Copying Sessions in Oculus VR

May Khoury and Yasmin Irshied

Mentor – Shacham-Barr Ori

Contents

Abstract

Virtual Reality (VR) has become an integral part of modern technology, with people using it for education, gaming, and even remote collaboration in professional environments. As VR adoption grows, ensuring the security of network communications within these systems becomes increasingly important.

The Oculus Quest, developed by Meta, is one of the most popular VR devices on the market, offering a seamless wireless experience with built-in casting capabilities. However, this raises a critical question: Is it truly secure? Can a network casting session between two Oculus devices be intercepted, copied, or exploited?

Our research aimed to explore whether it is possible to copy a session in Oculus VR network communication. While we gained insights into possible risks, we could not conclusively determine if session duplication is feasible. Further research is needed to fully understand the security risks involved in Oculus VR network communication.

1. Background

1.1 Oculus Quest headsets

Oculus Quest is a standalone Virtual Reality headset developed by Meta. Unlike traditional VR headsets that require a wired connection to a PC or gaming console, the Quest series operates independently, with built-in hardware capable of running VR applications wirelessly.

1.2 Oculus Quest operating system

The Oculus Quest operates on Horizon OS, a customized version of Android built on top of a Linux kernel that is engineered specifically for immersive virtual reality experiences. Horizon OS is designed to handle the unique demands of VR, optimizing system resources for real-time 3D graphics rendering, precise head tracking, and responsive controller input. Meta has enhanced the underlying Android platform with a proprietary code layer ensures the smooth run of a virtual reality headset and interacts seamlessly with the device's hardware. Beyond performance, security is a foundational aspect of Horizon OS. The operating system employs Android's sandboxing feature to isolate each application, thereby preventing unauthorized access to sensitive system resources. In addition, the verified boot process that Meta uses ensures that only authenticated, properly signed firmware is loaded, effectively safeguarding the device against tampering and malicious software. Overall, the Horizon OS embodies a strategic integration of high-performance VR capabilities with robust security measures, laying the groundwork for the Oculus Quest's immersive user experience.

1.3 Oculus Quest Casting

Oculus Quest casting allows users to mirror their VR experience onto an external device such as a browser, phone, or TV. This feature is useful for sharing gameplay, streaming VR content, or demonstrating VR applications. The casting process involves wireless communication over Wi-Fi, making it a potential target for network-based security threats.

Note

Initially, our research was conducted using the Oculus Quest 2, but we later switched to using the Oculus Quest 1 instead. Throughout this paper, we will specify which device was used for each observation to ensure clarity and distinction between the two models.

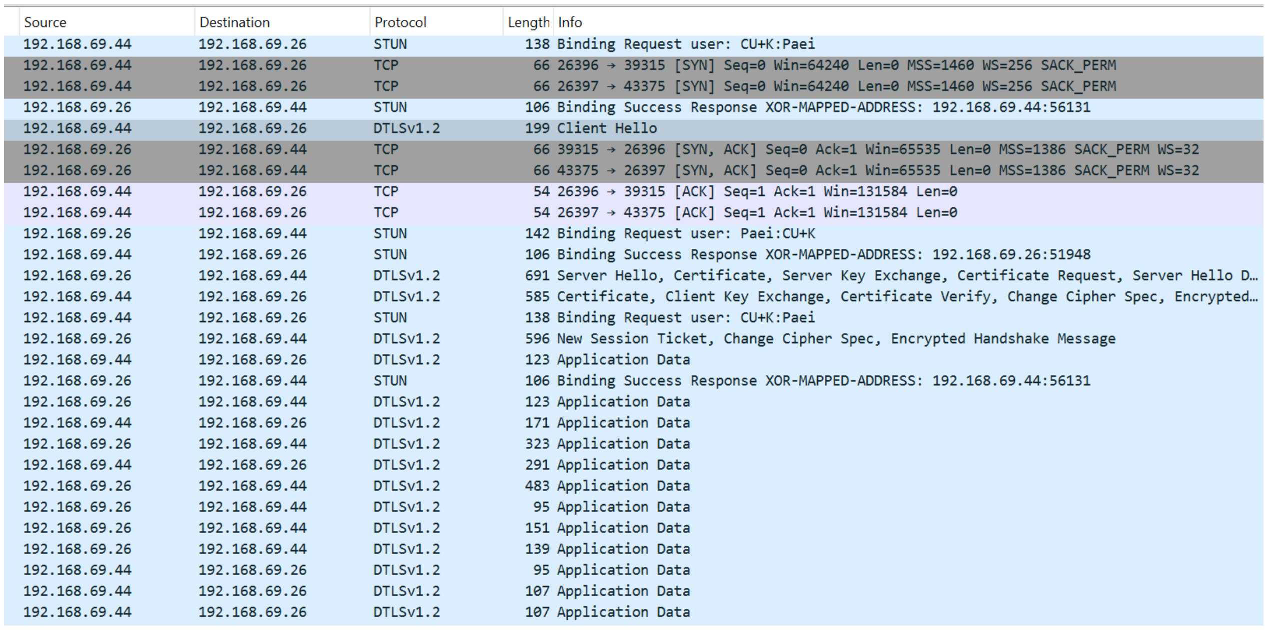
2. Casting traffic analysis

To understand how Oculus Quest casting works, first we began by analyzing network traffic between the Oculus2 VR headset and the casting destination (according to the instructions on Meta’s website [1]).

Our setup involved creating a controlled network environment to capture and inspect the data transmitted during a casting session.

First, we casted the Oculus Quest to a web browser while ensuring that both the VR headset and the receiving computer were on the same Wi-Fi network. To achieve this and also be able to see all the packets without the router filtering them, we configured our computer as a Wi-Fi hotspot, connecting both the Oculus Quest and the casting computer to the same network.

To capture network traffic, we used Wireshark, a network protocol analyzer. We filtered the captured data based on the Oculus Quest’s IP address, allowing us to isolate and examine all packets exchanged between the VR device and the browser. This provided a detailed view of the communication process which is provided in figure 1.



Source	Destination	Protocol	Length	Info
192.168.69.44	192.168.69.26	STUN	138	Binding Request user: CU+K:Paei
192.168.69.44	192.168.69.26	TCP	66	26396 → 39315 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
192.168.69.44	192.168.69.26	TCP	66	26397 → 43375 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
192.168.69.26	192.168.69.44	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 192.168.69.44:56131
192.168.69.44	192.168.69.26	DTLSv1.2	199	Client Hello
192.168.69.26	192.168.69.44	TCP	66	39315 → 26396 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1386 SACK_PERM WS=32
192.168.69.26	192.168.69.44	TCP	66	43375 → 26397 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1386 SACK_PERM WS=32
192.168.69.44	192.168.69.26	TCP	54	26396 → 39315 [ACK] Seq=1 Ack=1 Win=131584 Len=0
192.168.69.44	192.168.69.26	TCP	54	26397 → 43375 [ACK] Seq=1 Ack=1 Win=131584 Len=0
192.168.69.26	192.168.69.44	STUN	142	Binding Request user: Paei:CU+K
192.168.69.44	192.168.69.26	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 192.168.69.26:51948
192.168.69.26	192.168.69.44	DTLSv1.2	691	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello D…
192.168.69.44	192.168.69.26	DTLSv1.2	585	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted…
192.168.69.44	192.168.69.26	STUN	138	Binding Request user: CU+K:Paei
192.168.69.26	192.168.69.44	DTLSv1.2	596	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
192.168.69.44	192.168.69.26	DTLSv1.2	123	Application Data
192.168.69.26	192.168.69.44	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 192.168.69.44:56131
192.168.69.26	192.168.69.44	DTLSv1.2	123	Application Data
192.168.69.44	192.168.69.26	DTLSv1.2	171	Application Data
192.168.69.26	192.168.69.44	DTLSv1.2	323	Application Data
192.168.69.44	192.168.69.26	DTLSv1.2	291	Application Data
192.168.69.26	192.168.69.44	DTLSv1.2	483	Application Data
192.168.69.26	192.168.69.44	DTLSv1.2	95	Application Data
192.168.69.26	192.168.69.44	DTLSv1.2	151	Application Data
192.168.69.26	192.168.69.44	DTLSv1.2	139	Application Data
192.168.69.44	192.168.69.26	DTLSv1.2	95	Application Data
192.168.69.44	192.168.69.26	DTLSv1.2	107	Application Data
192.168.69.44	192.168.69.26	DTLSv1.2	107	Application Data

Figure 1.

During packet inspection, we observed that casting traffic primarily relied on UDP packets. A deeper examination revealed the use of DTLSv1.2 (Datagram Transport Layer Security) as the security protocol securing communication between the VR headset (server) and the browser (client). Additionally, we noticed the presence of STUN (Session Traversal Utilities for NAT) messages, indicating the involvement of peer-to-peer communication or NAT traversal mechanisms in establishing the casting session.

At this point, understanding how DTLS secures the data in transit became crucial. The following section provides a detailed breakdown of the DTLS protocol, its mechanisms, and its role in securing Oculus Quest casting.

2.1 DTLSv1.2 Protocol

Datagram Transport Layer Security (DTLS) is a protocol designed to provide security for datagram-based applications, such as those using UDP. Unlike TLS, which operates over TCP, DTLS ensures that encryption, authentication, and integrity checks are maintained even in unreliable, connectionless environments. DTLS is built on the foundation of TLS 1.2, making it highly similar in terms of cryptographic security mechanisms but adapted for datagram transport. One of its key features is the stateless cookie exchange mechanism, which is designed to prevent Denial-of-Service (DoS) attacks. In a DTLS protocol a handshake is performed at the start where the two sides participating in the handshake agree on cryptographic keys and info that will help encrypt the transport following the end of the handshake.

2.1.1 DTLS handshake

As we mentioned, at the start of the DTLS protocol a handshake is performed. We will now discuss the different parts of the handshake shown in figure 2.

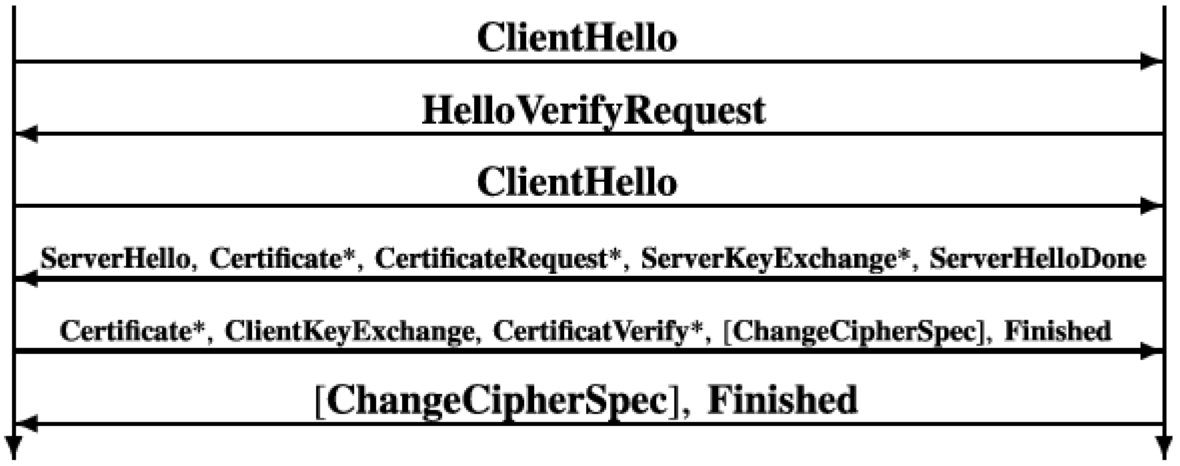

ClientHello
HelloVerifyRequest
ClientHello
ServerHello, Certificate*, CertificateRequest*, ServerKeyExchange*, ServerHelloDone
Certificate*, ClientKeyExchange, CertificatVerify*, [ChangeCipherSpec], Finished
[ChangeCipherSpec], Finished

Figure 2.

First off, the client is the one that starts the handshake by sending the ClientHello message that contains, among other things, a list of the cipher suites that are suitable for the client. Cipher suites are a list of encryption and verification algorithms that are sent to the server so it can choose one to use. Then the server may send back a HelloVerifyRequest message to the client containing a cookie. The client must return the same cookie in a second ClientHello message, proving that it can receive responses at its claimed IP address. This prevents attackers from flooding the server with fake handshake requests since they would need to receive and return a valid cookie [2]. After that, the handshake resumes much like the TLS handshake. The server sends a ServerHello message that contains, among other things, the cipher suite that he chose. At the end, the client and server agree on cryptographic keys and info and finish the handshake.

2.1.2 How does DTLS handle UDP problems

With DTLS securing UDP based packets, and since UDP has some problems (that TCP doesn't have), DTLS handles these problems by using different mechanisms:

- UDP does not guarantee ordered packet delivery and so DTLS introduces sequence numbers to help with the proper reconstruction of messages. The protocol also incorporates retransmission timers that allow the handshake process to recover from missing packets in case of packet loss or reordering.
- UDP packets are typically limited to a maximum size of 1500 bytes in standard networks to avoid fragmentation at the IP layer. To address this limitation, DTLS enables large handshake messages to be split across multiple smaller records, ensuring successful message delivery without exceeding packet size constraints.
- DTLS also features a record layer like that of TLS, responsible for maintaining message security throughout communication. Each record includes an epoch number, which increments whenever the encryption state changes, and a sequence number, which helps prevent replay attacks and ensure message integrity. The

fragmented payload structure ensures that large messages can be properly split and reassembled without data loss.

2.1.3 DTLS implementation libraries

To understand how DTLS is implemented in the Oculus Quest, we examined potential libraries and frameworks that could be responsible for handling DTLS encryption. In the firmware analysis that will be described in the following chapters, we found many indications that hint at what DTLS implementation libraries are being used, such as the strings shown in figure 3.

```
# Android's provider of OpenSSL backed implementations
security.provider.1=com.android.org.conscrypt.OpenSSLProvider
# Android's version of the CertPathValidator and CertPathBuilder
security.provider.2=sun.security.provider.CertPathProvider
# Android's stripped down BouncyCastle provider
security.provider.3=com.android.org.bouncycastle.jce.provider.BouncyCastleProvider
# Android's provider of OpenSSL backed implementations
security.provider.4=com.android.org.conscrypt.JSSEProvider
```

Figure 3.

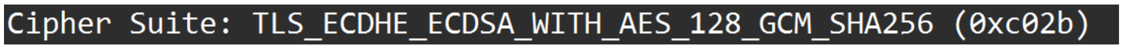
There are several cryptographic libraries that support DTLS, and based on our findings, Oculus Quest may be using a combination of Java and C-based implementations. OpenSSL is one of the most widely used cryptographic libraries for DTLS and is typically integrated into applications using Java Native Interface (JNI) to bridge the gap between Java and C [3]. Other possible implementations include WolfSSL JSSE provider (Java Secure Socket Extension), which is Java's built-in framework for handling TLS and DTLS connections [4].

Additionally, the Bouncy Castle DTLS implementation offers a lightweight solution for securing datagram communications in environments with packet loss and reordering. Its modular cryptography APIs ensures confidentiality and integrity while adhering to industry standards [5].

To test the vulnerabilities of these implementations we found papers that extensively test their security, using state inference machines. Security research has shown that DTLS implementations vary in terms of their resistance to attacks. OpenSSL's DTLS implementation has been extensively tested and has not demonstrated any critical vulnerabilities related to handshake manipulation or replay attacks. However, some Java-based DTLS implementations, such as JSSE, have been reported to contain authentication bypass vulnerabilities when certain out-of-order handshake messages are received [6]. Furthermore, analysis of TLS-Attacker, a Java-based framework for analyzing TLS implementations that was used in said paper, has demonstrated the TLS-attacker's ability to test DTLS behaviour under unexpected protocol flows. This suggests that while the DTLS implementation in Oculus Quest casting is robust, we could use this TLS-attacker to test the implementations in the Oculus Quest, and thus further analysis is needed to determine whether any vulnerabilities exist that could be exploited in an attack scenario.

2.1.4 DTLS package manipulation

As mentioned above, in the DTLS handshake, the server chooses a cipher suite from the ones the client offers in the first ClientHello message. Cipher suites are collections of algorithms that work together to secure network communications. They typically include a key exchange algorithm, which establishes a secure channel for sharing keys, an authentication algorithm, which verifies the identities of the parties involved, a symmetric encryption algorithm, which protects data privacy during transmission, and a message authentication code (MAC) algorithm, which ensures data integrity and authenticity. Together, these components enable encrypted connections that are both secure and efficient. In our network analysis we discovered that the cipher suite chosen by the server is the one shown in figure 4.



```
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
```

Figure 4.

The key exchange algorithm chosen by the server (in our case the VR) is ECDHE or Elliptic Curve Diffie-Hellman Ephemeral algorithm, that leverages the efficiency of elliptic curve cryptography to establish secure, temporary keys for each session. By generating ephemeral keys, ECDHE ensures forward secrecy, meaning that even if long-term keys are compromised, past communications remain secure. In addition, the digital signature algorithm chosen is Elliptic Curve Digital Signature Algorithm (ECDSA) which is a digital signature scheme that employs elliptic curve cryptography to provide strong security with smaller key sizes. It works by using a private key to generate a signature for a message, which can then be verified by others using the corresponding public key.

Interestingly, Wireshark can decrypt DTLS traffic if the private key is available, and the cipher suite allows for RSA key exchange [7]. But since Oculus Quest employs ECDHE, we can't decrypt the packages using Wireshark, unless we try modifying the DTLS handshake to force a downgrade to an RSA-based cipher suite. To do so, we thought about performing a Man in The Middle attack in order to perform package manipulation using the Scapy python library, which is a tool designed for interactive packet manipulation and network analysis. It allows users to create, send, and dissect packets at various network layers. This way, we can possibly change the first ClientHello message to have only one cipher suite suggested to the server, and for that cipher suite to allow RSA key exchange. By doing so, we force the server to choose this cipher suite, and we force the connection to be based on RSA key exchange algorithm, and then we can decrypt the packages using Wireshark, if only we had the private key of the device.

Of course, there may be other ways that help with package decryption such as python scripts tailored specifically for this purpose.

2.2 STUN Protocol in Oculus Quest Casting Security

During our network analysis, we observed that, in addition to DTLS, the Oculus Quest casting process relies on the STUN protocol. This suggests that STUN is being used to handle NAT traversal, which is essential for establishing a connection between the VR headset and the casting device, particularly when they are on separate networks. STUN plays a critical role in allowing devices behind NAT firewalls to determine their public IP address and network accessibility, enabling direct peer-to-peer (P2P) communication whenever possible. It is commonly employed in WebRTC applications, where real-time communication between browsers or devices requires low-latency data exchange. In the case of Oculus Quest casting, STUN likely facilitates the identification of the public IP address of the casting device, allowing for the initial connection setup between the VR headset and the browser.

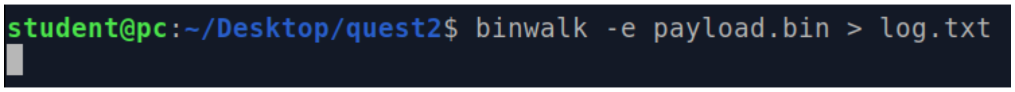
Beyond assisting in the discovery of public IP addresses, STUN is integral to maintaining active network connections. NAT devices frequently remove inactive sessions from their tables, which can cause unexpected disconnections in real-time applications. To counter this, STUN incorporates a keep-alive mechanism in which periodic refresh requests are sent to ensure that the NAT binding remains active. The reliance on STUN makes it possible for Oculus Quest to maintain stable casting connections across different network environments.

3. Firmware Analysis of Oculus Quest

To further investigate the security architecture of the Oculus Quest 2, we obtained firmware files corresponding to version 69.0.0.578.352.641254960 from an external repository [8]. Our purpose was to analyze the structure of the firmware, identify critical components, and try to find the filesystem of the device.

3.1 Extraction and Initial Analysis

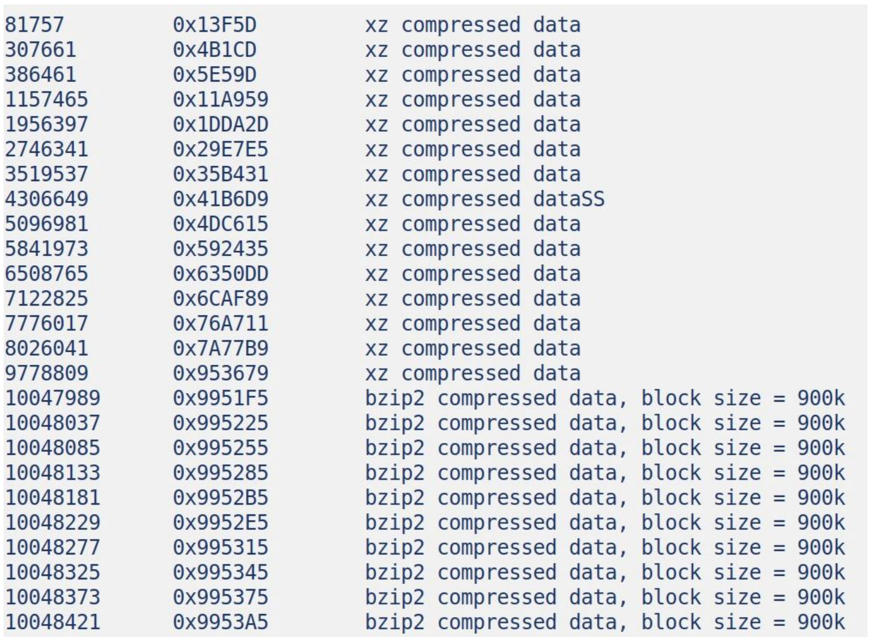
Using binwalk, a widely used tool for firmware reverse engineering, we analyzed the binary firmware files and identified multiple compressed archives containing various system components. Many of these files appeared to be structured as compressed system partitions, including SquashFS (sqsh), JFFS2 (jffs), and UBI (ubi) file systems. To better understand the structure, we extracted the binaries within these partitions and searched for strings related to system operations.



```
student@pc:~/Desktop/quest2$ binwalk -e payload.bin > log.txt
```

Payload.bin is the firmware binary file. We couldn't run a recursive binwalk using the **–Me** flag due to limited storage on the device. Instead, we performed binwalk on segments of the file and analyzed each segment individually. We examined the files, discovered many zipped folders, and then manually ran *binwalk* on each one again to extract partial files and try to understand how they relate to our research.

After the second round of running *binwalk*, we obtained many binaries and unreadable files. To make sense of these files, we ran the *strings* command to extract readable text and gain insight into their contents.

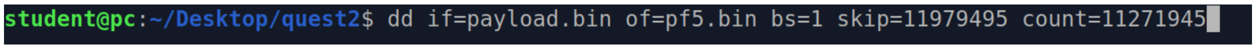


```
81757       0x13F5D     xz compressed data
307661      0x4B1CD     xz compressed data
386461      0x5E59D     xz compressed data
1157465     0x11A959    xz compressed data
1956397     0x1DDA2D    xz compressed data
2746341     0x29E7E5    xz compressed data
3519537     0x35B431    xz compressed data
4306649     0x41B6D9    xz compressed dataSS
5096981     0x4DC615    xz compressed data
5841973     0x592435    xz compressed data
6508765     0x6350DD    xz compressed data
7122825     0x6CAF89    xz compressed data
7776017     0x76A711    xz compressed data
8026041     0x7A77B9    xz compressed data
9778809     0x953679    xz compressed data
10047989    0x9951F5    bzip2 compressed data, block size = 900k
10048037    0x995225    bzip2 compressed data, block size = 900k
10048085    0x995255    bzip2 compressed data, block size = 900k
10048133    0x995285    bzip2 compressed data, block size = 900k
10048181    0x9952B5    bzip2 compressed data, block size = 900k
10048229    0x9952E5    bzip2 compressed data, block size = 900k
10048277    0x995315    bzip2 compressed data, block size = 900k
10048325    0x995345    bzip2 compressed data, block size = 900k
10048373    0x995375    bzip2 compressed data, block size = 900k
10048421    0x9953A5    bzip2 compressed data, block size = 900k
```

Screenshot after the first round of *binwalk* command

3.2 Key Observations from Extracted Files

Each partial file was acquired by performing the *dd command* in order to cut the large binary firmware file into segments like described above -



```
student@pc:~/Desktop/quest2$ dd if=payload.bin of=pf5.bin bs=1 skip=11979495 count=11271945
```

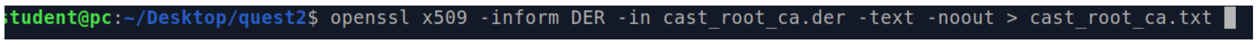
In our analysis we reached offset of 23251440 in decimal.

Below is a table summarizing some key findings from the partial files we acquired.

Partial file	Information
Pf1	The file contains strings that are closely tied to power management, clock control, and performance scaling on a Qualcomm-based platform, especially for subsystems like DDR (memory), WLAN, and other hardware components. The recurring identifiers—cx.lvl, mx.lvl, xo.lvl, and ddr.lvl—likely denote different power or voltage levels for specific components or subsystems. These markers are typically linked to power management and resource control, managed by a system such as the Qualcomm RPM or a similar subsystem.
Pf2	Empty
Pf3	The list of strings appears to be associated with a Qualcomm-based embedded system that handles battery charging, and USB/DC input. They offer valuable insights into how the firmware manages power-related events such as overvoltage, under voltage, temperature monitoring, and battery status.
Pf4	The file contains a range of strings and references specific to Qualcomm-secured environments. It includes mentions of QSEE, Widevine DRM, memory management, and hypervisor functionality, along with various configuration keys, error messages, and device-specific terminology.
Pf5	The file contains strings that suggest a debugging or error log output, likely originating from a video processing, decoding, or rendering pipeline. They are closely associated with resource allocation, buffer management, and the creation of task queues within a multimedia subsystem.
Pf6	The file contains strings that are logs and error messages from a Linux kernel running on a Qualcomm-based platform. These logs are associated with various system functions, including thermal and power management, device drivers, CPU frequency scaling, cryptographic operations, and device initialization. The data also indicates that the firmware corresponds to an Android 12-based build for an Oculus/Meta VR device with the codename "hollywood."

3.3 Cryptographic and Security Components

During our analysis, we found multiple certificates embedded within the firmware in partial file 4, indicating the presence of secure communication and authentication mechanisms. Using OpenSSL, we were able to extract and convert some of these certificates to a readable format, revealing details about firmware validation and secure boot policies.



```
tudent@pc:~/Desktop/quest2$ openssl x509 -inform DER -in cast_root_ca.der -text -noout > cast_root_ca.txt
```

Additionally, our findings strongly suggest that the Oculus Quest relies on a Trusted Execution Environment (TEE) for handling sensitive operations. Strings found in the firmware reference Qualcomm Secure Execution Environment (QSEE), an implementation of TEE designed to isolate cryptographic operations and key storage from the main operating system. The extracted firmware files contained paths related to secure storage, such as **/persist/data/app_g/sfs/keybox_lvl3.dat** and **license_data_store_path**, indicating that cryptographic keys and licensing data are securely stored within QSEE.

Our analysis also uncovered QSEE Secure File System (SFS) function calls, including:

- qsee_sfs_open
- qsee_sfs_read
- qsee_sfs_write
- qsee_sfs_getSize
- qsee_sfs_rm

These functions are from QSEE Secure Filesystem API, they suggest that QSEE manages secure storage, ensuring that sensitive files such as keyboxes, DRM licenses, and authentication tokens are protected from unauthorized access.

The next section will explore TEE and QSEE in detail, analyzing their security architecture, attack surface, and potential implications for Oculus Quest security.

4. Trusted Execution Environment and QSEE in Oculus Quest

The firmware analysis of the Oculus Quest strongly suggested the presence of a Trusted Execution Environment (TEE), specifically Qualcomm's Secure Execution Environment (QSEE), responsible for handling sensitive operations such as cryptographic key storage, certificate verification, and secure file management. To fully understand the security implications of this component, it is essential to explore how TEE and QSEE function within the system architecture and how their design contributes to, or potentially compromises, the overall security of the device.

4.1 What Is a Trusted Execution Environment (TEE)?

A Trusted Execution Environment (TEE) is a secure area of a main processor used to run code and process data that must be protected from external interference or tampering. It operates in parallel with the main operating system but is isolated from it, ensuring that data processed within the TEE is protected from attacks occurring in the normal (non-secure) environment. The most widespread TEE implementation in ARM-based devices is TrustZone, a hardware-based isolation technology that virtually splits the processor into a "secure world" and a "non-secure world" [9].

Within the secure world, the TEE operates its own minimalistic operating system and executes Trusted Applications (TAs) that perform security-sensitive tasks like digital rights management (DRM), secure key handling, and hardware-backed cryptographic operations. The normal world, meanwhile, runs the regular operating system (Android, in the case of Oculus Quest). The processor can only be in one world at a time, and the transition between them is managed through a privileged instruction called a Secure Monitor Call (SMC).

4.2 Qualcomm Secure Execution Environment (QSEE)

QSEE is Qualcomm's proprietary implementation of TrustZone and is one of the most widely deployed TEEs in Android-based devices, including smartphones and VR headsets such as the Oculus Quest.

QSEE adopts a monolithic architecture, bundling the secure kernel, monitor, and service handlers into a single ELF image. This design increases execution efficiency but also makes analysis and debugging more difficult, particularly because QSEE lacks standard debugging hooks and relies heavily on vendor-specific configurations [10].

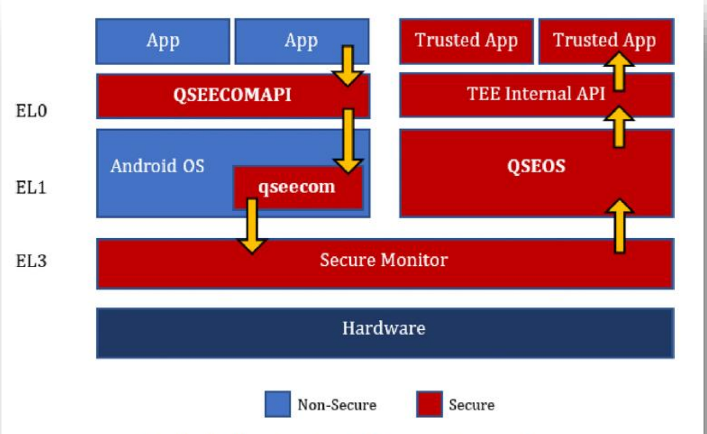


Fig. 2. QSEE Architecture of Software Components.

4.3 Security Considerations and Known Vulnerabilities

Although QSEE provides strong security assurances, several critical vulnerabilities have been identified over the years. For instance, a vulnerability in the Widevine Trusted Application (TA), which is used for decrypting DRM-protected media, enabled out-of-bounds memory access and ultimately key leakage through a carefully crafted buffer manipulation attack. Another common category of issues involves improper input validation in Secure Monitor Calls (SMCs), which can lead to privilege escalation or code execution in the secure world [11].

One of the most persistent security challenges within QSEE is related to the semantic gap between the secure and non-secure worlds. If the TEE assumes that input data from the normal world has already been sanitized or validated, it may blindly trust corrupted data structures, leading to memory corruption or logic errors. Vulnerabilities have also been found in the QSEECOM driver, which facilitates communication between Android and QSEE. These flaws allow attackers with root access to gain unauthorized entry into the secure world through buffer overflows or race conditions [12].

Despite these issues, Qualcomm has continuously hardened QSEE by introducing support for RSA-PSS and ECDSA cryptographic algorithms, deprecating older insecure schemes, and enforcing anti-rollback mechanisms to prevent firmware downgrades to vulnerable versions.

A study titled "Vulnerability Analysis of Qualcomm Secure Execution Environment (QSEE)" examined vulnerabilities across various QSEE components. They manually mined CVEs for these components and created a heat map to visually highlight the areas that are most susceptible to attacks. The results showed that Trusted Applications (TAs) are the most vulnerable, with 33% of the total attacks targeting them.[9]

4.4 Role of QSEE in Oculus Quest Security

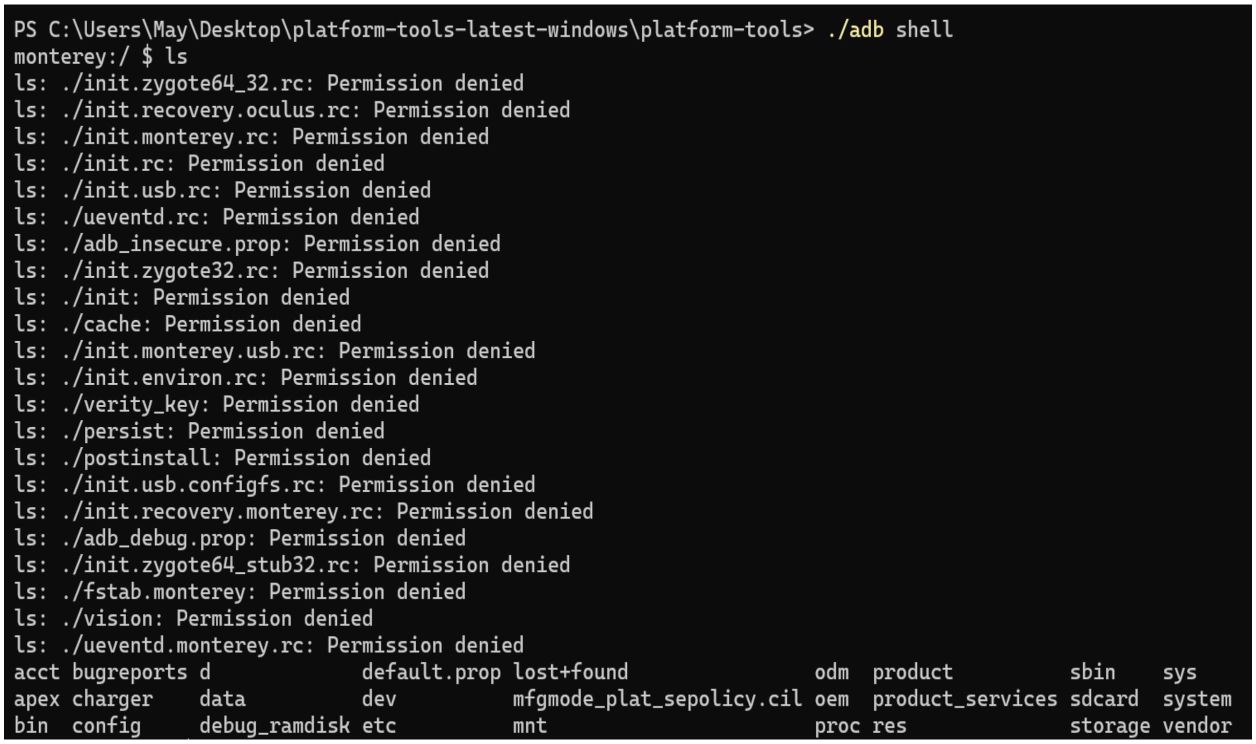
The reliance on QSEE introduces a dependency on Qualcomm's proprietary code and security patching cycle. Since QSEE is closed-source, identifying or validating its internal behaviour is challenging without advanced reverse engineering. This limitation makes it difficult for third-party researchers to verify whether the Oculus Quest implementation is up to date with the latest security fixes or vulnerable to legacy exploits targeting older versions of QSEE.

5. Android debug bridge

To further investigate the internal structure of the Oculus Quest system and identify where sensitive data such as private keys might be stored, we used Android Debug Bridge (ADB) to explore the device's file system.

ADB is a standard command-line tool that allows communication and file access on Android-based systems. Since the Oculus Quest runs a modified Android OS (Horizon OS), enabling developer mode on the headset and in the account also enables ADB access. This allowed us to initiate local shell sessions and navigate the file structure in an attempt to locate cryptographic key material or configuration files related to casting or firmware security.

This is a screenshot taken after connecting the device via a wire to the Oculus Quest and running ls command.



```
PS C:\Users\May\Desktop\platform-tools-latest-windows\platform-tools> ./adb shell
monterey:/ $ ls
ls: ./init.zygote64_32.rc: Permission denied
ls: ./init.recovery.oculus.rc: Permission denied
ls: ./init.monterey.rc: Permission denied
ls: ./init.rc: Permission denied
ls: ./init.usb.rc: Permission denied
ls: ./ueventd.rc: Permission denied
ls: ./adb_insecure.prop: Permission denied
ls: ./init.zygote32.rc: Permission denied
ls: ./init: Permission denied
ls: ./cache: Permission denied
ls: ./init.monterey.usb.rc: Permission denied
ls: ./init.environ.rc: Permission denied
ls: ./verity_key: Permission denied
ls: ./persist: Permission denied
ls: ./postinstall: Permission denied
ls: ./init.usb.configfs.rc: Permission denied
ls: ./init.recovery.monterey.rc: Permission denied
ls: ./adb_debug.prop: Permission denied
ls: ./init.zygote64_stub32.rc: Permission denied
ls: ./fstab.monterey: Permission denied
ls: ./vision: Permission denied
ls: ./ueventd.monterey.rc: Permission denied
acct bugreports d            default.prop lost+found                odm  product          sbin    sys
apex charger    data         dev          mfgmode_plat_sepolicy.cil oem  product_services sdcard  system
bin  config     debug_ramdisk etc         mnt                       proc res              storage vendor
```

However, during this process, we encountered significant permission restrictions. Most sensitive directories and system partitions—including **/data, /vendor**, **/persist** and many more of the paths returned permission denied errors.

In theory, rooting the device would bypass these restrictions and grant access to otherwise protected areas of the file system, potentially allowing us to extract cryptographic material or observe secure runtime behaviour. However, rooting the Oculus Quest is non-trivial and risky. The process involves modifying the bootloader and flashing unsigned firmware, which triggers the device's secure boot verification and may result in bricking the device if tamper detection is triggered or rollback protection is enforced. Given the risk of permanently disabling the hardware, we opted not to proceed with rooting at this stage of our analysis.

According to this GitHub repository [13] , certain vulnerable versions of the Oculus Quest 1 and 2 allow the bootloader to be unlocked. This capability could pave the way for rooting the device in the future, as it would grant users enhanced access to the system by bypassing some built-in security measures.

Interestingly, while our local attempts via USB-connected ADB were limited by system permissions, recent research has demonstrated a more alarming remote attack vector using wireless ADB access. The study by Yang et al. (2024), *Inception Attacks: Immersive Hijacking in Virtual Reality*, describes a method in which attackers exploit the presence of ADB over Wi-Fi in VR headsets like the Meta Quest. When developer mode is enabled ADB services can be accessed remotely over the local network without any authentication prompts. This creates an opportunity for a remote attacker on the same Wi-Fi network to run shell commands, install malicious apps, and manipulate the headset's behaviour without physical access to the device.

In their implementation, the researchers embedded an ADB client into a malicious app that was either sideloaded via platforms like SideQuest or silently authorized through deceptive popups. Once access was granted, the app used ADB to install a custom VR environment that mimicked the official home interface, effectively trapping the user inside a fake system layer. From there, they demonstrated full hijacking capabilities—including surveillance of user input, replication of installed apps, interception of VR browser traffic, and modification of social interactions inside applications like VRChat.

This finding not only illustrates the power and flexibility of ADB as a tool for legitimate research, but also highlights its potential misuse in VR environments, especially when access controls and user authentication mechanisms are lacking. While Oculus Quest does enforce certain permission boundaries under ADB, the combination of developer mode, Wi-Fi exposure, and user trust in sideloaded apps creates a serious vulnerability vector—one that is increasingly relevant as more users enable developer access for customization and experimentation.

Attacks on the ADB can help us gain higher access rights, allowing us to open files that were not accessible before. Therefore, researching ADB vulnerabilities is important for advancing the work.

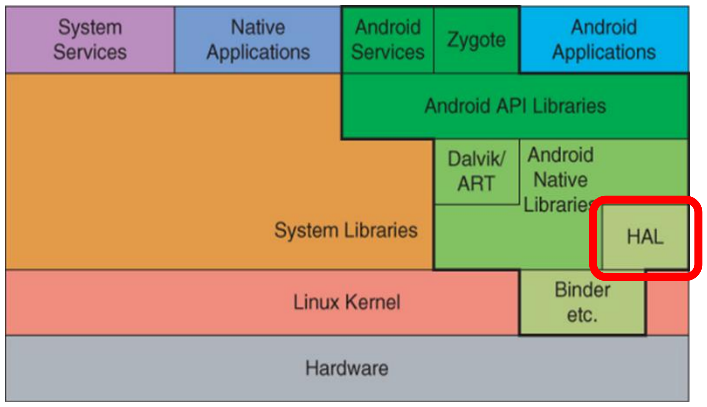
6. Android Keystore system

The Android Keystore System is a secure container provided by Android for managing and storing cryptographic keys. It allows developers to generate, store, and use keys for cryptographic operations—such as encryption, decryption, signing, and verification—without having to expose the actual key material to the application layer. Also, the Keystore system lets you restrict when and how keys can be used, such as requiring user authentication for key use or restricting keys to use only in certain cryptographic modes. [14]

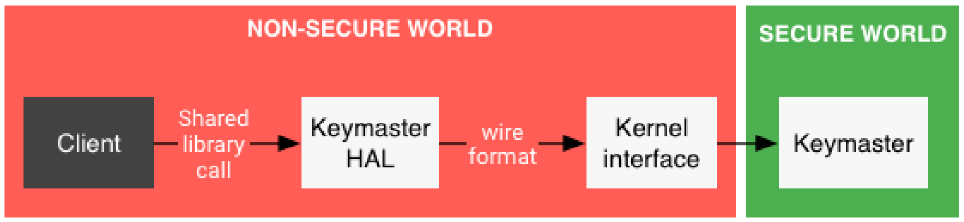
The system enforces security measures in two ways:

1. When an app uses a key from the Android Keystore, the actual key never leaves the secure system area. Instead, data like plaintext, ciphertext, or messages is sent to a separate system process that handles the cryptographic work. This means if the app is hacked, the attacker could use the key for operations but cannot steal the key itself.
2. When the device is backed by a secure hardware like TEE that we talked about in section 4, key material is never exposed outside of secure hardware.

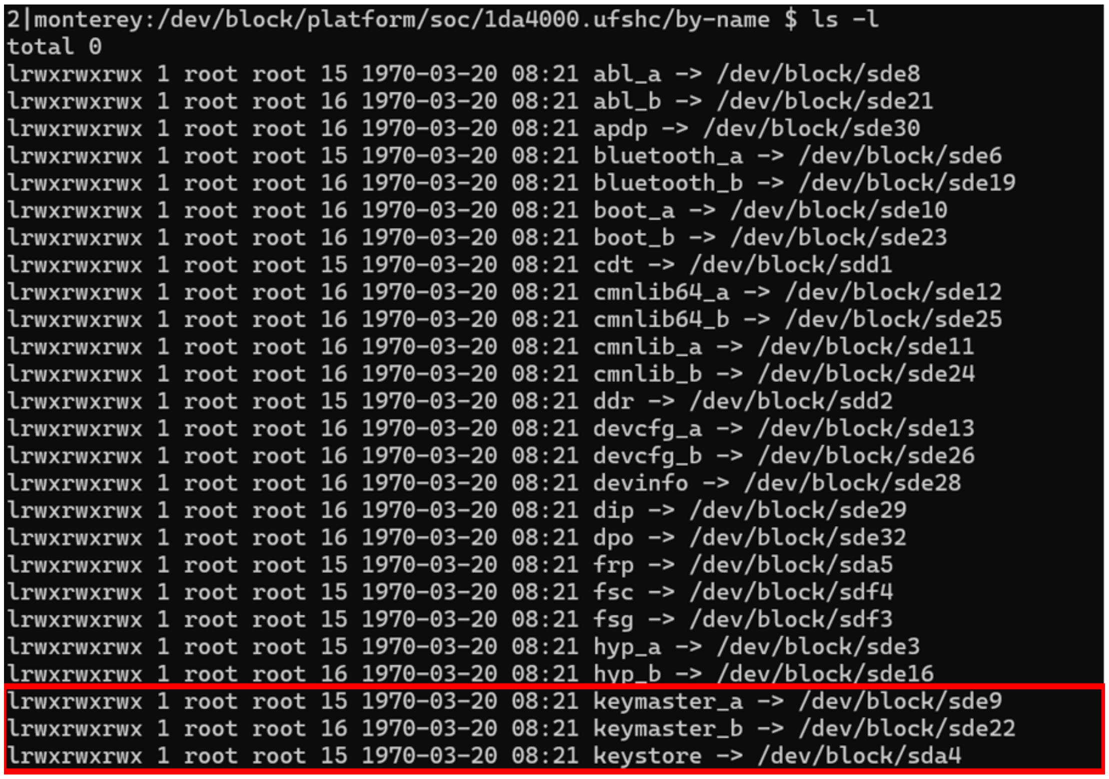
6.1 Keymaster HAL



The system component that manages and enforces secure use of those keys is called the Keymaster HAL. The **Keymaster HAL (Hardware Abstraction Layer)** is a layer within Android that serves as an interface between the Android operating system and the device's underlying hardware-based security modules (like TEE in our case). It allows Android to perform cryptographic operations in a secure and standardized way, regardless of the specific hardware details of the device. Basically, the Keymaster HAL abstracts the hardware specifics, enabling secure key management and cryptographic functions without exposing key material outside the secure hardware.



While exploring the ADB file tree, we found indications that the device uses the Android Keystore system. Below are symlinks to memory blocks found in path **/dev/block/platform/soc/lda4000.ufshc/by-name.**



```
2|monterey:/dev/block/platform/soc/1da4000.ufshc/by-name $ ls -l
total 0
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 abl_a -> /dev/block/sde8
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 abl_b -> /dev/block/sde21
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 apdp -> /dev/block/sde30
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 bluetooth_a -> /dev/block/sde6
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 bluetooth_b -> /dev/block/sde19
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 boot_a -> /dev/block/sde10
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 boot_b -> /dev/block/sde23
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 cdt -> /dev/block/sdd1
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 cmnlib64_a -> /dev/block/sde12
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 cmnlib64_b -> /dev/block/sde25
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 cmnlib_a -> /dev/block/sde11
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 cmnlib_b -> /dev/block/sde24
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 ddr -> /dev/block/sdd2
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 devcfg_a -> /dev/block/sde13
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 devcfg_b -> /dev/block/sde26
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 devinfo -> /dev/block/sde28
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 dip -> /dev/block/sde29
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 dpo -> /dev/block/sde32
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 frp -> /dev/block/sda5
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 fsc -> /dev/block/sdf4
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 fsg -> /dev/block/sdf3
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 hyp_a -> /dev/block/sde3
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 hyp_b -> /dev/block/sde16
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 keymaster_a -> /dev/block/sde9
lrwxrwxrwx 1 root root 16 1970-03-20 08:21 keymaster_b -> /dev/block/sde22
lrwxrwxrwx 1 root root 15 1970-03-20 08:21 keystore -> /dev/block/sda4
```

This symlink that point to **`/dev/block/sde9`** and labeled “keymaster_a” for example—indicates that this partition is being used as part of the secure key management infrastructure. Essentially, it’s a low-level storage area allocated for Keymaster’s operations, ensuring that key material and cryptographic processing are isolated and protected from the rest of the system.

7. Future Research Recommendations

Future research should focus on exploring vulnerabilities and testing the stability of the VR system's security measures. Our analysis has identified key areas for improvement. The following points outline our recommendations for further investigation:

1. **DTLS Vulnerability Assessment:**

   Use TLS-Attacker to test the DTLS implementation on the Oculus Quest. The goal is to find any weaknesses or misconfigurations that attackers could exploit. By simulating various handshake conditions, this assessment will help pinpoint areas where security vulnerabilities lie.

2. **Package Tampering Analysis:**

   Modify the cipher suite list in ClientHello packet to prioritize RSA key exchange, forcing a protocol downgrade in order to decrypt the packets automatically using Wireshark.

3. **ADB Jailbreaking Exploration:**

   Find ways to jailbreak the Android Debug Bridge (ADB) to gain elevated access. This could allow a deeper analysis of restricted system areas, secure storage, and cryptographic components that we were denied permission of before.

4. **Hardware Component Examination:**

   Focus on exploring the device's hardware, with special attention to the Qualcomm Secure Execution Environment (QSEE). Try to exploit QSEE vulnerabilities in order to search for the private key in the hardware.

5. **Exploit the presence of ADB over Wi-Fi:**

Explore possible attacks based on the paper by Yang et al. (2024), *Inception Attacks: Immersive Hijacking in Virtual Reality*.

References

1. Meta Casting guide - https://www.meta.com/help/quest/192719842695017/?srsltid=AfmBOoq8PiJ-we5foYiPXm1h23FhxbPjAn7Sbdxp3Q4SypDRcOHet7LW
2. Rescorla, E., & Modadugu, N. (2012). *Datagram Transport Layer Security Version 1.2* (RFC 6347). Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/html/rfc6347
3. OpenSSL. (2024). *OpenSSL Cryptographic and SSL/TLS Toolkit*. https://www.openssl.org
4. 5Oracle. (2023). *Java Secure Socket Extension (JSSE) Documentation*. https://www.wolfssl.com/products/wolfssl-jni-jsse/
5. Bouncy Castle. (2024). Bouncy Castle DTLS Implementation. https://www.bouncycastle.org/
6. TLS-Attacker. (2023). *TLS-Attacker: Open-Source TLS Security Testing Framework*. https://github.com/tls-attacker/tls-attacker
7. Romero, D. (2017). *Decrypting DTLS Traffic with Wireshark*. https://www.davidromerotrejo.com/2017/07/decrypting-dtls-traffic-with-wireshark.html
8. Meta Quest Firmware files https://cocaine.trade/
9. Khalid, F., & Masood, A. (2022). *Vulnerability analysis of Qualcomm Secure Execution Environment (QSEE)*. Computers & Security, 116, 102628. https://doi.org/10.1016/j.cose.2022.102628
10. Zhao, Q. (2021). *Wideshears: Investigating and Breaking Widevine on QTEE*. 360 Alpha Lab.
11. Beniamini, G. (2016). *Extracting Qualcomm's KeyMaster Keys*. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html

12. Zimperium. (2020). *Multiple Kernel Vulnerabilities Affecting All Qualcomm Devices*. https://www.zimperium.com/blog/multiple-kernel-vulnerabilities-affecting-all-qualcomm-devices
13. https://github.com/darknight1050/quest-bootloader-unlocker/blob/main/README.md
14. https://developer.android.com/privacy-and-security/keystore