RADBOUD UNIVERSITY

Faculty of Science

Analysing open-source 5G core networks for TLS vulnerabilities and 3GPP compliance

March 24, 2023

Author:
Alex Bui Nhat Linh
S1040308

Daily supervisor:
David Rupprecht

First assessor:
Erik Poll

Second assessor:
Katharina Kohls

Abstract

TLS is a cryptographic protocol used to provide security and integrity for connections over the internet. It is widely used in web browsers, email, and instant messaging and it has now been implemented in 5G core networks. While TLS is considered to be a relatively secure protocol, it is known to have a number of vulnerabilities. With the rise of open-source software, there are now publicly available implementations for 5G core networks and coupled with TLS vulnerabilities, this could lead to an increase in attack vectors. In order to implement the core network securely, it is essential for the open-source 5G core networks to follow the 3GPP requirements, an organization that produces technical specifications for mobile communications systems. Therefore, this research focuses on analysing the TLS vulnerabilities and 3GPP requirements compliance in three different open-source 5G core networks, namely free5GC, Open5gs and OAI 5G CN. The analysis was done using automated scanning tools, PySSLScan, SSLyze and TLS-Scanner. The results showed that free5GC and Open5gs support TLS within the core network while OAI 5G CN does not, failing the 3GPP requirements. In addition, free5GC implements weak ciphersuites and suffers from various bugs and vulnerabilities, specifically Sweet32 and version intolerance while Open5gs only implemented weak ciphersuites. Additional analysis revealed that both free5GC and Open5gs have default TLS private keys and certificates available to the public.

Contents

Chapter 1

Introduction

It has been decades since the first generation of wireless communications was launched. In the last couple of years, mobile network technology has steadily evolved, with the most recent development being the worldwide deployment of 5G. In 2016, the first trials of 5G were carried out by various companies such as Samsung, Nokia, and Verizon [1] and by 2019, the first commercial 5G mobile networks began to roll out in the United States [2]. Now in 2023, the majority of European countries have deployed 5G networks. Not only does it enable the virtual connection of machines, but 5G is also designed to be faster, have lower latency, have more network capacity, and be more reliable than previous generations of mobile networks [3].

Unlike the previous generations of mobile networks, which were largely aimed at consumer demand for mobile data services, 5G's capability extends beyond broadband access to the internet. The use cases for 5G are classified into three categories [4]:

- Enhanced Mobile BroadBand (eMBB): aims to offer high bandwidth for services such as cloud-based applications accesses, high definition videos, wireless internet access, etc.
- Massive Machine Type Communications (mMTC): the ability to connect large numbers of devices (Internet of Things). Examples of such use cases in this category would be smart cities, healthcare, manufacturing, transportation, etc.
- Ultra-Reliable and Low Latency Communications (URLLC): used for mission-critical applications where low-latency and network reliability is paramount. For example, self-driving cars, industrial robotics, industrial remote control, smart grids, etc.

In 2019, the first technical specifications of 5G, Release 15, was released by 3GPP (3rd Generation Partnership Project) [5]. 3GPP is an engineering group comprised of seven telecommunications standard development organisations (ARIB, ATIS, CCSA, ETSI, TSDSI, TTA, TTC) that develop technical specifications for cellular telecommunication such as 3G, 4G and now, 5G. In Release 15, the backbone of the 5G mobile network, the 5G core network (5GC), is specified to have two possible architectures: 5G Non-Standalone (5G-NSA) and 5G Standalone (5G-SA). The former is where the 5G network is supported by the existing 4G infrastructure while the latter has an independent infrastructure. The core network handles a variety of essential functions in the mobile network such as connectivity, mobility management, authentication and authorisation, subscriber data management and more.

There are many vendors offering 5G core, the most prevalent being Ericsson and Huawei [6]. With the benefits that 5G brings, many companies are deploying 5G networks on their campus. However, the price of deploying the network is relatively high. For instance, the cost of installing a 5G Non-Standalone Core with support of up to 50,000 subscribers is between $250,000 and $1.2 million depending on the provider [7]. Due to the high costs of deploying a 5G network, an alternative is to deploy open-source 5G core networks instead. Open-source means the code of a software is publicly available, allowing any and all persons to read, edit and distribute. The most popular open-source 5G core networks at the moment of writing are Open5gs [8], free5GC [9] and OAI 5G CN [10]. Similar to their paid 5G core counterparts, these open-source core networks offer both 5G-NSA and 5G-SA. In this research, we will focus on the 5G-SA core.

The 5G-SA core is made up of several Network Functions (NF), which are software-based and are designed to be cloud-native. This allows 5G to have higher deployment agility and flexibility. However, this software-based architecture (SBA) may open up new attack vectors for threat actors if the core is not configured properly. For example, a group of security specialists from Penthertz not only succeeded in intruding into a 5G-SA core but also managed to delete the original NF instances and created fake NF instances on the core due to misconfigured network functions [11]. This could be catastrophic if the network was deployed in a company as attackers can steal sensitive subscriber information, monitor communications, gain access to other infrastructures of the company, etc. As a result, it is imperative that each Network Function in the 5G core be appropriately configured and properly protected.

Each Network Function has a role of its own such as session management, policy control, authentication, etc. As a result, each NF acts as a service producer as well as a service consumer to other NFs and they communicate with one another by using one of two methods [12]:

- Request-response method where the consumer NF requests services from the producer NF over HTTP/2 requests. The producer NF then responds.
- Subscribe-notify method where the consumer NF subscribes to the producer NF events. The producer NF can then notify the consumer NF when an event occurs.

In the request-response method, the requests from NF to NF are HTTP/2 requests. The HTTP/2 network protocol, which is a revision of the original HTTP (HyperText Transfer Protocol) network protocol, can be configured to use TLSv1.2 and higher [13]. However, it is not mandatory and for the purposes of this research, the focus will be on HTTP/2 over TLS. TLS (Transport Layer Security) is a cryptographic protocol that aims to provide security and integrity to the communication between two applications [14]. It is currently widely used in applications, emails, etc but its most prevalent use is in HTTPS. However, TLS is not without vulnerabilities. The most notable vulnerabilities of TLS include but are not limited to RSA key transport, CBC mode ciphers, and RC4 stream ciphers [15]. Due to the fact that 5G core network functions use HTTP/2 with TLS to communicate with one another, it is important to analyse the 5G core networks for TLS vulnerabilities as these vulnerabilities could potentially open new attack vectors.

5G is a relatively new wireless technology, therefore, there not much research has been done regarding the security aspect of the 5G core. This is especially true for the 5G-SA core as its 3GPP requirements were only released in 2019 (Release 15) [5]. Most security research

regarding the security of 5G has been focused on the assumption that the 5G core will inherit the vulnerabilities of 4G as the 5G-NSA will be based on the infrastructure of the 4G network [16][17]. There are studies that looked into possible security challenges of the 5G core, focusing on its cloud-based infrastructure [18] or comparing open-source core networks based on their characteristics and functionalities [19]. Furthermore, there is research that focuses on the security aspect of HTTP/2 usage in the 5G-SA network usage, however, TLS vulnerabilities were not mentioned [20].

With the rise of open-source software, especially in 5G technology, it is imperative that the security of open-source 5G core is researched and analysed as these could potentially be deployed in a professional setting. Despite extensive research on TLS vulnerabilities and open-source 5G core, there is a significant gap in how these vulnerabilities can affect the security of open-source 5G core. Therefore, the purpose of this research is to address the gap in the existing knowledge by analysing three different open-source 5G-SA core networks, namely Open5gs, free5GC, and OAI 5G CN for TLS vulnerabilities. During the research, the 3GPP compliance of the core networks will be checked in order to better ensure the security of the open-source 5G core.

The thesis is structured as follows: Chapter 2 presents the theoretical background and related works, Chapter 3 covers the methodology, Chapter 4 outlines the results of the analysis and Chapter 5 offers discussion as well as the conclusion, incorporating the limitations of the study and possible future works.

Chapter 2

Background

In this chapter, we present the theoretical background of the thesis. This includes an overview of the 5G core network and its network functions, the TLS protocol and its vulnerabilities as well the 3GPP organisation and its role in securing mobile communication systems. Following this, we introduce related works to highlight the knowledge gap in the literature.

2.1 Overview of the 5G network

2.1.1 The 5G network and its capabilities

The 5G network is the 5th generation of cellular technology. It boasts unprecedented advancements in data transfer speeds, connectivity and capacity. It aims to deliver faster, more reliable wireless communication services to a plethora of devices, ranging from IoT (Internet of Things) devices and OT (Operational Technology) devices to autonomous devices. In order to fully grasp the improvements of 5G over previous generations of mobile networks, we shall explore briefly the evolution of the predecessors.

1. 1G network
   The 1st generation of mobile networks (1G) was first introduced in the 1980s. It used analogue signals to allow basic voice services. This generation had many limitations such as poor voice quality, bulky handheld devices, and limited cell coverage and functionality. Regarding the security aspect, this network was extremely insecure due to the fact that it used analogue signals which allow calls to be decoded with a device that can extract information from a signal wave (FM demodulator) [21].

2. 2G network
   In order to overcome the limitations of 1G, the second generation of mobile networks was developed. 2G introduced digitalisation to cellular networking, which means the radio signals in 2G are digital instead of analogue. This allowed for significant improvement in voice quality, and more efficient use of bandwidth as well as the introduction of SMS and MMS services. In addition, due to the digitalisation of radio signals, data could be digitally encrypted which enhanced security. However, 2G suffered from low data rates and limitations from hardware [22]. Regarding its security capabilities, it is vulnerable to several man-in-the-middle-attacks due to its use of GSM (Global System for Mobile Communications) [23].

3. 3G network
   Building on top of 2G, 3G developed to have an increased data rate and supported multimedia services in early smartphones such as video calling, access to email, weather, news, etc. 3G would become the foundation for future mobile broadband services and introduce a new generation of smartphones. Nonetheless, 3G had a major security vulnerability that resulted in attackers violating users' private data [24].

4. 4G/LTE network
   The need for higher data rates, a broader range of multimedia capabilities and the extended abilities of modern smartphones gave birth to the fourth generation of mobile networks. 4G allowed for simultaneous transmission of voice as well as data, which resulted in a much higher data rate compared to 1G, 2G, and 3G with data rates up to 1Gbps. In addition, 4G could use voice-over LTE network which has more data capacity than 3G and 2G [25]. However, 4G suffered from security vulnerabilities which leaked the user's location [26].

This brings us to 5G, which aims to substantially improve all of 1G, 2G, 3G and 4G shortcomings as well as introduce new use cases outside of mobile communications. The use cases, as mentioned in the introduction, include Enhanced Mobile BroadBand (eMBB), Massive Machine Type Communications (mMTC) and Ultra-Reliable and Low Latency Communications (URLLC). Out of these use cases, eMBB is the most widespread use case as it is aimed at users. The main purpose of this use case is to enhance users' experience and quality of service by providing users with the highest data rate possible (multi-Gbps) while maintaining super low latency (less than 1 millisecond). In addition, it also offers higher bandwidth (24GHz+) and faster speeds for densely populated regions. This would allow for high-definition video streaming such as 4K/8K and smart cities functionalities [27].

2.1.2 The 5G-SA network architecture

Figure 2.1 illustrates the 5G-SA network architecture. The components of the network architecture include:

- UE: user equipment such as 5G compatible smart devices.
- RAN: Radio Access Network, which is a combination of telecommunications network components with the purpose of connecting the UE to the 5G core.
- Control Plane: the communication path for managing and maintaining the network. Some of its tasks are establishing connections, routing data, maintaining network configuration, etc. In the case of 5G, it is also the core network.
- Data Plane (or User Plane): responsible for user data transmission.
- UPF: User Plane Function is used to connect user data coming from the RAN to the DN.
- DN: Data Network which could refer to internet access, service provider or 3rd part services.

We can see that the network architecture is split into two distinct planes: Control Plane and User Plane. This separation is known as CUPS (Control and User Plane Separation). The main reason for separating the Control Plane and the User Plane is so that the Control Plane

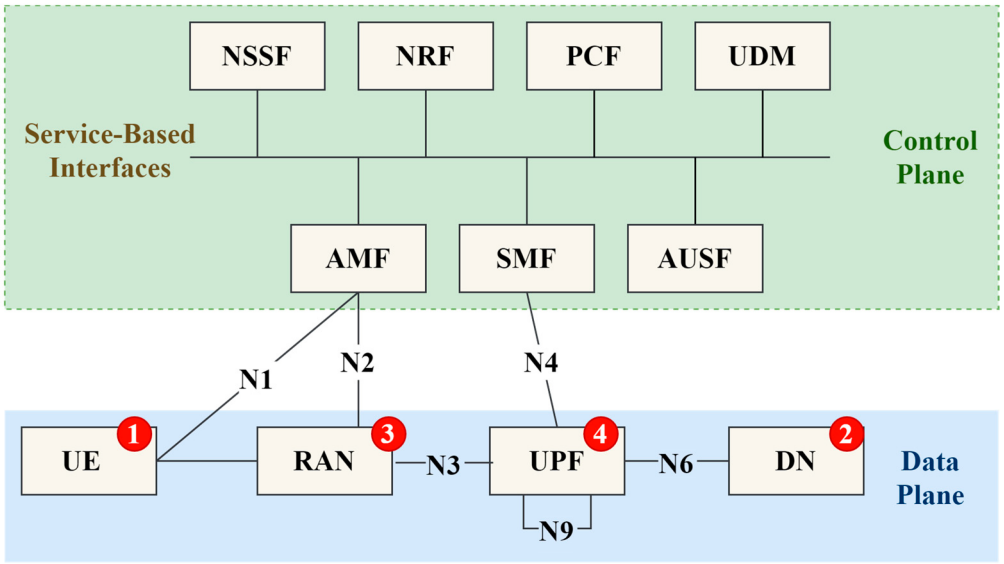

Figure 2.1: 5G network architecture with two distinct planes [29]

Functions can be centralized while allowing the user plane devices to be placed closer to users. This would result in lower latency and faster data transfer thereby improving user experience [28]. The components inside the control plane shall be explained in the next section.

2.2 The 5G-SA core network

The 5G core not only maintains and manages the network but also creates a reliable and secure connection to the network for users while granting them access to 5G services. In the first version of Release 15 by 3GPP, the 5G-SA core network was defined to be a Service-Based Architecture (SBA) [5]. A benefit of 5G SBA is that the core can be deployed as containers, for example, using Docker which would allow the core to run on non-proprietary infrastructures, saving costs [12]. Furthermore, this architecture enables network slicing, meaning various networks can be initialized on top of an existing physical structure and in turn, different parts of the network can be distributed based on the distinct needs of a use case [30]. This allows for a much more efficient and dynamic use of resources. However, the most noteworthy benefit of this architecture is its modularity. In this architecture, components of the core network are called Network Functions (NFs) instead of Network Entities. These NFs are software-based and are designed to be cloud-native, meaning they are built to fully take advantage of the cloud computing model. This allows the core network to be highly scalable and flexible for deployments and upgrades.

Within the core, there are multiple NFs, each with a different responsibility. However, we will only introduce the main NFs, which are AMF (Access and Mobility Management Function), AUSF (Authentication Server Function), NRF (NF Repository Function), NSSF (Network Slice Selection Function), PCF (Policy Control Function), SMF (Session Management Function) and UDM (Unified Data Management).

2.2.1 Access and Mobility Management Function (AMF)

The main functionalities of the AMF are registration management, reachability management, connection management and mobility management [31]. In order for the UE to connect to 5G services, it must be able to reach the network. The AMF's reachability management is responsible for this. It notifies the UE while the UE is in the "Idle" state, which in turn triggers the N1 signalling connection. After the N1 signalling connection has been established, the connection management function of the AMF will make sure that the UE is connected to the AMF by changing the state of the UE from "Idle" to "Connected". Once the connection has been made, the UE must register itself with the AMF in order to use the 5G services. If the registration was successful then the state of the UE will change from "Deregistered" to "Registered" and a UE context will be generated inside the core network. Upon complete registration of the UE, the UE must periodically update its status to the AMF. This allows the mobility management function to make sure that the UE is still connected to the network [32].

2.2.2 Authentication Server Function (AUSF)

During the registration process between the UE and the AMF, the AMF will contact the AUSF in order to authenticate the UE. The AUSF is not only used to authenticate the UE but is also used to store the security keys [33]. When authenticating a UE, it uses one of two mandatory authentication protocols as specified by 3GPP: 5G-AKA (5G Authentication and Key Agreement) and EAP-AKA' (Extensible Authentication Protocol) [34]. After the AUSF has finished authenticating a UE, it will then notify the UDM of the outcome of the authentication. As the authentication server, it ensures the communication between the UE and the 5G core is authentic, confidential and reliable. This helps guard the network as well as the user by preventing unauthorized access to the network.

2.2.3 NF Repository Function (NRF)

The NRF is a crucial network function that mainly manages network functions, storing and distributing their information throughout the 5G core network [35]. In network function management, each NF in the core network must register its services and capabilities with the NRF, which results in a network function repository that is stored within the NRF. This repository allows a network function to discover other NFs and allows the network function to subscribe to the NRF so that it can be notified if any modifications are made to other NFs. In addition, the NRF is also responsible for generating authorization tokens for other NFs, which allows the NFs to access certain resources and services in the network.

2.2.4 Session Management Function (SMF)

The SMF manages the sessions between the UE and the core network and it works hand-in-hand with the AMF to achieve this. To be more specific, the SMF creates, updates and deletes sessions between the UE and the control plane while managing the quality-of-service (QoS) of these sessions [36]. It then binds the UE's sessions to the AMF, so that the AMF has the necessary information regarding the UE's network access. In addition, the SMF aids the PCF in enforcing policy and charging rules, relaying these rules to the UPF.

2.2.5 Policy Control Function (PCF)

As its name suggests, the PCF controls the policy and charging rules in the core network. In addition to defining policy rules that control how data flows in the network, the PCF also determines charging rules i.e the cost of network services based on quality-of-service (QoS), usage, volume, etc [37]. The network function makes adjustments to the rules as needed based on the data that it collects and analyses from the core network. In order to enforce the policy and charging rules, the PCF communicates with the AMF and the SMF [38].

2.2.6 Network Slice Selection Function (NSSF)

The main role of the NSSF is to manage network slices. More specifically, this function is used by the AMF to help select the appropriate network slice instance based on the user's and use case requirements as well as the network conditions. After which, the NSSF will allocate suitable resources to the network slices such that they meet the service requirements [39]. It also takes care of other tasks such as user mobility across network slices as well as network slice termination.

2.2.7 Unified Data Management (UDM)

The UDM's main task is to store and manage user data. User data includes users' subscriber information, their authentication credentials, user profiles, service data, etc. It then provides relevant user information to other network functions whenever requested. In addition, the UDM is also responsible for authenticating and authorizing subscribers for certain service access [40]. For instance, when the UDM receives a request for service access by a user, it will check the user's credentials in its database and once the user has been authenticated, it will allow the user to access certain network services based on their subscription data.

Due to the fact that the 5G-SA core is designed to be an SBA, the NFs use a Service Based Interface (SBI) to communicate with one another. As mentioned in the Introduction, each NF acts as a service producer and a service consumer. They communicate with one another using API-based communication, where an NF can invoke an API call to another NF over the SBI to request service from the other NF. In the 3GPP version 16, it is specified that the SBI shall use the HTTP/2 protocol, which is the successor of HTTP/1.x and a request-response based protocol, for NFs communication [41].

2.3 The Transport Layer Security Protocol

2.3.1 Overview of TLS

The Transport Layer Security Protocol (TLS) is a cryptographic protocol that is used to secure a connection between a client and a server. It is the successor of the outdated Secure Socket Layer (SSL) protocol. What makes TLS stands out from SSL is that TLS is more secure than SSL in several aspects as TLS supports newer ciphersuites, stronger encryption algorithms and hash functions, as well as a more reliable handshake.

TLS secures a connection by protecting the confidentiality and integrity of the data being transferred using encryption and verification methods to make sure that the data did not get

tampered with during transmission and that only the intended parties may access the data. In addition, TLS makes sure that the client is communicating with an authentic server and vice versa by authenticating the server and client [42]. Due to its security, TLS is widely used in web browsing, online banking, email, etc. Every TLS connection starts with a TLS handshake, which uses asymmetric encryption and once the TLS handshake succeeds, symmetric encryption is used to ensure authenticity, privacy and reliability in the communication [43].

Asymmetric encryption, which is shown in Figure 2.2, is the use of a key pair, consisting of a private key and a public key. The process of sharing public keys does not have to be secure. However, due to the fact that the public key is mathematically related to the private key, the size of the keys must be large enough such that it is infeasible to derive the private key from the public key. Because of this, asymmetric encryption is computationally intensive and therefore, slow. On the other hand, symmetric encryption, illustrated in Figure 2.3 uses the same secret key to encrypt and decrypt data, making it much faster than asymmetric encryption [44]. As a result, TLS uses asymmetric encryption to generate and exchange a session key securely and uses symmetric encryption to secure the data transfer in a TLS communication.

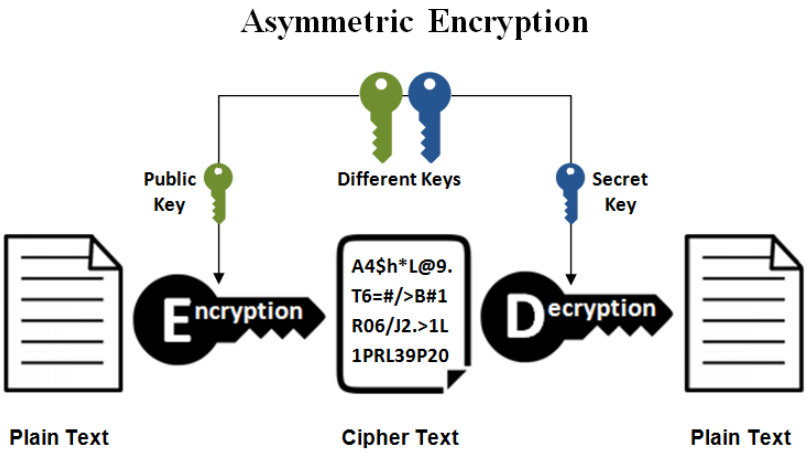

Figure 2.2: Asymmetric encryption [45]

The TLS handshake is a crucial component in setting up a secure TLS connection as it establishes the security details of the connection such as session keys, cryptographic algorithms, TLS versions, etc. The basic steps of a TLS handshake are executed as follows [46]:

1. The client sends a ClientHello message to the server with the client's supported ciphersuites, TLS version, and a string of random bytes known as the client random.

2. The server responds to the client with a ServerHello message, containing the server's certificate, chosen TLS version, ciphersuites, and a string of random bytes known as the server random.

3. The client authenticates the server based on the server's certificate and extracts the server's public key from the certificate.

4. The asymmetric encryption process starts. The client generates a premaster secret,

encrypts this secret using the server's public key and sends it to the server. The encrypted secret can only be decrypted using the server's private key.

5. The server decrypts the premaster secret with its private key.
6. The client and the server both generate a session key using the client's random, the server's random and the premaster secret.
7. The server and client then exchange Finished messages that are encrypted with the session key.

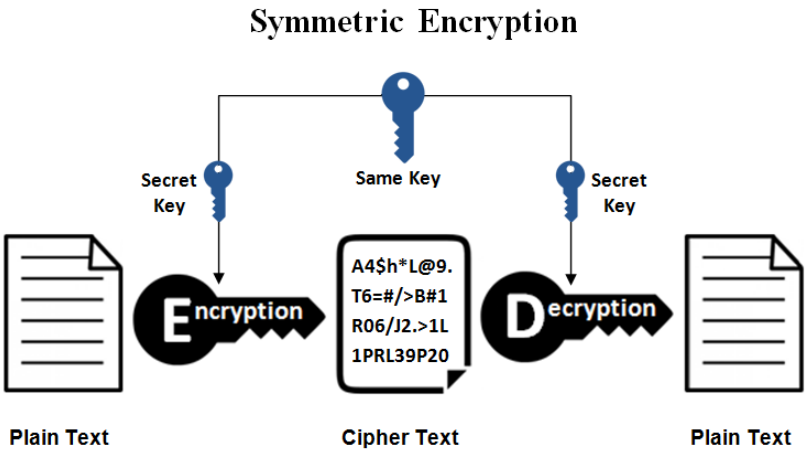

Figure 2.3: Symmetric encryption [45]

To make the process more clear, the TLS handshake is illustrated in Figure 2.4 After the TLS handshake is completed successfully, data transmitted in the connection will then be encrypted and decrypted using the session key. However, it should be noted that the specific TLS handshake process may differ between TLS versions, but the information that is exchanged is similar.

2.3.2 TLS vulnerabilities and versions vulnerabilities

There are several known attacks against TLS. In this section, we shall explain some of the most concerning vulnerabilities and specify which TLS versions suffer from them.

2.3.2.1 TLS vulnerabilities

1. **BEAST**
   BEAST stands for Browser Exploit Against SSL/TLS and it targets a vulnerability in TLSv1.0 and older using CBC-mode encryption. In the CBC-mode encryption of TLSv1.0 and older, each block of plaintext is XORed with the previous ciphertext block, essentially acting as the Initialization Vector, and then encrypted with a key. This process is shown in Figure 2.5 To make the message unique, a randomly generated IV is used in the first block [48]. Due to the fact that CBC does not use a random IV for every block, which makes the IV predictable, an attacker may perform a man-in-the-middle attack

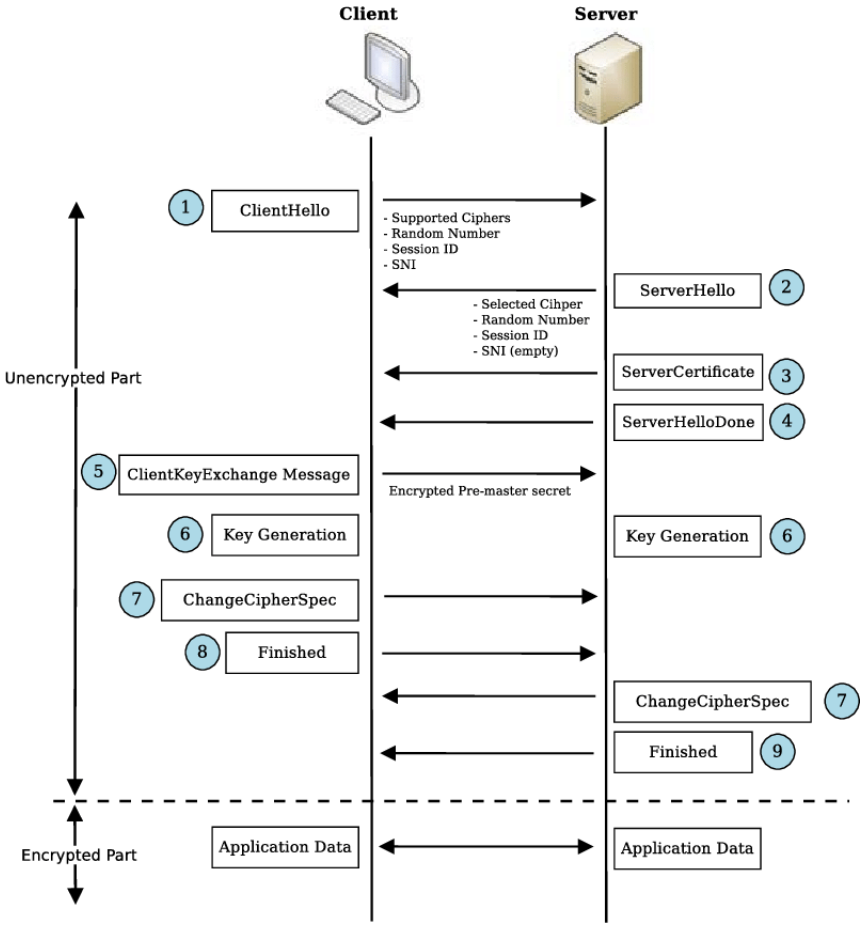

Figure 2.4: The TLS handshake [47]

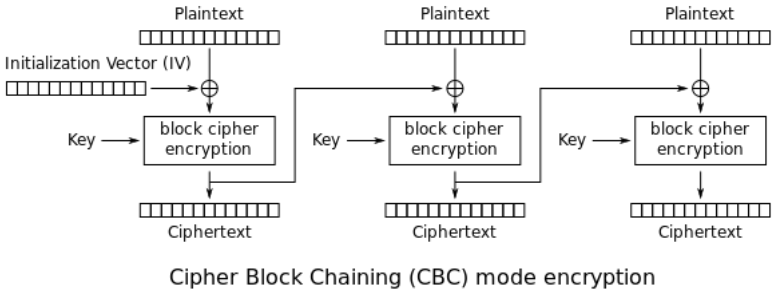

Figure 2.5: CBC encryption mode [50]

to execute BEAST. BEAST works by the attacker sending their chosen IV XORed with chosen plaintext to the server and observing the server's response. The attack allows the adversary to retrieve sensitive information such as credit card numbers, passwords, etc [49].

2. **Padding Oracle Attack**
   According to Paterson and Yau [51], these types of attack require an oracle that when receiving a ciphertext, will decrypt it and replies whether the ciphertext was padded correctly or incorrectly. The attack relies on the assumption that the adversary can intercept padded messages using CBC-mode with some key K while also having access to the previously mentioned padding oracle (using the same key K). The attacker can then retrieve the plaintext by making an average of $128b$ queries to an oracle where b is the number of bytes in a ciphertext block. Examples of a padding oracle attack are POODLE (Padding Oracle on Downgraded Legacy Encryption) [52] and LUCKY13 [53].

3. **Bleichenbacher Attack**
   This attack is possible when the key-exchange in TLS uses the RSA-based PKCS#1 v1.5 encryption scheme and assumes that the attacker has access to an "oracle" that checks whether the given ciphertext has the correct padding when decrypted [54]. The RSA algorithm is a public-key cryptography system, which uses a public encryption key to encrypt plaintexts and uses a private decryption key to decrypt the ciphertext. The algorithm also has homomorphic multiplication property, meaning when an encrypted text is multiplied with another encrypted text, the decryption of the result will be meaningful [55]. PKCS#1 v1.5 (Public Key Cryptography Standard v1.5) is a padding scheme recommended for the RSA algorithm, whereby the plaintext is padded with a random string starting with $0x000x02$, before being encrypted. Using the homomorphic multiplication property, the padding scheme and public encryption key knowledge, an attacker can retrieve the plaintext.

4. **Raccoon Attack**
   The Raccoon Attack relies on precise timing measurements and specific conditions in order to successfully break the encrypted connection [56]. It targets the Diffie-Hellman key exchange method used in servers running TLSv1.2 or lower. In Diffie-Hellman key exchange, the server and client first generate their own private keys and use these to generate their public keys. The public keys are then exchanged, allowing the server and client to compute a shared key, called the *premaster secret*. This premaster secret is

then used to derive all TLS session keys with a key derivation function. In order for an attacker to carry out this vulnerability, the following conditions must be met:

- The server must be configured for TLSv1.2 or below
- The victim connection must use Diffie-Hellman key exchange and the server must reuse ephemeral Diffie-Hellman public keys
- The attacker must be able to observe the connection between client and server *from the very beginning*
- The attacker must be near the server in order to perform precise timing measurements

If all of these conditions are met, then the attacker can retrieve the original premaster secret and use this to decrypt the connection between the client and server.

5. **CRIME Attack**
   The CRIME (Compression Ratio Info-leak Made Easy) vulnerability exploits the use of compression in TLS, allowing an attacker to hijack a user's authenticated web session which will open other attack vectors. Whenever we log into a web application, authentication information such as usernames and passwords are stored in a cookie. The cookie is compressed using a compression algorithm, encrypted and then sent to the server. The authentication information inside this cookie allows our browser to send HTTPS requests to the web application during the session, without needing us to authenticate each time we send a request. Therefore, if an attacker has access to the authentication cookie, then they can hijack the user's session. In order to obtain this cookie, the adversary will make the browser repeatedly send HTTPS requests to the web application, after which they can observe the difference in request length after compression to determine the value of the user's cookie [57].

6. **BREACH Attack**
   Unlike the CRIME attack which targets the compression used in TLS, the BREACH attack targets HTTP responses that uses HTTP compression. This type of compression is much more common than TLS compressions. The attack takes advantage of the compressed size of the text when there are repetitive terms, meaning if a webpage has many repetitive terms, the size of the compressed output will be reduced [58]. In order for the attack to be carried out, the server must use compression at the HTTP level and the website should have some parameter that reflects the input text so that the attacker can control the input. If these two conditions are met, then an attacker can change the input text and analyze the different sizes of the compressed output to deduce the user's sensitive information.

7. **Invalid Curve Attack**
   This attack exploits the Elliptic Curve Diffie-Hellman (ECDH) key exchange in TLS in order to get the server's private key. Once the private key is known, an attacker can decrypt all session communications. In this attack, a client can force a TLS server to compute a common secret with a point outside the defined curve. This invalid point could belong to a different curve, which could contain a small number of elements, making it possible for an attacker to calculate all valid points on this curve [59]. Using this information, an attacker can derive the server's secret key.

8. **LogJam Attack**
   The LogJam Attack targets the Diffie-Hellman key exchange used in TLS, which results in the downgrading of a TLS connection to use 512-bit export-grade ciphersuites. Export-grade ciphersuites are defined as low-grade ciphersuites that are generally considered insecure. Once the connection has been downgraded, an adversary can carry out a Man-in-the-Middle attack exploiting the 512-bit export-grade ciphersuites to eavesdrop and manipulate data in the connection [60]. In addition, the attacker can decrypt old sessions with the same Diffie-Hellman parameters.

9. **Sweet32 Attack**
   The Sweet32 Attack, a type of Birthday Attack, exploits the usage of 64-bit block cipher algorithms such as 3DES. In TLS, block cipher algorithms are used to encrypt data that is sent between the server and the client. Before being encrypted, the data is broken into fixed-length blocks and each block is then encrypted separately, according to a mode of operation. The security of a block cipher is dependent on the size of its key, however, the block size also plays a crucial role in securing the block cipher. For example, in 3DES-CBC (3DES using CBC mode of operation), two ciphertexts will be identical after $2^{n\div 2}$ message blocks are encrypted with the same key [61]. This collision in output means that the plaintexts are the same and also leaks the xor of two plaintext blocks. In order for this attack to work, a fixed secret i.e sensitive data such as HTTP cookies must be sent repeatedly and a fraction of the plaintext must be known. Once these conditions are fulfilled, a collision would reveal the secret.

10. **DROWN Attack**
    DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) is a cross-protocol attack that exploits SSLv2 in order to decrypt TLS connections. It allows attackers to decrypt, read and steal users' sensitive information such as passwords, credit card numbers, etc. There are two versions of this attack: General DROWN and special DROWN. General DROWN leverages the vulnerabilities in SSLv2 protocol to carry out a stronger variant of the Bleichenbacher Attack mentioned above. On the other hand, special DROWN exploits an OpenSSL vulnerability in its implementation of the SSLv2 server, making this version of the attack cheaper and easier to execute [62]. The attack is carried out in three stages:

    (a) An attacker observes and records sessions between client and server using RSA ciphersuites over TLS or SSL connections.

    (b) The attacker establishes several SSLv2 connections to the server, using modified handshake messages that they recorded in step a. These modified handshakes target the RSA ciphersuites.

    (c) The attacker can observe the server response to guess the encryption key.

    A server is vulnerable to DROWN if it permits both TLS and SSLv2 connections, or if it shares the private key or the public key with a different server that allows SSLv2 connections [63].

11. **Heartbleed attack**
    This attack exploits the Heartbeat extension in the TLS protocol, specified in RFC 6520 [64]. This extension allows the secure connection to be kept alive if no data is

exchanged. The attack is a type of buffer-overread attack, which means that the system can read restricted data outside of the buffer. This vulnerability is executed as follows [65]:

(a) A client sends a malicious heartbeat request to the server with a 1-byte data payload and a payload length of 65535 bytes.

(b) A server responds with 65535 bytes of payload data, copied from memory, due to the fact that there is no payload length validation.

The data in the server response can contain sensitive information such as private keys, passwords, etc.

12. EarlyCCS Attack
EarlyCCS (Early ChangeCipherSpec) Attack exploits a vulnerability in the OpenSSL 1.0.1 implementation. The goal of a CCS message is to notify the parties (client or server) that the connection should use a different CipherSpec and keys, specified in the CCS message. Usually, this CCS request would be sent during the handshake after the security parameters (private keys and master keys) have been agreed upon by the client and the server, but before the Finished message is sent [14][46]. However, due to a flaw in the implementation of OpenSSL 1.0.1, the CCS request can be sent out of order, meaning that it can be sent before the security parameters have been established, resulting in a zero-length premaster secret key being used. Since session keys and future session keys are derived using the zero-length premaster secret key, this would allow an attacker to decrypt and modify packets in the connection and future connections using these compromised keys [66].

13. Session Ticket Zero Key (CVE-2020-13777)
Session Ticket Zero Key utilizes an implementation flaw in GnuTLS version 3.6.x. GnuTLS is an open-source library implementing TLS, SSL and DTLS [67]. In versions 3.6.x until 3.6.14, GnuTLS used insecure cryptography to encrypt a session ticket resulting in the session ticket encryption key using the wrong data until the first key rotation. This allowed an attacker to carry out a Man-in-the-Middle (MitM) attack that can recover past communications in TLSv1.2 and, in addition, bypass TLSv1.3 authentication [68].

14. ALPACA
ALPACA is an application layer protocol content confusion attack. TLS servers that implement different protocols but use compatible certificates, for example, multi-domain or wildcard certificates are vulnerable to the ALPACA attack. An attacker can establish a valid TLS session by leveraging this attack to redirect traffic from one subdomain to another. Not only does this result in the TLS authentication being broken, but it may also allow for other cross-protocol attacks to be carried out [69].

15. Renegotiation attack
During a secure SSL/TLS connection, either the client or server can initiate a new handshake, in order to request new ciphersuites or key materials. This process is called renegotiation [70]. Due to the fact that the renegotiation process puts more strain on the server compared to the client, an attacker can carry out a Denial-of-Service (DoS) attack on the server by starting multiple handshakes, resulting in the server becoming unresponsive. In addition, this vulnerability may allow a MitM injection attack. This

means that an attacker can inject malicious data into the HTTPS session through unauthenticated requests, which can result in the attacker being able to execute commands using an authorized user's credentials and obtain other users' credentials [71].

2.3.2.2 TLS versions and their vulnerabilities

TLSv1.0 was introduced in 1999 while TLSv1.1 came out in 2006, and since then several vulnerabilities have been discovered targeting these protocols, including BEAST (Brower Exploit against SSL/TLS) [49], POODLE (Padding Oracle On Downgraded Legacy Encryption) [52] and DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) [62]. These attacks are possible due to the fact the insecure TLS versions use old, weak ciphersuites. For example, the weak ciphersuite `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA` is required in TLSv1.0 [14]. Furthermore, according to RFC 8996 [72], the integrity of the TLSv1.0 and TLSv1.1 depends on the SHA-1 hash of the exchanged messages. SHA-1 has officially been broken in 2017 [73] and the use of this hash algorithm in handshakes can result in possible downgrade attacks, such as LogJam, in 2^{77} operations, which is far below the modern security margin. The authentication of the handshakes also depends on SHA-1 signatures or a concatenation of MD-5 and SHA-1 hashes, which allows an attacker to impersonate a server if the SHA1 hash is broken.

TLSv1.2 was released in 2008 and it is still widely supported today by many web servers. It improves upon TLSv1.0 and TLSv1.1 by offering support for stronger ciphersuites and newer, more secure key exchange protocols such as Diffie-Hellman Ephemeral (DHE) and Elliptic Curve Diffie-Hellman (ECDHE). Depending on its configuration, TLSv1.2 may have multiple vulnerabilities such as Raccoon, Lucky13, POODLE, Heartbleed and CRIME. Some of these attacks require certain conditions to be fulfilled in order to be executed successfully. For example, in order for the Raccoon attack to be successful, some of the conditions are that the TLS connection needs to use the Diffie-Hellman or the Ephemeral Diffie-Hellman key exchange algorithm and it also relies on precise timing requirements [56].

TLSv1.3 is the most recent TLS version released in 2018. It eliminates most of the vulnerabilities found in previous TLS versions by only supporting ciphersuites that currently have no known vulnerabilities, removing the renegotiation capabilities, enabling PFS by default, etc [74]. However, there is a known vulnerability of TLSv1.3 that was discovered in 2020 that allowed for downgrade attacks. This vulnerability exploits the fallback mechanism in TLSv1.3 that results in the client and server using an older version of TLS if there are compatibility issues [75].

2.3.3 TLS in 5G core network

As mentioned in the 5G-SA core network section of this chapter, NFs communicate with one another using the HTTP/2 protocol. While the HTTP/2 protocol itself does not require the implementation of TLS, the 3GPP requirements state that the use of TLS is mandatory in network functions [41]. Within the 5G core network, it is necessary to use TLS due to the fact that TLS offers encryption, authentication, and other security guarantees to the connection between NFs, which helps to boost the security strength of the communication in the 5G core substantially.

First, TLS ensures that the network functions are authentic as the first step to establishing a TLS connection is the execution of the TLS handshake. The handshake allows the network

functions to authenticate one another, verifying the identity of the network functions before they are permitted to exchange data. Second, TLS prevents the modification of data and makes sure that sensitive information is not leaked, should an attacker get access to the 5G core, by encrypting the data that is being transferred between network functions. In addition to encrypting data, TLS also implements integrity checks such as Message Authentication Code (MAC) in order to further make sure that the data did not get tampered with during transit.

However, in order for the 5G core to fully utilize TLS to secure the connection between its network functions, TLS needs to be configured properly. As we mentioned above, TLS suffers from many vulnerabilities depending on the TLS version and its configurations. If an attacker gets access to the 5G core and the TLS configuration is improper, the attacker can leverage the vulnerabilities to retrieve sensitive data such as network functions information, subscribers information, user information, etc. They can then use these data to further exploit the 5G core, such as creating a fake network function that the attacker controls or performing a Denial-of-Service (DoS) attack to bring down the 5G core network entirely.

Therefore, for the 5G core to fully utilize TLS to secure its communication, the TLS configuration needs to be implemented appropriately.

2.4 The 3GPP requirements

As mentioned in the Introduction, the 5G network was designed to support many use cases in different industries. Therefore, it is crucial that the 5G network is secure to guarantee the safety and reliability of these use cases. The 3rd Generation Partnership Project (3GPP) helps to ensure the security of 5G networks. 3GPP is an engineering group comprised of seven telecommunications standard development organisations: ARIB, ATIS, CCSA, ETSI, TSDSI, TTA and TTC [76]. The group develops technical specifications for mobile communications systems, including cellular telecommunication networks such as 3G, 4G and 5G.

3GPP specifies and continuously maintains the security standards and guidelines for the 5G network. The security specifications cover all aspects of the 5G network, including the core network, the user equipment the access network, etc. This makes sure that important security measures and features are integrated into the design and operation of the network which helps 5G mitigate potential network-related attacks. For example, some of the security features that are defined by the 3GPP in the 5G core network include encryption, authentication, communication protocols and more [77]. In addition, the 3GPP group partners with several other organizations to keep track of emerging security threats and in turn updates the requirements of the 5G network to mitigate these threats. Not only does 3GPP define the security standards of 5G networks, but they also specify the protocols and network interfaces of the entire mobile system such as call and session control, mobility management, service provisioning, etc [78]. This makes sure that the network can be operated with other 5G networks regardless of vendors and technology.

With the rise of publicly available implementations for 5G core networks, it is crucial that these open-source core networks follow and update in accordance with each release of the 3GPP requirements for 5G core. Open-sourced means that anyone can access and modify the implementation of the core network. For malicious users, this allows them to analyse the code

for any vulnerabilities that could be exploited. It is also possible for a malicious user to submit their own code or to propose changes to the existing code that could result in a vulnerability in the core network [79]. In order to mitigate the above-mentioned risks, the open-source 5G core network should strictly follow the 3GPP security standards as well as have strict reviews for code changes and code submissions.

2.5 Related Work

5G has been gaining attention in the last few years with the 5G core specifications published by 3GPP in 2019 [77]. With the introduction of any new technology, there are always concerns over its security. Many studies have explored the security of 5G in all aspects including its network protocols, its core network, etc.

Shaik et al. presented a security study of 5G at the BlackHat USA 2019 conference that revealed new vulnerabilities affecting the 5G-NSA core [17]. They presented and performed three classes of attacks: identification attacks, bidding down attacks and battery-draining attacks. The first class allows an attacker to discover user equipment on the 5G network and reveal hardware and software information of the device. The second class can degrade the data rate of the device and downgrade the device to use 3G/2G instead. The final class results in the rapid drainage of the device's battery life. All three attack classes target devices and networks that support LTE. This meant that the 5G-NSA core network, which depends on the 4G LTE infrastructure will also be affected. The researchers concluded the paper by providing countermeasures against the attacks.

A study by researchers from The National Digital Switching System Engineering & Technological Research Center looked into whether the usage of the HTTP/2 protocol is secure in the 5G core network [20]. They conducted a security analysis into current and new features of HTTP/2 and identified a number of possible attack vectors such as stream reuse attacks, flow control attacks, dependency and priority attacks, header compression attacks, man-in-the-middle (MitM) attacks and interconnect attacks. After this, the researchers proposed security mechanisms such as encryption schemes in order to mitigate the above-mentioned threats. They conclude that the challenges that HTTP/2 poses to the 5G core can be addressed with appropriate security measures.

There was also a study by researchers at CPQD, a telecommunication R&D Center in Brazil, comparing three different open-source 5G core networks implementations such as Magma, Open5gs and free5GC [19]. The aim of the comparison was to choose a suitable open-source core network so that the CPQD can further build upon it. The comparison was based on the analysis of five characteristics of the implementations namely, Infrastructure, Documentation, Community and Code Maturity, Base Projects and Management. Magma core was chosen as a result of the analysis due to better user management, more active community, better separation of the control and user plane, etc.

Finally, a research team from Penhertz successfully conducted an attack that allowed them to access a 5G-SA core [11]. The attack started from conducting web-based exploits which ultimately resulted in the group accessing the core network. After this, they launched other exploits that allowed them to hijack existing network functions, create new network functions

that they control, etc.

While these studies have covered a wide range of security issues for both the 5G-SA core and 5G-NSA core, there was limited research on the security of open-source 5G core networks. Open-source software is becoming increasingly popular and this also applies to the implementations of 5G core networks. There are already three well-known open-source 5G core networks namely Open5gs, free5GC and OAI 5G CN. While open-source software has many benefits such as reduced costs, easier license management, etc, there are also several security risks that are associated with it as mentioned earlier in this chapter. As we have seen from the study by researchers from CPQD, open-source 5G core networks may be used to be further developed in the future or companies may opt to use open-source 5G core networks to reduce costs [19]. Therefore, the security of these 5G core networks should be researched to mitigate possible threats. Furthermore, none of the above-mentioned studies looked into how TLS vulnerabilities could affect the security of a 5G-SA core. In 3GPP version 16, it was specified that the use of TLS is mandatory within the core network. Despite being a widely used protocol and considered to be relatively secure, TLS still suffers from a number of vulnerabilities which may open up new attack vectors for adversaries in the 5G core network.

In this study, we aim to build on top of existing research by analysing three open-source 5G core networks namely free5GC, Open5gs and OAI 5G CN for TLS vulnerabilities. We shall also investigate whether these core networks comply with version 17.1.0 of the 3GPP specifications [80] as this release defines the TLS requirements. This is due to the fact the security standards defined by 3GPP help mitigate possible threats in the 5G core network and being open-sourced, where anyone can access and analyse the code, it is even more important for these core networks to follow the requirements.

Chapter 3

Methodology

The main goal of this research is to analyse the open-source 5G core networks, Open5gs, free5GC and OAI 5G CN, for TLS vulnerabilities. In this section, we will explain the setup, the tools that will be used, and how the results of these tools will be used to analyse the core networks.

3.1 The set up

The core network will be running in a local environment using Docker. Docker is a platform that allows developers to package their software and dependencies into lightweight "containers", which can run on different computing environments [81]. In this setup, each Network Function will be a Docker container and these containers will be able to communicate with one another. In order to analyse these NFs for TLS vulnerabilities, we need to scan the network functions for their SSL/TLS configurations. From these configurations, we can then evaluate whether there are any misconfigurations of TLS and whether these misconfigurations can be used to exploit the 5G core. We will be using three open-source tools to scan for the core's TLS configurations: PySSLScan, SSLyze and TLS-Scanner.

The second objective of this study is to check whether the core networks are compliant with the 3GPP requirements. After applying the tools on the core networks, we will analyse and compare the result of the tools, based on the level of detail of the output. This is so that we can choose which scanner would be the most appropriate to use, in order to develop a comparison script. After which, the results of the chosen tool will be used as input for an evaluator script which cross-checks the results with the 3GPP requirements. We shall be using Release 16 and Release 17 of the 3GPP requirements for the check. This is because they are the 3GPP specifications of how to implement a 5G core network.

3.2 The tools

3.2.1 TLS-Scanner

TLS-Scanner is a tool developed by researchers at Ruhr-Universität Bochum. It helps with evaluating TLS configurations for TLS-enabled servers by scanning them thoroughly [82].

TLS-Scanner scans for a number of configurations, including the server's certificates, which are explained below. It does this by impersonating a client and trying to connect to the server.

Ciphersuites

A ciphersuite is a combination of algorithms that are used to secure a TLS network connection. The algorithms that compose a ciphersuite are key-exchange algorithm, authentication algorithm, bulk encryption algorithm, and MAC (Message Authentication Code) algorithm [83]. During the TLS handshake between a client and server, the ciphersuite that the connection will use is agreed upon. It is important to note that the use of a weak ciphersuite will compromise the security of the connection as it will enable attackers to decrypt the communication. A typical ciphersuite looks like:

`TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`

We can break down the example ciphersuite:

- `TLS` indicates the protocol.
- `ECDHE` (Elliptic-Curve Diffie-Hellman) signifies the key exchange algorithm.
- `RSA` indicates the authentication algorithm.
- `AES_256_GCM` signifies the bulk encryption algorithm.
- `SHA384` indicates the MAC algorithm

Named Groups

Named Groups are a predefined set of elliptic curves (EC) and finite fields (DH) that are used for key exchange, determined during the TLS handshake between the client and server. The named group established should satisfy the security requirements in order to ensure a secure and reliable TLS connection [84].

Signature and Hash Algorithms

Signature and Hash algorithms in TLS are used to sign and authenticate messages. The signature algorithm is used to sign data, verifying the message's integrity while the hash algorithm is used to verify the message's authenticity by hashing the message. The signature and hash algorithms are also decided upon by the server and client during the TLS handshake [85].

Ordering

TLS-Scanner also checks for ciphersuite ordering, named group ordering and signature hash algorithm ordering. The ciphersuites, named group and signature hash should be ordered from strongest to weakest so that the most secure algorithms are used for the connection between the server and client [86]. The selection process is done during the TLS handshake. For example, the client will send an ordered list of ciphers that it supports and the server will reply to the client with a cipher that the server chose from the client's list.

Perfect Forward Secrecy

Perfect Forward Secrecy (PFS) is a security measure to make sure that past encrypted communications will not be decrypted should the current private key be exposed. PFS is accomplished by changing the encryption and decryption keys automatically whenever a new session is established, without using any of the previous keys. This means that even if an attacker can get a hold of a session private key, they will not be able to decrypt the communication in previous sessions as the keys are no longer the same [87]. PFS is important for applications or connections that require long-term secrecy such as banking apps or VPN connections because it helps to secure users' sensitive personal information if an attacker obtains the private keys.

Online Certificate Status Protocol

Online Certificate Status Protocol (OCSP) is a protocol that can be queried by the client in order to check whether the digital certificate the client received from the server is revoked or not [88]. The protocol is essential to the security and trustworthiness of digital certificates as a certificate may need to be revoked if its private key was leaked or if the certificate was issued improperly.

Common Bugs

TLS-Scanner does not only scan the server for TLS configuration but also checks whether the server suffers from common bugs. Some important bugs include:

- **Version, Ciphersuites, Extension, Named Group, Signature Hash Algorithm Intolerance:** occurs when a server terminates the connection prematurely or the server uses outdated TLS configurations when it does not recognize (or support) the TLS version, ciphersuites, or extension that the client advertises during the TLS handshake.
- **ClientHello, Ciphersuites Length Intolerance:** occurs when a server crashes and becomes unresponsive if the ClientHello length or ciphersuites length is longer than the server expects. This bug can be exploited by threat actors to carry out a Denial-of-Service attack on the server.
- **Ignores offered Ciphersuites, Named Groups, Signature Hash Algorithms:** is when a server ignores the client's preferred list of Ciphersuites, Named Groups and Signature Hash Algorithms. An attacker can exploit this bug by forcing the server to use a weaker ciphersuite, named group or signature hash algorithm which will result in the connection from a client to a server being insecure.
- **Reflects offered Ciphersuites**: happens when a server echoes back the client's list of preferred Ciphersuites. This can expose the client's configuration and will open other attack vectors for adversaries.

Vulnerabilities

In addition to common bugs, TLS-Scanner also checks if the server is vulnerable to the following attacks, which have been explained in the Background chapter.

- Padding Oracle Attack

- Bleichenbacher
- Racoon
- CRIME
- BREACH
- Invalid Curve
- POODLE
- LogJam
- Sweet32
- DROWN
- Heartbleed
- EarlyCCS
- Zero key
- ALPACA
- Renegotiation Attack

In addition to scanning TLS configurations, TLS-Scanner checks the scan results with existing technical guidelines on TLS security such as BSI TR-02102-2 [89] and NIST SP 800-52r2 [90]. It also uses the scan results to give recommendations on securing the server.

3.2.2 SSLyze

SSLyze is another open-source SSL/TLS configuration scanning tool created by nabla-c0de [91]. It can retrieve the server's certificates, ciphersuites, TLS versions, TLS compression and supported Elliptic Curve key exchange curves. In addition, SSLyze scans for TLS vulnerabilities such as CCS Injection, Heartbleed, ROBOT (Return of Bleichenbacher Oracle Threat) and Renegotiation attacks. The ROBOT attack is another version of the Bleichenbacher attack as mentioned in TLS-Scanner. Using the scan results, SSLyze will check the server's TLS configurations against Mozilla TLS configuration for compliance [92], which is a generator that gives recommendations for the TLS implementation of servers.

3.2.3 PySSLScan

PySSLScan is a scanning tool, developed by DinoTools, that is used to scan servers with TLS/SSL enabled. It can scan for enabled TLS versions and protocols in a server. The configurations it can scan are TLS ciphersuites, preferred ciphers, compression methods, elliptic curves compression, server certificates, and renegotiation. Furthermore, it can test for TLS vulnerabilities such as Heartbleed [93].

3.3 The 3GPP TLS requirements

The specifications for the implementation of the TLS protocol in a 5G-SA core were defined in Release 16 and Release 17. It specifies the TLS versions that should be allowed and made a distinction in requirements between TLS version 1.2 and TLS version 1.3 While compiling the TLS requirements, we noticed that there were no differences in the TLS specifications between Release 16 and Release 17. Below is a summary of the requirements.

3.3.1 TLS versions

The SSL/TLS versions SSLv1.0, SSLv2.0, SSLv3.0, TLSv1.0, TLSv1.1 and DTLSv1.0 shall not be supported. However, TLSv1.2 defined in RFC 5246 [46], TLSv1.3 defined in RFC 8446 [74] should be supported. In addition, should DTLS be implemented then the TLS endpoint should use DTLSv1.2 defined in RFC 6357 [94]. It should be noted that RFCs use keywords such as "MUST", "SHOULD", etc to specify technical requirements. According to RFC 2119 [95], if the keyword "MUST" is used, then requirement is mandatory to implement

3.3.2 Requirements for TLSv1.3

If the core supports TLSv1.3, then it should use the ciphersuites, the key exchange curves, the signature schemes and the TLS extensions as specified in RFC 8446. These are:

- The ciphersuite TLS_AES_128_GCM_SHA256 MUST be used.
- The ciphersuites TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256 SHOULD be used.
- The key exchange curve secp256r1 MUST be used.
- The key exchange curves secp384r1, X25519 SHOULD be used.
- The signature schemes ecdsa_secp256r1_sha256, rsa_pkcs1_sha256, rsa_pss_rsae_sha256 MUST be supported.
- The signature schemes ecdsa_secp384r1_sha384 SHOULD be supported.
- The extensions "supported_versions", "cookie", "signature_algorithms", "signature_algorithms_cert", "supported_groups", "key_share", "server_name" MUST be implemented

Furthermore, Release 16 and Release 17 state that the additional OSCP extension defined in RFC 6066 [96] and RFC 8446[74] SHOULD be supported.

3.3.3 Requirements for TLSv1.2

TLS ciphersuites

If the core also supports TLSv1.2, then the TLS specifications of ciphersuites in RFC 5246 shall be followed. These are:

- The cipher suites TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, TLS_DHE_RSA_WITH_AES_128_GCM_SHA256, TLS_RSA_WITH_AES_128_CBC_SHA MUST be supported.

- The ciphersuites TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384, TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 SHOULD be supported.

TLS PSK ciphersuites

Release 16 and 17 also specify the support of pre-shared-key (PSK) ciphersuites should they be implemented:

- The PSK ciphersuites TLS_DHE_PSK_WITH_AES_128_GCM_SHA256, TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256 MUST be implemented.
- The PSK ciphersuite TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384 SHOULD be supported.

TLS hash and signature algorithms

The following hash and signature algorithms should be implemented according to the 3GPP TLS specification:

- The hash SHA-256 MUST be supported.
- The hash SHA-384 SHOULD be supported.
- The hashes MD5, SHA-1 MUST NOT be supported.
- The signature schemes ecdsa, rsa_pss_rsae, rsa_pkcs1 MUST be supported, but the usage of rsa_pkcs1 is not recommended.
- The signature schemes ecdsa_secp384r1_sha384 SHOULD be supported

TLS Compression and Extensions

The following compression method specified in RFC 5246 [46] and extensions should be supported:

- The compression method NULL MUST be supported.
- The extensions SNI (Server Name Indication), TLS renegotiation, Extended Master Secret, Signature Algorithms, Supported Groups MUST be supported.
- The extensions TLS Session Resume, and OCSP status SHOULD be supported.
- The extension Truncated HMAC MUST NOT be supported.

To check the 3GPP compliance of the core networks, we process the output of the three tools with a Python script. The result of this script is a report which specifies the vulnerabilities the core networks suffer from and the 3GPP specification that the core networks violate. The complete Python script can be found in Appendix A, Listing 5.1.

Chapter 4

Results

In this chapter, we will present the results of the research. First, we will dive into the source code of the core networks to explore whether they support TLS between network functions. This is because the analysis of TLS vulnerabilities relies on the fact that the core networks support TLS. Second, we will present the results from each of the scanning tools and based on this, we will decide which tool will be used for the development of the comparison script. Third, we will go deeper into the functionality of the Python comparison script and explain its output. In addition, we will also explain how all of these results from the tools and from the script correlate with the research topic.

4.1 Results from inspecting the source code

4.1.1 free5GC

To make sure that free5GC supports TLS communication between its network functions, we will take a closer inspection at free5GC's version 3.2.1 source code [9].

A quick way to check whether free5GC supports TLS is to investigate the configuration files of the network functions. This is because a configuration file is used to define the parameters and the settings of a network function, such as the function's ID, name, IP, etc. As an example, Listing 4.1 presents a portion of the AMF network function configuration file.

```
info:
  version: 1.0.3
  description: AMF initial local configuration

configuration:
  amfName: AMF # the name of this AMF
  ngapIpList:  # the IP list of N2 interfaces on this AMF
    - 127.0.0.18
  sbi: # Service-based interface information
    scheme: http # the protocol for sbi (http or https)
    registerIPv4: 127.0.0.18 # IP used to register to NRF
    bindingIPv4: 127.0.0.18  # IP used to bind the service
    port: 8000 # port used to bind the service
    tls: # the local path of TLS key
      pem: config/TLS/amf.pem # AMF TLS Certificate
      key: config/TLS/amf.key # AMF TLS Private key
```

Listing 4.1: free5GC AMF configuration

Taking a closer look, we can see that the file specifies the SBI information (Service-Based Interface) starting from line 9. The SBI, as mentioned in the Background section, is the communication method between two NFs. The most important information from the SBI settings in the AMF configuration file is the `scheme` parameter. In the `scheme` parameter, we can choose between HTTP and HTTPS. This meant that we can configure the AMF to communicate with the other network functions using HTTP over TLS. However, it is not always the case that the SBI settings in the configuration file are an indication of TLS support in the core network, as we will see in the case of Open5gs. Therefore, in order to verify that free5GC allows the usage of TLS within its core, we will look at the implementation of a network function.

```
1  bindAddr := factory.NrfConfig.GetSbiBindingAddr()
2    logger.InitLog.Infof("Binding addr: [%s]", bindAddr)
3    server, err := httpwrapper.NewHttp2Server(bindAddr, nrf.KeyLogPath,
       router)
4    if err != nil {
5      logger.InitLog.Warnf("Initialize HTTP server: +%v", err)
6      return
7    }
8
9    serverScheme := factory.NrfConfig.GetSbiScheme()
10   if serverScheme == "http" {
11     err = server.ListenAndServe()
12   } else if serverScheme == "https" {
13     // TODO: support TLS mutual authentication for OAuth
14     err = server.ListenAndServeTLS(
15       factory.NrfConfig.GetNrfCertPemPath(),
16       factory.NrfConfig.GetNrfPrivKeyPath())
17   }
18
19   if err != nil {
20     logger.InitLog.Fatalf("HTTP server setup failed: %+v", err)
21   }
```

Listing 4.2: free5GC NRF implementation

We will be analysing the code for the NRF. We chose to inspect the NRF because it is certain that it will communicate with the other NFs, as the other NFs must register themselves to the NRF on the core network start-up. The file that we examined was the `init.go` file in the NRF directory, which is the initialization file for the NRF function when the core network is launched [97]. Listing 4.2 shows a snippet of this file. From line 9 to line 17, we can see that the code first checks the SBI scheme and initializes the server based on the scheme. If the scheme uses `https` then the code will call the method `ListenAndServeTLS`. This method is implemented by the HTTP package of Golang and according to its documentation, the function starts an HTTP server that expects TLS connections i.e HTTPS server [98]. In this case, the NRF will be started as an HTTPS server.

As a good measure, we also analysed the AMF, AUSF, SMF `init.go` initialization files and they yielded the same results as the NRF. This meant that the open-source core network free5GC supports TLS communications between the network functions. After adjusting the configuration files for the network functions, we started up the free5GC core network. To further confirm that the core network is using TLS we used Wireshark to capture the communication. Figure 4.1 shows the TLS communication between the NRF, with IP `172.44.0.12`, and other

network functions.

```
#  o TLS enable/disable
#  sbi:
#    server|client:
#      no_tls: false|true
#    - false: (Default) Use TLS
#    - true:  TLS disabled
...
#  o SBI Server(https://<all address available>:443)
#  sbi:
#    server:
#      key: /etc/open5gs/tls/amf.key
#      cert: /etc/open5gs/tls/amf.crt
#  amf:
#    sbi:
```

Listing 4.3: Open5gs AMF configuration file

```
172.44.0.12    172.44.0.15    TLSv1.3   1552 Server Hello, Change Cipher Spec, Application Data, Application
172.44.0.13    172.44.0.12    TLSv1.3    857 Application Data
172.44.0.13    172.44.0.12    TLSv1.3     99 Application Data
172.44.0.12    172.44.0.13    TLSv1.3    112 Application Data
172.44.0.12    172.44.0.13    TLSv1.3    123 Application Data
172.44.0.13    172.44.0.12    TLSv1.3    234 Application Data
172.44.0.13    172.44.0.12    TLSv1.3    154 Application Data
172.44.0.13    172.44.0.12    TLSv1.3    132 Change Cipher Spec, Application Data
172.44.0.12    172.44.0.13    TLSv1.3   1552 Server Hello, Change Cipher Spec, Application Data, Application
172.44.0.15    172.44.0.12    TLSv1.3    320 Client Hello
172.44.0.13    172.44.0.12    TLSv1.3    320 Client Hello
```

Figure 4.1: Wireshark capture of free5GC traffic

4.1.2 Open5gs

For Open5gs, we followed the same procedure as free5GC to evaluate whether Open5gs supports TLS within its core network. It should be noted that the first analysis for Open5gs was done in September 2022 with Open5gs version 2.4.9.

We first checked the configuration files for the network functions. A part of the AMF configuration file is shown in Listing 4.3. From lines 1 to 6, we can enable or disable TLS, with the default using TLS. Furthermore, in lines 8 to 14, we can configure the paths for the AMF TLS certificate and the AMF TLS key. This meant that we should be able to use TLS to communicate between the network functions. However, after changing the configuration files and starting the Open5gs core network, the core network failed to launch and there were errors in the SBI. As it turned out, Open5gs did not yet support TLS in version 2.4.9 and this issue was raised in the discussion of the Open5gs github [99].

However, we carried out a second analysis on Open5gs in January 2023 when Open5gs released v2.6.0 which supports TLS communication within the core network [100]. In Open5gs, there is a dedicated file that starts up an HTTP server called `nghttp2-server.c`, which the network functions use [101]. On closer inspection of this file, we see that it uses the `TLS_server_method()` function from the OpenSSL API, as seen in Listing 4.4 line 4. According to the OpenSSL documentation, this function initializes a server that uses TLS [102]. Furthermore, in lines 37 and 41, the code checks for the SBI scheme in order to deploy the proper server. It should be noted that on February 18th 2023, Open5gs enabled TLS communications within the core network by default [103]. In addition, going back to the Open5gs

official GitHub in March 2023, we could no longer find the version tagged v2.6.0. On our local repository of Open5gs, however, the git commit that used to be tagged v2.6.0 had the hash `739cb599d1998d4f87199eeada323358f162744d` [100]. We could not find the reason why the version had been removed from the official GitHub.

```
    static SSL_CTX *create_ssl_ctx(const char *key_file, const char *
    cert_file) {
    SSL_CTX *ssl_ctx;

    ssl_ctx = SSL_CTX_new(TLS_server_method());
    if (!ssl_ctx) {
        ogs_error("Could not create SSL/TLS context: %s",
    ERR_error_string(ERR_get_error(), NULL));
        return NULL;
    }

    SSL_CTX_set_options(ssl_ctx,
                            SSL_OP_ALL| SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3
     |
                            SSL_OP_NO_COMPRESSION |
                            SSL_OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION
    );

#if OPENSSL_VERSION_NUMBER >= 0x30000000L
    if (SSL_CTX_set1_curves_list(ssl_ctx, "P-256") != 1) {
        ogs_error("SSL_CTX_set1_curves_list failed: %s", ERR_error_string
    (ERR_get_error(), NULL));
        return NULL;
    }
#endif /* !(OPENSSL_VERSION_NUMBER >= 0x30000000L) */


    if (SSL_CTX_use_PrivateKey_file(ssl_ctx, key_file, SSL_FILETYPE_PEM)
    != 1) {
        ogs_error("Could not read private key file - key_file=%s",
    key_file);
        return NULL;
    }
    if (SSL_CTX_use_certificate_chain_file(ssl_ctx, cert_file) != 1) {
        ogs_error("Could not read certificate file - cert_file=%s ",
    cert_file);
        return NULL;
    }

...

 hostname = ogs_gethostname(addr);
    if (hostname)
        ogs_info("nghttp2_server() [%s://%s]:%d",
                server->ssl_ctx ? "https" : "http",
                hostname, OGS_PORT(addr));
    else
        ogs_info("nghttp2_server() [%s://%s]:%d",
                server->ssl_ctx ? "https" : "http",
                OGS_ADDR(addr, buf), OGS_PORT(addr));

    return OGS_OK;
```

```
172.22.0.28          172.22.0.12          TLSv1.3    416 Client Hello
172.22.0.12          172.22.0.28          TLSv1.3   1513 Server Hello, Change Cipher Spec, Application Data, Application
172.22.0.28          172.22.0.12          TLSv1.3     74 Change Cipher Spec
172.22.0.28          172.22.0.12          TLSv1.3    142 Application Data
172.22.0.12          172.22.0.28          TLSv1.3    323 Application Data
172.22.0.12          172.22.0.28          TLSv1.3    323 Application Data
172.22.0.28          172.22.0.12          TLSv1.3    114 Application Data
172.22.0.28          172.22.0.12          TLSv1.3    117 Application Data
```

Figure 4.2: Wireshark capture of Open5gs traffic

```
45 }
```

Listing 4.4: Open5gs TLS implementation

After upgrading the Open5gs core network to version v2.6.0 and enabling TLS using the network function configuration files, the core network could be started. In Figure 4.2, we can see that the NRF with IP `172.22.0.12` communicates with other network functions over TLSv1.3. This confirms that the open-source core network Open5gs supports TLS within the core.

4.1.3 OpenAirInterface5G (OAI 5G CN)

We did not find any support for TLS within the core network in the OAI 5G CN version 1.4.0 source code. In order to further ensure that OAI 5G CN does not support TLS, we contacted the developers and they confirmed that OAI 5G CN currently does not yet use TLS communications between the network functions.

This could be considered a security risk as the connection within the core network is not protected. Once an attacker gets access to the core network, the fact that the communication is not encrypted makes it easier for them to steal sensitive information, for example, subscriber information, user data, NFs' configurations, etc. They can also leverage the data to mount other attacks within the core network such as deleting NFs, creating malicious NFs, carrying out a DoS attack on the core network, and so on.

From the source code analysis, free5GC and Open5gs support TLS within their core network while OAI 5G CN does not.

4.2 Results from the tools

We scanned free5GC and Open5gs using PySSLScan, SSLyze and TLS-Scanner. In this section, we shall determine which tool is the most appropriate for the core network analysis based on the level of detail of its scan results. After which, we shall present the chosen tool's results on free5GC and Open5gs.

To evaluate each of the tool's scan results, we executed them on the NRF in the free5GC core network. Based on the results, we created a table 4.1 specifying the TLS configurations that each tool tests for. The scan results of each tool can be seen in Appendix B, Figure 5.1, 5.2, 5.3 and 5.4. In the PySSLScan column of Table 4.1, ee see that it scans the NRF for its TLS versions and ciphersuites as well as the preferred ciphersuites. In this case, the NRF accepts TLSv1.0, TLS v1.1 and TLSv1.2. Furthermore, it also specifies the ciphersuites that each version uses. Along with checking for basic TLS configurations such as session compression, TLS

Tools / Configs	PySSLScan	SSLyze	TLS-Scanner
TLS versions	1.0 v1.1 v1.2	1.0 v1.1 v1.2 v1.3	1.0 v1.1 v1.2 v1.3
Cipher Suites	Yes	Yes	Yes
Compression	Yes	No	Yes
Renegotiation	Yes	Yes	Yes
EC point format	No	Yes	Yes
HTTP Info	Yes	No	Yes
Certificate Info	No	Yes	Yes
Compliance documents	No	Mozilla	BSI TR-02102-2 NIST SP 800-52r2
Extensions	No	No	Yes
Named Groups	No	Yes	Yes
Signature Hash Algorithms	No	No	Yes
Bugs	No	No	Yes
Vulnerabilities	1	5	15
Recommendations	No	Yes	Yes

Table 4.1: Comparison of tools output.

renegotiation, and HTTP Information, PySSLScan also tests the NRF for one vulnerability.

The scan results from SSLyze, which are illustrated in the SSLyze column of Table 4.1, seem to be more detailed than that of PySSLScan. For instance, SSLyze indicates that the NRF also accepts TLS version 1.3, but PySSLScan did not show this. Additionally, SSLyze also scans for the NRF's certificate information as well as the named groups. Moreover, SSLyze analyses the NRF for more vulnerabilities than PySSLScan. While PySSLScan only tests for one vulnerability, SSLyze tests for five TLS vulnerabilities. Another interesting piece of information that SSLyze inspects for is compliance against the Mozilla TLS configuration, which is a document maintained by Mozilla that gives recommendations for TLS configurations in order to aid developers with the deployment of websites on Mozilla. This compliance check can be useful as it gives a list of suggestions on how to secure the server, in this case, the NRF.

Similar to PySSLScan and SSLyze, TLS-Scanner also scans the server for its TLS versions and the ciphersuites that the server supports. In addition, it specifies the TLS extensions and signature hash algorithms that the NRF supports. Unlike PySSLScan and SSLyze, TLS-Scanner also assesses the NRF for common TLS bugs. This is useful for configuring a secure server as attackers can leverage these TLS bugs to execute TLS attacks. Furthermore, the scanner inspects for more diverse vulnerabilities than SSLyze as it evaluates the server for 15 vulnerabilities. Additionally, TLS-Scanner scans for server compliance against official TLS guidelines such as NIST SP 800-52r2 [90] and BSI TR-02102-2 [89]. The tool then makes recommendations for securing the server based on the scans and the guidelines. These recommendations are more detailed compared to SSLyze's suggestions.

TLS versions	v1.0 v1.1 v1.2 v1.3
Ciphersuites	TLS_CHACHA20_POLY1305_SHA256 TLS_AES_128_GCM_SHA256 TLS_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA TLS_RSA_WITH_3DES_EDE_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA TLS_AES_256_GCM_SHA384 TLS_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_256_GCM_SHA384
Bugs	Version Intolerance
Vulnerabilities	Sweet32

Table 4.2: Part of free5GC NRF TLS-Scanner output

Based on the results of the three scanners, we decided to choose TLS-Scanner as the primary tool to analyse the core networks free5GC and Open5gs. This is because TLS-Scanner gives far more in-depth information about the TLS configuration of the server compared to PySSLScan and SSLyze. The common bugs and the extensive list of TLS attacks will be especially helpful for the topic of this research: the analysis of the core networks for TLS vulnerabilities.

4.2.1 free5GC TLS-Scanner results

We used TLS-Scanner to inspect the free5GC core network by scanning all of its network functions and we compared the outputs to make sure that the TLS settings of the NFs are identical. The result of this was that the network functions had the same TLS configurations. We will then present the most important sections of the scan results in a table, namely the TLS versions, the ciphersuites, the bugs and the vulnerabilities. As an example, we shall use the scan outcomes of the NRF, illustrated in Table 4.2. The complete scan results are attached in Appendix B, Figure 5.3 and Figure 5.4.

4.2.1.1 TLS versions

From the scan results, free5GC supports TLSv1.0, TLSv1.1, TLSv1.2 and TLSv1.3. The IETF (Internet Engineering Task Force) specifically states that TLSv1.0 and TLSv1.1 should be deprecated by the end of 2019 [72], therefore, the fact that free5GC supports TLSv1.0 and TLSv1.1 is a concerning security issue, which has been further explained in the Background section. In the case of free5GC, if an attacker succeeds in intruding the core network, they can impersonate an NF. This is a huge security risk as the attacker will be able to retrieve sensitive user data,

such as subscriber information, passwords, etc. Additionally, free5GC supports all four TLS versions, which can increase the chances of misconfiguration and in turn, increase the attack vectors.

4.2.1.2 Ciphersuites

free5GC supports 14 ciphersuites in total and it also supports ciphersuites ordering from strongest to weakest. As mentioned in the Ordering section of the Methodology chapter, ciphersuite ordering is essential in order to secure TLS communications as it ensures that the strongest ciphersuites will always be negotiated first. We checked the strengths of these ciphersuites using Ciphersuite Info, which is an open-source searchable directory of every TLS ciphersuites that was defined by the IETF. The ciphersuites are evaluated by breaking them down to their containing algorithms, which are then individually assessed [104]. According to Ciphersuite Info, free5GC supports 8 weak ciphersuites while the rest are strong ciphersuites. The reason these ciphersuites are deemed weak is due to the fact that they use algorithms such as RSA, 3DES, SHA-1, and CBC. These algorithms are known to be vulnerable to a number of attacks, as mentioned in the Background section. In addition, SHA-1 has been broken since 2017 by a group of researchers [73]. These weak ciphersuites are:

TLS_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_256_GCM_SHA384

```
1   func NewHttp2Server(bindAddr string, preMasterSecretLogPath string,
    handler http.Handler) (*http.Server, error) {
2   if handler == nil {
3     return nil, errors.New("server needs handler to handle request")
4   }
5
6   h2Server := &http2.Server{
7     // TODO: extends the idle time after re-use openapi client
8     IdleTimeout: 1 * time.Millisecond,
9   }
10  server := &http.Server{
11    Addr:    bindAddr,
12    Handler: h2c.NewHandler(handler, h2Server),
13  }
14
15  if preMasterSecretLogPath != "" {
16    preMasterSecretFile, err := os.OpenFile(preMasterSecretLogPath, os.
    O_WRONLY|os.O_CREATE|os.O_TRUNC, 0o600)
17    if err != nil {
18      return nil, fmt.Errorf("create pre-master-secret log [%s] fail: %s"
    , preMasterSecretLogPath, err)
19    }
20    server.TLSConfig = &tls.Config{
21      KeyLogWriter: preMasterSecretFile,
22    }
```

```
23    }
24
25    return server, nil
26 }
```

Listing 4.5: free5GC NewHttp2Server method

In order to find out the reason that free5GC uses weak ciphersuites, we had to look deeper into the free5GC source code. Specifically, analysing how the core network initializes the server and which TLS library it uses. According to the free5GC GitHub page, the core network is written almost entirely in Go programming language [9]. This gives us a hint into how the ciphersuites are chosen in free5GC as Go has a package that implements TLS, `crypto/tls`. Once again, we will analyse the implementation of the NRF as an example. In Listing 4.2 line 3, we see that the NRF is started up as an HTTP server using the method `NewHttp2Server` in the package `httpwrapper`. Investigating this package and the method shows that the TLS settings for the server are configured by a method in the Go TLS package, `tls.Config`, which is illustrated in Listing 4.5 on line 20. According to the documentation of the `tls.Config` method [105], if the `CipherSuites` parameter is left blank then a safe default ciphersuites list is used and the list may change over time. Furthermore, when inspecting the list of default ciphers, it states that the default cipher suites which are selected by the `tls` package may be different from the static list [105]. Consequently, we conclude that the free5GC ciphersuites are automatically generated by the `tls.Config` method and the list produced by the method contained weak ciphersuites.

The use of weak ciphersuites could decrease the security strength in the free5GC core network significantly. Due to the fact that the weak ciphers use 3DES, RSA, SHA, and CBC, an attacker can leverage these ciphers to carry out attacks such as Sweet32, BEAST, POODLE, etc. These attacks would allow sensitive information to be leaked, whether it is information about the core network or user data.

4.2.1.3 Common bugs and vulnerabilities

From the scan results of TLS-Scanner, free5GC suffers from the version intolerant bug. Briefly mentioned in the Common Bugs section of the Methodology chapter, version intolerance occurs when the server receives an unknown TLS version from the client. This results in the server using an outdated TLS version or the connection being terminated prematurely. This bug can affect the security of the core network.

Version intolerance can result in a TLS version downgrade attack. If the client and the server cannot agree on a recent TLS version, then the server may try to reconnect with the client using an older TLS version. As shown by Antoine Delignat-Lavaud at the Black Hat USA 2014 Conference, a man-in-the-middle-attacker can force the version downgrade by blocking the ClientHello messages with recent TLS versions [106]. Furthermore, older versions of TLS utilize outdated ciphersuites, which would open up other attack vectors for an adversary. For the free5GC core network, this would reduce the security of the connection between network functions, making it easier for attackers to decrypt sensitive data, monitor and/or tamper with the communication.

In addition, free5GC suffers from the Sweet32 vulnerability according to the TLS-Scanner

TLS versions	v1.2 v1.3
Ciphersuites	TLS_CHACHA20_POLY1305_SHA256 TLS_RSA_WITH_AES_128_CBC_SHA256 TLS_RSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 TLS_RSA_WITH_AES_256_CBC_SHA256 TLS_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_AES_128_GCM_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 TLS_RSA_WITH_AES_128_CBC_SHA
Bugs	None
Vulnerabilities	None

Table 4.3: Part of Open5gs NRF TLS-Scanner output

results. This is most likely due to the fact that the TLS communications may be using the ciphersuites such as TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA and TLS_RSA_WITH_3DES_EDE_CBC_SHA, which both utilizes the 3DES block cipher. The Sweet32 vulnerability would allow an attacker to decrypt messages in secure TLS communications between network functions, revealing subscriber data or sensitive information about the core network such as network configurations.

4.2.2 Open5gs TLS-Scanner results

Following the same scanning procedure as free5GC, the TLS configurations for the network functions are identical. Therefore, we shall present as an example the TLS-Scanner results of the NRF. The most relevant sections of the output are shown in Table 4.3. The complete scan results are Figure 5.5 and Figure 5.6 in Appendix B.

4.2.2.1 TLS versions

According to the TLS-Scanner results in Table 4.3, Open5gs only accepts TLS connections that uses TLSv1.2 and TLSv1.3 with the latter being the most recent TLS version.

The support of only two TLS versions is already an increase in the security of Open5gs as it decreases the chances of misconfiguration, which in turn, decrease the attack vectors. In addition, Open5gs is not vulnerable to the Raccoon Vulnerability, a complex timing attack targeting the Diffie-Hellman and Ephemeral Diffie-Hellman key exchange algorithm affecting TLSv1.2, as mentioned in the Background section. This is because Open5gs does not use these key exchange algorithms, according to the TLS-Scanner scan results. Depending on the

TLSv1.2 configuration of the Open5gs core network, the protocol will offer good security for the connection between the network functions. Furthermore, with the support of TLSv1.3, which offers several security improvements compared to the previous TLS versions, Open5gs is protected against a number of vulnerabilities such as BEAST, CRIME, POODLE, etc.

4.2.2.2 Ciphersuites

Open5gs supports 16 ciphersuites in total, 10 of which are weak ciphersuites. These ciphersuites are composed of compromised algorithms, similar to the ciphersuites that free5GC supports:

```
TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_RSA_WITH_AES_128_CBC_SHA
```

Furthermore, Open5gs does not support Ciphersuites ordering.

In Listing 4.4, we have seen that Open5gs initializes its TLS server by using the method `SSL_CTX_new(TLS_server_method())`, which is implemented in the OpenSSL library. According to the OpenSSL documentation [107], the method `SSL_CTX_new()` will initialize a default list of ciphers if the list is not specified otherwise. Diving into the source code of OpenSSL on GitHub, we explored the `ssl_ciph.c` file [108]. This file implements the ciphersuites management functions, ciphersuites selection functions, ciphersuites validation functions, etc. This is where we found the default list of TLSv1.2 and TLSv1.3, illustrated in Listing 4.6.

```
1     /*
2  * Default list of TLSv1.2 (and earlier) ciphers
3  * SSL_DEFAULT_CIPHER_LIST deprecated in 3.0.0
4  * Update both macro and function simultaneously
5  */
6 const char *OSSL_default_cipher_list(void)
7 {
8     return "ALL:!COMPLEMENTOFDEFAULT:!eNULL";
9 }
10
11 /*
12  * Default list of TLSv1.3 (and later) ciphers
13  * TLS_DEFAULT_CIPHERSUITES deprecated in 3.0.0
14  * Update both macro and function simultaneously
15  */
16 const char *OSSL_default_ciphersuites(void)
17 {
```

```
18      return "TLS_AES_256_GCM_SHA384:"
19             "TLS_CHACHA20_POLY1305_SHA256:"
20             "TLS_AES_128_GCM_SHA256";
21  }
```

Listing 4.6: OpenSSL implementation of TLSv1.2 and TLSv1.3 default ciphersuites list

For TLSv1.2, the ciphersuites are defined by `ALL:!COMPLEMENTOFDEFAULT:!eNULL`. According to the OpenSSL documentation [109], this means that the list for TLSv1.2 will contain all ciphersuites that are part of the default set and use encryption, which also includes the weak ciphersuites listed above. On the other hand, the default list for TLSv1.3 is fairly straightforward with three ciphersuites, which are used by Open5gs in Table 4.3.

Over half of the supported ciphersuites are considered weak ciphersuites and without ciphersuites ordering, there is no guarantee that the strongest ciphersuite will be negotiated first. Therefore, the connection may use a weaker ciphersuite, which may allow adversaries to leverage this weakness to gain access to sensitive information and possibly set the stage for a more serious attack. As a result, the usage of weak ciphers and the lack of support for ciphersuites ordering could negatively affect the security of the communication between the network functions.

4.2.2.3 Common bugs and vulnerabilities

As indicated by the TLS-Scanner scan results, the open-source Open5gs core network is resistant to a number of bugs and vulnerabilities. This is a good indication that the TLS configuration for the connection between network functions of Open5gs is secure as it eliminates an extensive list of known attacks while reducing the attack vectors significantly due to the lack of bugs.

4.3 Results from the scripts

As mentioned in the Methodology chapter, we created a script in order to test the open-source 5G core networks for 3GPP compliance. We also extended the capabilities of the script to output the bugs and vulnerabilities that the TLS-Scanner tests for. In order to present the results of the compliance script in a structured manner, we constructed a table which, for each TLS configuration, specifies the accepted implementations, the missing implementations and the extra implementations.

4.3.1 Results of compliance check

4.3.1.1 free5GC

The results of the compliance check for free5GC are shown in Table 4.4.

We can see that free5GC is not fully compliant with version 17 of the 3GPP requirements. However, it is compliant with one part of the requirements, where it implements all specified key exchange curves as well as an additional curve `SECP521R1`.

First, the Extra Support row of the TLS versions in Table 4.4 shows that free5GC accepts the TLS versions TLSv1.0 and TLSv1.1 while the requirements state that these versions should not

Table 4.4: Results of free5GC compliance check

TLS versions	**Accepted**	v1.2 v1.3
	Extra support	v1.0 × v1.1 ×
Ciphersuites	**Accepted**	TLS_AES_128_GCM_SHA256: MUST HAVE TLS_RSA_WITH_AES_128_CBC_SHA: MUST HAVE TLS_CHACHA20_POLY1305_SHA256: SHOULD HAVE TLS_AES_256_GCM_SHA384: SHOULD HAVE TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384: RECOMMENDED
	Missing	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256: MUST HAVE TLS_DHE_RSA_WITH_AES_128_GCM_SHA256: MUST HAVE TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384: RECOMMENDED TLS_DHE_PSK_WITH_AES_128_GCM_SHA256: MUST IF PSK TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256: MUST IF PSK TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384: RECOMMENDED IF PSK
	Extra Support	TLS_RSA_WITH_AES_256_CBC_SHA ⊗ TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA ⊗ TLS_RSA_WITH_AES_128_GCM_SHA256 ⊗ TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ⊗ TLS_RSA_WITH_AES_256_GCM_SHA384 ⊗ TLS_RSA_WITH_3DES_EDE_CBC_SHA ⊗ TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA ⊗ TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
Extensions	**Accepted**	SESSION_TICKET SUPPORTED_VERSIONS KEY_SHARE RENEGOTIATION_INFO
	Missing	SIGNATURE_ALGORITHMS SUPPORTED_GROUPS STATUS_REQUEST SIGNATURE_ALGORITHMS_CERT EXTENDED_MASTER_SECRET
	Extra Support	ALPN EC_POINT_FORMATS
Key exchange	**Accepted**	SECP256R1: MUST SUPPORT ECDH_X25519: SHOULD SUPPORT SECP384R1: SHOULD SUPPORT
	Missing	
	Extra Support	SECP521R1
Signature Hash	**Accepted**	RSA_PSS_RSAE_SHA256 RSA_PSS_RSAE_SHA384
	Missing	RSA_PKCS1_SHA256 RSA_PKCS1_SHA384 ECDSA_SHA256 ECDSA_SHA384 ECDSA_SECP384R1_SHA384
	Extra Support	RSA_SHA384 RSA_SHA1 × RSA_SHA256 RSA_SHA512 RSA_PSS_RSAE_SHA512
Bugs		HAS_VERSION_INTOLERANCE: TRUE
Vulnerabilities		VULNERABLE_TO_SWEET_32: TRUE

⊗: Weak ciphers

×: Should not be supported

be supported. TLSv1.0 and TLSv1.1 are vulnerable to a plethora of attacks, which we have explained in the Background chapter and the support of these versions will negatively affect the security of free5GC.

Second, while free5GC does implement the majority of the ciphersuites listed by the 3GPP requirements, it is missing two mandatory to implement ciphersuites which are:

`TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
`TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`

Furthermore, free5GC implements additional ciphersuites that include a mix of strong and weak ciphersuites, which can be seen in the Extra Support row of the Ciphersuites section of Table 4.4. While the inclusion of weak ciphersuites is not ideal in terms of security, the extra secure ciphersuites, `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` and `TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256` are an added bonus. Since free5GC also supports ciphersuite ordering, where the strong ciphersuites are negotiated first, the additional secure ciphersuites could increase the chances of stronger ciphersuites being used in the communication between the network functions.

Third, the TLS extensions implemented in free5GC are incomplete. The core network only supports three out of the nine mandatory TLS extensions. The missing extensions, such as `EXTENDED_MASTER_SECRET` and `STATUS_REQUEST` are essential to bolster the core network's ability to defend against TLS-targeted attacks. Furthermore, freee5gc has implementations for additional extensions that increase the efficiency and security of the TLS connection such as `EC_POINT_FORMATS` and `ALPN`.

Finally, free5GC does not support the majority of the signature hash algorithms listed in the 3GPP requirements. The core network only supports the algorithms `RSA_PSS_RSAE_SHA256` and `RSA_PSS_RSAE_SHA384`. However, it implements an extra signature hash algorithm that is recommended by RFC 8446, `RSA_PSS_RSAE_SHA512`. On the other hand, free5GC also supports `RSA_SHA1` which could be a security risk as RSA and SHA1 are not secure.

4.3.1.2 Open5gs

The results of the compliance check for the open-source Open5gs are shown in Table 4.5.

Similar to free5GC, Open5gs does not fully comply with 3GPP version 17. However, it does satisfy the TLS version requirements, only supporting TLSv1.2 and TLSv1.3. Furthermore, Open5gs implements all required key exchange curves while also adding support for the `ECDH_X448` and `SECP521R1` curves. The satisfaction of these requirements mitigates attack risks and strengthens the security of Open5gs.

First, Open5gs supports most of the ciphersuites that are compulsory to implement with the exception of the mandatory ciphersuites `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` and `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`. The core network also implements additional weak and strong ciphersuites. Due to the lack of ciphersuites ordering, a weaker ciphersuite could be used to secure the communication between network functions. This may expose Open5gs to attacks that target weak ciphersuites such as POODLE, Bleichenbacher and other

Table 4.5: Results of Open5gs compliance check

TLS versions	**Accepted**	v1.2 v1.3
	Extra support	
Ciphersuites	**Accepted**	TLS_AES_128_GCM_SHA256: MUST HAVE TLS_CHACHA20_POLY1305_SHA256: SHOULD HAVE TLS_AES_256_GCM_SHA384: SHOULD HAVE TLS_RSA_WITH_AES_128_CBC_SHA: MUST HAVE TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384: RECOMMENDED
	Missing	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256: MUST HAVE TLS_DHE_RSA_WITH_AES_128_GCM_SHA256: MUST HAVE TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384: RECOMMENDED TLS_DHE_PSK_WITH_AES_128_GCM_SHA256: MUST IF PSK TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256: MUST IF PSK TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384: RECOMMENDED IF PSK
	Extra Support	TLS_RSA_WITH_AES_256_CBC_SHA ⊗ TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA ⊗ TLS_RSA_WITH_AES_128_GCM_SHA256 ⊗ TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ⊗ TLS_RSA_WITH_AES_256_GCM_SHA384 ⊗ TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
Extensions	**Accepted**	EXTENDED_MASTER_SECRET SESSION_TICKET KEY_SHARE RENEGOTIATION_INFO SUPPORTED_VERSIONS
	Missing	SIGNATURE_ALGORITHMS SUPPORTED_GROUPS STATUS_REQUEST SIGNATURE_ALGORITHMS_CERT
	Extra Support	ENCRYPT_THEN_MAC ALPN ELLIPTIC_CURVES MAX_FRAGMENT_LENGTH
Key exchange	**Accepted**	SECP256R1: MUST SUPPORT ECDH_X25519: SHOULD SUPPORT SECP384R1: SHOULD SUPPORT
	Missing	
	Extra Support	SECP521R1 ECDH_X448
Signature Hash	**Accepted**	RSA_PSS_RSAE_SHA256 RSA_PSS_RSAE_SHA384
	Missing	RSA_PKCS1_SHA256 RSA_PKCS1_SHA384 ECDSA_SHA256 ECDSA_SHA384 ECDSA_SECP384R1_SHA384
	Extra Support	RSA_SHA384 RSA_SHA224 RSA_SHA256 RSA_SHA512 RSA_PSS_RSAE_SHA512
Bugs		
Vulnerabilities		

⊗: Weak ciphers

×: Should not be supported

padding oracle attacks.

Second, Open5gs is missing a number of extensions specified by the 3GPP requirements which may negatively affect the security and performance of the TLS connection. For instance, the OSCP status request extension, `STATUS_REQUEST` is not supported. This extension is required so that it reduces the risk of a man-in-the-middle attack as the server may use this extension to make sure that the client's certificate is valid. Furthermore, Open5gs implements extra extensions that may benefit the security and performance of the TLS connection. For example, the support of the `ENCRYPT_THEN_MAC` extension is a boost in security for the TLS connections between network functions. Traditionally, a message authentication code (MAC) is generated based on the plaintext before being encrypted together with the plaintext. However, this allowed attackers to carry out padding oracle attacks. The `ENCRYPT_THEN_MAC` extension eliminates the threat entirely by ensuring that the plaintext is encrypted and only then does the MAC get generated using the ciphertext.

Finally, similar to free5GC, Open5gs is missing a majority of the required signature hash algorithms. It only supports `RSA_PSS_RSAE_SHA256` and `RSA_PSS_RSAE_SHA384`, which are strong algorithms and are recommended for use in TLS connections that require a high level of security [74]. In addition, Open5gs supports an additional number of signature hash algorithms, including `RSA_PSS_RSAE_SHA512` which is considered to be more secure than `RSA_PSS_RSAE_SHA256` and `RSA_PSS_RSAE_SHA384` due to the fact that it uses the SHA-512, which is highly resistant to collision and pre-image attacks [74] Unlike free5GC, Open5gs does not support signature hash algorithms that utilize the SHA-1 hash, eliminating attacks that target this outdated hash in Open5gs.

From the TLS-Scanner scan results for free5GC and Open5gs, we can conclude that Open5gs is more secure than free5GC as it is more resistant to TLS vulnerabilities and bugs while supporting the latest TLS versions. According to the compliance check, Open5gs satisfies the versions and the key exchange algorithms listed in the 3GPP requirements, however, it does not follow the other specified implementations. Meanwhile, free5GC only satisfies the key exchange algorithms requirement and does not comply with the other specifications.

4.3.2 Certificates check

While going through the source code of the core networks, we came across a folder that contained the network functions' default TLS private keys and certificates [110][111]. This is a serious vulnerability if the core networks are deployed in a company setting and the default TLS private keys and certificates are not changed. Since the keys and certificates are stored in a public repository, anyone can access and see them. This means that a malicious user can use these default private keys to decrypt all communications between the network functions and gain access to sensitive information. Therefore, we developed another script that checks whether the TLS private keys and certificates are used in a deployment, based on the TLS-Scanner scan results. The script also assesses if the certificates were generated by the default private keys. The code for this script is attached in Appendix A, Listing 5.2.
The script takes three inputs: the 5G instance, the TLS-Scanner scan results and the current directory. In addition, the script requires two directories containing free5GC's and Open5gs' default TLS private keys and certificates, respectively. After execution, the script will output a

JSON file, specifying whether the certificates and private keys that a deployment uses were found in the core networks' public repositories.

Using the TLS-Scanner results, the script will extract the certificates and compare them with the default certificates stored in the directory. If a match is found, then the parameter `CERT_FOUND_IN_PUBLIC_REPO` in the output file will be updated with the name of the certificate file in the public repository. This check is done in the function `check_server_cert`, shown in Listing 4.7.

```
    def check_server_cert(input_file_name,files_list,dirname):
    """
    Checks if given server certificate is in the public
    repository of open-source 5G instance.

    :param input_file_name: name of json input file
    :param files_list: list of network functions' crt and key
                       files taken from open-source 5G instance repo
    :param dirname: directory where crt and key files are stored
    :return updates cert_key_report if certificate is found in public
    repo
    """
    file_json = open(input_file_name, encoding='utf-8')
    json_data = json.load(file_json)
    file_json.close()
    for filename in files_list:
        file = open(os.getcwd() + "/" + dirname + "/" + filename,
    encoding='utf-8')
        contents = file.read()
        file.close()
        if 'crt' in filename or 'pem' in filename:
            public_crt = ssl.PEM_cert_to_DER_cert(contents).hex()
        cert = json_data['certificateChainList'][0]['certificate']['
    certificates']
        if cert[0].lower() == public_crt:
            cert_key_report['CERT_FOUND_IN_PUBLIC_REPO']= filename
    return cert_key_report
```

Listing 4.7: Check server certificate function

After the certificate has been discovered in the public repository, the script will attempt to check whether the certificate was generated using the default private key using the function `check_public_private_key`, shown in Listing 4.8. In order to do this, we extracted the public key from the certificate and for each private key in the public repository, we generated a corresponding public key and compared this key with the public key extracted from the certificate. This is because every public-private key pair are unique, that is, every public key matches only to one private key [112]. Therefore, if the public key generated using the private key matches the public key extracted from the certificate, then we know that the certificate was generated using that private key. Once there is a match, the parameter `PRIVATE_KEY_FOUND_IN_PUBLIC_REPO` in the output file will be updated with the name of the private key file in the public repository.

```
    def check_public_private_key(cert_file,list_files,dirname):
    """
    Checks if given server certificate has matching private key
    in public repository of open-source 5g instance

```

```
6       :param cert_file: name of certificate file
7       :param files_list: list of network functions' crt and key
8                          files taken from open-source 5G instance repo
9       :param dirname: directory where crt and key files are stored
10      :return updates cert_key_report if private key is found in public
        repo
11      """
12      crt_file = open(os.getcwd() + "/" + dirname + "/" + cert_file,
        encoding='utf-8')
13      cert_str = crt_file.read()
14      crt_file.close()
15      certificate = load_certificate(FILETYPE_PEM,cert_str)
16      public_cert_key = dump_publickey(FILETYPE_PEM,certificate.get_pubkey
        ())
17
18      for filename in list_files:
19          if 'key' in filename:
20              priv_key_file = open(os.getcwd() + "/" + dirname + "/" +
        filename, encoding='utf-8')
21              priv_key_str = priv_key_file.read()
22              priv_key_file.close()
23              private_key = load_privatekey(FILETYPE_PEM,priv_key_str)
24              priv_pub_key = dump_publickey(FILETYPE_PEM,private_key)
25              if public_cert_key == priv_pub_key:
26                  cert_key_report['PRIVATE_KEY_FOUND_IN_PUBLIC_REPO'] =
        filename
27      return cert_key_report
```

Listing 4.8: Check private key function

Executing the script on our local Open5gs and free5GC instance, we found that they both use the default TLS private keys and certificates. Since they are local deployments used for research, they are not exposed to the Internet and therefore, they are less likely to be attacked by malicious users. However, as mentioned in the introduction of this section, if the core networks are deployed in a company setting, connecting many devices over the internet, the use of default TLS private keys and certificates will pose a serious security vulnerability.

Chapter 5

Conclusion

During the research, we analysed different open-source 5G core networks, namely Open5gs, free5GC and OAI 5G CN - to identify TLS vulnerabilities and assess their compliance with the 3GPP requirements. We discovered that free5GC and Open5gs have support for TLS in the communication between network functions while OAI 5G CN does not. This is a critical concern as the lack of support for TLS in OAI 5G CN may make the core network vulnerable to a number of exploits.

We used PySSLScan, SSLze and TLS-Scanner in our analysis for TLS vulnerabilities and bugs. We found that free5GC suffered from the version intolerance bug and Sweet32 vulnerability. In addition, the core network also implemented several weak ciphersuites. These weaknesses are concerning as they may allow attackers to access sensitive information such as user data, subscriber information and NFs' configurations. On the other hand, while Open5gs did not have any bugs or vulnerabilities, it does support many weak ciphersuites which could also be exploited by malicious users.

The compliance analysis carried out using our Python script revealed that neither free5GC nor Open5gs completely follow the 3GPP requirements. Free5GC only complies with the key exchange algorithms required while failing the other specifications. Open5gs also complies with the key exchange algorithms requirement and in addition, satisfies the TLS version specifications, but like free5GC, Open5gs did not comply with the other requirements. On the other hand, OAI 5G CN, which does not support TLS within its core network despite TLS being mandatory-to-implement, completely fails to meet the 3GPP requirements. This is a significant security issue as non-compliance with the 3GPP standards can increase the attack vectors. Therefore, it is crucial for open-source 5G core networks to strictly follow the 3GPP requirements in order to increase the security strengths of the system while making it more difficult for attackers to exploit vulnerabilities in the core network.

Furthermore, while carrying out the research, we found that our local instances of free5GC and Open5gs were using the default TLS private keys and certificates, which can be found in the public repository of the core networks. This is a significant finding because if these core networks were deployed in a company setting, an attacker could use the private keys from the public repository to effortlessly break the encryption in the communication between the network functions, resulting in sensitive information being leaked.

Our research not only highlights the importance of following the 3GPP standards but also emphasizes the need for proper implementations of TLS configurations in order to increase the security strengths within the open-source 5G core networks while decreasing the risk of exploitation by malicious users. In addition, it is crucial to ensure that the TLS private keys and certificates are not stored in the public repository, but perhaps generated and stored privately when a user installs these core networks or issues a warning when the core networks recognise that the default TLS private keys and certificates are being used. The findings of this study could help future and current developers to enhance the security of their core networks' implementations.

A limitation of our research lies in the number of open-source 5G core networks that we analysed. In this study, we explored only three open-source 5G core networks, namely free5GC, Open5gs and OAI 5G CN. However, this number was further reduced to two due to the fact that OAI 5G CN did not support TLS in the communication between network functions. In addition, we analysed version 3.2.1, version 2.6.0 and version v1.4.0 of free5GC, Open5gs and OAI 5G CN respectively. Being open-sourced projects, these core networks are constantly being updated and patched, therefore the results of our study may not be generalizable to future versions of free5GC, Open5gs and OAI 5G CN or to other open-source 5G core networks.

Another limitation of our research is that we used automated tools to scan the core networks for their TLS configurations as well as for any vulnerabilities and bugs that they may suffer from. While these tools proved to be useful, they only test for a limited number of known vulnerabilities and bugs. This means that the core networks could have potentially been vulnerable to other weaknesses that are outside the scanning capabilities of the tools.

The last limitation of this study is that we only conducted a server-side scanning of the network functions. Using the tools, we were able to simulate a service consumer NF i.e the client, connecting over TLS to a service producer NF i.e the server, in order to scan the server-side TLS configurations. While this enabled us to identify certain TLS misconfigurations and vulnerabilities on the server side, the research did not include client-side scanning i.e simulating a server and having clients connect to it to test the TLS configurations of a service consumer NF. Incorporating client-side scanning into our research would have allowed us to assess the TLS configurations of both the client and the server and check for discrepancies between the two.

Future works following this research could focus on addressing the above-mentioned limitations. For example, as an extension of this research, client-side scanning should be performed in order to provide a more complete picture of the TLS security of these open-source 5G core networks. Furthermore, other open-source 5G core networks as well as future versions of free5GC, Open5gs and OAI 5G CN could be analysed in order to give a more thorough understanding of the state of TLS security in the networks. This can be done using more advanced tools that test for a wider range of vulnerabilities and bugs. In addition, future research could conduct a comparison of TLS implementations between open-source 5G core networks and commercialized 5G core networks to give an insight into the effectiveness of different approaches to implementing TLS in 5G core networks.

Bibliography

[1] *Verizon sets roadmap to 5G technology in U.S.; Field trials to start in 2016*, en, Sep. 2015. [Online]. Available: https://www.verizon.com/about/news/verizon-sets-roadmap-5g-technology-us-field-trials-start-2016.

[2] *T-Mobile 5G: It's On! America's First Nationwide 5G Network Is Here - T-Mobile Newsroom*, en-US, Dec. 2019. [Online]. Available: https://www.t-mobile.com/news/press/americas-first-nationwide-5g-network/.

[3] M. Agiwal, A. Roy, and N. Saxena, "Next Generation 5G Wireless Networks: A Comprehensive Survey", *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, 2016, ISSN: 1553-877X. DOI: 10.1109/COMST.2016.2532458.

[4] ITU-R, *IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond*, Sep. 2015.

[5] 3GPP, *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; 5G; Release description; Release 15 (3GPP TR 21.915 version 15.0.0 Release 15)*, Oct. 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_tr/121900_121999/121915/15.00.00_60/tr_121915v150000p.pdf.

[6] M. Kapko, *Operators, Vendors Scramble for Standalone 5G Cores*, en-US, Jul. 2020. [Online]. Available: https://www.sdxcentral.com/articles/news/operators-vendors-scramble-for-standalone-5g-cores/2020/07/.

[7] M. Dano, *Here's how much a 5G wireless network really costs*, en, Apr. 2021. [Online]. Available: https://www.lightreading.com/open-ran/heres-how-much-5g-wireless-network-really-costs/d/d-id/769114.

[8] *Open5gs*, en-US. [Online]. Available: https://open5gs.org/.

[9] *free5GC*, en. [Online]. Available: https://www.free5gc.org/.

[10] *OpenAirInterface – 5G software alliance for democratising wireless innovation*, en-US. [Online]. Available: https://openairinterface.org/.

[11] S. Dudek, *Intruding 5G SA core networks from outside and inside | PentHertz Blog*, Dec. 2021. [Online]. Available: https://penthertz.com/blog/Intruding-5G-core-networks-from-outside-and_inside.html.

[12] N. Muley, *The advantages of 5G service-based architecture (SBA)*, en-US, Mar. 2021. [Online]. Available: https://www.alepo.com/the-advantages-of-5g-service-based-architecture-sba/.

[13] M. Belshe, R. Peon, and M. Thomson, *RFC7540*, en, May 2015. [Online]. Available: https://httpwg.org/specs/rfc7540.html.

[14] C. Allen and T. Dierks, "The TLS Protocol Version 1.0", Internet Engineering Task Force, Request for Comments RFC 2246, Jan. 1999, Num Pages: 80. DOI: 10.17487/RFC2246. [Online]. Available: https://datatracker.ietf.org/doc/rfc2246.

[15] Cloudflare, *Why use TLS 1.3? | SSL and TLS vulnerabilities | Cloudflare*. [Online]. Available: https://www.cloudflare.com/learning/ssl/why-use-tls-1.3/.

[16] P. Technologies, *5G security issues*, Nov. 2019. [Online]. Available: https://www.gsma.com/membership/wp-content/uploads/2019/11/5G-Research_A4.pdf.

[17] A. Shaik, R. Borgaonkar, S. Park, and J.-P. Seifert, "New vulnerabilities in 4G and 5G cellular access network protocols: Exposing device capabilities", in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '19, New York, NY, USA: Association for Computing Machinery, May 2019, pp. 221–231, ISBN: 978-1-4503-6726-4. DOI: 10.1145/3317549.3319728. [Online]. Available: https://doi.org/10.1145/3317549.3319728.

[18] I. Ahmad, T. Kumar, M. Liyanage, J. Okwuibe, M. Ylianttila, and A. Gurtov, "5G security: Analysis of threats and solutions", in *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*, Sep. 2017, pp. 193–199. DOI: 10.1109/CSCN.2017.8088621.

[19] F. J. De Souza Neto, E. Amatucci, N. A. Nassif, and P. A. Marques Farias, "Analysis for Comparison of Framework for 5G Core Implementation", in *2021 International Conference on Information Science and Communications Technologies (ICISCT)*, Nov. 2021, pp. 1–5. DOI: 10.1109/ICISCT52966.2021.9670414.

[20] X. Hu, C. Liu, S. Liu, W. You, and Y. Zhao, "Signalling Security Analysis: Is HTTP/2 Secure in 5G Core Network?", in *2018 10th International Conference on Wireless Communications and Signal Processing (WCSP)*, ISSN: 2472-7628, Oct. 2018, pp. 1–6. DOI: 10.1109/WCSP.2018.8555612.

[21] N. Bhandari, S. Devra, and K. Singh, *Evolution of Cellular Network: From 1G to 5G*, Oct. 2017. [Online]. Available: https://oaji.net/articles/2017/1992-1515158039.pdf.

[22] O. Eluwole, N. Udoh, M. Ojo, C. Okoro, and A. Akinyoade, *From 1G to 5G, What Next?*, Dec. 2021. [Online]. Available: https://www.iaeng.org/IJCS/issues_v45/issue_3/IJCS_45_3_06.pdf.

[23] M. Toorani and A. A. Beheshti, *Solutions to the GSM Security Weaknesses*, Mar. 2012. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1002/1002.3175.pdf.

[24] M. Arapinis, L. Mancini, E. Ritter, *et al.*, *New Privacy Issues in Mobile Telephony: Fix and Verification*, Aug. 2012. [Online]. Available: https://www2.seas.gwu.edu/~cheng/6547/Readings/p205-arapinis.pdf.

[25] Rajiv, *Evolution of wireless technologies 1G to 5G in mobile communication*, en-US, May 2018. [Online]. Available: https://www.rfpage.com/evolution-of-wireless-technologies-1g-to-5g-in-mobile-communication/ (visited on 02/08/2023).

[26] A. Shaik, R. Borgaonkar, N. Asokan, V. Niemi, and J.-P. Seifert, *Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems*, Aug. 2017. [Online]. Available: https://arxiv.org/pdf/1510.07563.pdf.

[27] *Making 5G NR a reality: Leading the technology inventions for a unified, more capable 5G air interface*, Dec. 2016.

[28] *Control and User Plane Separation of EPC nodes (CUPS)*, Jul. 2017. [Online]. Available: https://www.3gpp.org/news-events/3gpp-news/cups.

[29] Y.-B. Lin, C.-C. Tseng, and M.-H. Wang, "Effects of Transport Network Slicing on 5G Applications", *Future Internet*, vol. 13, p. 69, Mar. 2021. DOI: 10.3390/fi13030069.

[30] *5G Network Slicing, What is it? | 5G Slicing Architecture and Solutions*, eng, Last Modified: 2022-11-01, Sep. 2019. [Online]. Available: https://www.viavisolutions.com/en-us/5g-network-slicing.

[31] N. Singh, *What is the 5G Access and Mobility Management Function (AMF)?*, en, Section: Azure for Operators Blog, Jan. 2023. [Online]. Available: https://techcommunity.microsoft.com/t5/azure-for-operators-blog/what-is-the-5g-access-and-mobility-management-function-amf/ba-p/3707685.

[32] *Exploring the 3GPP AMF – Access & Mobility Management Function*, en-gb, Sep. 2022. [Online]. Available: https://emblasoft.com/blog/exploring-the-3gpp-amf-access-mobility-management-function.

[33] *5G; Security architecture and procedures for 5G System (3GPP TS 33.501 version 16.3.0 Release 16)*, Aug. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/133500_133599/133501/16.03.00_60/ts_133501v160300p.pdf.

[34] *Authentication and Key Management for Applications (AKMA) in 5G*, Dec. 2022. [Online]. Available: https://www.3gpp.org/technologies/akma.

[35] *5G; 5G System; Network function repository services; Stage 3 (3GPP TS 29.510 version 15.3.0 Release 15)*, Apr. 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/129500_129599/129510/15.03.00_60/ts_129510v150300p.pdf.

[36] N. Singh, *What is the 5G Session Management Function (SMF)?*, en, Section: Azure for Operators Blog, Dec. 2022. [Online]. Available: https://techcommunity.microsoft.com/t5/azure-for-operators-blog/what-is-the-5g-session-management-function-smf/ba-p/3693852.

[37] P. Malushte, *5G policy control and charging functions | 5G PCF CHF*, en-US, Mar. 2022. [Online]. Available: https://www.alepo.com/how-policy-control-and-charging-network-functions-harness-5g-potential/.

[38] 3GPP, *5G; 5G System; Policy and Charging Control signalling flows and QoS parameter mapping; Stage 3 (3GPP TS 29.513 version 16.9.0 Release 16)*, Sep. 2021. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/129500_129599/129513/16.09.00_60/ts_129513v160900p.pdf.

[39] 3GPP, *5G; 5G System; Network Slice Selection Services; Stage 3 (3GPP TS 29.531 version 16.3.0 Release 16)*, Jul. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/129500_129599/129531/16.03.00_60/ts_129531v160300p.pdf.

[40] 3GPP, *5G; 5G System; Unified Data Management Services; Stage 3 (3GPP TS 29.503 version 17.6.0 Release 17)*, May 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/129500_129599/129503/17.06.00_60/ts_129503v170600p.pdf.

[41] 3GPP, *5G; 5G System; Technical Realization of Service Based Architecture; Stage 3 (3GPP TS 29.500 version 16.5.0 Release 16)*, Nov. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/129500_129599/129500/16.05.00_60/ts_129500v160500p.pdf.

[42] *What is SSL/TLS Encryption?*, en-US. [Online]. Available: https://www.f5.com/glossary/ssl-tls-encryption.

[43] C. Kemmerer, *The SSL/TLS Handshake: An Overview*, en-US, May 2015. [Online]. Available: https://www.ssl.com/article/ssl-tls-handshake-overview/.

[44] *What is TLS & How Does it Work? | ISOC Internet Society*, en-US, Aug. 2022. [Online]. Available: https://www.internetsociety.org/deploy360/tls/basics/.

[45] *Symmetric vs. Asymmetric Encryption - What are differences?*, en-US. [Online]. Available: https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences (visited on 03/24/2023).

[46] E. Rescorla and T. Dierks, "The Transport Layer Security (TLS) Protocol Version 1.2", Internet Engineering Task Force, Request for Comments RFC 5246, Aug. 2008, Num Pages: 104. DOI: 10.17487/RFC5246. [Online]. Available: https://datatracker.ietf.org/doc/rfc5246.

[47] W. Shbair, "Service-Level Monitoring of HTTPS Traffic", Ph.D. dissertation, May 2017.

[48] S. Wang, *The difference in five modes in the AES encryption algorithm - Highgo Software Inc.* en-US, Aug. 2019. [Online]. Available: https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/.

[49] T. Duong and J. Rizzo, "Here Come The xor Ninjas", Buenos Aires, Argentina, May 2011. [Online]. Available: https://www.semanticscholar.org/paper/Here-Come-The-%E2%8A%95-Ninjas-Thai-Duong-Rizzo/64464198f4e6c10611cfb7dfe26bbb7ca4ddd344.

[50] *File:CBC encryption.svg - Wikipedia*, en, Jun. 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:CBC_encryption.svg (visited on 03/24/2023).

[51] K. G. Paterson and A. Yau, *Padding Oracle Attacks on the ISO CBC Mode Encryption Standard*, 2004. [Online]. Available: https://www.isg.rhul.ac.uk/~kp/padding.pdf.

[52] T. Duong, B. Möller, and K. Kotowicz, *This POODLE Bites: Exploiting The SSL 3.0 Fallback*, Sep. 2014. [Online]. Available: https://www.openssl.org/~bodo/ssl-poodle.pdf.

[53] N. J. Al Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols", en, in *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA: IEEE, May 2013, pp. 526–540, ISBN: 978-0-7695-4977-4 978-1-4673-6166-8. DOI: 10.1109/SP.2013.42. [Online]. Available: http://ieeexplore.ieee.org/document/6547131/.

[54] T. Jager, S. Schinzel, and J. Somorovsky, *Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption*, Dec. 2012. [Online]. Available: https://nds.ruhr-uni-bochum.de/media/nds/veroeffentlichungen/2012/12/19/XMLencBleichenbacher.pdf.

[55] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, *A Survey on Homomorphic Encryption Schemes:Theory and Implementation*, Oct. 2017. [Online]. Available: https://arxiv.org/pdf/1704.03578.pdf.

[56] R. Merget, M. Brinkmann, N. Aviram, J. Somorovsky, J. Mittmann, and J. Schwenk, *Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)*, 2020. [Online]. Available: https://raccoon-attack.com/RacoonAttack.pdf.

[57] A. Balasinor, *SSL/TLS attacks: Part 2 – CRIME Attack*, en-US, Dec. 2013. [Online]. Available: https://niiconsulting.com/checkmate/2013/12/ssltls-attacks-part-2-crime-attack/.

[58] Rorot, *The breach attack*, en-US, Oct. 2013. [Online]. Available: https://resources.infosecinstitute.com/topic/the-breach-attack/.

[59] T. Jager, J. Schwenk, and J. Somorovsky, "Practical Invalid Curve Attacks on TLS-ECDH", en, in *Computer Security – ESORICS 2015*, G. Pernul, P. Y A Ryan, and E. Weippl, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 407–425, ISBN: 978-3-319-24174-6. DOI: 10.1007/978-3-319-24174-6_21.

[60] D. Adrian, K. Bhargavan, Z. Durumeric, *et al.*, *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*, Aug. 2015. [Online]. Available: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf.

[61] K. Bhargavan and G. Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 456–467, ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978423. [Online]. Available: https://doi.org/10.1145/2976749.2978423.

[62] N. Aviram, S. Schinzel, J. Somorovsky, *et al.*, *DROWN: Breaking TLS using SSLv2*, Aug. 2016. [Online]. Available: https://drownattack.com/drown-attack-paper.pdf.

[63] S. Koussa, *How to Confirm Whether You are Vulnerable to the DROWN Attack*, en-US, Mar. 2016. [Online]. Available: https://www.softwaresecured.com/how-to-confirm-whether-you-are-vulnerable-to-the-drown-attack/.

[64] M. Williams, M. Tüxen, and R. Seggelmann, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", Internet Engineering Task Force, Request for Comments RFC 6520, Feb. 2012, Num Pages: 9. DOI: 10.17487/RFC6520. [Online]. Available: https://datatracker.ietf.org/doc/rfc6520.

[65] B. Kiprin, *Heartbleed Bug - Definition, Explanation and Prevention*, en-GB, Running Time: 474 Section: Vulnerability Prevention, Apr. 2021. [Online]. Available: https://crashtest-security.com/prevent-heartbleed/.

[66] H. Sidhpurwala, *OpenSSL MITM CCS injection attack (CVE-2014-0224)*, en, Jun. 2014. [Online]. Available: https://www.redhat.com/en/blog/openssl-mitm-ccs-injection-attack-cve-2014-0224.

[67] *Gnutls.org*, en. [Online]. Available: https://www.gnutls.org/.

[68] *CVE-2020-13777- Red Hat Customer Portal*, Jun. 2020. [Online]. Available: https://access.redhat.com/security/cve/cve-2020-13777.

[69] M. Brinkmann, C. Dresen, R. Merget, *et al.*, *ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication*, Jun. 2021. [Online]. Available: https://alpaca-attack.com/ALPACA.pdf.

[70] *Renegotiation*, en-US, Aug. 2021. [Online]. Available: https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/i/7.1?topic=properties-renegotiation.

[71] B. Kiprin, *What Is the SSL Renegotiation Vulnerability?*, en-GB, Running Time: 417 Section: Vulnerability Prevention, Apr. 2021. [Online]. Available: https://crashtest-security.com/secure-client-initiated-ssl-renegotiation/.

[72] K. Moriarty and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", Internet Engineering Task Force, Request for Comments RFC 8996, Mar. 2021, Num Pages: 18. DOI: 10.17487/RFC8996. [Online]. Available: https://datatracker.ietf.org/doc/rfc8996.

[73] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The First Collision for Full SHA-1", en, in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds., vol. 10401, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 570–596, ISBN: 978-3-319-63687-0 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_19. [Online]. Available: http://link.springer.com/10.1007/978-3-319-63688-7_19.

[74] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", Internet Engineering Task Force, Request for Comments RFC 8446, Aug. 2018, Num Pages: 160. DOI: 10.17487/RFC8446. [Online]. Available: https://datatracker.ietf.org/doc/rfc8446.

[75] S. Lee, Y. Shin, and J. Hur, "Return of version downgrade attack in the era of TLS 1.3", in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '20, New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 157–168, ISBN: 978-1-4503-7948-9. DOI: 10.1145/3386367.3431310. [Online]. Available: https://doi.org/10.1145/3386367.3431310.

[76] *About 3GPP*, en-us. [Online]. Available: https://www.3gpp.org/about-3gpp (visited on 03/15/2023).

[77] 3GPP, *5G; NR; Overall description; Stage-2 (3GPP TS 38.300 version 15.5.0 Release 15)*, May 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/138300_138399/138300/15.05.00_60/ts_138300v150500p.pdf.

[78] *5G System Overview*, Aug. 2022. [Online]. Available: https://www.3gpp.org/technologies/5g-system-overview.

[79] *Understanding the Risks of Open-Source Software*, Mar. 2022. [Online]. Available: https://www.contrastsecurity.com/hubfs/Understanding-the-Risks_WhitePaper_042020_Final.pdf?hsLang=en.

[80] 3GPP, *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; 5G; Network Domain Security (NDS); IP network layer security (3GPP TS 33.210 version 17.1.0 Release 17)*, Sep. 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/133200_133299/133210/17.01.00_60/ts_133210v170100p.pdf.

[81] *Docker overview*, en, Feb. 2023. [Online]. Available: https://docs.docker.com/get-started/overview/.

[82] *TLS-Scanner*, Feb. 2023. [Online]. Available: https://github.com/tls-attacker/TLS-Scanner.

[83] J. C. Villanueva, *An Introduction To Cipher Suites | JSCAPE*, en-us, Oct. 2022. [Online]. Available: https://www.jscape.com/blog/cipher-suites.

[84] I. Ristić, *OpenSSL Cookbook 3rd Edition - 2.9 Testing Named Groups*. [Online]. Available: https://www.feistyduck.com/library/openssl-cookbook/online/testing-with-openssl/testing-named-groups.html.

[85] *How TLS provides identification, authentication, confidentiality, and integrity*, en-US, Jan. 2023. [Online]. Available: https://ibm.com/docs/en/ibm-mq/9.1?topic=tls-how-provides-identification-authentication-confidentiality-integrity.

[86] *Ensure TLS Cipher Suite ordering is configured*, en. [Online]. Available: https://www.tenable.com/audits/items/CIS_v1.8_MS_IIS_7_Level_2.audit:ba7c9bdc0e43b69cd0a4cd0ab468dae1.

[87] M. Vojtko, *Perfect Forward Secrecy Explained*, en-US, Dec. 2020. [Online]. Available: https://www.thesslstore.com/blog/perfect-forward-secrecy-explained/.

[88] S. Galperin, C. Adams, M. Myers, R. Ankney, and A. N. Malpani, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", Internet Engineering Task Force, Request for Comments RFC 2560, Jun. 1999, Num Pages: 23. DOI: 10.17487/RFC2560. [Online]. Available: https://datatracker.ietf.org/doc/rfc2560.

[89] *Technical Guideline TR-02102-2 Cryptographic Mechanisms: Recommendations and Key Lengths*, 2023. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-2.pdf?__blob=publicationFile&v=5.

[90] K. A. McKay and D. A. Cooper, "Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations", in *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*, NIST, Aug. 2019. DOI: 10.6028/NIST.SP.800-52r2.

[91] A. Diquet, *SSLyze*, Sep. 2022. [Online]. Available: https://github.com/nabla-c0d3/sslyze.

[92] J. Vehent, *Security/Server Side TLS*. [Online]. Available: https://wiki.mozilla.org/Security/Server_Side_TLS.

[93] *pySSLScan*, Aug. 2022. [Online]. Available: https://github.com/DinoTools/pysslscan.

[94] V. Hilt, E. Noel, C. Shen, and A. Abdelal, "Design Considerations for Session Initiation Protocol (SIP) Overload Control", Internet Engineering Task Force, Request for Comments RFC 6357, Aug. 2011, Num Pages: 25. DOI: 10.17487/RFC6357. [Online]. Available: https://datatracker.ietf.org/doc/rfc6357.

[95] S. O. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", Internet Engineering Task Force, Request for Comments RFC 2119, Mar. 1997, Num Pages: 3. DOI: 10.17487/RFC2119. [Online]. Available: https://datatracker.ietf.org/doc/rfc2119.

[96] D. E. Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions", Internet Engineering Task Force, Request for Comments RFC 6066, Jan. 2011, Num Pages: 25. DOI: 10.17487/RFC6066. [Online]. Available: https://datatracker.ietf.org/doc/rfc6066.

[97] *Free5gc/nrf*, original-date: 2020-04-26T19:52:49Z, Feb. 2023. [Online]. Available: https://github.com/free5gc/nrf/blob/059f35e51383b8a419b3ebc6503ab1bbdcb2c77d/pkg/service/init.go.

[98] *Go http package*. [Online]. Available: https://pkg.go.dev/net/http#ListenAndServeTLS.

[99] *TLS on SBI Interfaces · Discussion #1102 · open5gs/open5gs*, en, Jul. 2021. [Online]. Available: https://github.com/open5gs/open5gs/discussions/1102.

[100] open5gs, *[UDR] Read framed routes from DB send them in sm-data · open5gs/open5gs@739cb59*, Jan. 2023. [Online]. Available: https://github.com/open5gs/open5gs/commit/739cb599d1998d4f87199eeada323358f162744d (visited on 03/23/2023).

[101] *Open5gs/nghttp2-server.c at main · open5gs/open5gs*. [Online]. Available: https://github.com/open5gs/open5gs/blob/v2.6.0/lib/sbi/nghttp2-server.c.

[102] *OpenSSL TLS Method documentation*. [Online]. Available: https://www.openssl.org/docs/man1.1.1/man3/TLSv1_2_method.html.

[103] *[SBI] HTTP2-TLS verification - ConfFile Changed · open5gs/open5gs@05fbaf6*, en. [Online]. Available: https://github.com/open5gs/open5gs/commit/05fbaf69587488e53b5e741a9ada9f9fa5749322.

[104] H. C. Rudolph, *Ciphersuite.info - a directory of TLS cipher suites*, original-date: 2017-05-09T20:50:22Z, Mar. 2023. [Online]. Available: https://github.com/hcrudolph/ciphersuite.info.

[105] *Tls package - crypto/tls - Go Packages*. [Online]. Available: https://pkg.go.dev/crypto/tls#Config.

[106] A. Delignat-Lavaud and K. Bhargavan, "Virtual Host Confusion: Weaknesses and Exploits", en, 2014. [Online]. Available: https://bh.ht.vc/vhost_confusion.pdf.

[107] */docs/man3.0/man3/SSL_ctx_new.html*. [Online]. Available: https://www.openssl.org/docs/man3.0/man3/SSL_CTX_new.html.

[108] *Welcome to the OpenSSL Project*, Mar. 2023. [Online]. Available: https://github.com/openssl/openssl/blob/08a11ba20461ce14b0a6b9c9e374fbea91fbd8cf/ssl/ssl_ciph.c.

[109] */docs/man1.1.1/man1/ciphers.html*. [Online]. Available: https://www.openssl.org/docs/man1.1.1/man1/ciphers.html.

[110] *Open5gs/configs/open5gs/tls at main · open5gs/open5gs*. [Online]. Available: https://github.com/open5gs/open5gs/tree/main/configs/open5gs/tls.

[111] *Free5gc/free5gc/config/TLS*, original-date: 2020-01-31T08:45:17Z, Mar. 2023. [Online]. Available: https://github.com/free5gc/free5gc.

[112] *Public and private encryption keys*, Dec. 2021. [Online]. Available: https://www.preveil.com/blog/public-and-private-key/.

Appendix A

```
"""The script compares 5G scan results for 3GPP compliance"""
import argparse
import json

valid_versions = {
    "TLS12": True,
    "TLS13": True,
}

valid_ciphers = {
    "TLS_AES_128_GCM_SHA256": "MUST HAVE",
    "TLS_AES_256_GCM_SHA384": "SHOULD HAVE",
    "TLS_CHACHA20_POLY1305_SHA256": "SHOULD HAVE",
    "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256": "MUST HAVE",
    "TLS_DHE_RSA_WITH_AES_128_GCM_SHA256": "MUST HAVE",
    "TLS_RSA_WITH_AES_128_CBC_SHA": "MUST HAVE",
    "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384": "RECOMMENDED",
    "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384": "RECOMMENDED",
    "TLS_DHE_PSK_WITH_AES_128_GCM_SHA256": "MUST IF PSK",
    "TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256": "MUST IF PSK",
    "TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384": "RECOMMENDED IF PSK"
}

weak_ciphers = {
    "TLS_RSA_WITH_AES_128_CBC_SHA": True,
    "TLS_RSA_WITH_AES_256_CBC_SHA": True,
    "TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA": True,
    "TLS_RSA_WITH_3DES_EDE_CBC_SHA": True,
    "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA": True,
    "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA": True,
    "TLS_RSA_WITH_AES_128_GCM_SHA256": True,
    "TLS_RSA_WITH_AES_256_GCM_SHA384": True
}

strong_ciphers = {
    "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256": True,
    "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384": True,
    "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256": True,
    "TLS_AES_128_GCM_SHA256": True,
    "TLS_CHACHA20_POLY1305_SHA256": True,
    "TLS_AES_256_GCM_SHA384": True
}

valid_extensions = {
    "SUPPORTED_VERSIONS": True,
    "KEY_SHARE": True,
```

```
    "SIGNATURE_ALGORITHMS": True,
    "SUPPORTED_GROUPS": True,
    "STATUS_REQUEST" : True,
    "SIGNATURE_ALGORITHMS_CERT": True,
    "EXTENDED_MASTER_SECRET": True,
    "SESSION_TICKET": True,
    "RENEGOTIATION_INFO": True,

}

valid_key_exchange = {
    "SECP256R1": "MUST SUPPORT",
    "SECP384R1": "SHOULD SUPPORT",
    "ECDH_X25519": "SHOULD SUPPORT"
}

valid_signature_hash = {
    "RSA_PSS_RSAE_SHA256": "MUST SUPPORT",
    "RSA_PSS_RSAE_SHA384": "SHOULD SUPPORT",
    "RSA_PKCS1_SHA256": "MUST SUPPORT",
    "RSA_PKCS1_SHA384": "SHOULD SUPPORT",
    "ECDSA_SHA256": "MUST SUPPORT",
    "ECDSA_SHA384": "SHOULD SUPPORT",
    "ECDSA_SECP384R1_SHA384": "SHOULD SUPPORT"
}

common_bugs = {
    "HAS_VERSION_INTOLERANCE": True,
    "HAS_CIPHER_SUITE_INTOLERANCE": True,
    "HAS_EXTENSION_INTOLERANCE": True,
    "HAS_CIPHER_SUITE_LENGTH_INTOLERANCE": True,
    "HAS_COMPRESSION_INTOLERANCE": True,
    "HAS_ALPN_INTOLERANCE": True,
    "HAS_CLIENT_HELLO_LENGTH_INTOLERANCE": True,
    "HAS_NAMED_GROUP_INTOLERANCE": True,
    "HAS_EMPTY_LAST_EXTENSION_INTOLERANCE": True,
    "HAS_SIG_HASH_ALGORITHM_INTOLERANCE": True,
    "HAS_BIG_CLIENT_HELLO_INTOLERANCE": True,
    "HAS_SECOND_CIPHER_SUITE_BYTE_BUG": True,
    "IGNORES_OFFERED_CIPHER_SUITES": True,
    "REFLECTS_OFFERED_CIPHER_SUITES":True,
    "IGNORES_OFFERED_NAMED_GROUPS": True,
    "IGNORES_OFFERED_SIG_HASH_ALGOS": True,
    "HAS_GREASE_NAMED_GROUP_INTOLERANCE": True,
    "HAS_GREASE_CIPHER_SUITE_INTOLERANCE": True,
    "HAS_GREASE_SIGNATURE_AND_HASH_ALGORITHM_INTOLERANCE": True
}

vulnerabilities = {
    "VULNERABLE_TO_PADDING_ORACLE": True,
    "VULNERABLE_TO_BLEICHENBACHER": True,
    "VULNERABLE_TO_RACCOON_ATTACK": True,
    "VULNERABLE_TO_DIRECT_RACCOON": True,
    "VULNERABLE_TO_CRIME": True,
    "VULNERABLE_TO_BREACH": True,
    "VULNERABLE_TO_INVALID_CURVE": True,
    "VULNERABLE_TO_INVALID_CURVE_EPHEMERAL": True,
    "VULNERABLE_TO_INVALID_CURVE_TWIST": True,
```

```
    "VULNERABLE_TO_POODLE": True,
    "VULNERABLE_TO_LOGJAM": True,
    "VULNERABLE_TO_SWEET_32": True,
    "VULNERABLE_TO_GENERAL_DROWN": True,
    "VULNERABLE_TO_EXTRA_CLEAR_DROWN": True,
    "VULNERABLE_TO_HEARTBLEED": True,
    "VULNERABLE_TO_EARLY_CSS": True,
    "VULNERABLE_TO_SESSION_TICKET_ZERO_KEY": True,
    "ALPACA_MITIGATED": True,
    "VULNERABLE_TO_RENEGOTIATION_ATTACK_EXTENSION_V1": True,
    "VULNERABLE_TO_RENEGOTIATION_ATTACK_EXTENSION_V2": True,
    "VULNERABLE_TO_RENEGOTIATION_ATTACK_CIPHERSUITE_V1": True,
    "VULNERABLE_TO_RENEGOTIATION_ATTACK_CIPHERSUITE_V2": True
}


out_report = {
                "accepted_versions": [],
                "supported_versions_but_should_not": [],
                "accepted_suites": [],
                "missing_suites": [],
                "supported_suites_but_should_not":
                {
                    "weak_ciphers": [],
                    "secure_ciphers": []
                },
                "accepted_extensions": [],
                "missing_extensions": [],
                "extra_extensions": [],
                "accepted_key_exchange": [],
                "missing_key_exchange": [],
                "extra_key_exchange": [],
                "accepted_signature_hash": [],
                "missing_signature_hash": [],
                "supported_signature_hash_but_should_not": [],
                "bugs": [],
                "vulnerabilities": []
            }

def get_version_specs(data_file):
    """
    Retrieves the versions of the network functions and compares this
    with the 3GPP requirements,
    updating out_report

    :param data_file: the network function json input file
    :return updates out_report
    """
    for ver in data_file['versions']:
        if ver in valid_versions:
            out_report["accepted_versions"].append(ver)
        else:
            out_report["supported_versions_but_should_not"].append(ver)
    return out_report

def get_ciphers_specs(final_out, data_file):
    """
    Retrieves the ciphersuites of the network functions and compares this
```

```
    with the 3GPP requirements,
    updating final_out

    :param data_file: the network function json input file
    :param final_out: the output dictionary
    :return updates final_out
    """
    for cipher_input in data_file['cipherSuites']:
        if cipher_input in valid_ciphers:
            final_out["accepted_suites"].append(cipher_input + ": " + valid_ciphers[cipher_input])
        elif cipher_input not in valid_ciphers:
            if cipher_input in weak_ciphers:
                final_out["supported_suites_but_should_not"]["weak_ciphers"].append(cipher_input)
            elif cipher_input in strong_ciphers:
                final_out["supported_suites_but_should_not"]["secure_ciphers"].append(cipher_input)
    for cipher in valid_ciphers:
        if cipher not in data_file['cipherSuites']:
            final_out["missing_suites"].append(cipher + ": " + valid_ciphers[cipher])
    return final_out

def get_extensions(final_out, data_file):
    """
    Retrieves the TLS extensions of the network functions and compares this with the 3GPP requirements,
    updating final_out

    :param data_file: the network function json input file
    :param final_out: the output dictionary
    :return updates final_out
    """
    for extensions in data_file['supportedExtensions']:
        if extensions in valid_extensions:
            final_out["accepted_extensions"].append(extensions)
        else:
            final_out["extra_extensions"].append(extensions)
    for missing in valid_extensions:
        if missing not in data_file['supportedExtensions']:
            final_out["missing_extensions"].append(missing)
    return final_out

def get_key_exchange(final_out, data_file):
    """
    Retrieves the key exchange algorithm of the network functions and compares this with the 3GPP requirements,
    updating out_report

    :param data_file: the network function json input file
    :param final_out: the output dictionary
    :return updates final_out
    """
    for key in data_file['supportedNamedGroups']:
        if key in valid_key_exchange:
            final_out["accepted_key_exchange"].append(key + ": " + valid_key_exchange[key])
```

```
212             else:
213                 final_out["extra_key_exchange"].append(key)
214         for missing in valid_key_exchange:
215             if missing not in data_file['supportedNamedGroups']:
216                 final_out["missing_key_exchange"].append(missing)
217         return final_out
218 
219 def get_signature_hash(final_out, data_file):
220         """
221         Retrieves the signature hash algo of the network functions and
        compares this with the 3GPP requirements,
222         updating out_report
223 
224         :param data_file: the network function json input file
225         :param final_out: the output dictionary
226         :return updates final_out
227         """
228         for sig_hash in data_file['supportedSignatureAndHashAlgorithmsSke']:
229             if sig_hash in valid_signature_hash:
230                 final_out["accepted_signature_hash"].append(sig_hash)
231             else:
232                 final_out["supported_signature_hash_but_should_not"].append(
        sig_hash)
233         for missing in valid_signature_hash:
234             if missing not in data_file['
        supportedSignatureAndHashAlgorithmsSke']:
235                 final_out["missing_signature_hash"].append(missing)
236         return final_out
237 
238 def get_bugs_and_vulns (final_out,data_file):
239         """
240         Retrieves the bugs and vulnerabilities of the network functions and
        updates out_report
241 
242         :param data_file: the network function json input file
243         :param final_out: the output dictionary
244         :return updates final_out
245         """
246         for bugs_and_vulns in data_file['resultMap']:
247             if data_file['resultMap'][bugs_and_vulns] == "TRUE":
248                 if bugs_and_vulns in vulnerabilities:
249                     final_out["vulnerabilities"].append(bugs_and_vulns + ": "
250                                                         + data_file['
        resultMap'][bugs_and_vulns])
251                 if bugs_and_vulns in common_bugs:
252                     final_out["bugs"].append(bugs_and_vulns + ": "
253                                              + data_file['
        resultMap'][bugs_and_vulns])
254         return final_out
255 
256 if __name__ == "__main__":
257         parser = argparse.ArgumentParser(prog = "Check 3GPP compliance +
        vulnerabilities and bugs")
258         parser.add_argument('filename', help='Required 5G scan results json
        file')
259         args = parser.parse_args()
260         file_name = args.filename
261         open_file = open(file_name,encoding="utf-8")
```

```
    json_in = json.load(open_file)
    json_out = json.dumps(get_bugs_and_vulns
                          (get_signature_hash
                          (get_key_exchange
                          (get_extensions
                          (get_ciphers_specs
                          (get_version_specs(json_in), json_in),json_in),
    json_in),json_in), json_in),
                          indent=4)

    #print(json_out)
    with open("3GPP_specs_results_" + file_name, "w",encoding="utf-8") as
     outfile:
        outfile.write(json_out)
    open_file.close()
    print("Done generating 3GPP check for " + file_name)
```

Listing 5.1: 3GPP compliance check script

```
import json
import argparse
import ssl
import os
from OpenSSL.crypto import load_certificate, FILETYPE_PEM, dump_publickey
    , load_privatekey


cert_key_report = {
    "CERT_FOUND_IN_PUBLIC_REPO": "",
    "PRIVATE_KEY_FOUND_IN_PUBLIC_REPO" : ""
}

def check_server_cert(input_file_name,files_list,dirname):
    """
    Checks if given server certificate is in the public
    repository of open-source 5G instance.

    :param input_file_name: name of json input file
    :param files_list: list of network functions' crt and key
                       files taken from open-source 5G instance repo
    :param dirname: directory where crt and key files are stored
    :return updates cert_key_report if certificate is found in public
    repo
    """
    file_json = open(input_file_name, encoding='utf-8')
    json_data = json.load(file_json)
    file_json.close()
    for filename in files_list:
        file = open(os.getcwd() + "/" + dirname + "/" + filename,
    encoding='utf-8')
        contents = file.read()
        file.close()
        if 'crt' in filename or 'pem' in filename:
            public_crt = ssl.PEM_cert_to_DER_cert(contents).hex()
        cert = json_data['certificateChainList'][0]['certificate']['
    certificates']
        if cert[0].lower() == public_crt:
            cert_key_report['CERT_FOUND_IN_PUBLIC_REPO']= filename
    return cert_key_report
```

```
37
38 def check_public_private_key(cert_file,list_files,dirname):
39     """
40     Checks if given server certificate has matching private key
41     in public repository of open-source 5g instance
42
43     :param cert_file: name of certificate file
44     :param files_list: list of network functions' crt and key
45                        files taken from open-source 5G instance repo
46     :param dirname: directory where crt and key files are stored
47     :return updates cert_key_report if private key is found in public
       repo
48     """
49     crt_file = open(os.getcwd() + "/" + dirname + "/" + cert_file,
       encoding='utf-8')
50     cert_str = crt_file.read()
51     crt_file.close()
52     certificate = load_certificate(FILETYPE_PEM,cert_str)
53     public_cert_key = dump_publickey(FILETYPE_PEM,certificate.get_pubkey
       ())
54
55     for filename in list_files:
56         if 'key' in filename:
57             priv_key_file = open(os.getcwd() + "/" + dirname + "/" +
       filename, encoding='utf-8')
58             priv_key_str = priv_key_file.read()
59             priv_key_file.close()
60             private_key = load_privatekey(FILETYPE_PEM,priv_key_str)
61             priv_pub_key = dump_publickey(FILETYPE_PEM,private_key)
62             if public_cert_key == priv_pub_key:
63                 cert_key_report['PRIVATE_KEY_FOUND_IN_PUBLIC_REPO'] =
       filename
64     return cert_key_report
65
66
67
68
69 if __name__ == "__main__":
70     parser = argparse.ArgumentParser(prog = "Check open5gs and free5gc
       certificate")
71     parser.add_argument('instance', help='Required 5G instance. open5gs
       or free5gc')
72     parser.add_argument('jsonInput', help='Required TLS-Scanner network
       function json file')
73     args = parser.parse_args()
74     if args.instance.lower() == "open5gs":
75         dir_name = "open5gs_public_crt_key"
76     elif args.instance.lower() == "free5gc":
77         dir_name = "free5gc_public_crt_key"
78     else:
79         raise Exception("No instance found")
80     files = os.listdir(dir_name)
81     input_name = args.jsonInput
82     report = check_server_cert(input_name,files,dir_name)
83     if len(cert_key_report['CERT_FOUND_IN_PUBLIC_REPO']) != "":
84         report = check_public_private_key(cert_key_report['
       CERT_FOUND_IN_PUBLIC_REPO'],files,dir_name)
85     json_out = json.dumps(report, indent=4)
```

```
86      print(json_out)
```

Listing 5.2: Certificate check script

Appendix B

```
Supported Server Cipher(s):
  Accepted  TLSv10   128 bits TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  Accepted  TLSv10   256 bits TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  Accepted  TLSv10   128 bits TLS_RSA_WITH_AES_128_CBC_SHA
  Accepted  TLSv10   256 bits TLS_RSA_WITH_AES_256_CBC_SHA
  Accepted  TLSv10   168 bits TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  Accepted  TLSv10   168 bits TLS_RSA_WITH_3DES_EDE_CBC_SHA
  Accepted  TLSv11   128 bits TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  Accepted  TLSv11   256 bits TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  Accepted  TLSv11   128 bits TLS_RSA_WITH_AES_128_CBC_SHA
  Accepted  TLSv11   256 bits TLS_RSA_WITH_AES_256_CBC_SHA
  Accepted  TLSv11   168 bits TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  Accepted  TLSv11   168 bits TLS_RSA_WITH_3DES_EDE_CBC_SHA
  Accepted  TLSv12   128 bits TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  Accepted  TLSv12   256 bits TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
  Accepted  TLSv12   128 bits TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  Accepted  TLSv12   256 bits TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  Accepted  TLSv12   128 bits TLS_RSA_WITH_AES_128_GCM_SHA256
  Accepted  TLSv12   256 bits TLS_RSA_WITH_AES_256_GCM_SHA384
  Accepted  TLSv12   128 bits TLS_RSA_WITH_AES_128_CBC_SHA
  Accepted  TLSv12   256 bits TLS_RSA_WITH_AES_256_CBC_SHA
  Accepted  TLSv12   168 bits TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  Accepted  TLSv12   168 bits TLS_RSA_WITH_3DES_EDE_CBC_SHA

Preferred Server Cipher(s):
  SSLv3   Protocol version not supported
  TLSv10   128 bits TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  TLSv11   128 bits TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  TLSv12   128 bits TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

Session:
  Compression: none

TLS renegotiation:
  Supported: yes
  Secure: yes

EC Pointer Format(s):
  uncompressed

HTTP Information
  Strict-Transport-Security: no
  Status-Code: 404
  Status-Message: Not Found
  Version: HTTP/1.1

Heartbleed(Vulnerability)
```

Figure 5.1: free5GC PySSLScan NRF scan results

```
CHECKING CONNECTIVITY TO SERVER(S)
 ----------------------------------

   172.44.0.12:8000          => 172.44.0.12

 SCAN RESULTS FOR 172.44.0.12:8000 - 172.44.0.12
 -----------------------------------------------

 * Certificates Information:
       Hostname sent for SNI:             172.44.0.12
       Number of certificates detected:   1


     Certificate #0 ( _RSAPublicKey )
       SHA1 Fingerprint:                  873a7657cc202774c2384bf82a0661edc2716490
       Common Name:                       ab
       Issuer:                            ab
       Serial Number:                     546410697906342172358321033622269960657191592753
       Not Before:                        2019-07-19
       Not After:                         2029-07-16
       Public Key Algorithm:              _RSAPublicKey
       Signature Algorithm:               sha256
       Key Size:                          2048
       Exponent:                          65537
       DNS Subject Alternative Names:     []

     Certificate #0 - Trust
       Hostname Validation:               FAILED - Certificate does NOT match server hostname
       Android CA Store (12.1.0_r5):      FAILED - Certificate is NOT Trusted: self signed certificate
       Apple CA Store (iOS 15.1, iPadOS 15.1, macOS 12.1, tvOS 15.1, and watchOS 8.1):FAILED - Certificate is NOT Trusted: self signed certificate
       Java CA Store (jdk-13.0.2):        FAILED - Certificate is NOT Trusted: self signed certificate
       Mozilla CA Store (2022-04-03):     FAILED - Certificate is NOT Trusted: self signed certificate
       Windows CA Store (2022-04-10):     FAILED - Certificate is NOT Trusted: self signed certificate
       Symantec 2018 Deprecation:         ERROR - Could not build verified chain (certificate untrusted?)
       Received Chain:                    ab
       Verified Chain:                    ERROR - Could not build verified chain (certificate untrusted?)
       Received Chain Contains Anchor:    ERROR - Could not build verified chain (certificate untrusted?)
       Received Chain Order:              OK - Order is valid
       Verified Chain contains SHA1:      ERROR - Could not build verified chain (certificate untrusted?)

     Certificate #0 - Extensions
       OCSP Must-Staple:                  NOT SUPPORTED - Extension not found
       Certificate Transparency:          NOT SUPPORTED - Extension not found

     Certificate #0 - OCSP Stapling
                                          NOT SUPPORTED - Server did not send back an OCSP response

 * SSL 2.0 Cipher Suites:
     Attempted to connect using 7 cipher suites; the server rejected all cipher suites.

 * SSL 3.0 Cipher Suites:
     Attempted to connect using 80 cipher suites; the server rejected all cipher suites.

 * TLS 1.0 Cipher Suites:
     Attempted to connect using 80 cipher suites.

 * TLS 1.0 Cipher Suites:
     Attempted to connect using 80 cipher suites.

     The server accepted the following 6 cipher suites:
        TLS_RSA_WITH_AES_256_CBC_SHA                    256
        TLS_RSA_WITH_AES_128_CBC_SHA                    128
        TLS_RSA_WITH_3DES_EDE_CBC_SHA                   168
        TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA              256       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA              128       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA             168       ECDH: prime256v1 (256 bits)

     The group of cipher suites supported by the server has the following properties:
       Forward Secrecy                  OK - Supported
       Legacy RC4 Algorithm             OK - Not Supported

 * TLS 1.1 Cipher Suites:
     Attempted to connect using 80 cipher suites.

     The server accepted the following 6 cipher suites:
        TLS_RSA_WITH_AES_256_CBC_SHA                    256
        TLS_RSA_WITH_AES_128_CBC_SHA                    128
        TLS_RSA_WITH_3DES_EDE_CBC_SHA                   168
        TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA              256       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA              128       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA             168       ECDH: prime256v1 (256 bits)

     The group of cipher suites supported by the server has the following properties:
       Forward Secrecy                  OK - Supported
       Legacy RC4 Algorithm             OK - Not Supported

 * TLS 1.2 Cipher Suites:
     Attempted to connect using 156 cipher suites.

     The server accepted the following 11 cipher suites:
        TLS_RSA_WITH_AES_256_GCM_SHA384                 256
        TLS_RSA_WITH_AES_256_CBC_SHA                    256
        TLS_RSA_WITH_AES_128_GCM_SHA256                 128
        TLS_RSA_WITH_AES_128_CBC_SHA                    128
        TLS_RSA_WITH_3DES_EDE_CBC_SHA                   168
        TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256     256       ECDH: X25519 (253 bits)
        TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384           256       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA              256       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256           128       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA              128       ECDH: prime256v1 (256 bits)
        TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA             168       ECDH: prime256v1 (256 bits)

     The group of cipher suites supported by the server has the following properties:
       Forward Secrecy                  OK - Supported
       Legacy RC4 Algorithm             OK - Not Supported

 * TLS 1.3 Cipher Suites:
     Attempted to connect using 5 cipher suites.

     The server accepted the following 3 cipher suites:
        TLS_CHACHA20_POLY1305_SHA256                    256       ECDH: X25519 (253 bits)
        TLS_AES_256_GCM_SHA384                          256       ECDH: X25519 (253 bits)
        TLS_AES_128_GCM_SHA256                          128       ECDH: X25519 (253 bits)


 * Deflate Compression:
                                          OK - Compression disabled

 * OpenSSL CCS Injection:
                                          OK - Not vulnerable to OpenSSL CCS injection

 * OpenSSL Heartbleed:
                                          OK - Not vulnerable to Heartbleed

 * ROBOT Attack:
                                          OK - Not vulnerable.

 * Session Renegotiation:
       Client Renegotiation DoS Attack:   OK - Not vulnerable
       Secure Renegotiation:              OK - Supported

 * Elliptic Curve Key Exchange:
       Supported curves:                  X25519, prime256v1, secp384r1, secp521r1
       Rejected curves:                   X448, prime192v1, secp160k1, secp160r1, secp160r2, secp192k1, secp224k1, secp224r1, secp256k1, sect163k1,

 SCANS COMPLETED IN 0.684986 S
 -----------------------------

 COMPLIANCE AGAINST MOZILLA TLS CONFIGURATION
 --------------------------------------------

    Checking results against Mozilla's "intermediate" configuration. See https://ssl-config.mozilla.org/ for more details.

    172.44.0.12:8000: FAILED - Not compliant.
        * certificate_hostname_validation: Certificate hostname validation failed for 1.2.840.113549.1.9.1=ab,CN=ab,OU=ab,O=ab,L=ab,ST=ab,C=ab.
        * certificate_path_validation: Certificate not path validation failed for 1.2.840.113549.1.9.1=ab,CN=ab,OU=ab,O=ab,L=ab,ST=ab,C=ab.
        * maximum_certificate_lifespan: Certificate life span is 3650 days, should be less than 366.
        * tls_versions: TLS versions {'TLSv1', 'TLSv1.1'} are supported, but should be rejected.
        * ciphers: Cipher suites {'TLS_RSA_WITH_AES_256_GCM_SHA384', 'TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA', 'TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA', 'T
```

Figure 5.2: free5GC SSLyze NRF scan results

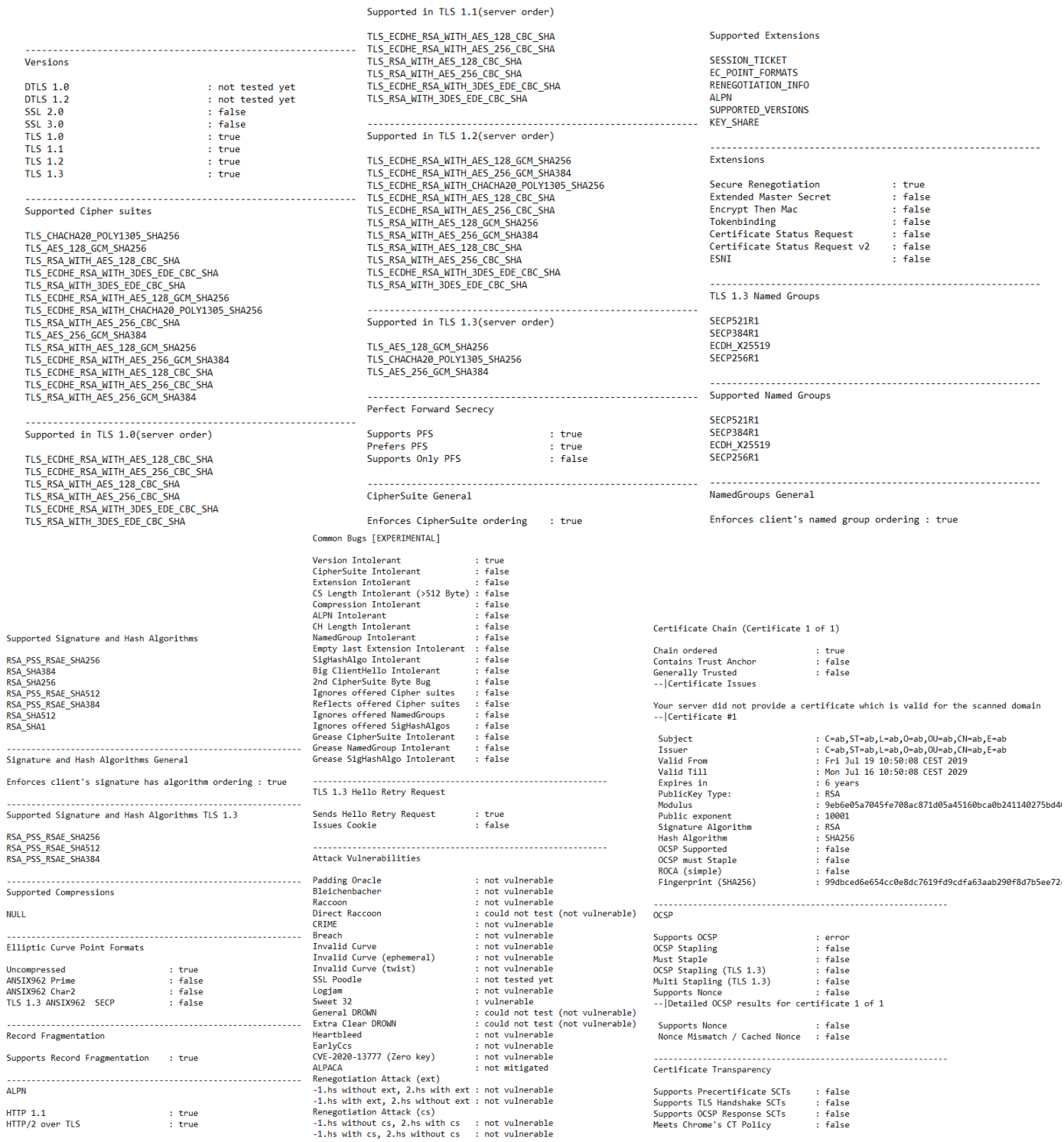

Figure 5.3: free5GC TLS-Scanner NRF scan results 1

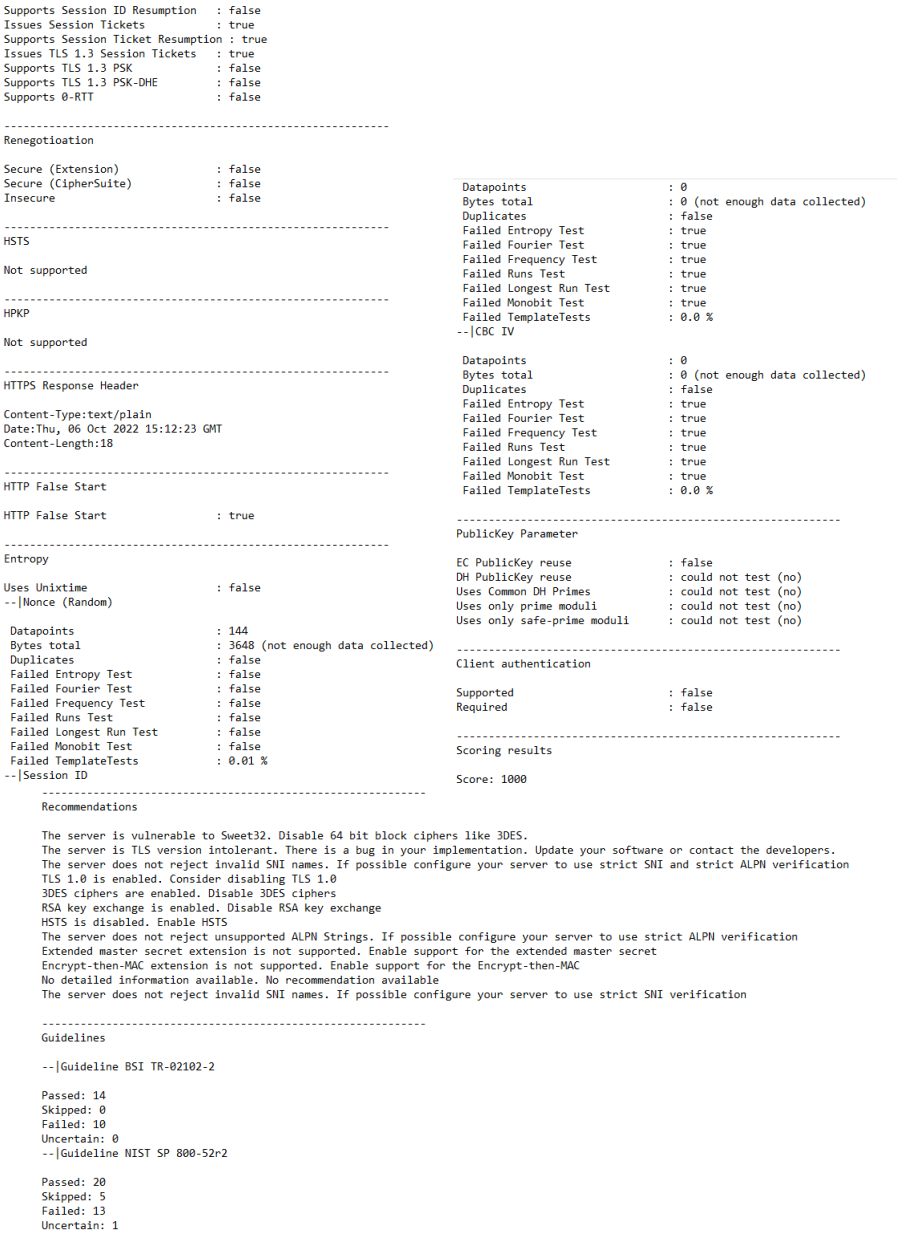

Figure 5.4: free5GC TLS-Scanner NRF scan results 2

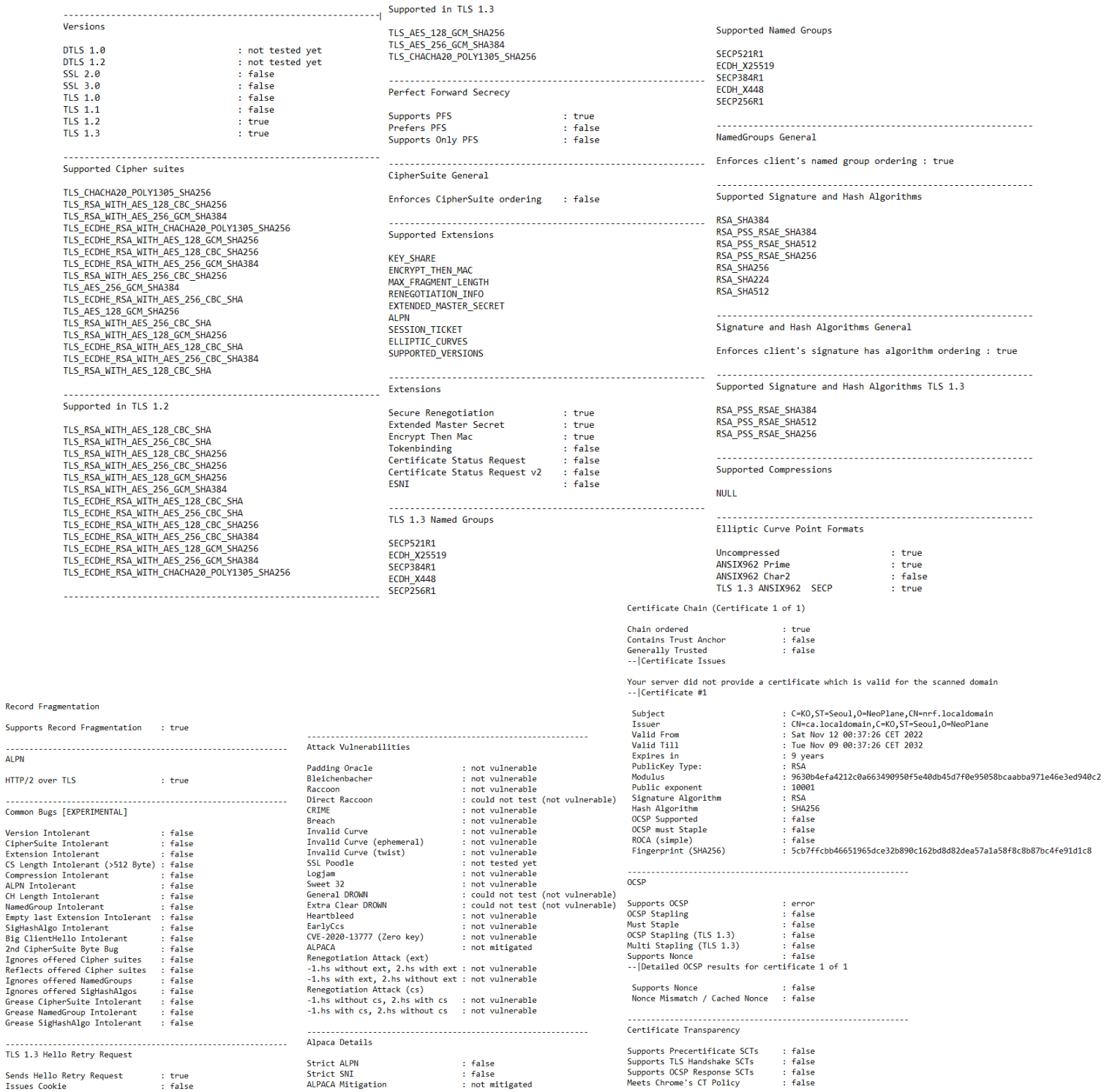

Figure 5.5: Open5gs TLS-Scanner NRF scan results 1

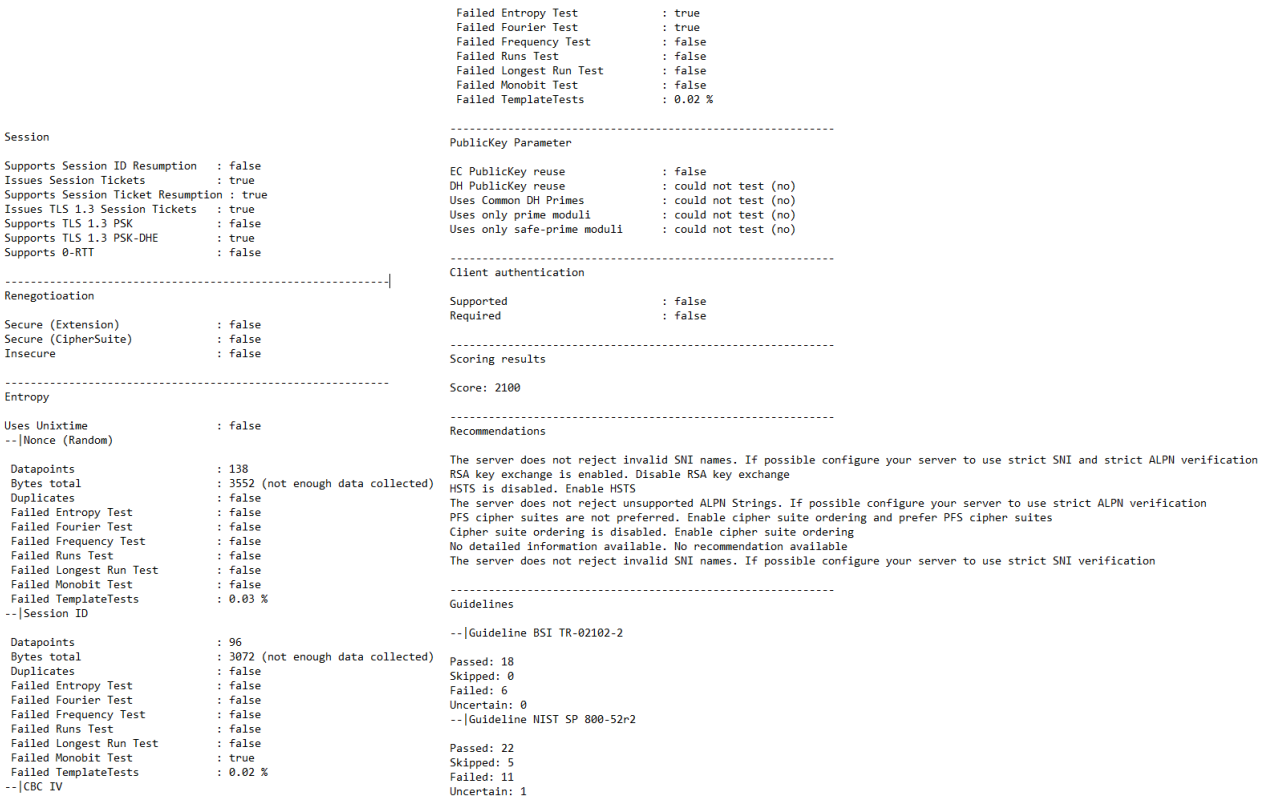

Figure 5.6: Open5gs TLS-Scanner NRF scan results 2