of a security team activity fighting you back every step of the way.

King of the hill is a version of attack/defence but instead of one server per team, there are many neutral servers. They have different vulnerabilities. The goal is to get access to as many servers as possible and then holding them by protecting them in the same fashion as in attack/defence. The team holding a server gets point for each time unit they hold a server multiplied by the number of servers they hold.

(a) The original image (b) The encrypted image

Figure 4.1: Image encrypted using ECB mode

Making crypto challenges

Most crypto challenges fall into two categories, namely breaking a cipher or breaking the implementation of a cipher. In the first case one executes an attack directly against the chiper/cryptographic primitive. Examples of this can be cracking an MD5 hash or breaking a rotor machine like Enigma. The common theme her is an older cryptographic primitive with a known flaw. One can of course also make a weak primitive with an intended flaw. These challenges are valuable tools for learning about mistakes done in the past but will probably not be seen in modern software.

The other case is when the cryptographic primitives are unbreakable but the implementation is flawed. A good example of this is using AES for encrypting an image. AES with a 128-bit key is not breakable by today's computing power, but if AES is directly applied block for block to the image, then the same plaintext values get the same chiphertext values every time. This results in patterns in the ciphertext emerging as one can see in Figure 4.1. One can not find the key used to encrypt the image but it is not needed either. Another example of this which we will get back to is the padding oracle where one can decrypt entire messages in a linear fashion without knowing the key.

4.1 CTF crypto tasks

4.1.1 Certificate pinning

Introduction

When analysing an app, one good way to get a lot of information without reading through all the code, or even worse, the compiled code, is to look at the network traffic. A common thing to do when analysing an app is to set up some form of network traffic capture. Wireshark is often used for this purpose as it lets one capture all incoming and outgoing traffic on a network interface.

The problem with the Wireshark approach is that if the traffic is encrypted and one does not have the decryption key, little information can be extracted.This is because most traffic is http with tls encryption. Creating a TLS connection means fetching the server's certificate and validating it. If the certificate is valid one uses the server's public key to negotiate a session key and use it to encrypt the session. Since this is done with asymmetric cryptography one can not retrieve the key with Wireshark and thereby not read the content of the encrypted

traffic.

Burp solves this problem by acting as a man in the middle. When the client asks for a certificate, Burp generates one that Burp itself signed and knows the private key of. This allows Burp to unmask the encryption. Burp then establishes a legit connection to the server and forwards the unmasked traffic over its connection to the server. The server responses are in the same way forwarded to the client.

However, this does not explain why the client would validate the certificate issued by Burp as a valid one. Simply explained, all certificate are issued by a Certificate Authority and these are trusted with blind faith. The public keys of these CA's are shipped with the operating system itself, and stored in some form of certificate storage, for example in the case of a Mac OS it is in the keychain application. One can add more CA's to this cache. So Burp simply adds itself to the blind trust list and all certificates signed with Burps root certificate are now treated as if they came from any other CA.

As mentioned above OS'es comes prebundeled with certificates this in turn forces an application to trust the certificates the os trust. The problem then aeries that with a proxy like burp all traffic encrypted using a certificate can be decrypted by burp. This is done by burps web proxy, when an application tries to establish a connection and fetches the servers certificate burp exchanges it with one it has signed with its root certificate. Knowing the private key of the newly forged certificate burp can decrypt the traffic and the application can not detect this as the root certificate used by burp is trusted by the OS.

To prevent this applications instead comes bundled with the expected certificate. Thus the application first checks if the certificate sent by the server is the same as the one it came bundled with. If the certificates matches it precedes as normal and if they don not match the application assumes fowl play. This bundling and checking of certificates is referred to as certificate pinning.

Why this attack

Certificate pinning is a common occurrence in smart phone apps. In fact it is so common that http libraries like Volley [13] and OkHttp [10] have support for certificate pinning. The reason certificate pinning is so popular on mobile devices is that it gives added security and it is easy to update certificates if they expire. If a pinned certificate is about to expire one fixes it by pushing an update to the app. This will be automatically downloaded by the phone resulting in no connectivity issues.

The motivation for undoing the certificate pinning is that an independent third party can analyse an app for security purposes. Examples of this are checking if an app is malicious, if it leaks unnecessary information or analysing it for security flaws.

The attack

Certificate pinning as stated above is done by implementing a custom way to validate certificates provided by the server. This entails that each attack will be different, so I will give some common examples on how to unpin apps.

In the case of no certificate pinning one simply needs to add the certificate of one's own CA to the Android system certificate cache, as user trusted certificates are not trusted by an app by default.

Another common way to do certificate pinning is simply shipping the certificate with the application. This can be done by by adding the file asset folder and loading it to the trust manager during run time. To bypass this method simply decompile the app with apk-tool and then change the certificate with one with a known private key and recompile with the apk-tool, resign the app and zip-align it.

The last case is when the certificate or a hash of the certificate is hard coded into the app. In that case one has to decompile the app and find the hard coded check and patch it or change the app's byte code in such a way that it will accept a different certificate.

Frida can be combined with some static code analysis to ease the patching of the code by letting one overwrite the functions performing evaluations dynamically. Also worth noting is the ”Universal Android SSL Pinning Bypass with Frida project. This is a prebuilt Frida hook for removing common forms of certificate pinning.

4.1.2 Validation and Enumeration

Introduction

This attack is a continuation of the certificate pinning challenge. After the certificate challenge is completed the participant has a way to intercept the traffic between the server and Android app. The participant should therefore be able to discover the simplified JWT token sent between the client and server to do authentication.

This token based authentication flow has two flaws in it, which when combined allows the participant to login as any user. The first flaw is in the JWT token library used on the server, as it accepts a token with the None algorithm type. This allows the contestant to forge tokens. The second flaw is an enumeration flaw, allowing the contestant to enumerate user ids as they are only integers. These two flaws combined lets the contestant log in as any user.

Why the attack

Enumeration is a common attack vector allowing one to extract information which was not intended to be found. It is a common attack vector and is an essential part to solving CTF challenges. Common examples of this is enumerating paths on web servers to find ”hidden” directories not linked to by the main page or other services running behind an obfuscated URL. There a countless examples of critical information being exposed through a web server and easily found with a some simple enumeration. There exist pre-built lists of common patterns to check, and automated tools for the enumeration.

An example of this was the flaw in the booking site collaboration between the municipalities Bergen, Stavanger, Ålesund and Fjell [32]. The booking system allowed one to book facilities and equipment like sport courts, venues, meeting rooms, music instruments, etc. There where thousands of such ”resources” that can be booked. The booking system lets one view what was booked or available. The response containing the booked items returned more data then it should. It included information like, full name, phone number, e-mail address, full address, personal identity number, comment, attached documents and gender and age groups of attendees. The URL returned by the system for the attached documents should have been a secret URL that should be unguessable, but an integer id was used resulting in an enumeration attack being possible. By changing the integer one would get access to other documents on the server. Some of the attached documents were ID cards, scanned passports,

tickets, visas, family photos, contracts and e-mails. This is a good example of why enumeration flaw is often paired with some form of broken authentication.

Broken authentication [28] is listed as #2 in the OWASP top 10 vulnerabilities, and is described as follows

> Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

The JWT token libraries with the None flaw is an example of this. A JWT token consists of tree parts: a header, payload and a signature. The header contains two properties the ”alg”, short for algorithm, and the type, which is JWT in this case. When a JWT token is validated the algorithm is collected from the header and a signature is computed using this algorithm. This signature is matched against the tokens own signature part. If they match the token is considered valid. Note that signature algorithms requires some key. Either a private key if an asymmetric algorithm is used or a pre-shared key if a symmetric one is used. The problem is that there is a third category of signing algorithm, the None algorithm.

Tokes with the None algorithm were supposed to be used on already trusted tokens. For example, if the token is validated by the public facing part of the server and is to be sent further in to backend to other services, the singing part of the token is not needed as there is no point to re-validating it every step of the way. Thus the public facing part of the front end could validate the token, set the alg type to None and strip away the signing part of the token. The problem with this is how the libraries validating the token were implemented. They would have a function to validate taking an argument of a key and a token. The key would be optional so when it was called it would check the header, find the None algorithm, and validate it as valid as a None token has no signature to be checked and is therefore always valid. This would allow the attacker to forge a token with any values desired in the payload, opening for impersonation attacks.

4.1.3 Padding oracle

Introduction

The padding oracle attack was discovered by Serge Vaudenay [33], where he introduces an attack against crypto systems utilizing the CBC mode of encryption/decryption. The attack exploits information gained from the system about the correctness of the padding of a crafted message. The ability to query the system about the state of padding for any message is called the oracle. This information can be leveraged to retrieve the plaintext from previously sent messages. Giving the user an error message if the padding is wrong seems like reasonable information to disclose, but as this attack demonstrates it can lead to a vulnerability.

Definitions

Throughout this section the following definitions will be used

- b is the number of bytes in a block
- $N = b *$ number of blocks

- PL is the plaintext, i.e. the decrypted text of the original transmission.
- CL is the cleartext, the decrypted message of the current transmission.
- AC means attacker controlled and is a block chosen by the attacker
- D is the bytes of a decrypted block
- OC means original ciphertext, and refers to the previous block sent under the original transmission.

Why this attack

The reason for choosing this attack as a CTF task is that it is a good example of something which is easy to implement in a way that can lead to a vulnerability. As stated in the introduction, giving the sender an indication that a sent message was misformated is reasonable, but as this attack demonstrates it can lead to compromising previously sent messages. This attack was first discovered in 2002, but until this day padding oracle vulnerabilities are still found. A few examples are Zombie POODLE and GOLDENDOODLE [27]

Requirements for the attack

1. The system utilizes the CBC mode of a block cipher
2. A padding oracle
3. Being able to intercept the traffic between client and server
4. Padding failure should not reset the connection's session key

Padding

There are many implementations of padding. For the purpose of explaining the attack we will here introduce a simple padding scheme. Given a block length of b bytes and a chunk of message of length m where $b > m$, padding is used to extend the length such that $m + p = b$ where p is the number of padded bytes. In the case of $m = b$ one adds a block of only padding.

The padding also needs to be distinguishable from the rest of the message. To solve this the last byte of the cleartext is simply the number of padding bytes, and the other bytes of the padding hold the same value. This makes the padding easy to remove. Simply read the last byte and remove this many bytes from the message. The padding can be checked, testing that all the padding bytes have the same value.

The attack

This attack can be described in 3 steps

1. recover a single byte of the plaintext
2. recover one block of the plaintext
3. recover the full plaintext

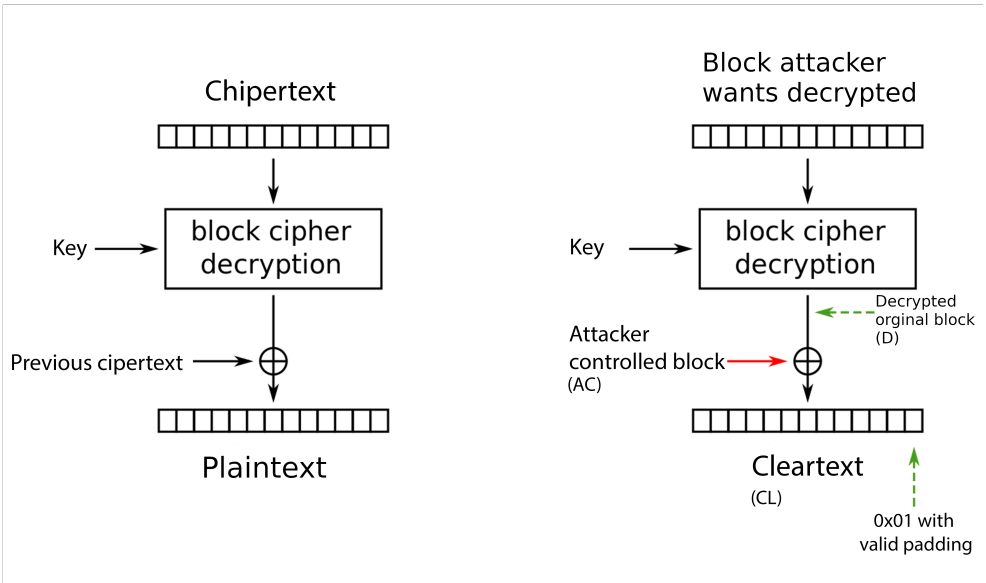


Figure 4.2: left the original scenario, right the attacker scenario

To start, one chooses a random block of length b and sends it as the second to last block. Then one chooses the block one wants to decrypt and sends it as the last block (the padded block).

One byte From the definition of the padding one can observe that any message ending in the byte 0x01 has a valid padding. Looking at Figure 4.2 the decrypted ciphertext gets xor'ed with the previous ciphertext to get the plaintext. The expression for the last byte then is

$$CL_b = D_b \oplus AC_b \tag{4.1}$$

where $D_b = OC_b \oplus PL_b$.

The AC block is controlled by the attacker, OC_b is known and PL_b is unknown. One key observation is that in $1/256$ cases the byte CL_b resulting from the xor will be 0x01 which is a valid padding, resulting in no padding error. With CL_b known, all that is unknown is D_b and it can be computed from AC_b and CL_b. We have that $D_b = OC_b \oplus PL_b$, and since D_b and OC_b are now know, PL_b can be computed.

$1/256$ seems bad for one byte. But the oracle will give padding error for all cases except the correct one. So if there is no protection in place one can simply try all 256 combinations and that would be fairly cheap and fast.

One block The method above describes how to get one byte, but this can be generalised. From above, all values in (4.1) are known. This makes it such that by changing AC_b the resulting value of CL_b can be any chosen value. Choose AC_b such that CL_b gets the value 0x02. Then if CL_{b-1} also is 0x2 one gets a valid padding. For CL_{b-1} the equation from the byte can be modified and applied as $CL_{b-1} = D_{b-1} \oplus AC_{b-1}$. Since AC_{b-1} is also controlled by the attacker the strategy from the single byte case applies. Brute forcing AC_{b-1} until a valid padding is found gives us the value of D_{b-1} and from it we can retrieve PL_{b-1}. This strategy can then be repeated for CL_{b-2} by choosing AC_{b-1} and AC_b such that CL_{b-1} and

CL_b are both 0x03 and one can brute force CL_{b-2}. This can be repeated for 0x04, 0x05 and so on until the entire block is decrypted.

One Message To decode the entire message one simply decodes block by block as described above by letting every block play the role of the last padded block in the message.

Complexity

The run time of this attack is linear in the number of bytes in the plaintext. N bytes takes at most $N * 256$ tries, giving a $\mathcal{O}(N)$ algorithm.

4.1.4 Replay attack

Introduction

Motivation for including replay attack

A replay attack was chosen as a CTF task because it is an attack vector which can be easily overlooked due to its nature. A server has no reason to deny a correctly formatted request containing all valid fields and tokens. It is also a classical and interesting cryptographic problem.

Requirements for the attack

1. Ability to capture the network traffic between the communicating parties.
2. Ability to send network traffic to the communicating parties.

What is a replay attack?

A replay attack is when an attacker can send previously sent messages to a receiver and have the message interpreted as a valid message. Take for example a bank transaction: Assume you are sending a small amount of money to an account (preferably controlled by the attacker) and the attacker intercepts that message. What is stopping the attacker from sending this message 1000 times to the server, gaining 1000 times the original amount? The integrity of the message remains unchanged as it has not been tampered with in any way, so the server will evaluate the message as correct.

In the scenario described above the server has no way of identifying if the message is an intended second money transfer, or if it has been re-sent by an attacker. Securing the cipher by encrypting the messages will not fix the problem as the process of accepting or denying the transfer comes after decryption. A possible solution would be for the server to keep some state where it is accepting the transmission once. After the packet is received, the server state changes in such a way that it will not allow the packet to be accepted a second time.

Session keys provides such functionality to some extent as the time-limit limits the window in which a packet can be resent. If the session key is invalidated or replaced, the replayed packed would not be properly decrypted, thereby reducing the window of opportunity for a replay attack to be successful, but it does not prevent it. This can be taken further by introducing timestamps, this will limit the window of opportunity even further. The timestamp

still does not entirely resolve the problem as some transfer delay has to be factored in. If the attacker is closer to the servers than the user, than the attacker my still be able to abuse the small time frame and resend the message. To completely prevent such attacks, both the client and server will need some associated information, as suggested in the paper [16].

By always appending previously sent messages to the current message body, a replay attack becomes impossible as the replayed packet does not contain the previously sent packets. Replayed messages would have to include all the previous versions of themselves in the chain of previously sent messages. Sending a complete history of all previous messages will cause increasingly bigger messages conveying the same information, making it impractical. However, this problem can be solved by hashing the previous messages, ensuring that the history is mostly preserved in a more space effective manner, as well as maintaining the integrity of the history. As the hashing functionally is a one-way compression with minuscule probability of collisions, the integrity of the history is not perfect compared to sending the actual history in its entirety. But it is much more space efficient and a good trade off.

The method described over can be a bit simplified if one uses a MAC on the message. Using HMAC and then including a counter in the message makes it impossible for the attacker to change the sequence as the HMAC provides message integrity by a shared key between the server and client which the attacker does not have. This can be exchanged in a secure way between the client and server by for example using the Diffie–Hellman key exchange algorithm.

Other methods that are commonly used for https is to use a one time token. When a user sends a get request to a site with a form in it, the get request contains a one time valid token. This token is sent in an invisible field with the rest of the form data in the following post request. When the server gets the post request it checks if it contains the same token it sent with the get request and if they match the request is processed and the token is devalidated. This makes replays impossible as replaying the post request the sent token would not be valid and resending both get and post request will not work as a new token is generated and it will not match the token in the replayed post request. This method can also be implemented on other channels then http. The only drawback is the extra round trip to get the one time token.

4.1.5 Bad random number generator

Introduction

Random numbers are a very important aspect of modern cryptography. They are used to create session keys, used as initial vectors and as nonces in key establishment protocols like the Diffie–Hellman key exchange. If a number becomes predictable this can be used to break the crypto system. The most obvious example of this is if the session key can be guessed/recreated. The problem with making random number generators is that a computer by itself can not generate random numbers. There are mathematical functions that given an initial input, often called a seed, will produce a random sequence of numbers. The property the sequence should have is that given the first n numbers, the $n + 1$'th number can not be derived using the previous n numbers in a computationally feasible way, without knowing the seed value used.

The seed still needs to be random as a given seed will always produce the same sequence. So to produce a random seed, one has to have a source of randomness. Computers can not

produce this, so external factors are used. Examples of this is current system time in nano seconds, number of running processes on the system, the spin on disk based storage mediums (hard drives), key stroke timings from the user and mouse movements. In the case where the seed is poorly chosen or the mathematical function that is used has a flaw, the attacker might be able to break the the crypto system as we shall see in the case of RSA and not so random primes.

Motivation for including this attack

This attack was chosen because bad random generators is something that has often occurred. There are examples where session tokens have been found by clever brute force, allowing an attacker to generate session tokens and impersonate other people [22]. In recent years with the surge of cheap flash storage, hard drives in system is becoming less common. Hard drives are one of the sources of entropy (random numbers) used by the Linux kernel, other sources are user input as mouse and keyboard. The problem with these is that a server does not have those available or they are seldom used [26]. This can result in systems having little entropy at boot as the kernel has had no chance to build up its entropy store.

Common primes

The public key part of an RSA key contains a public modulus which is the product of the two secret primes p and q. Let us assume we have two public keys, one with n_1 as its modulus and one with n_2 as its modulus. If $n_1 = p \times q$ and $n_2 = p \times r$ then n_1 and n_2 would have a common factor p. This factor could easily be found using the Euclidean algorithm for finding greatest common divisors on n_2 and n_1.

If p is found, some simple division would yield both q and r as well. From the primes p and q, the number $\lambda(n_1)$ can be computed as $lcm(q-1, p-1)$. The exponent d which is used to decrypt the RSA messages can be retrieved by computing $d = e^{-1} \mod \lambda(n)$. Since $\lambda(n_1)$ is now known we can run the extended euclidean algorithm on e and $\lambda(n_1)$. This gives the result $a \times e + b \times \lambda(n_1) = gcd(e, \lambda(n_1))$. From the definition of RSA we know that $gcd(e, \lambda(n_1)) = 1$ and $b \times \lambda(n_1) \mod \lambda(n_1) = 0$. This leaves us with the expression $a \times e = 1$, thus a is the inverse of e meaning that $a = d_1$. Now that all the parts of the private key are recovered, d_1 can be used to decrypt any message. This has actually occurred. Quote from [24]:

> More worrisome is that among the 4.7 million distinct 1024-bit RSA moduli that we had originally collected, 12720 have a single large prime factor in common.

Chapter 5

Implementation of tasks

This chapter describes tools, languages and technologies used to make the different challenges and the reasoning for choosing them.

5.1 Technologies

5.1.1 Languages

Two languages were primarily used for programming: Rust and C. Rust was primarily used as the backend language and for creating the certificates from prime numbers tool. The reasoning behind using Rust was twofold. The first is that it is a language I am comfortable using. The second is that it is an efficient language requiring no runtime environment. There are alternatives like C++ that meet the same criteria, but Rust provides a unique memory management model that provides memory safety during compilation. This means Rust is not susceptible to uninitialized memory accesses, buffer overflows, null dereferences, dangling pointers and other memory related errors [19]. This is with the exception of the so called ”unsafe” code as segments declared unsafe relaxes the compile time guarantees but gaining some flexibility like the ability to call C library functions.

C was used for the padding oracle challenge. This was done because the challenge requires the source code for the server to be viewed by the contestant. C being a common language used for code snippets in CTF’s is the main reason for choosing it for this challenge. It is also a typical language for implementing crypto systems and other high performance pieces of code. It is also a reasonable language to expect someone to understand as languages are often classified as ”C-like” or not. This makes C a good common denominator. Lastly C is relatively speaking a small language and the syntax is simple and common for most C-like languages.

There were to front ends that were made: One Android application and a web game. For the web game plain JavaScript was chosen as it was the simplest alternative yielding fast results. The Android app was created using Java. The only other choice for making the Android app is Kotlin. Java was chosen because I am familiar with the language and not Kotlin. Kotlin is in many ways a modern version of Java with Java backwards compatibility, and it would probably be used if not for the time constraint.

5.2 Tools

5.2.1 Docker

Docker is a platform allowing one to create and deploy/run containers. It uses OS-virtualisation to achieve this. In a traditional setup one would usually run a hypervisor service on the bare metal. Examples of this is Microsoft's hyper-v, VMwares EXSI, or KVM. The hypervisor in turn runs a virtual machine image containing a complete OS. The application is then installed to that virtual machine, or is part of the image.

When a virtual machine is created resources needs to be statically allocated, like disk space and memory, as the each virtual machines needs its own OS. Docker on the other hand is an OS-level-virtualisation. This means that the segregation of its services is done on OS level. Instead of running a full virtual machine docker uses the Linux kernel and its namespace technology to separate the applications. This allows docker to create an isolated user space for the application while having the benefit of not needing to allocate static amounts of memory for the application. All the container share that same kernel thus removing the need for multiple OS duplicates and an hypervisor.

Docker uses Dockerfiles to build images. A Dockerfile specifies files to copy from the host in to the image. Images can inherit from other images. An image is usually based on a small linux distro like apline. The apline image contains the bare bones to get an application running. A sh shell and a packet manger to install packages with. The Dockerfile file allows to specify command to be ran inside the image and using the packet manger it is easy to make an enviorment for ons's application to run inn.

When an image is built with all needed dependencies and the application to be run is copied into the image, a container can be started. Docker makes use of kernel namespaces to run an isolated process with its own file system called the container. Each aspect of a container is run in its own namespace and access to resources outside its namepaces is prohibited [15].

5.2.2 Docker Compose

Docker Compose [5] is command line tool for declaring multiple container setups using a yaml config file. This makes docker compose a good tool for creating several application services. For example, it makes it easy to run an API. It allows for running a web proxy in front of the API, running the API backed application and a database, all described in one file with a simple ”docker-compose up” command to start it all.

Traditional Docker allows for multiple customizations on a container with its run command. One can mount folders from the host into the container and create a virtual network for the container to run in. Unless a network is specified the containers run in Docker's default bridged network. It can also allow a container to use the host network directly. Environment variables for the container can also be set using the run command. The Docker Compose file is used to define multiple containers and their run time parameters like the ones mentioned above. It also allows for the declaration of the resources like networks and file mounts, also called volumes. The Docker Compose file describes all the containers and their run time settings.

Unless otherwise specified, it will create a network for the containers isolating them from

other containers. In this network all the containers get their own DNS. The container name as declared in the Docker Compose file serves as a domain, so the other containers can connect to each other using names.

The virtual network that is created is set up such that all outgoing traffic is allowed and is forwarded to the host's interface, but traffic from the world to the host interface is not forwarded to the virtual network. It works in the same manner as a traditional NAT'ed home network, where the router in this scenario is the host's network interface and the clients of the NAT are the container services. The router Docker Compose file also supports port forwarding. Ports of containers can be directly exposed on the host interface if specified by the Docker Compose file. The file specifies the port of the container and which port it is to be mapped to on the host.

This works well for the API example as it becomes trivial to set it up in such a manner that the web proxy is exposed to the world, while the API backend and database is shielded from traffic originating from the outside world. The containers also maintain the ability for cross communication between themselves.

5.2.3 Android Studio

Android studio [4] is a tools suite built upon the popular IDE from Intelij used for creating Android apps. This was used as it contains the necessary tooling for compiling and installing Android apps to a virtual machine or a phone. It also provides templates for creating Android apps and some management tools for downloading the correct version of the SDK.

5.2.4 OpenSSL

OpenSSL [11] is an open-source library implementing the SSL and the later TLS protocols. TLS and SSL are protocols that allow for secure communication between two parties. This includes a validation of the identity of the communication parties using asymmetric cryptography. This validation is also used to establish a secret session key.

The communication is a symmetrically encrypted connection using the session key. Messages that are sent are also reliable as they include a message authentication code. This fills the two of criteria for information security, confidentiality and integrity. The OpenSSL library is used by both the app and the server for the certificate pinning challenge.

The OpenSSL library also comes with a command line tool. This tool provides access to many of the library features. In two of the challenges it was used to generate public and private keys and generating self-signed certificates from config files. It was also used to generate a public private key pair from a config specifying the primes, exponent and key size to be used.

5.3 Implementation of the CTF challenges

5.3.1 Certificate pinning

Implementation choices

The certificate pinning challenge was implemented as an Android banking app, called the Bit-bank, with a corresponding API backend. The player gets the app's apk file and a test username and password for the app. When the player installs and loads the app the player is met with a simple login screen. The credentials that were provided will log the player into the app, where the player can interact with the app's main screen. The main screen contains a simple button for fetching the name and balance of the logged in user's account. The objective for the player is to read all the traffic between the server and app and not just what the app shows to the user.

The server is implemented in Rust because of the good compile-time guaranties the language provides, such as memory safety. The http-library used is called Rouille [3]. It is a small library and provides the minimum needed for creating the banking API. In front of the API is an Nginx [9] server used as a reverse http-proxy to provide TLS connections. This is done so that players cannot sniff the server's traffic directly.

The server code

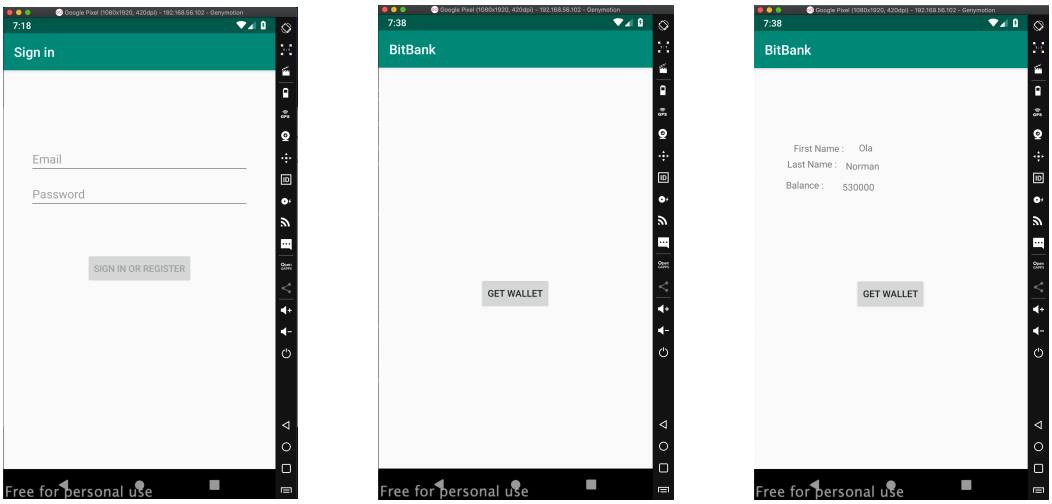
The server (code in Appendix A.1.2) provides the banking API for the app. The banking API consists of two API endpoints, one for login and one for fetching the user's balance.

The login endpoint takes a POST request with two arguments, namely the username and password. When the server receives the POST request, the username is used as a key to retrieve the corresponding password and this password is matched against the one provided in the login request. If they do not match a 404 error code is returned, otherwise a JWT token is generated and sent as the response. The token contains the user's id and it is signed using a secret server side key and the HS256 algorithm.

The endpoint for fetching the balance, called getWallet, is a get request taking a token as its argument. When the endpoint is called it first validates the token using the secret key. If the token is invalid, a 404 response is sent. In the case of a valid token the user id is extracted from the token and used to fetch the user's name and balance. The balance and the user's name is sent back to the client as a json object. This json object also contains the flag the player wants to obtain.

The client code

The client app (code in Appendix A.1.1) contains two screens, the login screen and the balance screen, see Figure 5.1. The login screen has two input fields, one for username and one for the password. Below the input field is a login button. When the login button is pressed the client loads a certificate field from file and adds it to a trust store. A hash of this certificate is also generated. The hash and the trust store containing the certificate is passed to the okhttp client's certificate pinning builder and a http client is created. This client will only accept the cert in the store and will also check that the hash matches. This is so that even if an attacker



(a) The login screen of bitBank

(b) The users dashboard before interaction

(c) The user dashboard after interaction

Figure 5.1: Screenshots of the bitBank app

controlled CA is added to the system store and a valid cert is provided, the hash will still not match and the connection will be refused.

With the new client a https POST request is made with the username and password form values populated by the user input. If the app receives a 404 it will close itself and a login failed popup is presented on the screen. If the login was successful the app spawns the balance screen and passes the token received during the login to it.

The balance screen starts out empty with a single get balance button. When this button is clicked a new client with the pinned certificate is spawned in the same fashion as with the login screen. With the http client a getWallet request is sent with the token as the only parameter. If the request fails, an error message is displayed to the user on the screen and the user can retry by clicking the getBalance button a second time. In the case of a valid request a json object is returned. This json object contains the user's first and last name, the balance and the flag. The fist name, last name and the balance is displayed in the app, while the flag is ignored.

The challenge

The player is given the banking app and a test user for the app and is tasked with discovering how this app communicates. The goal is to be able to read the encrypted traffic sent between the server and client. This will lead to the discovery of the flag hidden in one of the messages.

Intended solution

To accomplish finding the flag the player needs to decrypt the traffic between the server and the app. The app has the certificate used for communication built in to it. The player has to first decompile the apk file to get the byte code and resources of the app. The decompiled app contains a resources folder with the certificate in it. By replacing the certificate with one the player has crafted, and recompiling the app the player can set up a MITM proxy like Burp. Burp will act as a proxy between the app and the server using the crafted certificate to com-

municate with the app, decrypting it and forwarding it to the server on the TLS connection it has established. As both the session key to the app and the server is known by Burp all traffic can be read as plaintext by Burp, and in turn, the player.

5.3.2 Validation and enumeration

Implementation choices

As this challenge builds on certificate pinning it uses the same implementation with a few additions. The frank_jwt library is used as in the previous exercise to validate the JWT tokens. This library does not possess the JWT None token flaw. To solve this, a local modified version of the library is used instead. The changes that were made to the JWT library was making the decode_header_and_payload function public so that it can be used directly in the backend code.

The backend code was altered, so instead of validating the token the decode_header_and_payload function is called and the algorithm of the header is checked. If the type is None the validation step is skipped and the payload is processed. Otherwise the code works as described in the certificate pinning chapter.

The Challenge

The challenge here is to be able to impersonate another user of the bitbank to gain access to the their profile, which contains the flag. To achieve this one first has to complete the previous step of this challenge as the ability to decrypt traffic between the app and server is needed. Inspecting traffic one can observe that jwt tokens are used for authentication. The jwt none flaw can be exploited to forge valide tokens. This must be combined with user id enumeration to gain access as another user.

Intended Solution

From the previous challenge the contestant can view the plaintext version of the traffic going between the server and the application. By creating a None algorithm token and changing the user id of said token the contestant is able to call the getWallet endpoint for several user ids. As the None-token is valid the endpoint will accept each of the requests, and only failing because of invalid user ids. The solution is to simply try user ids until an actual user is found. As a hint, the user id of the test user is close to 1337 in value. So the observant participant might try that value first, and succeed. Regardless a exhaustive search in both directions will quickly yield a result.

5.3.3 Padding oracle

Implementation choices

The server was implemented with the intent to be as simple as possible but still requiring a proper knowledge of the attack to exploit it. The first ting that was done was to place the padding in front of the message as it is typically at the end. This does not change the attack as the IV used for xor'ing in the first block is sent as plaintext. The contestant simply has

to attack the first block instead of the last. This will hopefully make existing tools on the internet solving the task obsolete or at least needing to be customized.

The server code

The server (code in Appendix A.2) code starts by setting up a socket at port 1337 and listens for connections. If a connection is established. It will try read the IV and the encrypted message sent from the client. It tests if the padding is valid and if it is not the server will respond with the taunt ”You know nothing, skid”. If the padding is correct it will check if the message itself is valid. In this case the message has to be the flag of the challenge to be considered valid. The user will get another taunt namely, ”You know nothing, hacker boy”, if the message does not match the flag file on the server. If the message and the flag matches the server will respond with ”You know the path , bro”.

The client code

This code (code in Appendix A.2.2) simply establishes a connection to the server and sends the flag as an encrypted message. So this program is an emulation of an actual user of the server with the correct password and the flag. This program is simply an aid to the CTF challenge as the traffic the program generates is to be captured as a pcap file and handed to the constant with the server code.

The challenge

The participant is given a valid package capture of a client connection to the server sending the flag and receiving the positive confirmation. The participant also receives the code for the server so the participant can understand the meaning and context of the error messages sent from the server and spot the oracle flaw within the server.

Intended solution

Every time the server sends the response ”You know nothing, skid”, the padding of a message is wrong. On the other hand, if ”You know nothing, hacker boy” is sent the padding test passed but the message sent was not the correct flag (with padding). These messages/taunts from the server can be used as an oracle as it leaks one byte of information about the plaintext. This can be used perform the padding oracle attack. If the first byte of the message is 0x1 then the padding is considered valid. Since the contestant controls the IV that is xored with the cleartext (cipher text decrypted by AES) before the padding is checked, the value of the last byte of the IV can be iterated through all the combinations until hitting a valid padding. Knowing the padding lets the player infer the last byte of the plaintext as we know the IV. Then (s)he can do the same for the second byte by setting the first byte to 0x2 and flipping the second byte until getting a valid padding. This can be repeated for the entire block and for multiple blocks until the entire flag/plaintext is recovered.

5.3.4 Replay attack

Implementation choices

This replay attack was implemented in the shape of a game. The player loads the game in their browser and is asked for a license key to play. Then a tile game is loaded. The player's mission is simply find the tile with the flag and receive the flag string. If the player enters an invalid license key the game loads, but the player's character is not able to move. The server only sends the visible part (from the players perspective) of the map to the player. More parts of the map are only sent from the server when the player's position changes. The API only allows the player to move if a correct license key is used to encrypt the packages.

The server is implemented in Rust because of the good compile-time guaranties the language provides such as memory safety. The http-library is called Rouille [3]. It is a small library and provides the minimum needed for this web game's API. In front of the game an nginx [9] server that is used as a reverse http-proxy to provide a https connection. This is done so that players cannot sniff each others traffic. If a player gets his hands on another player's token, the player can potentially move the other player's character.

The server code

The game server (code in Appendix A.3.2) provides a simple web API, as well as serving the game resources and the game itself. For the remainder of this text, a ”hacker” refers to the CTF-contestant and ”player” refers to the in-game player character.

When the game is served from the game server the game client fetches all the tiles (images) and makes an initial call to the move API. Since the hacker has no token, a player state is initialized to the starting position and a token is created. The game sets the hacker's cookie to the token value and returns a JSON-object containing an 11×11 grid with the tiles the player can observe from its position. The token is simply a long uid-string in the player's browser used to identify the player and find his player charterer's coordinates on the map.

When playing the game, the hacker sends ”move”-requests to the API. The get-request to move the player takes two parameters. The direction, which is a URL and base64 encoded encrypted text of the direction the player wants to move, and the initial value (IV) used to encrypt the direction.

The server fetches the current player position form its database based on the uid in the token in the cookie. It then tries to decrypt the player's direction. If the direction is not one of the predefined ”up”, ”down”, ”left”, ”right”, or if it can not be the decrypted, the game returns the player's current position. If the direction is valid, a second check is performed to see if the move is valid. A white list is used to check if the new position's tile is of a type that the player is allowed to stand on, eg. grass is allowed but water is not. If the tile type is invalid, the player's position remains unchanged. If both checks are valid, the new position is returned in the form of an 11×11 grid with the new tiles that are visible to the player. One additional check is performed where the game will check if the new position is the position of the flag. If so, a ”flag”-property is appended to the returned JSON-object, with a string containing the flag.

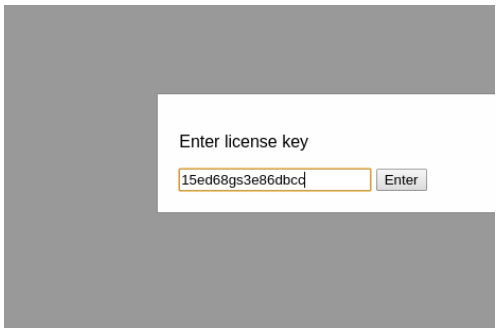


Figure 5.2: The license key screen

Figure 5.3: The player exploring the map

The client code

The client (code in Appendix A.3.1) code starts by loading all the tile images and asks the player to enter a license key. When the license key is entered the game sends a move request to the server. The license key and a random IV is used to encrypt the plain text string of the direction and sends the encrypted text together with the IV (in plain text). The server will always respond with an 11×11 tile grid with the player in the middle. The client then renders that tile map, see Figures 5.2, 5.3 and 5.4 for screenshots of the game.

The movement keys are bound to functions sending the direction pressed over the ”move” API, and are also encrypted with the license key. The server either returns the same state as before or moves the player to a new position, depending on if the move was legal and if the the license key was correct.

The challenge

The hackers are given the URL to the repeatable game and a packet dump of an earlier playthrough of the game, with the context that the packet dump was created before the developer installed a certificate. The challenge is simply to play the game and get to the flag tile in the game.

Figure 5.4: The challenge completed

Intended solution

This is a ”repeatable” game and intended as a replay attack challenge. If one studies the code of the client it is possible to see that the sent messages are one of four strings: ”up”, ”down”, ”left” or ”right”. The packages in the packet dump are unique, only because of the randomness of the IVs. The IV is also sent as plaintext. n After decryption, the only information left for the server is one of the strings describing the direction. As the server does not implement any rules regarding the iv, it is impossible to tell if the packet was original or retransmitted. Another observation is that the token itself is not part of the encrypted packet. This means that the token can be changed, but the http-request will still be valid. This is all a hacker needs to move their character.

To perform the attack and solve the challenge the hacker must first find four packets that correspond to moving in each of the directions in the game. By sending an empty token on each request the game server always creates a new player character at the starting point, resulting in the same output for the same move command each time. This makes the attack possible to script. Simply use the response as a key in a hashmap and the value as the encrypted direction, then try all the move-requests from the packet capture and update the value each time. When this is done, the hacker only have to iterate over the four entries in the hashmap and figure out which responses correspond to which encrypted direction. In the web-browser, open the console and simply overwrite the functions ”up”, ”down”, ”left” and ”right” to send the corresponding encrypted direction. Then it is just to play the game as if one had the license key, and navigate to the flag-tile.

Sample overwrite of the ”left” function:

```
left = () =>{    let Http = new XMLHttpRequest();
        let url =
        '/move?direction=18YxlQFWlTkJIldjBN%2'
        +'BeEA%3D%3D&iv=bMxkTUrLQWjPZRLxQodf4Q%3D%3D';
        Http.open("GET", url);
        Http.send();

        Http.onreadystatechange = (e) => {
```

```
        if (Http.readyState === 4 && Http.status === 200) {
            let map = JSON.parse(Http.responseText);
            if (map.flag){
                modal.style.display = "block";
                license_button.style.display = "none";
                license_key_field.style.display = "none";
                textField.innerText= map.flag;
            }
            clear();
            drawLayer(map.layer1);
            drawLayer(map.layer2);
            drawPlayer(direction);
        }

    };
    }
```

5.3.5 Bad random number generator

Implementation choices

This challenge uses the OpenSSL command line tool to generate keys and certificates. It is also used to encrypt and decrypt the flag using the generated private keys on the flag file.

Certificate generation

A script called getCert.sh is executed in the docker container. This start out by using the OpenSSL tool to create nine public and private key pairs.

Then the first primes in keys 4 and 7 are extracted and passed to certs, a Rust program (code in Appendix A.4) that takes two primes as an argument and prints an ASN1 file. This file is used to produce a new public private key pair with the OppenSSL tool. The newly created pair overwrites the seventh of the originally crated key pairs. Since this key is based on the first prime of the fourth key and the first prime of the original seventh key, it results in the new key having one prime in common with the fourth key.

All the nine key pairs are used to create nine self-signed certificates. For each certificate to be created their exist a prebuilt configuration file specifying required values like subject, organisation and so on. Then each of the private keys are used to encrypt the flag file, producing nine encrypted versions of the flag. Finally the script copies the encrypted flags and the certificates to the out-folder for use on the CTF platform.

The challenge

Find a way to decrypt one of the encrypted flag files. The contestant will get all of the public keys used to do the encryption. There is a mistake in two of the public keys that will allow the player to regenerate the private key. The exercise name hints at what the mistake in the private keys is, namely ”Who stole/copied my prime?!”

Intended solution

The constant must realize that one of the RSA keys share a prime with one of the others. This common factor can be found by running the gcd algorithm of the modulus on all the public keys against each other. One combination will yield a gcd not equal to one and this will give one of the primes uses to get the modulus, which is $n = p * q$. Since p is now found n/p gives q. From this one can produce a valid Ans1 file. The Ans1 file can be used with the help of the OpenSSL utility to produce a valid private key. This private key can be used to decrypt one of the encrypted flag files.

5.4 Deployments of tasks

The challenges were successfully deployed to Netsecurity using docker. The platform hosts the interactive challenges in docker containers. The challenges were compiled and run by the platform with the help of the files described below.

5.4.1 Metadata files

Each of the projects contain a few metadata files that either describe the project or contains values used during the build process. There were three files required by the platform, the readme file, the Docker json file and the DevBuildStart.sh file. A Dockerfile file were indirectly required as the platform runs challenges in containers. The Dockerfile was used to build and run the containers through the shell script. The other files and structures were added by me to allow for customization.

Docker.json

This file contains the metadata for the Docker container. The metadata file includes the following fields: the docker repo to be used for the image and the tag of the image, and the name of the challenge. The IP address of the container and its ports are also listed. Lastly, it contains the domain where the challenge can be found during a competition.

Key.txt

This file contains the key that are used for AES 128-bit encryption in the challenges. The key is a 128-bit key and therefore must always contain 16 bytes. The key can be changed freely but should be hard to brute force as the challenges are not intended to be solved using or finding this key. Also note that changing the key will affect the generated packet capture.

Flag(#)**.txt**

The flag file contains the flag of the challenge. The flag is a string that is passed to the CTF game server as proof that the challenge was completed. This file is also customizable with no restrictions to file length.

Readme.md

The readme file contains a short listing of metadata like the name of the flag/challenge, the name of the container, the flag type, the points the flag is worth, and the flag string itself. Note that the flag string must match the one in the flag file. This is followed by a description of the challenge that is presented to the contestant. The two last sections are not presented to the contestant and contains a short hint summary for the challenge and a longer text describing a complete solution.

5.4.2 Building and Running the Challenges

Source code

The source code of the challenges are placed in a source folder named src, with the exception of the C code that is placed directly in the root folder. In the cases where there is both client and server code, the src folders for each of them is wrapped in a parent folder named client or backend accordingly. If Rust is used there also exists a cargo.lock and cargo.toml file. These files manages the dependencies for the Rust programs.

Docker file

Each of the source folders are accompanied by a docker file. This docker file is used to build the source code and produce a container with the built binary.

Docker compose file

The certificate pinning challenge folder contains a docker compose file that is used to run the challenge as it creates the necessary reverse proxy needed by the API backend in the challenge.

DevBuildStart.sh

This file does the actual building and running of the tasks. It start by reading the values from the Docker.json file and uses the values to first assign a port on the host system for the challenge. Then it builds the images described in the docker file and runs them as containers.

There is one notable exception to this and that is the certificate pinning challenge where the port is dictated by the docker compose file and the script invokes the docker compose build and run command instead of docker's command.

Out folder

During the initialization of the container an out folder will be mounted into the container of a few challenges. This folder will be populated with files that are required for solving the challenge and should be copied to a place where the contestant has access.

Chapter 6

Summary

6.1 Lessons learned in designing CTF tasks

Five challenges were created, each with cryptography as their overarching theme. Creating the challenges came with challenges of its own. The CTF challenges hosted by NetSecurity is usually a day event, and this puts a limit on the time one can use per challenge in order to be effective as a player. Harder challenges can award more points but winning by only solving one or two challenges makes for a bad competition. The challenges have to be solvable in half an hour to and hour if the contestant has an idea of what the weakness might be and knows how to exploit it. To achieve this a few criteria needed to be met.

The name and description of the challenge has to be worded in such a fashion that it gives the contestant an idea of what the flaw could be, given that the contestant is familiar with it, but also not so obvious that performing a web search including a few words form the challenge text would give away the solution.

This brings up a particular problem with cryptography challenges in that one can not break cryptographic primitives. If that were the case they would not be used. So this leaves one with the attack surface of the implementation of the primitives, both in the form of flaws in the implementation of the primitive itself and how they are combined into a bigger cryptographic system or protocols. The challenges that were made touch on both areas.

The challenges are intended to be of the easier type so that extensive knowledge of cryptography is not needed. A fundamental understanding of the common primitives combined with an overview of a few well known implementation flaws should suffice. They should also be realistic in the sense that the flaw is something that has occurred in a ”real life” application or could have happened. The implementations were made realistic in the sense that the flaws were implemented as one would expect in ”the wild”, but all other aspects of the application one would expect is stripped away to prevent rabbit holes. An example of this is the bank app where only login and balance check API points are implemented and the app only has a login screen and a bare bones balance view. In the case where traffic sniffing would be required a pcap file is simply handed out.

The last criteria is to avoid so-called rabbit holes. Rabbit holes are things in the application that intentionally or not seems like something which could be the flaw that is to be exploited, but is not. This leads the contestant to spend time on something that in the end will not yield any results. The reasoning for avoiding this is twofold. One is that the limited time frame the challenge is to be completed in could cause problems for the contestant. The second

reason is that in my personal opinion, rabbit holes are at best an annoying way to stall one from completing a challenge and it can be argued that being stuck in a rabbit hole does not necessarily imply lack of skill. In an ideal competition lack of skill should be the only thing keeping one from completing a challenge!

Some of the attacks were at the beginning only something I knew by name, and thus researching and figuring out how they worked was sometimes difficult. The padding oracle was one of the harder ones to understand as there are many variables. The challenge also required me to build a working implementation with the flaw and the solution abusing the flaw, making small room for errors in either of the implementations. The RSA prime challenge required some mathematical understanding paired with an understating of how the OpenSSL library works and a partial recreation of the RSA key generation algorithm. The other challenges offered mainly programming difficulties.

Further development

The Certificate pinning challenge could have had more flags. As the code is, it generates the certificate's signature at run time. This allows for decompiling the app and replacing the certificate file with one with a known private key and recompile the app. A simpler entry flag could be built in where if the contestant manages to recompile the app with the debugging flag set the app prints a flag to the debug console. A common mistake done by developers is leaving debug code in the released product. Simply by putting the application in debug mode one gains vast amounts of information with minimal effort.

A fourth challenge that could be made is an attack as a logged in user. JWT token libraries introduced a new problem in addition to the problem of setting the algorithm to None. This is due to the fact that it is the JWT token that dictates the algorithm used to validate it. A few JWT libraries implement the verification method with a signature like this: ”verify(clientToken, Key)”. The verify function checks the token algorithm and assuming it finds an asymmetric algorithm the function will assumes that it is handed a public key and attempts to decrypt the token using it. In the scenario of a symmetric algorithm it will assume the key is a normal secret key [25].

The problem arises when legit tokens are signed with a private key. The server is then programmed to expect asymmetric tokens and therefore hands the verify method the public key, but the verify method on the other hand reads the token first and then decides how to use the key. If an attacker were to change the algorithm to a symmetric one the byte sequence of the public key would simply be interpreted as a symmetric secret key. The problem with this is that the byte sequence making up the public key also is known by the attacker. Thus he simply signs his token using the public key, in the role as the key to a symmetric algorithm.

As the need for security grows larger hopefully the challenges that were created can contribute by learning both developers and security experts about some crypto flaws and give an understanding on how they can be fixed or prevented in the future.

Appendix A

Source code

A.1 Certificate Pinning source

A.1.1 Android App

```
1  public class LoginDataSource {
2
3      private String hostname = "bitbank.ctf.cwat.no";
4      private OkHttpClient client;
5
6      public LoginDataSource(byte[] cert){
7
8
9          X509TrustManager trustManager;
10         SSLSocketFactory sslSocketFactory;
11         String certHash;
12         try {
13             trustManager = CustomTrust.trustManagerForCertificates(new ByteArrayInputStream(cert));
14             SSLContext sslContext = SSLContext.getInstance("TLS");
15             sslContext.init(null, new TrustManager[] { trustManager }, null);
16             sslSocketFactory = sslContext.getSocketFactory();
17             certHash = PeerCertificateExtractor.extract(new ByteArrayInputStream(cert));
18         } catch (GeneralSecurityException e) {
19             throw new RuntimeException(e);
20         }
21
22         CertificatePinner certificatePinner = new CertificatePinner.Builder()
23                 .add(hostname, certHash)
24                 .build();
25
26         client = new OkHttpClient.Builder()
27                 .certificatePinner(certificatePinner)
28                 .connectionSpecs(Arrays.asList(ConnectionSpec.MODERN_TLS))
29                 .sslSocketFactory(sslSocketFactory, trustManager)
30                 .hostnameVerifier(new HostnameVerifier(){
31                     @Override
32                     public boolean verify(String hostname, SSLSession session) {
33                         return hostname.equals(session.getPeerHost());
34                     }
35
36                 })
37                 .build();
38
39
40     }
41
42
43
44     String post(String url, String username, String password) throws IOException {
45         HttpUrl fullUrl = new HttpUrl.Builder()
46                 .scheme("https")
47                 .host(url)
48                 .port(443)
49                 .addPathSegment("login")
50                 .build();
51         RequestBody body = new MultipartBody.Builder()
52                 .setType(MultipartBody.FORM)
53                 .addFormDataPart("login", username)
54                 .addFormDataPart("password", password)
55                 .build();
56         Request request = new Request.Builder()
57                 .url(fullUrl)
58                 .post(body)
59                 .build();
60         try (Response response = client.newCall(request).execute()) {
```

```
61              if (response.code() == 401){
62                  throw new IOException();
63              }
64              return response.body().string();
65          }
66      }
67
68      public Result<LoggedInUser> login(String username, String password) {
69          try {
70              String response = post(hostname, username, password);
71              Log.d("hei", response);
72              return new Result.Success<>(new LoggedInUser(response));
73          } catch (Exception e) {
74              return new Result.Error(new IOException("Error logging in", e));
75          }
76      }
```

A.1.2 Backend

```
1   #[macro_use]
2   extern crate rouille;
3   extern crate frank_jwt;
4   #[macro_use]
5   extern crate serde_json;
6
7   use serde_json::Value as JsonValue;
8   use serde::Deserialize;
9   use serde::Serialize;
10
11  use frank_jwt::{Algorithm, encode, decode, ValidationOptions, decode_header_and_payload};
12
13
14  use std::collections::HashMap;
15  use std::io;
16  use std::sync::Mutex;
17  use rouille::Request;
18  use rouille::Response;
19  use std::convert::From;
20  use std::fs::File;
21  use std::io::{Read, BufReader, BufRead};
22
23  #[derive(Debug, Clone, Serialize, Deserialize)]
24  struct Token {
25      id: usize,
26  }
27
28  impl From<User> for Token {
29      fn from(user: User) -> Self {
30          Token { id: user.id }
31      }
32  }
33
34  #[derive(Debug, Clone, Default)]
35  struct User {
36      id: usize,
37      user: String,
38      password: String,
39      wallet: usize,
40      flag: String,
41  }
42
43  #[derive(Debug, Clone, Default, Serialize, Deserialize)]
44  struct GetWalletResponse {
45      first_name: String,
46      last_name: String,
47      wallet: usize,
48      flag: String,
49  }
50
51  #[derive(Debug)]
52  struct TokenValidationError {}
53
54
55  impl From<frank_jwt::Error> for TokenValidationError {
56      fn from(_: frank_jwt::Error) -> Self {
57          TokenValidationError {}
58      }
59  }
60
61  impl From<serde_json::Error> for TokenValidationError {
62      fn from(_: serde_json::Error) -> Self {
63          TokenValidationError {}
64      }
65  }
66
67
68  fn validate_token(request: &Request, key: &String) -> Result<Token, TokenValidationError> {
69      if let Some(token) = request.header("Authorization") {
70          let raw_segments: Vec<&str> = token.split(".").collect();
71          if raw_segments.len()< 2 {
72              return Err(TokenValidationError {});
73          }
```

```
74          let (header, payload) = decode_header_and_payload(raw_segments[0], raw_segments[1])?;
75          match header.get("alg") {
76              Some(none) => {
77                  if none.eq("none") {
78                      let payload: Token = serde_json::from_value(payload)?;
79                      return Ok(payload);
80                  }
81              }
82              _ => { return Err(TokenValidationError {}); }
83          }
84
85          let (_, payload) = decode(&token, key, Algorithm::HS256, &ValidationOptions::dangerous())?;
86          let payload: Token = serde_json::from_value(payload)?;
87          Ok(payload)
88      } else {
89          Err(TokenValidationError {})
90      }
91  }
92
93  fn handle_banking_request<>(request: &Request, token: Token, users: &Mutex<HashMap<usize, User>>) -> Response {
94      let user = users.lock().unwrap().get(&token.id).unwrap_or(&User::default()).clone();
95      let mut user_name = user.user.split(" ");
96      let first_name = user_name.next().unwrap().to_string();
97      let last_name = user_name.next().unwrap().to_string();
98      let response = GetWalletResponse {
99          first_name,
100         last_name,
101         wallet: user.wallet,
102         flag: user.flag,
103     };
104     router!(request,
105                 (GET) (/getWallet) => {
106                     Response::json(&response)
107                 },
108
109                 _ => Response::empty_404()
110             )
111 }
112
113 fn main() {
114     let mut users: HashMap<usize, User> = HashMap::new();
115     let mut flag1 = String::new();
116     let mut flag2 = String::new();
117     let mut key = String::new();
118     File::open("flag1.txt").unwrap().read_to_string(&mut flag1).unwrap();
119     File::open("flag2.txt").unwrap().read_to_string(&mut flag2).unwrap();
120     File::open("key.txt").unwrap().read_to_string(&mut key).unwrap();
121     users.insert(1330, User {
122         id: 1330,
123         user: "Ola Norman".to_string(),
124         password: "testUserPassword123".to_string(),
125         wallet: 530000,
126         flag: flag1,
127     });
128     users.insert(1337, User {
129         id: 1337,
130         user: "Elongated Muskrat".to_string(),
131         password: "dedc607e686df9f547b6f494d58bc8bc89018d09d5dbde9885312a54".to_string(),
132         wallet: 530000,
133         flag: flag2,
134     });
135     let users = Mutex::new(users);
136
137     println!("Now listening on 0.0.0.0:8080");
138
139     #[allow(unreachable_code)] //avoid compile time waring of unreachable code which is simply wrong.
140         rouille::start_server("0.0.0.0:8080", move |request| {
141         //rouille::log(&request, io::stdout(), || {
142             router!(request,
143             (POST) (/login) => {
144                 let data = try_or_400!(post_input!(request, {
145                     login: String,
146                     password: String,
147                 }));
148
149                 let user_id = match &*data.login {
150                    "ola@norman.no" => 1330,
151                    "musk@space.x" => 1337,
152                    _=> {
153                    return Response::html("Wrong login/password")}
154
155                 };
156
157                 let user = users.lock().unwrap().get(&user_id).unwrap().clone();
158
159                 if data.password.eq(&user.password) {
160                     let header = json!({});
161                     let jwt = encode(header, &key, &serde_json::to_value(&Token::from(user)).unwrap(), Algorithm::HS256).unwrap();
162                     return Response::text(jwt);
163                     } else {
164                     return Response::html("Wrong login/password");
165                 }
166             },
167
```

```
168             _ => ()
169         );
170
171             let validation_result = validate_token(&request, &key);
172             if validation_result.is_err() { return Response::empty_404(); }
173             handle_banking_request(&request, validation_result.unwrap(), &users)
174         })
175     //});
176 }
```

A.2 Padding oracle source

A.2.1 Backend

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <sys/socket.h>
4   #include <stdlib.h>
5   #include <netinet/in.h>
6   #include <string.h>
7   #include <math.h>
8   #include <stdbool.h>
9   #include <mcrypt.h>
10
11  #define PORT 1337
12
13  void display(char *ciphertext, int len) {
14      int v;
15      for (v = 0; v < len; v++) {
16          printf("%d ", ciphertext[v]);
17      }
18      printf("\n");
19  }
20
21  bool testPadding(char buffer[1024]) {
22
23      char paddingSize = buffer[0];
24      for (int i = 0; i < paddingSize; ++i) {
25          if (paddingSize != buffer[i]) {
26              return false;
27          }
28      }
29      return buffer[0];
30  }
31
32  bool testFlag(char *buffer, char *flag, int len) {
33      for (int i = 0; i < len; ++i) {
34          if (buffer[i] != flag[i]) {
35              return false;
36          }
37      }
38      return true;
39  }
40
41  int decrypt(
42          void *buffer,
43          int buffer_len,
44          char *IV,
45          char *key,
46          int key_len
47  ) {
48      MCRYPT td = mcrypt_module_open("rijndael-128",
49                                     NULL, "cbc", NULL);
50      int blocksize = mcrypt_enc_get_block_size(td);
51      if (buffer_len % blocksize != 0) { return 1; }
52
53      mcrypt_generic_init(td, key, key_len, IV);
54      mdecrypt_generic(td, buffer, buffer_len);
55      mcrypt_generic_deinit(td);
56      mcrypt_module_close(td);
57
58      return 0;
59  }
60
61  int main(int argc, char const *argv[]) {
62      int server_fd, new_socket, valread;
63      struct sockaddr_in address;
64      int opt = 1;
65      int addrlen = sizeof(address);
66      char buffer[1024] = {0};
67
68      char *greating = "You know the path , bro";
69
70      char iv[16] = {0};
71      char key[16] = {0};
72      char flag[36] = {0};
73
74      FILE *keyFile = fopen("key.txt", "r");
75      fread(key, 1, 16, keyFile);
```

```
76
77      FILE *flagFile = fopen("flag.txt", "r");
78      fread(flag, 1, 36, flagFile);
79
80      server_fd = socket(AF_INET, SOCK_STREAM, 0);
81      setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR
82                  | SO_REUSEPORT, &opt, sizeof(opt));
83      address.sin_family = AF_INET;
84      address.sin_addr.s_addr = INADDR_ANY;
85      address.sin_port = htons(PORT);
86
87      bind(server_fd, (struct sockaddr *) &address, sizeof(address));
88      listen(server_fd, 3);
89      while (true) {
90          if ((new_socket = accept(server_fd,
91                                   (struct sockaddr *) &address,
92                                   (socklen_t *) &addrlen)) < 0) {
93              continue;
94          }
95          valread = read(new_socket, iv, 16);
96          valread = read(new_socket, buffer, 1024);
97          printf("Resived: ");
98          display(buffer, valread);
99          decrypt(buffer, valread, iv, key, 16);
100         printf("Decrypted: ");
101         display(buffer, valread);
102         if (testPadding(buffer)) {
103             if (testFlag(&buffer[buffer[0]], flag, 36)) {
104                 send(new_socket, greating, strlen(greating), 0);
105             } else {
106                 send(new_socket,
107                      &"You know nothing, hacker boy\n", 29, 0);
108                 close(new_socket);
109                 continue;
110             }
111         } else {
112             send(new_socket, &"You know nothing, skid\n", 29, 0);
113             close(new_socket);
114             continue;
115         }
116     }
117 }
```

A.2.2 Client

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <sys/socket.h>
4   #include <stdlib.h>
5   #include <netinet/in.h>
6   #include <string.h>
7   #include <math.h>
8   #include <stdbool.h>
9   #include <mcrypt.h>
10
11  #define PORT 1337
12
13  void display(char *ciphertext, int len) {
14      int v;
15      for (v = 0; v < len; v++) {
16          printf("%d ", ciphertext[v]);
17      }
18      printf("\n");
19  }
20
21  bool testPadding(char buffer[1024]) {
22
23      char paddingSize = buffer[0];
24      for (int i = 0; i < paddingSize; ++i) {
25          if (paddingSize != buffer[i]) {
26              return false;
27          }
28      }
29      return buffer[0];
30  }
31
32  bool testFlag(char *buffer, char *flag, int len) {
33      for (int i = 0; i < len; ++i) {
34          if (buffer[i] != flag[i]) {
35              return false;
36          }
37      }
38      return true;
39  }
40
41  int decrypt(
42          void *buffer,
43          int buffer_len,
44          char *IV,
45          char *key,
46          int key_len
47  ) {
```

```
48      MCRYPT td = mcrypt_module_open("rijndael-128", NULL, "cbc", NULL);
49      int blocksize = mcrypt_enc_get_block_size(td);
50      if (buffer_len % blocksize != 0) { return 1; }
51
52      mcrypt_generic_init(td, key, key_len, IV);
53      mdecrypt_generic(td, buffer, buffer_len);
54      mcrypt_generic_deinit(td);
55      mcrypt_module_close(td);
56
57      return 0;
58  }
59
60  int main(int argc, char const *argv[]) {
61      int server_fd, new_socket, valread;
62      struct sockaddr_in address;
63      int opt = 1;
64      int addrlen = sizeof(address);
65      char buffer[1024] = {0};
66
67      char *greating = "You know da wae, bro";
68
69      char iv[16] = {0};
70      char key[16] = {0};
71      char flag[36] = {0};
72
73      FILE *keyFile = fopen("key.txt", "r");
74      fread(key, 1, 16, keyFile);
75
76      FILE *flagFile = fopen("flag.txt", "r");
77      fread(flag, 1, 36, flagFile);
78
79      server_fd = socket(AF_INET, SOCK_STREAM, 0);
80      setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt));
81      address.sin_family = AF_INET;
82      address.sin_addr.s_addr = INADDR_ANY;
83      address.sin_port = htons(PORT);
84
85      bind(server_fd, (struct sockaddr *) &address, sizeof(address));
86      listen(server_fd, 3);
87      while (true) {
88          if ((new_socket = accept(server_fd, (struct sockaddr *) &address,
89                                   (socklen_t *) &addrlen)) < 0) {
90              continue;
91          }
92          valread = read(new_socket, iv, 16);
93          valread = read(new_socket, buffer, 1024);
94          printf("Resived: ");
95          display(buffer, valread);
96          decrypt(buffer, valread, iv, key, 16);
97          printf("Decrypted: ");
98          display(buffer, valread);
99          if (testPadding(buffer)) {
100             if (testFlag(&buffer[buffer[0]], flag, 36)) {
101                 send(new_socket, greating, strlen(greating), 0);
102             } else {
103                 send(new_socket, &"You know nothing, hacker boy\n", 29, 0);
104                 close(new_socket);
105                 continue;
106             }
107         } else {
108             send(new_socket, &"You know nothing, skid\n", 29, 0);
109             close(new_socket);
110             continue;
111         }
112     }
113 }
```

A.3 Repeatable game source

A.3.1 Interface

```
1   <style>
2       body {
3           font-family: Arial, Helvetica, sans-serif;
4       }
5
6       /* The Modal (background) */
7       .modal {
8           display: block; /* Hidden by default */
9           position: fixed; /* Stay in place */
10          z-index: 1; /* Sit on top */
11          padding-top: 100px; /* Location of the box */
12          left: 0;
13          top: 0;
14          width: 100%; /* Full width */
15          height: 100%; /* Full height */
16          overflow: auto; /* Enable scroll if needed */
17          background-color: rgb(0, 0, 0); /* Fallback color */
18          background-color: rgba(0, 0, 0, 0.4); /* Black w/ opacity */
```

```
19      }
20
21      /* Modal Content */
22      .modal-content {
23          background-color: #fefefe;
24          margin: auto;
25          padding: 20px;
26          border: 1px solid #888;
27          width: 80%;
28      }
29
30  </style>
31  <canvas id="game" height="770" width="770"></canvas>
32  <div id="myModal" class="modal">
33      <div class="modal-content">
34          <p id="textField">Enter license key</p>
35          <input id="license_key_field" >
36          <button id="license_btn">Enter</button>
37      </div>
38  </div>
39  <button onclick="up()">Up</button>
40  <button onclick="down()">Down</button>
41  <button onclick="right()">Right</button>
42  <button onclick="left()">Left</button>
43  <script src=https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-1/crypto-js.js></script>
44  <script>
45
46      let images = [];
47      let player = [];
48      let modal = document.getElementById("myModal");
49      let license_button = document.getElementById("license_btn");
50      let license_key_field = document.getElementById("license_key_field");
51      let license_key = "";
52      let textField= document.getElementById("textField");
53
54      license_button.addEventListener("click", () => {
55              modal.style.display = "none";
56              add_game_controls();
57              license_key = document.getElementById("license_key_field").value;
58              up();
59      });
60
61     function add_game_controls(){
62         window.addEventListener("keydown", function (event) {
63             if (event.defaultPrevented) {
64                 return; // Do nothing if the event was already processed
65             }
66
67             switch (event.key) {
68                 case "Down": // IE/Edge specific value
69                 case "ArrowDown":
70                 case "s":
71                 case "S":
72                     down();
73                     break;
74                 case "Up": // IE/Edge specific value
75                 case "ArrowUp":
76                 case "w":
77                 case "W":
78                     up();
79                     break;
80                 case "Left": // IE/Edge specific value
81                 case "ArrowLeft":
82                 case "a":
83                 case "A":
84                     left();
85                     // Do something for "left arrow" key press.
86                     break;
87                 case "Right": // IE/Edge specific value
88                 case "ArrowRight":
89                 case "d":
90                 case "D":
91                     right();
92                 default:
93                     return; // Quit when this doesn't handle the key event.
94             }
95
96             // Cancel the default action to avoid it being handled twice
97             event.preventDefault();
98         }, true);
99     }
100
101     window.onload = function () {
102         for (let i = 0; i <= 4; i++) {
103             let image = new Image();
104             image.src = `/tile/Character${i}.png`;
105             image.onload = ()=>{};
106             player.push(image);
107         }
108
109         for (let i = 0; i <= 78; i++) {
110             let image = new Image();
111             image.src = `/tile/Outdoors_${i}.png`;
112             image.onload = ()=>{};
```

```
113             images.push(image);
114         }
115     };
116
117     function drawPlayer(pos) {
118         if (pos == "up"){
119             pos = 0;
120         } else if (pos == "down"){
121             pos = 3;
122         }else if (pos == "left"){
123             pos = 2;
124         }else{
125             pos = 1;
126         }
127         let c = document.getElementById("game");
128         let ctx = c.getContext("2d");
129         ctx.drawImage(player[pos], 70 * 5, 5 * 70, 70, 70);
130     }
131
132     function clear() {
133         let c = document.getElementById("game");
134         let ctx = c.getContext("2d");
135         ctx.fillStyle = "#70BB23";
136         ctx.fillRect(0, 0, c.width, c.height);
137     }
138
139     function drawLayer(layer) {
140         let c = document.getElementById("game");
141         let ctx = c.getContext("2d");
142         layer.forEach((row, j) => {
143             row.forEach((element, i) => {
144                 if (element === -1) return;
145                 ctx.drawImage(images[element], i * 70, j * 70, 70, 70);
146             });
147         });
148     }
149
150
151     function sendMove(direction) {
152         let key = CryptoJS.enc.Utf8.parse(license_key);
153
154         let encrypted = CryptoJS.AES.encrypt(direction, key, {
155             mode: CryptoJS.mode.CBC,
156             iv: CryptoJS.lib.WordArray.random(16),
157             padding: CryptoJS.pad.Pkcs7
158         });
159
160         const Http = new XMLHttpRequest();
161         const url = `/move?direction=${encodeURIComponent(encrypted)}&iv=${encodeURIComponent(CryptoJS.enc.Base64.stringify(encrypted.iv))}`
162         Http.open("GET", url);
163         Http.send();
164
165         Http.onreadystatechange = (e) => {
166             if (Http.readyState === 4 && Http.status === 200) {
167                 let map = JSON.parse(Http.responseText);
168                 if (map.flag){
169                     modal.style.display = "block";
170                     license_button.style.display = "none";
171                     license_key_field.style.display = "none";
172                     textField.innerText= map.flag;
173                 }
174                 clear();
175                 drawLayer(map.layer1);
176                 drawLayer(map.layer2);
177                 drawPlayer(direction);
178             }
179
180         };
181
182     }
183
184     function up() {
185         sendMove("up");
186     }
187
188     function down() {
189         sendMove("down");
190     }
191
192     function left() {
193         sendMove("left");
194     }
195
196     function right() {
197         sendMove("right");
198     }
199
200 </script>
```

A.3.2 Backend

```
1 extern crate serde;
```

```
#[macro_use]
extern crate rouille;
#[macro_use]
extern crate serde_derive;

use std::collections::HashMap;
use std::io;
use std::str::Split;
use std::str;
use std::str::FromStr;
use std::sync::Mutex;
use std::thread;
use std::fs::File;
use std::io::{Read, BufReader, BufRead};
use rouille::Request;
use rouille::Response;
use base64::decode;
use hex;
use aes::Aes128;
use block_modes::Cbc;
use block_modes::BlockMode;
use block_modes::block_padding::Pkcs7;

type Aes128Cbc = Cbc<Aes128, Pkcs7>;

static WALKABLE_TILE_VALUES: [i8; 43] = [-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18,
    19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
    54, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68];

#[derive(Debug, Clone, Copy)]
struct SessionData { x: usize, y: usize }

#[derive(Serialize)]
struct MapView { layer1: [[i8; 11]; 11], layer2: [[i8; 11]; 11] }

#[derive(Serialize)]
struct Flag { flag: String }

fn make_map_view(player: &SessionData, layer1: &Vec<Vec<i8>>, layer2: &Vec<Vec<i8>>) -> MapView {
    let mut map_view = MapView { layer1: [[0; 11]; 11], layer2: [[0; 11]; 11] };
    for y in 0..map_view.layer1.len() {
        for x in 0..map_view.layer1.len() {
            let map_x = (player.x + x) as i32 - 5;
            let map_y = (player.y + y) as i32 - 5;
            map_view.layer1[y][x] = layer1[map_y as usize][map_x as usize];
            map_view.layer2[y][x] = layer2[map_y as usize][map_x as usize];
        }
    }
    map_view
}

fn walkable_tile(position: &SessionData, layer1: &Vec<Vec<i8>>, layer2: &Vec<Vec<i8>>) -> bool {
    return WALKABLE_TILE_VALUES.contains(&layer1[position.y][position.x]);
}

fn load_tile_layer_from_file(filename: &str) -> Vec<Vec<i8>> {
    let mut layer = Vec::<Vec<i8>>::new();
    for line in BufReader::new(File::open(filename).unwrap()).lines() {
        layer.push(line.unwrap().split(",").map(|str| {
            i8::from_str(str).unwrap()
        }).collect());
    }
    layer
}

fn handle_move(session_data: &SessionData, move_data: &str, iv: &str, key: &Vec<u8>) -> SessionData {
    let decoded_data = decode(move_data).unwrap();
    let iv = decode(iv).unwrap();


    let cipher = Aes128Cbc::new_var(key, &iv).unwrap();
    let decrypted_ciphertext = cipher.decrypt_vec(&decoded_data).unwrap_or(vec![65]);
    let direction = str::from_utf8(&decrypted_ciphertext).unwrap_or("failed");

    match direction {
        "down" => SessionData { y: session_data.y + 1, x: session_data.x },
        "up" => SessionData { y: session_data.y - 1, x: session_data.x },
        "left" => SessionData { y: session_data.y, x: session_data.x - 1 },
        "right" => SessionData { y: session_data.y, x: session_data.x + 1 },
        _ => SessionData { y: session_data.y, x: session_data.x }
    }
}

fn main() {
    let sessions_storage: Mutex<HashMap<String, SessionData>> = Mutex::new(HashMap::new());
    let mut game_view = String::new();
    let mut flag = String::new();
    let mut key: Vec<u8> = vec![];
    File::open("game_view.html").unwrap().read_to_string(&mut game_view).unwrap();
    File::open("flag.txt").unwrap().read_to_string(&mut flag).unwrap();
    File::open("key.txt").unwrap().read_to_end(&mut key).unwrap();
    let layer1 = load_tile_layer_from_file("tiles/test_Tile Layer 1.csv");
    let layer2 = load_tile_layer_from_file("tiles/test_Tile Layer 2.csv");
    let url = envmnt::get_or("URL", "0.0.0.0");
```

```
96      let port = envmnt::get_or("PORT", "8000");
97
98      rouille::start_server(format!("{}:{}", url, port), move |request| {
99          rouille::log(&request, io::stdout(), || {
100             rouille::session::session(request, "SID", 3600, |session| {
101                 let session_data = sessions_storage.lock().unwrap().
102                     get(session.id()).unwrap_or(&SessionData { x: 13, y: 99 }).clone();
103
104                 router!(request,
105                     (GET) (/) =>{
106                         Response::html(format!("{}", game_view.as_str()))
107                     },
108
109                     (GET) (/tile/{id: String}) => {
110                         if let Some(request) = request.remove_prefix("/tile") {
111                             return rouille::match_assets(&request, "tiles");
112                         }
113                         Response::empty_404()
114                     },
115
116                     (GET) (/move) => {
117                         let new_pos = handle_move(&session_data, &request.get_param("direction").unwrap_or_default(), &request.get_param("iv
118                         if new_pos.x == 12 && new_pos.y == 82{
119                             return Response::json(&Flag{flag: flag.clone()});
120                         }
121                          return if walkable_tile(&new_pos, &layer1, &layer2){
122                             sessions_storage.lock().unwrap().insert(session.id().to_string(), new_pos);
123                             Response::json(&make_map_view(&new_pos, &layer1, &layer2))
124                           }else {
125                             Response::json(&make_map_view(&session_data, &layer1, &layer2))
126                           }
127                      },
128
129                     _=> {Response::empty_404()}
130                 )
131             })
132         })
133     });
134 }
```

A.4 Bad random number generator source

```
1   use num::bigint::{BigInt, Sign};
2   use hex;
3   use num::Integer;
4   use std::env;
5
6
7   pub fn modinverse(a: &BigInt, modulo: &BigInt) -> Option<BigInt> {
8       let test = a.extended_gcd(modulo);
9       if test.gcd != BigInt::from(1) {
10          None
11      }
12      else {
13          Some((test.x % modulo + modulo) % modulo)
14      }
15  }
16
17  fn main() {
18
19
20      let args: Vec<String> = env::args().collect();
21      let p = BigInt::from_bytes_be(Sign::Plus, &hex::decode(args[1].replace(":", "").trim()).unwrap());
22      let q = BigInt::from_bytes_be(Sign::Plus, &hex::decode(args[2].replace(":", "").trim()).unwrap());
23
24      let p_minus_one: BigInt = &p-1;
25      let q_minus_one: BigInt = &q-1;
26
27      let lambda_n = p_minus_one.lcm(&q_minus_one);
28
29      let modulus= &p*&q;
30      let pub_exp = BigInt::from(65537); //public exponent e. 65537
31      let priv_exp =  modinverse(&pub_exp, &lambda_n).unwrap(); //private exponent d.
32      let e1 = &priv_exp % &p_minus_one+ &p_minus_one % &p_minus_one; // d mod (p - 1)
33      let e2 = &priv_exp % &q_minus_one+ &q_minus_one % &q_minus_one; // d mod (q - 1)
34      let coeff = modinverse(&q, &p).unwrap();
35
36     /*
37     asn1=SEQUENCE:rsa_key
38          [rsa_key]
39      version=INTEGER:0
40      modulus=INTEGER:187
41      pubExp=INTEGER:7
42      privExp=INTEGER:23
43      p=INTEGER:17
44      q=INTEGER:11
45      e1=INTEGER:7
46      e2=INTEGER:3
47      coeff=INTEGER:14
48
49
```

```
50      */
51
52      println!("asn1=SEQUENCE:rsa_key");
53      println!("");
54      println!("\t[rsa_key]");
55      println!("version=INTEGER:0");
56      println!("modulus=INTEGER:{}", &modulus);
57      println!("pubExp=INTEGER:{}", &pub_exp);
58      println!("privExp=INTEGER:{}", &priv_exp);
59      println!("p=INTEGER:{}", &p);
60      println!("q=INTEGER:{}", &q);
61      println!("e1=INTEGER:{}", &e1);
62      println!("e2=INTEGER:{}", &e2);
63      println!("coeff=INTEGER:{}", &coeff);
64
65
66  }
```

Bibliography

[1] Let's Encrypt. https://letsencrypt.org/about/, 2000. [Online; accessed 29.05.20]. 14

[2] Ieee standard for information technology—telecommunications and information exchange between systems local and metropolitan area networks—specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, 2016. 11

[3] Rouille. https://github.com/tomaka/rouille, 2019. [Online; accessed 29.05.20]. 38, 42

[4] Android Studio. https://developer.android.com/studio/intro, 2020. [Online; accessed 29.05.20]. 37

[5] Docker Compose. https://github.com/docker/compose, 2020. [Online; accessed 29.05.20]. 36

[6] Ethash. https://www.hashrates.com/algorithms/ethash, 2020. [Online; accessed 29.05.20]. 9

[7] Frida. https://frida.re/, 2020. [Online; accessed 29.05.20]. 20

[8] Magisk. https://magisk.download/, 2020. [Online; accessed 29.05.20]. 20

[9] Nginx. https://github.com/nginx/nginx, 2020. [Online; accessed 29.05.20]. 38, 42

[10] OkHttp. https://github.com/square/okhttp, 2020. [Online; accessed 29.05.20]. 26

[11] OpenSSL. https://github.com/openssl/openssl, 2020. [Online; accessed 29.05.20]. 37

[12] The burp suite familiy. https://portswigger.net/burp, 2020. [Online; accessed 29.05.20]. 20

[13] Volley. https://github.com/google/volley, 2020. [Online; accessed 29.05.20]. 26

[14] Wireshark. https://www.wireshark.org/, 2020. [Online; accessed 29.05.20]. 20

[15] N. Agarwal. Understanding the Docker Internals. https://medium.com/@BeNitinAgarwal/understanding-the-docker-internals-7ccb052ce9fe, 2017. [Online; accessed 29.05.20]. 36

[16] T. Aura. Strategies against replay attacks. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 59–59. IEEE Computer Society, 1997. 32

[17] C. Bailey. Extended Validation Certificates: Warning Against MITM Attacks. https://blog.trendmicro.com/trendlabs-security-intelligence/files/2015/02/MITM-attack-diagram-2.jpg, 2015. [Online; accessed 29.05.20]. 17

[18] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280. https://tools.ietf.org/html/rfc5280f, 2008. [Online; accessed 29.05.20]. 12

[19] W. Crichton. Memory safety in rust: A case study with c. http://willcrichton.net/notes/rust-memory-safety/, 2018. [Online; accessed 29.05.20]. 35

[20] P. Gutmann. X.509 Style Guide. https://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt, 2000. [Online; accessed 29.05.20]. 13

[21] J. Håstad. Solving simultaneous modular equations of low degree. *SIAM J. Comput.*, 17:336–341, 1988. 6

[22] S. Kamkar. phpwn: Attacking sessions and pseudo-random numbers in PHP. https://samy.pl/phpwn/BlackHat-USA-2010-Kamkar-How-I-Met-Your-Girlfriend-wp.pdf, 2010. [Online; accessed 12.02.20]. 33

[23] A. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, and P. Leyland. Factoring estimates for a 1024-bit rsa modulus. In *Advances in Cryptology - ASIACRYPT 2003*, pages 55–74. Springer Berlin Heidelberg, 2003. 6

[24] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. https://eprint.iacr.org/2012/064.pdf, 2012. [Online; accessed 25.05.20]. 33

[25] T. McLean. Critical vulnerabilities in JSON Web Token libraries. https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries, 2015. [Online; accessed 29.05.20]. 50

[26] S. Müller. Linux Random Number Generator – A New Approach. http://www.chronox.de/lrng/doc/lrng.html, 2020. [Online; accessed 29.05.20]. 33

[27] NIST. CVE-2019-6593 Detail. https://nvd.nist.gov/vuln/detail/CVE-2019-6593, 2019. [Online; accessed 07.10.19]. 29

[28] OWASP Foundation Inc. A2:2017-Broken Authentication. https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A2-Broken_Authentication, 2017. [Online; accessed 29.05.20]. 28

[29] OWASP Foundation Inc. Use of hard-coded password. https://owasp.org/www-community/vulnerabilities/Use_of_hard-coded_password, 2020. [Online; accessed 29.05.20]. 18

[30] A. Roy, N. Memon, and A. Ross. Masterprint: Exploring the vulnerability of partial fingerprint-based authentication systems. *IEEE Transactions on Information Forensics and Security*, 12(9):2013–2025, 2017. 16

[31] B. Schneier. The Legacy of DES. https://www.schneier.com/blog/archives/2004/10/the_legacy_of_d.html, 2004. [Online; accessed 29.05.20]. 5

[32] R. Solberg. Case #22: Another booking leak. https://blog.roysolberg.com/, 2019. [Online; accessed 29.05.20]. 27

[33] S. Vaudenay. Security flaws induced by cbc padding — applications to ssl, ipsec, wtls... In L. R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 534–545, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. 28