



WhatsApp Contacts Security Assessment

Meta Platforms

Version 1.1 – October 11, 2024

©2024 – NCC Group

Prepared by NCC Group Security Services, Inc. for Meta Platforms, Inc. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By

Parnian Alimi
Gérald Doussot
Marie-Sarah Lacharité
Thomas Pornin
Javed Samuel
Eli Sohl

Prepared For

Meta Platforms, Inc.

1 Executive Summary

Synopsis

In May 2024, Meta engaged NCC Group's Cryptography Services practice to perform a cryptography security assessment of selected aspects of the WhatsApp Identity Proof Linked Storage (IPLS) protocol implementation. IPLS underpins the WhatsApp Contacts solution, which aims to store a WhatsApp user's in-app contacts on WhatsApp servers in a privacy-friendly way. WhatsApp servers do not have visibility into the content of a user's contact metadata. The user can later retrieve these contacts from WhatsApp servers to a new device where they re-register their WhatsApp account using the same phone number. The IPLS protocol makes use of Hardware Security Modules (HSMs) and an Auditable Key Directory (AKD).

This program of work consisted of two main phases:

- **Phase 1:** 20 person-days dedicated to protocol design review.
- **Phase 2:** implementation security assessment, further broken down into three stages:
 - **Stage 1:** 30 person-days focusing on HSM Secure Execution Environment (SEE) code, iOS client code, HSM scripts, usage of HSM administration cards, and retesting of Phase 1 design review fixes.
 - **Stage 2:** 15 person-days focusing on Android client code review, gap areas from the HSM SEE code review, HSM scripts, and usage of HSM administration cards.
 - **Stage 3:** 5 person-days dedicated to retesting implementation fixes, and public report creation and review.

Phase 1 was delivered in May, and Phase 2 in September. The program was delivered remotely by five consultants, with a total effort of 70 person-days.

Scope

NCC Group's evaluation included a review of the protocol specification from a security and privacy perspective, and an implementation review of the following Identity Proof Linked Storage components:

- Secure Execution Engine implementation of IPLS for the Marvell and Entrust HSMs.
- iOS and Android clients IPLS implementation.
- HSM Configuration and Deployment (Marvell).
- Middle-tier protocol implementation.

The WhatsApp team provided design documentation, threat model, and code pointers for all the above components.

Limitations

IPLS uses WhatsApp's Key Transparency system, based on an Auditable Key Directory, for authentication of users via their WhatsApp identity keys. Any transparency system has inherent limitations due to its reliance on *trust*. The aim of a transparency system is not to make all attacks infeasible, but to make them sufficiently visible to users and auditors and, where possible, to minimize the amount of trust necessary.

For example, account take-over (ATO) through, e.g., SIM card theft, SIM swapping, or social engineering to obtain an SMS OTP, presents a risk to the WhatsApp Contacts solution during (re-)registration with the Auditable Key Directory. Evidence of such an attack would be visible in the AKD. Further, it is understood that WhatsApp is aware of this risk, and has controls in place to mitigate it. These controls were not in scope for this review, therefore NCC Group did not attempt to identify issues related to these types of threats.



WhatsApp Identity Proof Linked Storage has a critical dependency on an attestation service provided by Cloudflare. This service was not in scope for this program, and thus was not reviewed. Nevertheless, NCC Group provided some analysis of this dependency as exercised by the HSM implementation in section [Protocol Specification Review](#).

Key Findings

During the assessment, NCC Group uncovered several issues. The most notable findings were:

- [HSM-Stored Keys Remain Accessible to the Host for Cryptographic Operations Despite Locking](#): a compromised infrastructure could impersonate Marvell HSMs, decrypt users' secret key material, and from that obtain their private contact metadata.
- [HSM Backup Files Require Secure Deletion](#): the backup files employed for deployment of a Marvell HSM fleet may be used later on to extract the fleet secret keys and compromise all operations, if they are not securely deleted.
- [Rogue Namespaces Allow Split-View Attack](#): clients' secrets could be retrieved by a compromised WhatsApp infrastructure (by malicious or compelled insiders, or other parties), with less observability.
- [Potential Mirror Attacks on Symmetric Encryption](#): a hostile server could send back the client payload to the client instead of the actual response payload from the HSM.
- [Potential Nonce Reuse in Encryption of HSM Session and Client Secret Data](#): This could allow an attacker to subvert the WhatsApp Contacts security protocol to ultimately obtain contact information.
- [Clients Can Provide Valid Signature Without Knowledge of HSM Challenge Value](#): Attackers may be able to impersonate legitimate clients, force the re-use of HSM ephemeral key material and bypass some steps of the protocol.

Retest Status

The WhatsApp team implemented a number of changes to address NCC Group's findings. NCC Group retested these changes at the end of September 2024. Upon completion of the assessment, 13 findings were reported to WhatsApp, along with recommendations. After retesting, and before the solution was rolled out to users, **all 13 findings were found to be fully fixed**.

Other Remarks

All findings are listed in the [Table of Findings](#) alongside with their retest status. NCC Group provides a diagram illustrating its understanding of the architecture of WhatsApp Contacts in the [WhatsApp Contacts Cryptographic Architecture](#) section. NCC Group also included section [Protocol Specification Review](#), which details Phase 1's notes and recommendations regarding the Identity Proof Linked Storage (IPLS) protocol as is described in the *ipls.md* document at commit [56f456e](#) (version 1.1). The consultants also captured notes and observations that did not warrant security findings in section [Implementation Review Engagement Notes](#), including the aforementioned considerations around the external third-party attestation service.



2 Dashboard

Target Data

Name	WhatsApp Contacts
Type	Protocol Specification, Software Service Implementation
Platforms	C, Erlang, Java, Kotlin, Objective-C, Rust, Swift
Environment	Local





Engagement Data

Type	Security Protocol Design And Implementation Review
Method	Code-assisted inspection
Dates	2024-05-13 to 2024-09-25
Consultants	5
Level of Effort	70 person-days, including retesting

Targets

Protocol specification	Design documents, including an outline of the IPLS protocol and a threat model.
Secure Execution Engine (SEE) program	Runs on Entrust and Marvell HSM backends.
HSM configuration and deployment	WhatsApp Contacts design documents, and management scripts for Entrust and Marvell HSMs.
Chatd module	Middle-layer. WhatsApp Contacts service integration only.
WhatsApp iOS and Android clients	WhatsApp Contacts service integration only.

Finding Breakdown














Critical issues	1	
High issues	0	
Medium issues	5	
Low issues	5	
Informational issues	2	
Total issues	13	

Category Breakdown

Cryptography	12	
Data Exposure	1	



Component Breakdown


Android Client	1	
Attestation	1	
Marvell HSM	2	 
Protocol Design	3	  
Secure Execution Environment (SEE)	4	   
iOS Client	2	 

 Critical

 High

 Medium

 Low

 Informational



3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Android Client

Title	Status	ID	Risk
Protocol May Execute with Weaker Forward Secrecy Assurance	Fixed	LL3	Low

Attestation

Title	Status	ID	Risk
Rogue Namespaces Allow Split-View Attack, Unauthorized Client Secret Retrieval	Fixed	2W7	Medium

Marvell HSM

Title	Status	ID	Risk
HSM-Stored Keys Remain Accessible to the Host for Cryptographic Operations Despite Locking	Fixed	MWE	Critical
HSM Backup Files Require Secure Deletion	Fixed	RV9	Medium

Protocol Design

Title	Status	ID	Risk
Potential Mirror Attacks on Symmetric Encryption	Fixed	46P	Medium
Contact Metadata Ciphertext Length Side-Channel	Fixed	V2H	Low
AKD Verification Occurs Too Late	Fixed	922	Low

Secure Execution Environment (SEE)

Title	Status	ID	Risk
Potential Nonce Reuse in Encryption of HSM Session and Client Secret Data	Fixed	P24	Medium
Clients Can Provide Valid Signature Without Knowledge of HSM Challenge Value	Fixed	WHT	Medium
AES-GCM Encryption in the SEE Application is not Constant-Time	Fixed	6RL	Low
Use of the Same Encryption Key for Session and Client Secret Data May Weaken Protocol	Fixed	ML2	Info

iOS Client

Title	Status	ID	Risk
Protocol May Execute with Weaker Forward Secrecy Assurance	Fixed	CN2	Low
Non Constant-Time AES-GCM Implementation For ARM Platforms	Fixed	GHL	Info



4 Finding Details – Android Client

Low

Protocol May Execute with Weaker Forward Secrecy Assurance

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E015746P-LL3

Component Android Client

Category Cryptography

Status Fixed

Impact

An attacker in a privileged network position may be able to decrypt past client secret information if HSM long term keys are compromised in the future.

Description

The HSM SEE system and client engage in a X3DH key agreement protocol to establish a shared secret key. This shared secret is employed by clients to recover and store their own key material which is used to protect client contact information. The X3DH protocol provides [forward secrecy](#); that is, future compromise of long term keys will not result in a compromise of past session keys.

NCC Group identified an issue in the Android client that may allow active adversaries to weaken forward secrecy, so that client secret data could be recovered in the event of a compromise of the long term HSM keys. The consultant team previously identified the exact same [finding "Protocol May Execute with Weaker Forward Secrecy Assurance"](#) in the iOS client, as part of Phase 2 Stage 2 of the program of work.

In its Server Hello message, the HSM SEE sends its long term public key `hk`, its ephemeral public key `hek`, and a signature for each of these values to the client. Both keys are signed with the HSM fleet key, which is the private counterpart of `hk`. A new `hek` is generated for each interaction with a client.

The client validates that both `hk` and `hek` are signed by the HSM fleet key using a hardcoded value for the fleet key. The client does not attempt to determine whether `hk` and `hek` correspond to respectively the long term and ephemeral/session public keys. An attacker may therefore intercept a Server Hello message, and replace the (`hk`, `hek`) fields to contain (`hk`, `hk`), or (`hek`, `hek`), or (`hek`, `hk`) instead. Tampering with the fields this way will force the client to derive a different session key than what was expected by the HSM server. Therefore, the protocol will abort (but may restart automatically, and this time without requiring adverse intervention, if the resulting error is a retryable event).

However, the client will have sent encrypted information that may be decryptable if the HSM long term keys are leaked later, in contradiction with the forward secrecy guarantees of X3DH.

The issue is illustrated in the `assertIplsHsmIdentity()` function in the `ClientIplsHandshakeUtils.kt` source file, where the implementation limits itself to checking the signatures of `hk`, and `hek`, identified as `hkPub`, and `hekPub` respectively in the Kotlin implementation below :

```
fun assertIplsHsmIdentity(  
    iplsServerHelloPayload: Ipls.IplsServerHelloPayload,  
    hsmFleetKey: CurvePublicKey  
): HSMAssertionResult {  
    val hekPub: ByteArray = iplsServerHelloPayload.hekPub.toByteArray()
```



```

val hkPub: ByteArray = iplsServerHelloPayload.hkPub.toByteArray()
val hkPubSig: ByteArray = iplsServerHelloPayload.hkKeySignature.toByteArray()
val hekPubSig: ByteArray = iplsServerHelloPayload.hekKeySignature.toByteArray()

val isHkPubSigValid: Boolean = CryptoUtils.verifySignature(hsmfleetKey, hkPub, hkPubSig)
if (!isHkPubSigValid) {
    // SNIP
    return HSMAssertionResult.Failure(
        SERVER_HELLO_PAYLOAD_ASSERTION_ERROR.INVALID_HSM_HK_PUB_SIGNATURE_ERROR)
}
}

val isHekPubSigValid: Boolean = CryptoUtils.verifySignature(hsmfleetKey, hekPub, hekPubSig)
if (!isHekPubSigValid) {
    // SNIP
    return HSMAssertionResult.Failure(
        SERVER_HELLO_PAYLOAD_ASSERTION_ERROR.INVALID_HSM_HEK_PUB_SIGNATURE_ERROR)
}
}

return HSMAssertionResult.Success
}

```

Recommendation

Consider validating that `hk`, and `hek` are at least different and that they are in their correct fields. Evaluate whether sending `hk` is actually required, as the client already has this key hardcoded as the fleet key.

Location

Function `verifyIplsIdentityOnServerHelloPayload()`, in file `ClientIplsHandshakeUtils.kt`

Retest Results

2024-09-24 – Fixed

NCC Group reviewed the changes WhatsApp made in code changes [D62240490](#) and [DD63463920](#) to address this issue. The client implementation now validates that the `hk` and `hek` fields have different values and that the `hk` public key equals the hard-coded HSM fleet key. These changes fully mitigate this issue.



The Cloudflare attestation signature is over the concatenation of the epoch, digest, Cloudflare timestamp, and Cloudflare namespace fields. The namespace appears to be a management feature, used by Cloudflare to differentiate tenants. According to the WhatsApp team, different namespaces are intended to allow recovery from corruption in the AKD Merkle tree: WhatsApp could reset the tree in the event of a bug or corruption. When validating the attestation, the HSM checks that the namespace consists of a hard-coded prefix, and one-byte integer, in function `check_cloudflare_namespace_format()` in file `auditor_signature_verifier.rs`:

```
pub fn check_cloudflare_namespace_format(namespace: &str) -> bool {
    // TODO(T195729761): Remove checks for non-prod namespaces before launch.
    // Check namespace starts with expected base for staging, prod (test namespaces), or
    // ↳ prod.
    let formats = [
        CLOUDFLARE_NAMESPACE_BASE_FORMAT,
        CLOUDFLARE_TEST_NAMESPACE_BASE_FORMAT,
        CLOUDFLARE_STAGING_NAMESPACE_BASE_FORMAT,
    ];

    formats.into_iter().any(|format| {
        namespace
            .strip_prefix(format)
            .is_some_and(|v| u8::from_str(v).is_ok())
    })
}
```

WhatsApp can request the creation of different namespaces using the “Create namespace” Cloudflare API. The resulting namespaces share the same Cloudflare public key. A compromised or coerced WhatsApp infrastructure could theoretically create a new namespace name that would pass HSM validation. Such a namespace could be backed by a concurrent rogue AKD infrastructure, allowing to potentially target some users in order to obtain their secret contact metadata information.

There do not appear to be preventative controls. However, users can monitor the list of namespaces, and changes to it, using the Cloudflare API.

Further note that currently, there is a `TODO` comment for WhatsApp to remove non-production namespaces from the hardcoded list of allowed namespaces.

Recommendation

- Consider augmenting the `DecryptedKeyVault` structure to include the Cloudflare namespace along with the user secret in the HSM database. Then, when handling client requests, have the HSM verify that the stored namespace matches the namespace from the signed attestation. Note, however, that if the AKD tree is ever reset and there is a transition to a new namespace for a legitimate reason, then users would not be able to retrieve their secrets from the previous namespace; they would need to re-register in the new namespace.
- Encourage users and auditors to monitor the list of namespaces. Create an expectation that new namespaces are exceptional events, and must be accompanied by an announcement.

Location

- Cloudflare “Create namespace” and “Create a new epoch” APIs



-
- Function `check_cloudflare_auditor_signature_params()` in SEE code, file `auditor_signature_verifier.rs`

Retest Results

2024-09-26 – Fixed

While WhatsApp did not implement the recommended mitigation of storing namespace with user records (because it would prevent the use of namespaces for disaster recovery), WhatsApp emphasized that all namespaces are publicly visible (<https://akd-auditor.cloudflare.com/namespaces>) and permanently stored on Cloudflare’s infrastructure. WhatsApp will leverage this public visibility as the primary deterrent to the creation of rogue namespaces.

WhatsApp also documented a draft of the internal protocol to follow in the event of a tree reset, when a new namespace must be provisioned. This protocol involves alerting Cloudflare, sharing a public post explaining the event, disabling the previous namespace using the Cloudflare “Patch namespace” API, and creating a new namespace, numbered sequentially relative to the previous one. Any namespace change outside of this protocol can be viewed as suspicious.

NCC Group also reviewed the changes WhatsApp made in code diff `D62551671`, which removed the non-production namespace formats in `check_cloudflare_namespace_format()`.

These changes partially address the finding, since the proposed mitigation to prevent the attack was not implemented. However, the presented draft protocol would make the attack sufficiently visible, in line with WhatsApp’s threat model for WhatsApp Contacts, where split-view attacks are not necessarily prevented, but are detectable.



6 Finding Details – Marvell HSM

Critical

HSM-Stored Keys Remain Accessible to the Host for Cryptographic Operations Despite Locking

Overall Risk Critical

Impact High

Exploitability High

Finding ID NCC-E015746P-MWE

Component Marvell HSM

Category Cryptography

Status Fixed

Impact

The fleet shared keys (AES and X25519) remain usable for cryptographic operations by the host, despite the locking policy. This permits a compromised infrastructure to impersonate HSMs, decrypt users' secret key material, and from that obtain their private contact metadata.

Description

The deployment procedure for the Marvell HSMs entails generating the fleet keys (AES and X25519) on an initial HSM, then taking a backup and restoring the backup files on all other HSMs in the fleet. When all HSMs have been configured with the intended SecureMachine application (SM-App), they are locked by enabling their `HSM_POLICY_HSM_LOCK` policy. The *Marvell LiquidSecurity HSM Adapter SDK User Guide 2.09-0702* specifies that once the policy is enabled, a number of operations are no longer feasible on the HSM from the host, including modifying or exporting keys, cloning, creating backups, or creating/deleting/modifying SM-Apps. All such operations can still be performed by the SM-App itself. The goal of locking down the HSM is to ensure that only the installed SM-App will ever be able to access the keys.

However, the `HSM_POLICY_HSM_LOCK` policy, when enabled, still allows the host to “[perform] all crypto operations”. The policy prevents the host from exporting keys, but operations such as AES/GCM encryption and decryption with that key are still possible. All the host needs to do is to authenticate as the `crypto_user` user, using the relevant password, which the host necessarily knows since it provides it to the SM-App upon startup. This is in direct violation of the security properties that were expected from the HSM (and the only reason why HSMs are used).

Recommendation

Meta pointed out the existence of a key attribute that can be optionally set on cryptographic keys, called `OBJ_ATTR_SECURE_MACHINE_ONLY`. The exact meaning of that attribute is not documented (the “modifiable key attributes” section of the Marvell guide states that its ID is 268435957 and that its value is a Boolean, but the “description” field is empty). Inquiries with Marvell are ongoing to obtain more information. If that attribute, when set, prevents any use of a cryptographic key by the host, limiting access to the SM-App, then that attribute, combined with the locking policy, would provide an effective mitigation for the issue described here.

A potential alternative mitigation would be to enforce a systematic key derivation step. Given the shared “AES key”, the SM-App would systematically, upon startup, export that key into raw bytes, and derive from these bytes the actual fleet key with a cryptographically secure key derivation function. The IND-CCA2 security of AES intuitively means that even an



attacker able to perform arbitrary encryption and decryption operations with a key still obtains no usable information on the key itself; the derived key would then be inaccessible to the host, but still easily obtained by the SM-App. The key derivation would have only to be performed at startup time, hence with negligible overhead for overall performance. Note that cryptographic operations performed by the HSM include not only encryption and decryption, but also key derivation (with `Cfm2DeriveKey()` and similar functions), so this mitigation would be successful only by using a key derivation process which is “cryptographically different” from the ones supported by the HSM; this notion of “difference” is hard to qualify formally, but the [HKDF](#) key derivation used with a hash function that the HSM hardware does not support (e.g. [BLAKE2](#)) would *informally* be enough.

If a mitigation is found for AES keys, then it can also be used for the X25519 private key. In the current deployment procedure, the X25519 key is generated under that type; however, since the X25519 and similar operations (X3DH, XEdDSA) are performed in the SM-App with pure Rust implementations, not leveraging the HSM’s accelerated hardware, then it is not really needed that the HSM recognizes that X25519 key as being an X25519 key; any mechanism that can generate and store 32 random bytes would be enough. Thus, the X25519 key could be stored in the HSM as an “AES-256 key”; the actual X25519 private key could then be obtained by applying some key derivation (as described above) and then the “clamping” process that clears four specific bits and sets another one (see [RFC 7748](#), section 5, function `decodeScalar25519()`).

Location

Marvell Fleet Commission Process, step 2.i

Retest Results

2024-09-24 – Fixed

WhatsApp indicated that they will not use Marvell HSMs in production, until the vendor can support mitigation via firmware update. This avoids the risk highlighted in this finding, and therefore fixes the issue.



HSM Backup Files Require Secure Deletion

Overall Risk Medium

Impact High

Exploitability Low

Finding ID NCC-E015746P-RV9

Component Marvell HSM

Category Data Exposure

Status Fixed

Impact

If the backup files used for deployment of a Marvell HSM fleet are not securely deleted, then they may be used later on to extract the fleet secret keys and compromise all operations.

Description

The *Marvell Fleet Commission Process* describes the deployment process envisioned by Meta for HSM fleets using the Marvell LiquidSecurity HSMs. In summary, an initial HSM is configured with the relevant SM-App and generates the fleet keys; then a backup is created from that HSM, and the backup files are transported (with `scp` or a similar mechanism) to the hosts for all other HSMs in the fleet, where the backup is restored. In that way, the fleet keys are transported to all HSMs. At the end of the ceremony, all HSMs are locked (`HSM_POLICY_HSM_LOCK` is enabled) and the backup files are deleted. No specific deletion process is specified; presumably, a simple `rm` command is used.

It is imperative for security that no attacker gets hold of these backup files; indeed, the attacker could restore the backup files on another Marvell HSM, and *not* perform the locking process; instead, the attacker could just export the fleet keys from their non-locked HSM. Simple file deletion is not enough, since it does not actually destroy the file data; it only ceases to reference that data, and marks the relevant disk space as reusable.

Recommendation

Modern hard disk hardware, especially SSDs, makes it difficult to securely delete a file. Instead, the deployment procedure should make it so that the files are never stored to a physical storage device. This is doable in the following way:

- Ensure that all host machines are physical systems (not virtual machines).
- Disable virtual memory (swap) support. On Linux system, this can be done with some `swaponoff` commands; the `free` command should report a total swap space size of zero.
- Create some RAM-based filesystem, i.e. some “disk” space which is really backed by RAM. On Linux, the `tmpfs` filesystem type can be used.
- Verify that no automatic system-wide backup system is in place and would cover the directory on which the RAM-based filesystem is mounted.
- All operations on HSM backup files (creation, copy, restoration) should write the files only in the RAM-based filesystem, on all the HSM host machines.
- At the end of the ceremony, all host machines should be rebooted. A restart ensures the destruction of the contents of all the RAM-based filesystems; moreover, the boot-time memory wiping normally performed by the BIOS ensures that the old RAM contents are properly overwritten.

Location

Marvell Fleet Commission Process, step 2.g



Retest Results

2024-09-24 – Fixed

WhatsApp indicated that they will not use Marvell HSMs in production, until the vendor can support mitigation via firmware update. This avoids the risk highlighted in this finding, and therefore fixes the issue.



7 Finding Details – Protocol Design

Medium

Potential Mirror Attacks on Symmetric Encryption

Overall Risk Medium

Impact Medium

Exploitability Medium

Finding ID NCC-E015746P-46P

Component Protocol Design

Category Cryptography

Status Fixed

Impact

Since the same symmetric key (session key) is used to encrypt payloads from both the client and the HSM, a hostile server may send back the client payload to the client instead of the actual response payload from the HSM.

Description

In the protocol specification (*ipls.md* version 1.1), the X3DH key exchange (optionally augmented with the Kyber768-based post-quantum KEM) results in a session key *SK*. That key is used to encrypt the client opaque payload (`IplsClientRequestOpaquePayload` protobuf message) but also the server's response (`IplsClientSecretData` protobuf message), in both cases with an IV chosen randomly by the server. Since the same key is used and the IV is arbitrary, a malicious intermediate (the host server) could replace the server's response with a copy of the client's payload, and the decryption would still work.

In the messages as currently defined in *ipls.proto*, an `IplsClientSecretData` object has a single field of tag value 1 and type "map":

```
message IplsClientSecretData {
  map<string, IplsClientSecretKeyList> client_secret_map = 1;
}
```

However, in the client payload (`IplsClientRequestOpaquePayload`) the field of tag 1 has type `IPLSRequestType`, which is an enumeration. As described in the [protobuf documentation](#), enumerations are serialized into a VARINT, while maps use a (repeated) sub-message for each map entry, the latter using LEN objects. Therefore, when the client is faced with a copy of its own payload and tries to decode it as an `IplsClientSecretData`, it *should* report an error, thereby preventing the substitution from having any deleterious consequence beyond a reported failure. However, this relies on details of the payloads in both directions, and on the behavior of the protobuf decoders when faced with abnormal data. Protobuf decoders are encouraged to be tolerant of encoding deviations, if only to smoothly support protocol upgrades. Thus, appropriate error detection when the client's payload is mirrored is never guaranteed and is fragile with regard to ulterior protocol enhancements.

Recommendation

Since the session key *SK* is derived using `HKDF`, which has a variable-size output, that `HKDF` call could be used to produce not *one* session key, but *two* session keys `SK_client` and `SK_server`, to be used for encrypting payloads from, respectively, the client and the server.

Another possible method, applicable here since AES/GCM is used for the encryption, is to enforce specific IV values, e.g. 0 for the client payload and 1 for the server answer. Fixed IV values can be used securely because the session key is freshly generated; the X3DH key exchange, coupled with ephemeral public keys, ensures that every handshake results in its



own session key, and no two different messages will be encrypted with the same key+IV pair. This scheme can naturally extend into a multi-message protocol, with the client using even numbers in sequence (0, 2, 4, 6...) while the server uses odd numbers for its responses (1, 3, 5, 7...).

Some inspiration can be taken from TLS, which uses both methods: [two separate symmetric keys](#) are derived from the handshake material, for the two traffic directions, and within each direction, each record payload uses a nonce built on [the record sequence number](#). The use of an explicit sequence number is needed in multi-message protocols to avoid attacks based on replay or reordering of messages.

Location

ipls.md IPLS specification document

Retest Results

2024-08-28 – Fixed

NCC Group reviewed design changes made in response to this finding during phase 2 of the WhatsApp Contacts security assessment project. The HKDF primitive is used to produce 64 bytes of key material. The first 32 bytes are used for the client session key, while the last 32 bytes are used for the server session key. These design changes fully mitigate this issue.



Contact Metadata Ciphertext Length Side-Channel

Overall Risk	Low	Finding ID	NCC-E015746P-V2H
Impact	Low	Component	Protocol Design
Exploitability	Medium	Category	Cryptography
		Status	Fixed

Impact

An attacker who can observe the resulting encryption of contact metadata may be able to infer the length of the plaintext prior to encryption, and/or what metadata fields were actually populated. This may assist in turn in mounting further attacks, including guessing correct plaintext values.

Description

WhatsApp Contacts adds a `metadata` attribute to the Usync Contact Protocol for each user contact. The attribute value is encrypted using the AES-GCM cipher block mode of operation, using the client secret key. According to the *Multi Device Native Contacts Management* document, the plaintext value before encryption consists of the following fields:

Contact Name: This is the name that is stored on the user's address book. This field will be in two parts : `first_name` and `last_name`.

Business Name: Business name associated with this contact.

Version: Internal representation of which version this definition belongs to.

Sync Policy: This tells us whether this contact is synced back to the user's address book or not.

Apart from the length of the tag, and IV (if serialized, the most likely case), the AES-GCM ciphertext length is identical to the plaintext length. Attackers may therefore infer some information about the plaintext, based on the size of the ciphertext. Contacts information, including encrypted metadata, is stored on a server outside of the HSM SEE environment; it is therefore accessible to attackers with a presence on these servers, and possibly other systems such as backup hosts.

There does not appear to be any formal description of the serialization of the plaintext value before encryption. The serialization format may facilitate or reduce the effectiveness of inferring information about the plaintext. For instance, it is unknown whether the lack of a populated field, e.g., Business Name, would result in an empty plaintext field ("`<FieldName/>`", or "`<FieldName></FieldName>`") or no field at all (""); a ciphertext length with a populated Business Name could increase by at least twice the tag name length + 5 (the size of all delimiters) in the worst case scenario, and may help attackers in deriving information about the plaintext metadata. It is assumed that the last two fields, Version and Sync Policy, are relatively static in size, and their lengths can probably be subtracted when attempting to guess the sizes of other fields.

Note that the use of AES-GCM encryption does not appear to be documented; the WhatsApp team communicated this information over Workplace chat.



Recommendation

Consider formally describing the serialized metadata format prior to encryption. Consider padding all plaintext fields to their maximum allowed size, even when not populated by the user, prior to encryption.

Consider documenting what cipher and block mode of operation are used to encrypt contact metadata, and any other relevant details.

Location

Multi Device Native Contacts Management document

Retest Results

2024-08-28 – Fixed

NCC Group reviewed design changes made in response to this finding during phase 2 of the WhatsApp Contacts security assessment project. Limits on the first name, last name, and business name sizes were introduced, as well as padding to ensure each contact's metadata has a fixed size when encoded. These design changes fully mitigate this issue.



AKD Verification Occurs Too Late

Overall Risk	Low	Finding ID	NCC-E015746P-922
Impact	Medium	Component	Protocol Design
Exploitability	Low	Category	Cryptography
		Status	Fixed

Impact

When the protocol is implemented, the data required at steps 11-17 will not be available and the protocol specification will need to be modified.

Description

In the protocol specification (`ipls.md` version 1.1), the initial client message (`xwa2_ipls_client_init`) contains the randomly chosen session ID; the server complements that message with a fresh AKD lookup proof (and associated data) that contains the current active public keys for the user. However, that proof is not verified immediately. The server responds with `XWA2IplsClientInitResponse`, which contains the encoded `IplsServerHelloPayload` from the HSM. Since the transmission of the response to the client, and reception of the next message, may take some time, the HSM stores the current session data in the “HSM session table” which is described in step 3. That table contains the user identifier (`account_jid`), the session ID, the ephemeral key pairs created by the HSM for this session (both Curve25519 and Kyber768), and a time stamp. The AKD lookup proof is not apparently stored in that table. However, after the client sends the next message (`xwa2_ipls_client_hello`, containing in particular the client handshake material and the client request payload), the HSM completes the handshake, and only then verifies the AKD lookup proof and associated elements:

- `cloudflare_signature` is verified in step 11;
- presence of the purported client public key `cik_pub` in the proof is checked in step 15;
- the proof itself is verified in step 16;
- `cloudflare_timestamp`, as received, is compared to the value associated with the stored record in step 17.

Since the `cloudflare_signature`, `cloudflare_timestamp` and AKD lookup proof were not saved in the HSM session table, they are not available to the HSM at this point, making these steps unimplementable.

Recommendation

The issue can be fixed by either augmenting the HSM session table with the relevant elements, so that they are available when performing steps 11-17, or by making the corresponding verifications earlier. From a perspective of resilience against denial-of-service attacks, an earlier verification is advisable: for all normal requests, the proof will have to be verified anyway; for flawed requests, an AKD proof verification failure, if detected earlier, would avoid the costs of generating ephemeral key pairs in the HSM. Moreover, the extra storage space required in the HSM session table is lower if the AKD proof verification is performed earlier, as follows:

- The client may send `cik_pub` in the `xwa2_ipls_client_init` message.
- When receiving the client initial message, augmented with the AKD proof and other data by the server, the HSM can verify the proof right away, including checking



`cloudflare_signature` and validating that the provided `cik_pub` is indeed part of the proof.

- The session data should then be augmented with a copy of `cik_pub` (which the HSM has now verified to be correct) and of `cloudflare_timestamp`.
- In the client handshake material, `cik_pub` does not need to be sent again, since the server already has it.

`cik_pub` and `cloudflare_timestamp` are only a few dozen bytes in length; this should not prevent the possibility of keeping the HSM session data in HSM RAM only, which is advisable for both performance reasons (RAM access is cheaper than using an external database) and also for security (rollback attacks do not apply to the HSM internal RAM).

Yet another possible strategy is to delay the fetching of the AKD proof and inclusion in the client message by the server until the client sends its handshake material and the payload. In that case, `lookup_proof`, `epoch`, `root_hash`, `account_jid`, `cloudflare_namespace`, `cloudflare_timestamp` and `cloudflare_signature` are to be added as extra fields in the `IplsClientHelloPayload` message, and not in the `IplsClientInit` message.

Location

ipls.md IPLS specification document

Retest Results

2024-08-28 – Fixed

NCC Group reviewed design changes made in response to this finding during phase 2 of the WhatsApp Contacts security assessment project. As explained by the WhatsApp team, the AKD proof fetch step was misplaced in the design specification (whereas the implementation had the correct sequence of steps). The AKD proof fetch step was placed in the correct location, and specifically after client sends its handshake material and the payload. These design changes fully mitigate this issue.



8 Finding Details – Secure Execution Environment (SEE)

Medium

Potential Nonce Reuse in Encryption of HSM Session and Client Secret Data

Overall Risk	Medium	Finding ID	NCC-E015746P-P24
Impact	High	Component	Secure Execution Environment (SEE)
Exploitability	Medium	Category	Cryptography
		Status	Fixed

Impact

An attacker may be able to recover the GHASH authentication key allowing them to forge encrypted sessions and client secret data which will pass authentication. This may allow an attacker to subvert the WhatsApp Contacts security protocol to ultimately obtain contact information.

An attacker may also be able to recover information about the encrypted session and user data; if the nonce repeats under the same key, then the result of the XOR operation on the affected ciphertexts is equal to the result of the XOR operation on the matching session plaintexts. If an attacker already knows any part of either plaintext, then they can recover the corresponding part of the other plaintext, up to and including the full message.

Additionally, the implementation does not comply with some of the NIST SP 800-38D standard requirements.

Description

The HSM SEE system sets up session information, including HSM challenge and ephemeral keys, when receiving an `InitClientRequest` message in its `InitHandler` component. This information is encrypted using the HSM fleet AES key for storage and later use. The encryption routine observes certain limits, such as how large the plaintext can be. It however does not place an apparent limit on how many times the encryption function can be invoked using the same AES key.

The `fleet_aes_encrypt()` function in the `hsm_context.rs` source file is responsible for encrypting session information. It does not, nor do its callers, track the number of encryptions per key:

```
pub fn fleet_aes_encrypt(&self, plaintext: &[u8]) -> Result<AesGcmEncryptedData> {
    #[cfg(profilite)]
    let _p = profilite::start_guard("HsmContext::fleet_aes_encrypt");

    crate::hsm::aes::encrypt(self, plaintext, self.fleet_aes_key.as_slice())
}
```

The `encrypt()` function in the `aes.rs` source file performs the actual encryption. The 12-byte nonce structure is fully populated with a newly generated random value.

```
// Encrypt plaintext using AES-GCM-256.
pub fn encrypt(
    hsm_context: &HsmContext,
    plaintext: &[u8],
```



```

    key: &[u8],
) -> Result<AesGcmEncryptedData> {
    #[cfg(profilite)]
    let _p = profilite::start\_guard\("aes::encrypt\_to\_components"\);

    ensure!(
        plaintext.len() <= AES_256_GCM_MAX_ENCRYPTION_SIZE_BYTES,
        "Overlimit 128 AES-GCM plaintext"
    );
    let cipher = Cipher::aes\_256\_gcm\(\);
    assert_eq!(cipher.key_len(), key.len());

    // AES IV.
    let iv_len = cipher.iv_len().unwrap();
    debug_assert_eq!(iv_len, AES_256_GCM_IV_SIZE_BYTES);

    let mut iv = \[0u8; AES\_256\_GCM\_IV\_SIZE\_BYTES\];
    hsm\_context.rng\_try\_fill\_bytes\(&mut iv\)?;

    // AES ciphertext.
    let mut encrypter = Crypter::new(cipher, Mode::Encrypt, key, Some(&iv))?;
    let mut ciphertext = vec![0u8; plaintext.len() + cipher.block_size()];
    let mut ciphertext_len = encrypter.update(plaintext, &mut ciphertext)?;
    ciphertext_len += encrypter.finalize(&mut ciphertext[ciphertext_len..])?;
    ciphertext.truncate(ciphertext_len);

    // AES tag.
    let mut tag = [0u8; AES_256_GCM_TAG_SIZE_BYTES];
    // Output tag is sized to fill entire passed array.
    // 16 bytes recommended by documentation for AES-GCM.
    encrypter.get_tag(&mut tag)?;

    Ok(AesGcmEncryptedData {
        ciphertext,
        iv,
        tag,
    })
}

```

NCC Group further discovered that the AES key used to protect HSM session information with clients is also used to protect HSM client secret data records (see [finding "Use of the Same Encryption Key for Session and Client Secret Data May Weaken Protocol"](#)).

Specifically, the `KeyVault` structure `to_xdb()` function calls the same `fleet_aes_encrypt()` function described above, thus reusing the same AES key.

As this key is used for two different functions, there is an increased risk that the implementation will generate and use the same nonce/IV, especially with WhatsApp's large (2 billion) user population. For AES in the GCM mode of operation, re-using the same nonce with a given key is catastrophic. It allows the recovery of information about the plaintext values for the two corresponding ciphertexts encrypted with the same nonce and key. More importantly, one can derive the GMAC key and forge authentication tags for arbitrary encrypted information, which in turn adversely impacts the security guarantees of the WhatsApp Contacts protocol. This is also a deviation from the [NIST SP 800-38D standard](#), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", which states the following in section 8.3:



The total number of invocations of the authenticated encryption function shall not exceed 2^{32} , including all IV lengths and all instances of the authenticated encryption function with the given key.

From a standards compliance perspective, assuming 2 billion users, this invocation limit (2^{32} divided by 2 billion \approx 2.15 invocations) will be nearly reached if each user performs one secret data store operation – that is, one (1) session, plus one (1) user secret data encryption equals two (2) invocations. From a security perspective, it would take approximately 165,705 invocations per user to have a 50% chance of collision.

Note that an attacker in a privileged position (i.e. with a direct connection to the HSM fleet, thus bypassing rate-limiting controls) may increase the likelihood of IV collisions, for instance by generating a large number of arbitrary sessions. Given this scenario, the number of users, and the threat model including WhatsApp as a malicious party (possibly by being compelled to act maliciously), the overall risk rating was raised from low to medium.

Recommendation

At the very least, do not encrypt more than 2^{32} plaintexts under a given key across any number of HSMs, as required by NIST. A new key should be generated before reaching this limit, and used for subsequent encryption operations. This will likely require maintaining multiple key versions, to be able to decrypt records encrypted with previous versions of the HSM fleet key.

Location

Function `fleet_aes_encrypt()` in file `hsm_context.rs`

Retest Results

2024-09-25 – Fixed

2024-09-24: NCC Group reviewed the changes WhatsApp made in code change `D62047580` in response to this finding. The implementation of the `fleet_aes_encrypt()` function now derives a per-user key using HMAC; the key is derived as HMAC(AES fleet key, user phone number). This derived key is used to encrypt both client secret data and client session data, with a freshly generated IV at each invocation, thus greatly reducing the probability of nonce reuse. While it is unlikely that any user will perform 2^{32} (about 4.3 billion) operations, there are no counters keeping track of how many times each key is used. The possibility of an insider attack artificially generating new sessions to increase the risk of IV collision exists. Therefore, this finding is considered “partially fixed”.

2024-09-25 update: WhatsApp decided to replace AES-GCM with AES-GCM-SIV, a nonce-misuse-resistant scheme, following NCC Group retest feedback. The HMAC solution, augmented with AES-GCM-SIV fully addresses the original issue, without introducing further weaknesses.



Clients Can Provide Valid Signature Without Knowledge of HSM Challenge Value

Overall Risk Medium

Impact Low

Exploitability High

Finding ID NCC-E015746P-WHT

Component Secure Execution Environment (SEE)

Category Cryptography

Status Fixed

Impact

An attacker may demonstrate knowledge of the HSM challenge string without access to it, in order to mount further attacks against the protocol. Specifically, they may be able to impersonate legitimate clients, force the re-use of HSM ephemeral key material and bypass some steps of the protocol.

Description

When receiving an `initClientRequest` client message, the HSM SEE system generates a random challenge string and provides it to the client in its response. Upon reception of the response, the client generates an ephemeral key as part of its X3DH key agreement protocol with the HSM SEE and signs this HSM challenge with this key. The client then submits its HSM challenge signature (along with other information, including its ephemeral public key) in its next message (`FinishClientRequest`) to the HSM SEE. When processing this message, the HSM SEE retrieves the HSM session information, which includes the challenge string for the client, and verifies the client signature with the client ephemeral key and challenge string. If the signature is not valid, the security protocol is aborted.

The HSM challenge appears to be designed to provide several security properties to the protocol, including retrieving the correct server X3DH key material, ensuring freshness of client requests and authentication of clients.

The HSM does not authenticate the client ephemeral key. Furthermore, when the HSM verifies that the signature of the client is correct, it does not validate that the ephemeral public key used to sign the challenge is not an elliptic curve point of low order. As a result, an attacker can submit signatures that will validate for any HSM challenge string without knowledge of them. Using this weakness, an attacker may attempt to bypass the security properties envisaged by the protocol designers for the use of HSM challenge.

The HSM SEE system validate client signature of HSM challenge data in function `validate_challenge_response()` in file `session_data.rs`:

```
pub fn validate_challenge_response(
    &self,
    cek_pub: PublicKey,
    challenge_response: &[u8],
) -> Result<(), ClientResponseError> {
    let signature: [u8; super::SESSION_CHALLENGE_RESPONSE_LEN_BYTES] =
        challenge_response.try_into().map_err(|err| {
            // TODO(T197786561): Remove value returning in response.
            ClientResponseError::InvalidRequest(FieldInfo::PayloadProtobuf(format!(
                "cek_pub: {cek_pub:?}, challenge_response: {challenge_response:?}, error:
                ↳ {err:?}"
            )))
        })
    )))
```



```

    })?;

    let xed_verifying_key = XedVerifyingKey::from(&cek_pub);
    xed_verifying_key
        .verify(&self.challenge, &signature)
        .map_err(|err| {
            // TODO(T197786561): Remove value returning in response.
            ClientResponseError::InvalidRequest(FieldInfo::PayloadProtobuf(format!(
                "cek_pub: {cek_pub:?}, signature: {signature:?}, error: {err:?}"
            )))
        })
    })
}

```

Signature verification is delegated to the Rust `xeddsa` library. As illustrated above, the HSM SEE implementation does not validate that the client public key is a low order point, before passing it to `xeddsa`.

Recommendation

Validate that the HSM challenge signature public key is not a point of low order and abort the protocol otherwise.

The `xeddsa` library uses the Rust `25519-dalek` crate. The latter has implemented function `is_weak()`, to check whether a point is a low order point, or not. The Meta WhatsApp team has vendored crate `xeddsa`. They could add a call to `is_weak()` as follows, to reject low order points:

```

impl Verify<Signature, [u8; PUBLIC_KEY_LENGTH]> for PublicKey
where
    PublicKey: ConvertMont<[u8; PUBLIC_KEY_LENGTH]>,
{
    // Get EdDSA public key and verify using standard Ed25519 implementation
    fn verify(&self, message: &[u8], signature: &Signature) -> Result<(), Error> {
        // Extract sign from signature.
        let mut signature_bytes = signature.to_bytes();
        let sign = (signature_bytes[SIGNATURE_LENGTH - 1] & 0b1000_0000_u8) >> 7;
        // Clear sign bit from signature.
        signature_bytes[SIGNATURE_LENGTH - 1] &= 0b0111_1111_u8;
        let final_signature = Signature::from_bytes(&signature_bytes);

        // Use the sign to convert.
        let public_key = self.convert_mont(sign)?;

        let verifying_key =
            VerifyingKey::from_bytes(&public_key).or(Err(Error::UnusablePublicKey))?;
        if public_key.is_weak() {
            return Err(Error::WeakPublicKey);
        }
        verifying_key
            .verify(message, &final_signature)
            .or(Err(Error::InvalidSignature))
    }
}

```

Location

Function `validate_challenge_response()` in file `session_data.rs`



Retest Results

2024-09-26 – Fixed

NCC Group reviewed the changes WhatsApp made in code change **D622320422** in response to this finding. The WhatsApp team implemented NCC Group recommendation to reject low order points. These changes fully mitigate this issue.



AES-GCM Encryption in the SEE Application is not Constant-Time

Overall Risk Low

Impact High

Exploitability Low

Finding ID NCC-E015746P-6RL

Component Secure Execution Environment (SEE)

Category Cryptography

Status Fixed

Impact

Non constant-time AES encryption and decryption can be leveraged in cache attacks, a sub-class of timing attacks, to reveal the secret key. An attacker in control of the systems that directly connect to the HSM may use such an attack to extract the AES fleet key, and then compromise the entire fleet.

Description

In the SEE application, when running in an Entrust HSM (nFast API), AES encryption and decryption operations are performed through the *nfast/aes.rs* source file. The key and plaintext are provided as byte array slices, and the operation uses an `openssl::ssym::Crypter` object, which is a wrapper for the OpenSSL library. The HSM hardware provides an hardware-accelerated (and presumably hardened) AES implementation, but it must be invoked explicitly through the nFast core API; it will not be used by the OpenSSL library, which will rely on its software implementations.

The HSM hardware uses an NXP QorIQ T1042 CPU, which relies on four Freescale e5500 cores. Each core implements the PowerPC v2.06 ISA, both in 32-bit and 64-bit modes. On PowerPC systems, OpenSSL may use a number of implementations for AES, GHASH, or the combination of AES and GHASH in AES-GCM, leveraging the relevant optional instruction sets. Unfortunately, the e5500 does not support such instructions, making it impossible to use the optimized implementations that use the hardware opcodes `vcipher` and `vpmsumd` (added in PowerPC v2.07); it does not support either the “AltiVec” SIMD instructions, for which OpenSSL has some [specific code](#). Instead, the used implementations will be classic table-based code; the AES implementation is [assembly-optimized](#) while the GHASH code is in [plain C](#). Both implementations perform dynamic memory lookups at addresses that depend on secret data, which makes them susceptible to cache attacks.

Cache attacks are a subclass of timing attacks, in which the attacker extracts information from a side channel based ultimately on timing measurements; in a cache attack, timing differences come from differing latencies when accessing memory, depending on whether the target value is in cache memory or not. Cache attacks were indeed [first demonstrated on table-based AES implementations](#), and the timing measurement was made over a local ethernet network. The SEE application runs in a similar situation: the application runs in a dedicated hardware system (the HSM) and responds to requests sent by the potential attacker (the non-HSM host) over a local low-latency connection (either a very fast Ethernet link, or a PCI bus, depending on the specific HSM model). While cache attacks are notoriously difficult to set up and require some substantial tuning, they must still be deemed



to apply to this case. In particular, the following features of the WhatsApp Contacts application tend to help the attacker:

- In a server-side attack context, the attacker can be very close (network-wise) to the target hardware system (the HSM), allowing low-noise precise timing measurements.
- HSMs respond to requests but do not have any network ability that is not controlled by the host; this allows the attacker to run experiments discreetly, without triggering external alerts. An HSM may furthermore be “rewinded”, i.e. restarted with a previous state, since it has no real internal permanent storage¹, allowing an attacker to repeat each experiment for more precise measurements.
- The HSM runs only the SEE application, which is furthermore mono-threaded, thus with a predictable memory access pattern, which allows a precise setup of the cache state before the actual measurement.

Recommendation

The recommended mitigation is to use the nFast core API, instead of OpenSSL, to perform all AES-GCM operations in the SEE application. This API invokes the hardware accelerator which Entrust has joined to the T1042 CPU, and which will provide a more secure operation (indeed, this module is covered by some [FIPS 140-2 and 140-3 certifications](#)). The hardware accelerator should also yield better encryption performance, at least for long messages.

Alternatively, if the nFast core API turns out to be inappropriate for use in the SEE application, e.g. for key derivation reasons, then a constant-time software implementation may be used. There are several implementation strategies that can compute both AES and GHASH without lookup tables, or at least using only non-secret addresses for any table access. One example is the [BearSSL library](#), in particular its [aes_ct_ctr](#) (for AES in CTR mode) and [ghash_ctmul](#) (for GHASH) implementations. Compared to table-based implementations, a slowdown by a factor of about 3 or 4 can be expected, though that should be measured on the actual hardware, and may vary depending on the encrypted plaintext size. Another constant-time implementation, in pure Rust, can be obtained with the [aes-gcm crate](#); indeed, its GHASH implementation is a direct translation of BearSSL’s “ghash_ctmul32” code, and its AES code is built along the same “bitslicing” mechanism.

Location

File `aes.rs`, lines 48-51

Retest Results

2024-09-25 – Fixed

NCC Group reviewed the changes WhatsApp made in code change [D62053155](#) in response to this finding. The WhatsApp team implemented one of the recommended NCC Group libraries, and specifically the `aes-gcm` crate. All AES-GCM operations now use this crate. These changes fully mitigate this issue.

1. Entrust HSMs embed a small amount of NVRAM which could be used to detect rewinding attempts, but it does not appear to be used in the current SEE application.



Use of the Same Encryption Key for Session and Client Secret Data May Weaken Protocol

Overall Risk	Informational	Finding ID	NCC-E015746P-ML2
Impact	High	Component	Secure Execution Environment (SEE)
Exploitability	Undetermined	Category	Cryptography
		Status	Fixed

Impact

An attacker may forge an encrypted HSM session and client secret data records which are interchangeable in an attempt to break the security protocol, including but not limited to, impersonating other users in order to access their secret data. NCC Group did not identify an exploitable scenario, but changes to the protocol and record formats may surface vulnerabilities in new iterations of the implementation.

Generally, the current reliance on the protobuf parser increases the attack surface of the protocol.

Description

The HSM fleet uses a single AES key to encrypt HSM/client session information and client secret data record before storage, for later use. The information is encrypted using the AES-GCM cipher block mode of operation, with a random 12 byte IV, and a 16 byte tag, without any additional authenticated data. This guarantees that only the HSM can encrypt and decrypt these records in the absence of other vulnerabilities. This method does not allow for the detection of whether one encrypted record was replaced with another during decryption. The HSM provides additional logic to determine whether a session information record was replaced with another session information record, and whether a user secret data record was replaced with another one. One example of this additional logic is that the HSM validates that the HSM challenge in the encrypted record matches the one provided by a user as part of a separate request to the HSM (for example, a request for their secret data). This in effect provides some level of assurance that a malicious party cannot impersonate other users by exchanging session records, thus obtaining their secret data.

The HSM does not appear to provide any explicit controls to determine whether a decrypted record type, be it HSM session or secret data, was the valid type which the HSM intended to read. The code implicitly relies on the correct parsing of the record based on its protobuf version 3 definition instead. This may open opportunities for attackers to smuggle data from one record type to another, when they have some control on the data which goes into the record type. The parsing of protobuf version 3 defined records may lead to misinterpretations. Indeed, for one, all fields are [optional](#).

The protobuf definition of secret user data and session data are implemented as messages `DecryptedKeyVault` and `DecryptedSessionData` respectively, in the file `xdb_data.proto`:

```
syntax = "proto3";  
  
package whatsapp_altheia;  
  
message DecryptedKeyVault {  
  bytes stored_user_secret = 1;
```



```

string account_jid = 2;
uint32 creation_timestamp = 3;
int64 creation_key_transparency_epoch = 4;
}

message DecryptedSessionData {
  bytes challenge = 1;
  bytes hsm_session_ephemeral_key = 2;
  string account_jid = 3;
  string session_id = 4;
  uint64 session_ttl = 5;
}

```

Encoding and decoding of records are implemented with the `to_xdb()` and `from_xdb()` methods of the structures `DecryptedKeyVault`, and `DecryptedSessionData` using the Rust `prost` library. The implementation currently decodes `DecryptedSessionData` as highlighted in the code snippet below in file `session_data.rs` as follows:

```

impl SessionData {
  // SNIP

  // Decrypt and deserialize the session data. The session data is serialized
  // as protobuf structure and then encrypted with the security worlds' 256
  // bit AES key.
  pub fn from_xdb(
    hsm_context: Arc<HsmContext>,
    encrypted_session_data: &[u8],
  ) -> Result<Self, ClientResponseError> {
  // SNIP

    let session_data_bytes = hsm_context
      .fleet_aes_decrypt(&encrypted_session_data)
      .map_err(|_| ClientResponseError::SessionKeyDecryption)?;

    let DecryptedSessionData {
      hsm_session_ephemeral_key,
      challenge,
      account_jid,
      session_id,
      session_ttl,
    } = DecryptedSessionData::decode(session_data_bytes.as_slice()).map_err(|_| {
      ClientResponseError::SessionDataRead(
        "Cannot deserialize DecryptedSessionData".to_owned(),
      )
    })?;

  // SNIP

```

The current implementation would successfully (decrypt, then) decode the following `DecryptedSessionData` message (secret user data record) represented in Rust pseudocode, as a `DecryptedKeyVault` message:

```

DecryptedSessionData {
  hsm_session_ephemeral_key: vec![65;32],
  challenge: vec![66;32],

```



```
account_jid: String::from(""),
session_id: String::from(""),
session_ttl: 1,
};
```

Specifically, the record `DecryptedKeyVault` message `account_jid` field would be set as `AAAAA` `AAAAAAAAAAAAAAAAAAAAAAAAAAAA` (smuggled from `DecryptedSessionData`'s `hsm_session_ephemeral_key` bytes field). This relies on the attackers ability to force empty `DecryptedSessionData` `account_jid` and `session_id` string fields, so that these fields are omitted from the serialized message before encryption.

The HSM currently checks that these fields are not empty in the request from the user before any encoding is performed, therefore preventing such attack. NCC Group did not identify another potential avenue of attack in the time allocated to the project. However, relying on successfully decoding data on an arguably lenient parser is fragile and increases the attack surface of the system.

Recommendation

Consider using different encryption keys to encrypt/decrypt different types of records so that records will be rejected if they can't be decrypted before parsing them. This can be achieved by using the HKDF primitive to generate different encryption keys from one key material.

Alternatively, consider using additional authenticated data (AAD) which is different based on the record type e.g., "DecryptedSessionData" for session data encrypted records, and "DecryptedKeyVault" for user secret data encrypted records. Records will be rejected during decryption if the AAD does not contain the intended value.

Location

Files `xdb_data.proto`, `key_vault.rs`, and `session_data.rs`

Retest Results

2024-09-25 – Fixed

NCC Group reviewed the changes WhatsApp made in code change `D62056575` in response to this finding. WhatsApp implemented different AADs depending of the record being encrypted: "session data" for session data records, and "key_vault" for user secret data records. This fix was implemented for Entrust HSMs only, and is not completely implemented for Marvell HSM. The WhatsApp team indicated that they will not use Marvell HSMs in production just yet. These changes fully mitigate this issue for Entrust HSMs.



9 Finding Details – iOS Client

Low

Protocol May Execute with Weaker Forward Secrecy Assurance

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E015746P-CN2

Component iOS Client

Category Cryptography

Status Fixed

Impact

An attacker in a privileged network position may be able to decrypt past client secret information if HSM long term keys are compromised in the future.

Description

The HSM SEE system and client engage in a X3DH key agreement protocol to establish a shared secret key. This shared secret is employed by clients to recover and store their own key material which are used to protect client contact information. The X3DH protocol provides [forward secrecy](#); that is, future compromise of long term keys will not result in a compromise of past session keys.

NCC Group identified an issue that may allow active adversaries to weaken forward secrecy, so that client secret data could be recovered in the event of a compromise of the long term HSM keys. Note that the consultant team later identified the exact same [finding "Protocol May Execute with Weaker Forward Secrecy Assurance"](#) in the Android client.

In its Server Hello message, the HSM SEE sends its long term public key `hk`, its ephemeral public key `hek` and a signature for each of their values to the client. Both keys are signed with the HSM fleet key, which is the private counterpart of `hk`. A new `hek` is generated for each interaction with a client.

The client validates that both `hk` and `hek` are signed by the HSM fleet key using a hardcoded value for the fleet key. The client does not attempt to determine whether `hk`, and `hek` correspond to respectively the long term and ephemeral/session public keys. An attacker may therefore intercept a Server Hello message, and replace the (`hk`, `hek`) fields to contain (`hk`, `hk`), or (`hek`, `hek`), or (`hek`, `hk`) instead. Tampering with the fields this way will force the client to derive a different session key, than what was expected by the HSM server. Therefore, the protocol will abort (but may restart automatically, and this time without requiring adverse intervention, if the resulting error is a retryable event).

However, the client will have sent encrypted information that may be decryptable if the HSM long term keys are leaked later, in contradiction with the forward secrecy guarantees of X3DH.

The issue is illustrated in the `verifyIplsIdentityOnServerHelloPayload()` function in the `IPLS HsmKeyServiceUtils.swift` source file, where the implementation limits itself to checking the signatures of `hk` and `hek`:

```
func verifyIplsIdentityOnServerHelloPayload(  
    _ serverHelloPayload: WAPBIplsServerHelloPayload,  
    isPostQuantumEnabled: Bool  
) -> Result<Bool, IPLSHsmServiceError> {  
  
    // -----
```



```

// Verify HSM key
guard let hsmPublicKeyBytes = serverHelloPayload.hkPub,
    let hsmPublicKeySignature = serverHelloPayload.hkKeySignature else {
    return .failure(.verifyServerHelloPayloadError("HSM public key bytes or signature
↳ is missing from hello payload"))
}
guard hsmFleetPublicKey.validateSignature(hsmPublicKeySignature, withMessage: hsmPublic
↳ KeyBytes) else {
    return .failure(.verifyServerHelloPayloadError("HSM public key signature failed
↳ validation"))
}

// -----
// Verify HSM ephemeral key
guard let hsmEphemeralPublicKeyBytes = serverHelloPayload.hekPub,
    let hsmEphemeralPublicKeySignature = serverHelloPayload.hekKeySignature else {
    return .failure(.verifyServerHelloPayloadError("HSM ephemeral key bytes or
↳ signature missing from hello payload"))
}
guard hsmFleetPublicKey.validateSignature(hsmEphemeralPublicKeySignature, withMessage:
↳ hsmEphemeralPublicKeyBytes) else {
    return .failure(.verifyServerHelloPayloadError("HSM ephemeral public key signature
↳ failed validation"))
}

// SNIP
}

```

Recommendation

Consider validating that `hk`, and `hek` are at least different and that they are in their correct fields. Evaluate whether sending `hk` is actually required, as the client already has this key hardcoded as the fleet key.

Location

Function `verifyIplsIdentityOnServerHelloPayload()`, in file `IPLSHsmKeyServiceUtils.swift`

Retest Results

2024-09-24 – Fixed

NCC Group reviewed the changes WhatsApp made in code changes `D62239026` and `DD63464458` to address this issue. The client implementation now validates that the `hk` and `hek` fields have different values, and that the `hk` public key equals the hard-coded HSM fleet key. These changes fully mitigate this issue.



Non Constant-Time AES-GCM Implementation For ARM Platforms

Overall Risk Informational
Impact High
Exploitability Undetermined

Finding ID NCC-E015746P-GHL
Component iOS Client
Category Cryptography
Status Fixed

Impact

A malicious mobile app, running on the same device as the WhatsApp iOS application, may be able to obtain the client AES-GCM keys when used by the client via cache attacks. This would allow attackers to encrypt/decrypt contact information, contact information encryption keys and to forge valid encrypted information.

Description

The IPLS protocol WhatsApp iOS client employs AES in the GCM block mode of operation to protect two assets:

- contact information, encrypted with client generated AES keys (contact information encryption keys),
- contact information encryption key, encrypted with a shared session key, which is derived from the X3DH key agreement protocol between the HSM SEE, and the client.

The client AES-GCM operations are implemented in the `NetworkUtils` Swift protocol extension methods `aesDecrypt()`, and `aesEncrypt()`, in file `NetworkUtils+Crypto.swift`, as illustrated below:

```
extension NetworkUtils {  
  
    /// Decrypts data using AES GCM  
    /// - Parameters:  
    ///   - key: the AES key  
    ///   - data: the data to decrypt  
    ///   - nonce: a nonce  
    ///   - auth: an auth tag  
    /// - Returns: the plaintext data  
    public static func aesDecrypt(_ key: Data, data: Data, nonce: Data, auth: Data) throws ->  
↳ Data {  
        guard let result = WABlocksAESGCMDecrypt(key, data, nonce, nil, auth) else {  
            throw EncryptedPayloadError.invalidResponse("error decrypting response")  
        }  
        return result  
    }  
  
    /// Encrypts data using AES GCM  
    /// - Parameters:  
    ///   - key: the AES key  
    ///   - data: the data to encrypt  
    ///   - nonce: a nonce  
    ///   - aad: additional account data  
    ///   - tagLength: auth tag len  
    /// - Returns: the plaintext data
```



```

public static func aesEncrypt(_ key: Data, data: Data, nonce: Data, aad: Data? = nil,
↳ tagLength: Int) throws -> (data: Data, auth: Data) {
    guard let encTuple = WABlocksAESGCMEncrypt(key, data, nonce, aad, Int32(tagLength)),
        let data = encTuple.firstObject as Data?,
        let auth = encTuple.secondObject as Data?
    else {
        throw EncryptedPayloadError.encryption
    }

    return (data, auth)
}

// (SNIP)

```

These functions wrap the MbedTLS cryptography library AES-GCM implementation in folder *third_party/mbedtls*. The library AES cipher in the GCM mode block mode of operation is mostly implemented in files *aes.c*, and *gcm.c*. The library appears to use dedicated hardware instructions for AES-GCM on x86 systems. It does not appear to use dedicated instructions for ARM systems, the primary iOS platform; instead, it performs table lookups for the AES and GHASH primitives, which underpin AES-GCM.

Table lookups may reveal the AES encryption keys, and GHASH hash subkeys via cache attacks.

Recommendation

Consider using Apple CryptoKit framework's AES-GCM [implementation](#). Apple [states](#) that CryptoKit has side channel-resistance.

Location

`NetworkUtils` Swift protocol extension methods in file *NetworkUtils+Crypto.swift*

Retest Results

2024-09-27 – Fixed

NCC Group reviewed the changes WhatsApp made in code diff [D63505288](#) in response to this finding. The WhatsApp team implemented NCC Group's recommendation to use Apple CryptoKit AES-GCM implementation, for devices running iOS 13+. WhatsApp plans to drop support for iOS 12, which represents a tiny fraction of their users. Apple does not appear to [support](#) iOS 12 anymore and is therefore not likely to provide security patches for this version. For these reasons, and considering the risk rating of this finding, NCC Group considers that these changes address the issue.



10 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

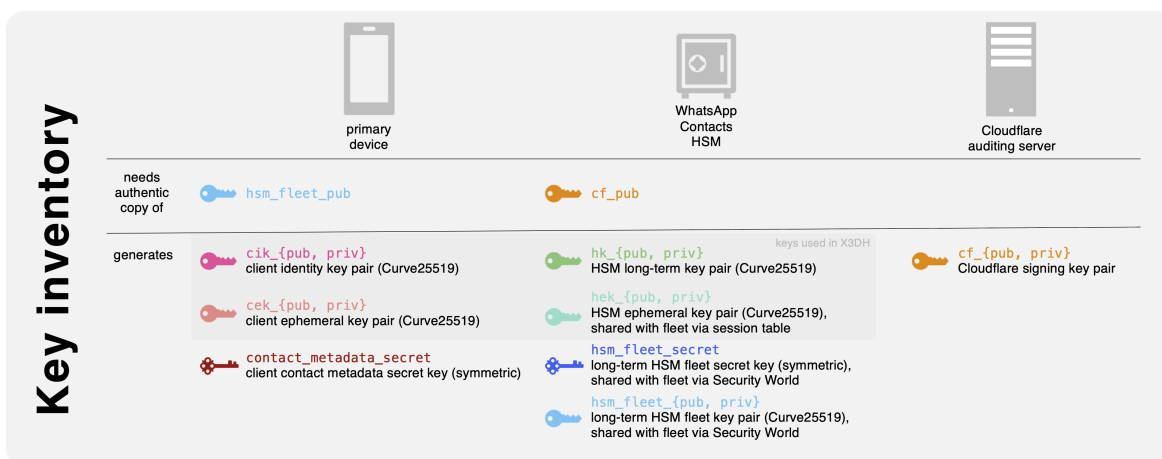
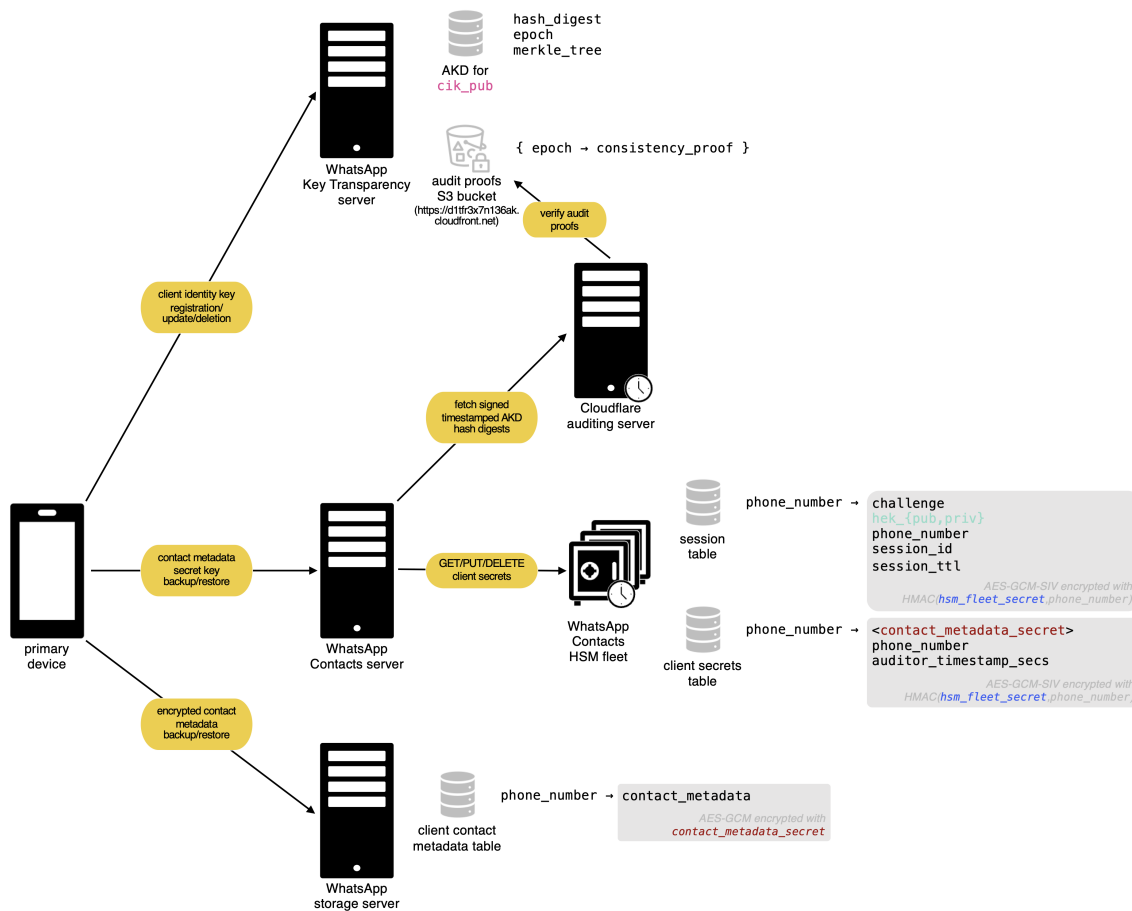
NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



11 WhatsApp Contacts Cryptographic Architecture

The following diagram illustrates NCC Group's understanding of the architecture of WhatsApp Contacts.



12 Protocol Specification Review

The first phase of the WhatsApp Contacts' privacy-preserving backup and restore feature audit focuses solely on reviewing the specifications. This section provides notes and recommendations regarding the Identity Proof Linked Storage (IPLS) protocol as is described in the *ipls.md* document at commit [56f456e](#) (version 1.1). Findings with security impact and their mitigations are described in the [Finding Details](#) section. NCC Group observations below may no longer fully apply, and are presented for informational purposes only.

Notes on Privacy Claims

The IPLS protocol leverages Hardware Security Modules (HSMs) and Auditable Key Directories (AKDs) to provide a privacy-preserving backup and restore solution. WhatsApp users create a keypair on their primary device, register their public key with AKD, and use their private key to prove possession of the key to the WhatsApp HSM. During the contact's metadata backup flow, the user performs a handshake with the HSM to establish a symmetric key which it will use to encrypt its secret data payload. The HSM securely stores the payload. Later, during the contacts' metadata restore flow, the user provides an AKD proof of the link between their phone number and the new key, and also proves possession of the new key by signing an HSM's challenge.

It is worth noting that the metadata mainly contains contact names as they are entered by the user. The metadata does not include contacts' phone numbers (referred to as `account_jid`s in the design document), and as such phone numbers are visible to the WhatsApp backend. It is also worth highlighting that WhatsApp servers can delete an account's stored data at any time.

There is also a notion of *profile names* (internally called "push names") that users set for themselves in their WhatsApp profiles. These profile names are visible to WhatsApp servers as (encrypted) chat messages pass through their systems, but WhatsApp does not explicitly store them. WhatsApp Contacts enables app users to give their in-app contacts names other than what those contacts have set for themselves in their profiles. For example, WhatsApp user Charlie may have contacts with the profile names "Alice" and "Bob", but Charlie could rename them in WhatsApp Contacts to "Mom" and "Dad".

Notes on the Core Handshake Protocol

The following remarks are not security findings, but are a list of inconsistencies and missing details that should be clarified in order to facilitate development.

- **Step 2:** The quoted `IplsServerHelloPayload` contains an extra `challenge_response` field which is unexplained and does not belong to this message (since the response to the challenge is a value that the client will send to the server, not vice versa). The definition in *ipls.proto* does not contain that extraneous field.
- **Step 3:** The `cloudflare_signature` is missing from the `ClientInit` structure. Also `session_create_t` is missing from the "HSM Session Table", or possibly `session_t` should be renamed to `session_create_t`.
- **Step 4:** Step 3 mentions that `session_id_signature` is `XEdDSA_SIGN(msg=HMAC-SHA-256(hek_pub, session_id), key=hek_priv)`, however step 4 specifies that: "Client will also verify `session_id_signature` is the `HMAC-SHA-256(hk_pub, session_id)` is signed by the HSM ephemeral key". There seems to be a typo in step 4, and `hk_pub` should be changed to `hek_pub`.
- **Step 9:** In the `IplsClientHelloPayload` message description, the IV index should be 3.
- **Step 11:** It is stated that HSMs should enforce a 10 minute time to live (TTL) for handshakes and abort if the creation timestamp was older than 10 minutes before now. Further, it is stated "Note: +/- 60 seconds drift between HSM nodes is acceptable" (this



applies only to the extent that the session table is shared between distinct HSMs). The difference between the current time (`now`) and the session creation timestamp (`session_create_t`) should therefore be checked to be at most 11 minutes; however, the specification shows an explicit assertion that uses “`20_minutes`” instead.

- **Step 11:** `cloudflare_signature_message` is a concatenation of `epoch`, `digest`, `cloudflare_timestamp`, and `cloudflare_namespace`. Here `digest` is undefined.
- **Step 12:** It is stated that “Client will update the `SK(Session key)` derived above in step 11 with `key_shared_secret` outputted by decapsulation, to yield a quantum-safe shared secret”. This step is performed by the HSM, and “Client” should be edited to “HSM”.
- **Step 16(2):** The `IplsClientSecretData` is required to include the client’s `account_jid`, however `IplsClientSecretData` as is defined in the `ipls.proto` file only includes a map of `key_ids` to client’s secret key lists and is missing the `account_jid`.

Contact Metadata Plaintext Format, and Encryption Are Not Sufficiently Documented

Both the metadata plaintext format, and encryption should be specified to facilitate further reasoning, and to ensure a correct implementation:

- **Metadata encryption:** The documents provided to aid this review do not appear to specify how contact metadata is encrypted. The team explained over chat that they intend to use AES-GCM encryption with the client secret key. *Updated 2024-10-02: The details of how AES-GCM encryption is used were fully reviewed during the implementation review phase.*
- **Metadata plaintext format:** The metadata plaintext data format should be carefully devised and documented. NCC Group explained in [finding "Contact Metadata Ciphertext Length Side-Channel"](#) how its serialization prior to encryption may affect the success of length side-channel attacks. Specifically, it may help in determining whether a field was populated by a user, or not.

No Authentication of Cloudflare Timestamp Field in HSM Record

In the course of the project, WhatsApp introduced a Cloudflare timestamp freshness assertion over the lookup proof provided by the server to the HSM, to ensure client public keys are not stale, as a fallback until HSM storage rollback protection is designed and implemented. In the updated specification (`ipls.md` version 1.1), the HSM record is augmented to contain a Cloudflare timestamp (in addition to the client account JID, client wrapped secret data, and payload HMAC).

The Cloudflare timestamp does not appear to be validated for authenticity (specifically, the payload HMAC does not appear to encompass this field). Modification of the timestamp field of the HSM record, in addition to an HSM rollback by a malicious internal actor, would permit usage of a compromised client identity key. However, the window of opportunity for usage of a stale key and its associated lookup proof would be 6 hours, as the HSM enforces this limit when it receives a lookup proof, with an accompanying Cloudflare assertion, before proceeding with handling any request.

WhatsApp should ensure that the HSM record timestamp field is authenticated to mitigate such attack.

Updated 2024-10-02: The implementation was fully reviewed during the implementation review phase, and any findings were documented in the [Finding Details](#) section.

PBKDF2 could be Swapped with HKDF

Uploading or rotating IPLS client secret data requires generating a symmetric encryption key on the primary device to encrypt the contacts’ metadata. This key is generated by passing



64 pseudorandom bytes (referred to as the base secret) to a Password-Based Key Derivation Function version 2 (PBKDF2) with HMAC-SHA-256 as the PRF using 32 iterations. PBKDF2 is designed to derive a cryptographic key from user-provided randomness which often has low entropy. Presently the recommended number of iterations in such a scenario is on the order of 600,000. However, since a CSPRNG is used to generate the base secret, it presumably has acceptable entropy, and it can be expanded to a 128-byte secret key using a Hash-based Key Derivation Function (HKDF). In addition to simplifying the design, swapping PBKDF2 with HKDF will enhance performance without loss of security.

Duplicate Client Handshake Material

In the `IplsClientHelloPayload` message, the client sends its handshake material (`IplsClientHelloHandshakeMaterial`) twice: directly as a sub-element of `IplsClientHelloPayload`, and again as part of the encrypted payload (`IplsClientRequestOpaquePayload.handshake_material`). The HSM verifies that both versions are identical (step 13 of *ipls.md*), in order to strengthen the cryptographic binding of the payload with the handshake. The second handshake material instance (inside the encrypted payload) could be replaced with a hash of that material (e.g. with SHA-256), which would preserve the expected security properties, but reduce the size of the encrypted payload and therefore the network usage. In particular, the client handshake material, if using the post-quantum KEM, includes a Kyber768 ciphertext, which has a size of 1088 bytes, while a SHA-256 hash fits in 32 bytes.

Lack of Domain Separation

The protocol contains multiple instances of signatures of various objects. In particular, the HSM permanent key (`hk_pub`) is signed with the HSM fleet key, but so is the HSM ephemeral key (`hek_pub`). Since the two keys have the same format (32 bytes), an active attacker could swap the two keys and their signatures without incurring immediate detection. No such substitution attack seems to result in an exploitable vulnerability in the currently defined protocol, but in general it is highly recommended to apply strict domain separation to ensure that different uses of the same signing key over different kinds of objects cannot be exchanged. Domain separation is easily achieved by using as signed data not the object m that is to be signed, but the concatenation $p \parallel m$ for some prefix p which is specific to the type of object that is signed. The prefix should be such that the boundary between prefix and message is unambiguous; e.g. all prefixes may have a fixed, common size (say, 32 bytes), or all prefixes may consist of a zero-terminated ASCII string.

Updated 2024-10-02: key substitution was addressed by the WhatsApp team, as detailed in findings [finding "Protocol May Execute with Weaker Forward Secrecy Assurance" \(iOS\)](#), and [finding "Protocol May Execute with Weaker Forward Secrecy Assurance" \(Android\)](#).

Redundant Elements

The *ipls.md* protocol includes several elements which are redundant, and thus increase the computational cost, message size, and overall complexity, without providing additional security features. Such redundant elements include the following:

- **Session ID:** The session ID's main use is to ensure freshness of the HSM's response, by having the HSM sign the session ID in conjunction with the HSM's permanent public key (`hk_pub`) with its ephemeral private key (`hek_priv`); since the client generated the session ID randomly, and the HSM's ephemeral public key `hek_pub` was signed by the HSM fleet key, then this signature guarantees to the client that a real HSM has been involved in computing this specific response. However, this freshness property is already ensured by X3DH itself: the client has just generated its own ephemeral key pair (`cek_pub` and `cek_priv`), and already knows that the session key `SK` can be computed only by an entity knowing `hk_priv` (since `hk_pub` is used in X3DH) and using that freshly



generated `cek_pub` key. Moreover, session IDs have a one-to-one mapping to account IDs, since the HSM must maintain the invariant that for any `account_jid` value, a single session ID should exist at any one time.

Note: the signature on the session ID also allows the client to detect a replay attack by a hostile server one step earlier in the process, but this property has no real benefits since the client cannot do much in that situation except declare that the process failed. Making the server always return a payload encrypted with `SK`, even on `PUT` and `DELETE` requests (e.g. a simple acknowledgement message), would ensure that the client always ultimately gets explicit assurance that a real HSM was involved in the processing.

- **HSM challenge:** The HSM challenge is a random value generated by the HSM and signed by the client; it is similar to the session ID, but this time for the HSM to verify that the client's request is fresh. This is again redundant with X3DH: since the HSM generated a fresh ephemeral public key pair (`hek_pub` and `hek_priv`), which is used in X3DH, a client request payload may be successfully decrypted by the server only if the sender used that value of `hek_pub`.
- **Signatures on ephemeral EC keys:** The HSM's ephemeral public key `hek_pub` is signed with the HSM's fleet key, while the client's ephemeral public key `cek_pub` is signed with the client's identity key (`cik_priv`). These signatures are redundant with the use of these keys in X3DH, which already ensures mutual authentication: the client knows that the resulting session key `SK` can be computed only by the owner of `hk_priv`, since `hk_pub` is involved in the process; similarly, the server knows that the message comes from the expected client, through the use of `cik_pub` in X3DH. The extra ephemeral keys in X3DH do not need to have additional authentication; their role is to ensure freshness of communications and post-compromise security. *A contrario*, the signature on `hk_pub` with the HSM fleet key is important: the client needs an *a priori* assurance that only a true HSM running the expected software will be able to recompute the session key `SK`. The signature on the Kyber768 key (`kem_pub`) can also be deemed useful, though its lack would only allow rather contrived attack scenarios.

The overall protocol could be simplified by removing the redundant elements listed above and still provide the expected security guarantees.



13 Implementation Review Engagement Notes

In this section, we present various remarks about the implementation for informational purposes only. None of them are a security vulnerability, but they were deemed worth reporting for discussion and to avoid pitfalls with possible future extensions. As WhatsApp implemented changes to their code in several areas following intermediate feedback from the consultants during the course of the project, some initial observations from the NCC Group team may no longer fully apply.

Extra Check Of RNG Output Length

The Rust `new()` function implemented on the `HsmRng` struct in the `rand.rs` source file checks that the amount of requested entropy is equal to the amount of returned entropy as seen highlighted below, or returns an error otherwise:

```
impl HsmRng {
    pub fn new() -> Result<Self> {
        #[cfg(profilite)]
        let _p = profilite::start_guard("HsmRng::new");

        let mut seed: [u8; 32] = [0u8; 32];
        let mut seed_len_inout: std::os::raw::c_uint = 32;
        let fill_status: Status =
            unsafe { see_native::get_random_bytes(seed.as_mut_ptr(), &mut seed_len_inout) };

        if fill_status != STATUS_OK {
            let err = get_status_string(fill_status);
            bail!(
                "Error creating HsmRng. Error getting HSM random bytes: {:?}",
                err
            );
        }
        if seed_len_inout != 32 {
            bail!(
                "Error getting HsmRng seed. Only received {:?} out of 32 random bytes",
                seed_len_inout
            );
        }

        Ok(Self::from_seed(seed))
    }
}
```

Entropy is obtained by a call to the C function `get_random_bytes()` in the `rand.c` source file. This function in turn calls the `try_checked_memcpy()` function which verifies that both amounts are equal as highlighted below, or returns an error otherwise.

```
Status get_random_bytes(unsigned char* output, M_Word* num_bytes_inout) {
    M_Command command = {0};
    command.cmd = Cmd_GenerateRandom;
    command.args.generatorandom.lenbytes = *num_bytes_inout;

    M_Reply reply = {0};
    M_Status status = SEELib_Transact(&command, &reply);

    Status result = Ok;
    if (command_failed(status, &reply, "Cmd_GenerateRandom")) {
        result = GenerateRandomError;
    } else {
        const unsigned char* data = reply.reply.generatorandom.data.ptr;
        const M_Word data_len = reply.reply.generatorandom.data.len;
    }
}
```



```

if (try_checked_memcpy(output, *num_bytes_inout, data, data_len)) {
    result = SafeMemcpyFromHsmError;
} else {
    *num_bytes_inout = data_len;
}
}

// Free any data allocated into `reply` by API call.
SEELib_FreeReply(&reply);
return result;
}

```

The `get_random_bytes()` function does not document or commit to this guarantee. If it did, the extra validation in the Rust function could be omitted.

Non Constant-Time Comparison of Session ID

The `validate_session()` function in the `session_data.rs` source file does not compare the secret session ID value provided by the client in a request, with the value obtained from a decrypted record by the HSM, in constant time as highlighted below. This may in principle leak the correct value of the session ID and thus provide one of the steps required to impersonate a client by forging a valid session. Any potential timing side-channels resulting from this code are probably not exploitable in the current implementation (one of the first difficulties in this case would be to guess the value within the session lifetime of 10 minutes). Nevertheless, NCC Group recommends using constant-time code to compare secret data.

```

// Validate 1. session_ttl is valid, 2. session_id matches the one in encrypted session
↳ data
pub fn validate_session(
    &self,
    session_id: &str,
    user_id_with_namespace: String,
) -> Result<(), ClientResponseError> {
    if user_id_with_namespace != self.account_jid {
        return Err(ClientResponseError::SessionDataRead(
            "user ID does not match the one in persisted session data".to_owned(),
        ));
    }

    if session_id != self.session_id {
        return Err(ClientResponseError::SessionIdMismatch);
    }

    let utc_now: DateTime<Utc> = Utc::now();
    let now_ts: i64 = utc_now.timestamp();
    let converted_now_ts: u64 = now_ts
        .try_into()
        .map_err(|_| ClientResponseError::Unknown)?;

    if self.session_ttl < converted_now_ts {
        return Err(ClientResponseError::SessionExpired);
    }

    Ok(())
}
}

```



Entropy Usage Documentation And Implementation Discrepancies

Upon startup, the HSM SEE code instantiates a HSM context, which holds a pointer to a cryptographically secure random number generator implemented on the `HsmRng` struct. The `rng_try_fill_bytes()` function permits obtaining the required amount of entropy from the HSM context. `HsmRng` is underpinned by the low-level `get_random_bytes()` C function, which itself calls the Entrust CodeSafe `Cmd_GenerateRandom` HSM API. The source code states that all rng usage relies on `HsmRng`, in the `rand.rs` source file:

```
/* HsmRng: Interface to native HSM crypto rng.
 *
 * HSM doesn't allow system calls to `getrandom`, so all rng usage relies on
 * this struct.
 */
impl HsmRng {
    pub fn new() -> Result<Self> {
        #[cfg(profilite)]
        let _p = profilite::start_guard("HsmRng::new");

        let mut seed: [u8; 32] = [0u8; 32];
        let mut seed_len_inout: std::os::raw::c_uint = 32;
        let fill_status: Status =
            unsafe { see_native::get_random_bytes(seed.as_mut_ptr(), &mut seed_len_inout) };

        if fill_status != STATUS_OK {
            let err = get_status_string(fill_status);
            bail!(
                "Error creating HsmRng. Error getting HSM random bytes: {:?}",
                err
            );
        }
        if seed_len_inout != 32 {
            bail!(
                "Error getting HsmRng seed. Only received {:?} out of 32 random bytes",
                seed_len_inout
            );
        }

        Ok(Self::from_seed(seed))
    }
}
```

This is not strictly correct. While this structure is used for generating random IVs for the AES-GCM cipher block mode of operation, NCC Group noted the usage of the `rand crate th read_rng().fill_bytes()` function for generating HSM challenges, ephemeral elliptic curve keys and `OsRng` for signing. The `thread_rng()`, `ThreadRng`, and `OsRng` functionality implemented on `ThreadRng` appears to be backed by the `getrandom()` system call. SEE does not implement/whitelist this system call. Therefore, WhatsApp implemented a wrapper in file `syscall_wrapper.c` around the `getrandom()` system call, and linked its SEE binary with this wrapper. The wrapper calls the same `get_random_bytes()` described above.

The implementation team should consider updating the code documentation accordingly.

Support for Post-Quantum Cryptography

Note: Post-Quantum cryptography support was not in scope for the implementation review.

NCC Group notes that there is embryonic code for post-quantum cryptography support, but it is not functional at the time of review. For instance, a dummy public key is generated,



signed, and sent by the HSM to the client in the `handle()` function on the `InitHandler` struct, in the `init.rs` source file. This arguably unnecessarily increases the attack surface of the SEE system, while the code is not fully implemented.

NCC Group also notes that some of the documentation, including the protocol specification in the `ipls.md` file, was updated to include post-quantum cryptography since the last design review.

Integration Testing and Auditor Signature TTL Verification

During the course of the review, change `D61499433` was implemented to disable auditor signature TTL in integration tests, as shown in the code excerpt below, in the `auditor_signature_verifier.rs` source file.

```
// 4. timestamp not too old compared to HSM clock
let now_ts: i64 = utc_now.timestamp_millis();
let converted_now_ts: u64 = now_ts.try_into().map_err(|_| {
    ClientResponseError::InvalidRequest(FieldInfo::PayloadProtobuf(format!(
        "Failed to convert now timestamp to u64: {now_ts:?}"
    )))
})?;
if converted_now_ts > (message.timestamp + AUDITOR_SIGNATURE_TTL_IN_MILLISECONDS) {
    // Do not validate auditor signature TTL in integration tests, since we can't
    // get fresh auditor signatures for integration tests.
    #[cfg(not(feature = "integration_test"))]
    return Err(ClientResponseError::OldAuditorSignature);
}
```

The feature `integration_test` is enabled by default, thus disabling this check globally. The WhatsApp team should ensure this change will not impact production.

Erratum: `aes_gsm` for AES-GCM operations

In several locations within the codebase, AES-GCM is incorrectly identified as “AES-GSM”. The typo stems from definitions within the `utils.rs` source file, lines 75 and 97, which define helper functions for `aes_gsm_decrypt` and `aes_gsm_encrypt` respectively. These functions appear to perform standard AES-GCM operations, and should be named as such. It is noted in passing that the same typo is also present in unrelated code for FB Pay.

Duplicated Protobuf Files

The protobuf file specifying IPLS messages is duplicated in several codebases and does not appear to be synchronized among them. The main locations are the following, in order from most recently updated to least recently updated, with main differences noted:

- `whatsapp-common/./ipls.proto`: last updated 14 August, 2024
 - uses `proto2`
 - request type `enum` capitalized as `IPLSRequestType`
 - `session_id_signature` (field 8) of `IplsServerHelloPayload` marked as `deprecated`
 - `cek_pub_signature` (field 7) of `IplsClientHelloHandshakeMaterial` marked as `deprecated`
- `whatsapp/./ipls.proto`: last updated 17 July, 2024
 - uses `proto3`
 - request type `enum` capitalized as `IPLSRequestType`
 - `session_id_signature` (field 8) of `IplsServerHelloPayload` present
 - `cek_pub_signature` (field 7) of `IplsClientHelloHandshakeMaterial` present



- `whatsapp-iphone/./contact_encryption_ipls.proto`: last updated 2 August, 2024
 - uses `proto3`
 - request type `enum` capitalized as `IplsRequestType`
 - `session_id_signature` (field 8) of `IplsServerHelloPayload` absent
 - `cek_pub_signature` (field 7) of `IplsClientHelloHandshakeMaterial` present
- `whatsapp-android/./ipls.proto`: last updated 1 August, 2024
 - uses `proto3`
 - request type `enum` capitalized as `IplsRequestType`
 - `session_id_signature` (field 8) of `IplsServerHelloPayload` present
 - `cek_pub_signature` (field 7) of `IplsClientHelloHandshakeMaterial` present

There are also differences in comments and order of messages.

32-bit Assumption

In the `syscall_wrapper.c` source file, the code assumes that it is compiled and run on a 32-bit system (lines 30-33 and again lines 76-78), so that `size_t` and `uint32_t` have the same size. The HSM hardware is a PowerPC CPU (Freescale e5500 cores) that supports both 32-bit and 64-bit modes, hence the code may conceptually be compiled in 64-bit mode, which will break that assumption. It is thus recommended to at least include a protection mechanism that will abort compilation if the target mode is not 32-bit; for instance, the following could be added to `syscall_wrapper.c`:

```
#include <stdint.h>
#if (INTPTR_MAX >> 31) != 0
#error Only 32-bit mode is supported
#endif
```

This code will prevent compilation from completing if the target system uses pointers that are larger than 32 bits.

An alternative would be to make the code compatible with 64-bit mode, which could have performance benefits, e.g. for computing some hash functions such as SHA-384 or SHAKE. However, this is *not recommended* because the 64-bit multiplication opcodes of e5500 cores happen to have execution timings that depend on the values of the input operands. Specifically:

- In 32-bit mode, a multiplication of two unsigned 32-bit integers with a 64-bit result will use the `mullw` and `mulhww` instructions (for the low and high halves of the result), which both have fixed execution timings (4-cycle latency and 1-cycle reciprocal throughput).
- In 64-bit mode, similar opcodes `mulld` and `mulhdw` can multiply two unsigned 64-bit integers, into a 128-bit product. However, these instructions have varying latencies (4 to 7 cycles) and reciprocal throughputs (2 or 4 cycles) depending on the sizes and signs of the operands (note that `mulld` covers both the sign and unsigned cases, which do not differ in the low half of the output products, and thus may leverage the signed interpretation of operands).

When used in 64-bit mode, the X25519 and Ed25519 implementation (the [curve25519-dalek](#) library) will use the 64×64→128 multiplications in order to improve performance, which unfortunately leads to non constant-time operations that can at least conceptually be leveraged into a timing attack that leaks X25519 and Ed25519 private keys. Therefore, 64-bit mode *should not* be used in the SEE application in its current state.



Busy Loops

The SEE application is organized as several asynchronous tasks using the [Tokio framework](#). A custom Tokio scheduler is created by calling `tokio::runtime::Builder::new_current_thread()`, which will use a single thread to run all tasks. Within the application, a busy-looping pattern is used for several long-running tasks; it can be seen for instance in the `.../see_job_device/mod.rs` (lines 115 and 289), `.../see_job_processor/mod.rs` (line 101), and `.../client_request_processor/mod.rs` (line 50) source files. In this pattern, a loop is used, which repeatedly checks a condition (e.g. whether a given queue is empty or not), then calls `yield_now()` to let other tasks run:

```
loop {
    {
        let mut requests_guard = self.requests.lock().unwrap();
        let requests = &mut *requests_guard;

        if let Some(requests_vec) = requests {
            // <SNIP>
        }
    }

    // After processing the previous set of requests, SeeJobProcessor
    // will "block" in this method until the next batch of requests is
    // received. Since SEE is single threaded, to ensure other tasks
    // (e.g. RaftTask) can run, we yield here.
    let _ = tokio::task::yield_now().await;
}
```

This pattern has two undesirable characteristics:

- If the application is nominally idle and just waiting for the next request from the outside world, then all such long-running tasks will keep spinning, repeatedly checking the queues and yielding. From the point of view of the SEE kernel, the thread will be always busy, and the core on which it runs will never enter an idle state. This will induce a higher energy usage and operating temperature than if the CPU core was allowed to be idle. The HSM lifetime, before hitting a hardware failure, might be reduced.
- The yielding mechanism relies on the assumption that all runnable tasks are scheduled in a round-robin way, and that the task that calls `yield_now()` is moved to the very end of the list of runnable tasks, thus ensuring that all other runnable tasks are regularly scheduled. This is how the “current thread” Tokio scheduler operates for now, but the [Tokio documentation](#) explicitly denies any such guarantee. In a future version of Tokio, the loop-and-yield pattern might instead run only a subset of the runnable tasks and consistently avoid some other tasks, leading to an application failure. Also, the application might later on be modified to support using several native threads, in order to leverage the multiple cores of the CPU (the NXP QorIQ T1042 has four e5500 cores) for improved performance; in such a case, a different scheduler would be used, with a different behaviour upon explicit yielding.

An alternative pattern, which would allow CPU idling and avoid assumptions on the scheduler strategy, would be to make such loops block (with the [Tokio synchronization primitives](#)) until they have actual requests to process.

Avoidable Signature Generation

In the IPLS protocol specification, each HSM has its own permanent public/private key pair (`hk_pub / hk_priv`), and generates a new ephemeral public/private key pair (`hek_pub /`



`hek_priv`) when receiving a client request. The HSM has moreover access to the fleet private key (`fleet_priv`) and uses it to sign both `hk_pub` and `hek_pub` (and also `kem_pub`, if used). The client verifies these signatures relatively to the fleet public key, which is hardcoded in the client. The X3DH key exchange is performed with `hk`, `hek`, and the corresponding client-side permanent (`cik`) and ephemeral (`cek`) keys.

Since it is not actually important for security that the HSM's permanent public key (`hk`) and the fleet key be distinct, the SEE application conflates the two, i.e. it sends a copy of the fleet key as `hk_pub`, and the signature on the fleet key is then a self-signature (the fleet key signs itself). In the implementation, this self-signature is computed anew for every request (in `init.rs`):

```
let hk_key_signature: [u8; 64] = hk_signing_key.sign(&hk_pub, 0sRng);
```

Since the fleet key does not change, that signature value does not have to be recomputed each time; it could be cached locally. Such caching would reduce the computation cost in the HSM.

Alternatively, as pointed out in [finding "Protocol May Execute with Weaker Forward Secrecy Assurance"](#), since the security does not rely on `hk` being distinct from the fleet key, the protocol could be amended to simply remove the sending of `hk_pub` and the accompanying signature; instead, the client may simply assume the situation which is already applied by the server, i.e. that `hk_pub` is the fleet public key, that the client already knows. This would reduce the used network bandwidth (no need to transmit a key that the client already has) and allow removing both the self-signature generation (in the HSM) and the self-signature verification (in the client).

Allocation Checking

In the `see_job.c` source file, a buffer is allocated but there is no check for allocation failure:

```
SmPacketReq* packet_req = (SmPacketReq*)malloc(sizeof(SmPacketReq));  
packet_req->sm_packet_request = sm_req;
```

If the memory allocation fails, then `malloc()` will return `NULL` and the write access will be performed at an invalid (low) address, which triggers undefined behaviour. Low-memory conditions can usually be forced by the host, e.g. by sending a (fake) request split into a very large number of individual "pages".

Android Client Digital Signature Validation

In its Server Hello message, the HSM SEE sends its long term public key `hk`, its ephemeral public key `hek`, and a signature for each of their values to the client. Both keys are signed with the HSM fleet key, which is the private counterpart of `hk`. The client validates that both `hk` and `hek` are signed by the hard-coded HSM fleet key.

The Android client digital signature validation implementation wraps a vendored version of Signal's `curve25519-java` library with pure Java, and JNI/C language implementations in the general purpose `CryptoUtils` Java class. Both implementations do not check that the public key used to sign the `hk`, and `hek` values is not an elliptic curve point of low order. This would in principle enable attackers to forge valid signatures for arbitrary messages. However, it is assumed that the attacker would not have control over the client hard-coded public key for the HSM fleet, making this issue moot in the current implementation. It is worth keeping in mind though, if the `CryptoUtils` class is used for other purposes, and/or in other projects in the future.



Test HSM Fleet Key

The WhatsApp Android Client authenticates the HSM Hello message payload in a call to the `assertIplsHsmIdentity()` function, using a hard-coded HSM fleet public key passed as an argument variable named `MOCK_HSM_FLEET_KEY`, as illustrated in the code extract below of the `handleClientIplsInitSuccessResponse()` function in the `ContactMetaDataEncryptionHelper.kt` source file:

```
@WorkerThread
private fun handleClientIplsInitSuccessResponse(response: ClientIplsInitResponse.Success) {
    Log.i("${TAG}/handleClientIplsInitSuccessResponse ${requestType}")
    val serverHelloPayload = response.serverHelloPayload
    val responseByteArray = Base64.decode(serverHelloPayload, Base64.NO_PADDING)

    val serverHelloPayloadProto = Ipls.IplsServerHelloPayload.parseFrom(responseByteArray)
    val hsmAssertionResult =
        clientIplsHandshakeUtils.assertIplsHsmIdentity(serverHelloPayloadProto, MOCK_HSM_FLEET
        ↳ KEY)
    // SNIP
```

This note is a reminder to ensure that this variable is set to the correct value in production, and to consider renaming the variable accordingly.

Contact Metadata Padding Before Encryption

The Android client pads contact metadata before encryption, in order to not reveal its size to observers of encrypted data, in the `constructContactMetadataProto()` function within the `ContactMetaDataEncryptionHelper.kt` source file:

```
fun constructContactMetadataProto(
    waContact: WAContact
): ContactMetadataOuterClass.ContactMetadata {
    val contactMetadataBuilder = ContactMetadataOuterClass.ContactMetadata.newBuilder()
    contactMetadataBuilder.firstName = (waContact.given_name ?: "").take(MAX_FIRST_NAME_THRESHOL
    ↳ LD)
    contactMetadataBuilder.lastName = (waContact.family_name ?: "").take(MAX_LAST_NAME_THRESHOL
    ↳ D)
    contactMetadataBuilder.businessName =
        (waContact.company ?: "").take(MAX_BUSINESS_NAME_THRESHOLD)
    contactMetadataBuilder.syncPolicy =
        when (waContact.syncPolicy) {
            ContactSyncPolicyType.SYNC_WITH_DEVICE ->
                ContactMetadataOuterClass.ContactMetadata.SyncPolicy.SYNC_TO_DEVICE
            else -> ContactMetadataOuterClass.ContactMetadata.SyncPolicy.NOT_SYNC_TO_DEVICE
        }
    val metadataProtoWithoutPadding = contactMetadataBuilder.build()
    val requiredPaddingLength = paddingLength(metadataProtoWithoutPadding)

    if (requiredPaddingLength > 0) {
        return transformToProtoWithPadding(metadataProtoWithoutPadding, requiredPaddingLength)
    } else {
        return contactMetadataBuilder.build()
    }
}
```

The highlighted conditional branch implicitly assumes that the padding length might be zero, and that therefore a record does not require padding. If it is the case, then the size of a record that does not require padding will be two bytes less than a record that was padded,



because of the addition of the `padding` field for the latter. This may thus leak whether the contact metadata was the maximum allowed size, or not. However, the required amount of padding is computed as follows:

```
val requiredPadding = Math.max(PROTO_SIZE_THRESHOLD - totalBytes, 0)
```

where the current `PROTO_SIZE_THRESHOLD` value is large enough, such that the `else` conditional branch will never be taken. The WhatsApp team is still in the process of deciding the value of this threshold, and other thresholds (e.g., maximum name size). Therefore, the above should be taken into account when finalizing these values. Obviously, occurrences of records that do not require padding should be rare, or even inexistent. However, this may offer an avenue to forge, disseminate, and track such encrypted records for other malicious purposes.

Unnecessary Length Management

In the support code for AES encryption specific to the Marvell HSMs, within the `aes.rs` source file, when applying AES/GCM encryption, the plaintext length is checked to be less than a constant size, which is 4000000000:

```
ensure!(
    plaintext.len() <= AES_256_GCM_MAX_ENCRYPTION_SIZE_BYTES,
    "Overlimit 128 AES-GCM plaintext"
);

let plaintext_len: c_ushort = plaintext.len().try_into()?;
```

However, immediately after that check, the plaintext length is converted to the 16-bit type `c_ushort` (because the underlying native HSM function cannot process plaintexts larger than 65535 bytes). This conversion is checked (`try_into()` is used and any error result is propagated to the caller). Thus, the second conversion is much stricter than the first one, which is thereby rendered useless. It may be noted that in the corresponding *decryption* code, the same conversion (of the ciphertext length) to `c_ushort` is used, but there is no prior check of the ciphertext length against `AES_256_GCM_MAX_ENCRYPTION_SIZE_BYTES`. The extra test in the encryption path is probably a remnant of an initial code import from the support code for the Entrust HSM, which supports 32-bit plaintext/ciphertext lengths and for which that test is conceptually more meaningful. An unnecessary test is harmless in itself, but it might indicate that the implementation is not fully stabilized yet.

Another length-related quirk occurs in the same file; when encrypting, a `ciphertext` vector is created to receive the ciphertext, with the same length as the plaintext since GCM is a length-preserving mode:

```
let ciphertext_len: c_ushort = plaintext_len;
let ciphertext: Vec<u8> = vec![0u8; ciphertext_len.try_into()?];
```

This is followed by the invocation of the native encryption function (`encrypt_aes()`), which receives a native pointer to the content area of `ciphertext`; the native function is unaware that the destination area is part of a Rust vector (`Vec`), and the `ciphertext_len` variable is not referenced at all in that code chunk. Finally, a `truncate()` call is performed:

```
ciphertext.truncate(ciphertext_len.try_into()?);
```

Since, at that point, neither the length of `ciphertext` (as a vector) nor the `ciphertext_len` variable have been modified, the `ciphertext` vector length must still be equal to `ciphertext_len`, and the truncation does nothing. This truncation operation is probably a



trace of an older code version which would have not used GCM, but a different, non-length-preserving encryption mode, such as CBC. A similar unnecessary truncation is present in the decryption path.

Malicious Insider Could Prevent Epoch Creation and Attestation

Update 2024-09-26: The WhatsApp team validated that epochs are increasing, and sequential, rendering the observations in this subsection moot. It is kept for informational purposes only.

By publishing epochs with increasing – but not sequential – epoch numbers, a malicious WhatsApp insider could effectively brick the Cloudflare attestation system, preventing the HSM from successfully verifying any user’s identity, *and* preventing the creation of new epochs (i.e., new sets of public key changes in the AKD).

Cloudflare acts as a third-party attestation service for the AKD, providing signed attestations that the AKD root digest for a namespace has a certain value in a certain epoch. WhatsApp requests a signature from Cloudflare by invoking the “Create a new epoch” API, which is specific to a particular namespace, with a digest (32-byte value) and an epoch (encoded as `int64`). WhatsApp requires the Cloudflare signature on the epoch and digest at the time an epoch is published (i.e., when a set of changes to the AKD is made public). According to the API documentation, Cloudflare requires that “[e]pochs must be increasing” to prevent split-view attacks where there are different signed digests for the same epoch. However, the documentation does not state that epochs must be *sequential*.

Suppose the last published epoch, with a signed attestation from Cloudflare, is N . A malicious or compelled WhatsApp insider could invoke the “Create a new epoch” API with epoch value $N+2$. This would prevent WhatsApp from ever publishing epoch $N+1$ in that namespace, since it requires signed attestations to publish epochs, and audit proofs require published epochs to be sequential. The malicious or compelled WhatsApp insider could also invoke the “Create a new epoch” API with the maximum epoch value, $2^{64}-1$. In this case, the Cloudflare signing API would sign no further requests, rendering the AKD for that namespace entirely useless. The HSM would no longer service client requests without up-to-date signatures from Cloudflare.

The denial of service could be expanded if the insider further invokes the “Create namespace” API for all namespaces allowable by the HSM, and repeats this attack for each, preventing their legitimate use. There are 256 other namespaces allowed by the HSM, due to the use of `u8::from_str()` in `check_cloudflare_namespace_format()`.

While such a denial-of-service attack would be detectable (via Cloudflare signature logs and the list of namespaces), it is not preventable. However, requiring Cloudflare to enforce *sequential*, not just increasing, epoch numbers in each namespace would prevent it.

HSM Critical Reliance On Third-Party Service Availability

The WhatsApp Contacts service is underpinned by a read-only HSM configuration in production. The HSM application relies on a third-party attestation service from Cloudflare to operate normally. That is, if the Cloudflare service is not available, or does not produce the correct data, the HSM cannot validate, and will therefore reject user WhatsApp Contacts service requests, resulting in a denial of service. Most issues with the third-party service would likely to be transient in nature, and recoverable e.g. temporary connectivity loss because let’s say a router failed, until fail-over happen.

Destruction of the attestation service private key would result in bricking the HSM fleet in the current WhatsApp HSM software implementation, and permanent denial of service for the WhatsApp Contacts service. In general, WhatsApp should seek some level of assurance



that the attestation service is resilient, and secure. NCC Group also understands that WhatsApp may implement support for other attestation services to avoid reliance on a single provider.

Entrust HSMs Identify Keys With SHA-1

In Entrust SEE machines, permissions are enforced by checking that an operation is authorized by a key with a certain hash. For example, the AES and X25519 fleet keys' permissions allow exporting them to the host encrypted with a key whose hash matches `hkm` from the NFKM Info of the Security World. According to vendor documentation (*nshield-v13-6-3-utilities-reference.pdf*, p. 194), `hkm` is "the SHA-1 hash of the Security World key".

SHA-1 is not a collision-resistant hash function, and there have been practical implementations of chosen-prefix collision attacks (<https://shattered.io/>). While the HSM vendor documentation states (*ncore-v13-6-3-developer-tutorial.pdf*, p. 109) that "SHA-1 is a hash function that has been approved by NIST", NIST actually deprecated the use of SHA-1 for digital signatures in 2011 and announced in 2022 that SHA-1 has "reached the end of its useful life".

If the HSM setup ceremony were compromised, an insider could craft a pair of keys ahead of time that have the same hash, and use one of these illegitimately. This further illustrates the importance of auditing the ceremony to ensure that all keys were freshly generated by the HSM.

