Security and Privacy Assessment: WhatsApp Message Summarization Service

Meta Platforms

Version 1.0 – August 26, 2025

Prepared By
Elena Bakos Lang
Paul Bottinelli
David Brauchler
Marie-Sarah Lacharité
Carles Pey
Thomas Pornin
Javed Samuel
Eric Schorn

Prepared For
Meta

1 Table of Contents

2 Executive Summary

Synopsis

In late January 2025, Meta Platforms engaged several of NCC Group's specialty practices to conduct a security and privacy assessment of the WhatsApp Message Summarization Service, which is part of a broader Private Processing system.[1] This service allows WhatsApp users to send a batch of messages to a Meta-operated Large Language Model (LLM), which returns a summarization of the message contents. Several third parties play key roles in the service: Cloudflare maintains transparency logs of signed artifacts, such as Trusted Execution Environment (TEE) images and hashes of LLM prompts, and Fastly acts as an Oblivious Relay between WhatsApp users and Meta. The overall system is intended to ensure user data remains private, inaccessible to Meta, not persisted, and not used for any other purposes.

The review was performed remotely as a 115 person-day effort during the first half of 2025 by NCC Group's Cryptography Services team, Hardware and Embedded Security team, and AI/ML Security team.

Assessment Scope and Methodology

NCC Group's evaluation included the following areas:

- Overall design and architecture review, threat modelling.
- Applicability of Meta's stated security and privacy assurances.
- Detailed implementation review:
  - Oblivious HTTP (OHTTP) for connections initiated by WhatsApp users, using Hybrid Public Key Encryption (HPKE).
  - Remote Attestation TLS (RA-TLS) for confidentiality, server authentication, and attestation of hardware-based TEEs.
  - Transparency proofs for artifacts such as TEE images and LLM prompts.
  - TEE and Confidential Virtual Machine (CVM) configuration.
  - Orchestrator and Inference services running in CVMs.
  - LLM model selection, system prompt templates for Summarization and Output Guard LLMs.
  - Graphics Processing Unit (GPU) configuration.
  - Logging and metrics in CVMs.
  - Sandboxing using Linux containers with CVMs.
  - Patched Stage 0 bootloader.
- Retesting fixes to NCC Group findings.

NCC Group assessed the Message Summarization Service for common security vulnerabilities using primarily source code review. Consultants also performed some dynamic testing and interacted with deployments in development environments. Consultants reviewed code from the latest snapshots of the `fbsource` and `whatsapp-cf` repositories at the beginning of the review. Meta employees provided timely, much-appreciated help throughout the project in a dedicated Workplace chat group.

Limitations

NCC Group conducted a best effort, time-boxed review. Even with more time, no such review could ever be exhaustive.

1. See WhatsApp's introduction to Private Processing at https://engineering.fb.com/2025/04/29/security/whatsapp-private-processing-ai-tools/.

The code is under active development; the version of the code examined in this review may differ from the code used at the time of the Message Summarization Service's public launch.

Finally, the system relies on the participation of several third parties, some of which play key roles, such as Fastly and Cloudflare, and some of which are more auxiliary, like AMD and NVIDIA. NCC Group's review did not include any of their services or procedures concerning, e.g. key management and access control, or key derivation. Similarly, the code has many dependencies, both open-source and private, and these were not in scope.

Evaluation of Meta's Desired Security and Privacy Assurances

NCC Group evaluated the WhatsApp Message Summarization Service with respect to seven security and privacy assurances specified by Meta.

1. Client-side source code was reviewed to confirm that usage of the service is exclusively under **user control.**
2. Source code review across all participating systems confirmed that **privacy** of user data is maintained in software. Potential hardware attacks were deemed infeasible as long as transparency logs include device IDs. Some common opportunities for strengthening key management were identified. Any practical implementation has some unavoidable risk of traffic analysis, potentially leading to inference about user data.
3. Review of back-end systems confirmed that user data is not persisted after processing, providing **statelessness**; adherence to TLS/RA-TLS and SEV-SNP protocols ensures ephemeral secrets provide **forward security.**
4. **Targeted attacks** on individual users are infeasible under the assumption that Meta does not collude with a third party (e.g. Fastly, Cloudflare). For user anonymity, it is critical that clients obtain Oblivious HTTP key information from a third party.
5. No opportunities to circumvent **purpose limitation** were found in source code review. Several common, minor opportunities for strengthening security were identified.
6. The system's **enforceable guarantees** primarily rely on the robustness of AMD SEV-SNP and NVIDIA Confidential Compute technologies, which are relatively recent and have imperfect track records. However, no system design can ever fully cover unknown firmware- or hardware-level exploits.
7. Fully **verifiable transparency** would require all files and binaries to be open-source and covered by attestation.

Overall, the WhatsApp Message Summarization Service has ambitious security and privacy targets; Meta has invested a significant amount of resources to achieve the leading edge of what is currently possible.

Observations and Key Findings

The assessment uncovered a total of 21 findings, among which the most notable were:

- Finding "CVM Initializes Unnecessary Hypervisor-Provided NICs". The hypervisor could have assigned network interfaces to the CVM through which private data could be exfiltrated.
- Finding "Missing Freshness Check on Transparency Proof for Attested Image". Any old CVM image with known vulnerabilities could have been indefinitely used by an attacker.
- Finding "HPKE Key Configuration Served From Meta". The ability to serve malicious key configurations to WhatsApp clients could have allowed Meta to violate privacy and non-targetability assurances.

All three of these findings were addressed. By the end of NCC Group's review,

- 16 findings were considered "Fixed",
- 1 Low-risk finding was considered "Not Fixed" due to planned development, and
- 4 (3 Low-risk and 1 Informational) findings are considered "Risk Accepted".

Meta provided responses, included in Finding Details, to each of the latter 5 findings.

Conclusions and Strategic Recommendations

The Message Summarization Service uses modern cryptographic protocols and hardware-based security features while striving to be verifiably transparent. Users must trust Meta to not insert any malicious behavior in artifacts that are not open source, to tightly control access to artifact signing keys, and to not collude with Cloudflare or Fastly. NCC Group encourages fully verifiable transparency, which would require open-source code and reproducible builds for all artifacts.

Report Structure

The Section Architectural Overview and Inventory of Systems contains an overview of the Message Summarization Service's architecture and main components, as understood by NCC Group. Next, NCC Group's evaluation of Meta's desired assurances, with references to relevant findings, is in the Section Evaluation of Meta's Desired Security and Privacy Assurances. Sections Review of Cryptography Primitives and Protocols and Review of TEE and LLM Usage provide in-depth reviews of the lower-level components of the service, while higher-level attacks are considered in the Section Potential Service-Level Attacks and Mitigations. The Section Limitations and Opportunities discusses service availability, supply chain attacks, and transparency. The remainder of the report contains finding details, starting with the Table of Findings.

3 Architectural Overview and Inventory of Systems

The WhatsApp Message Summarization Service allows users on iOS and Android to obtain a summarization of a batch of private messages, such as unread group messages. The user initiates summarization for one specific batch of messages at a time, with an explicit tap or button press: initiating the summarization process is explicit, opt-in, and not done by Meta. Despite this service involving the external transmission of private messages, their application to a remote Meta-operated Large Language Model (LLM) in plaintext, and the return of a private summary, users should enjoy similar privacy protections consistent with the rest of the WhatsApp application. Specifically, Meta has articulated several security and privacy assurances for the WhatsApp Message Summarization Service. See the Section Evaluation of Meta's Desired Security and Privacy Assurances for NCC Group's evaluation of these assurances.

An architectural overview of the summarization service is provided through two complementary perspectives:

- following user content as it leaves the WhatsApp client application, arrives at the Meta-operated LLM for summarization, and is returned to the user in the form of an ephemeral message-batch summary; and
- through an inventory of systems and cryptographic materials – some of which are not directly involved with the summarization of user messages, but instead play a supporting role.

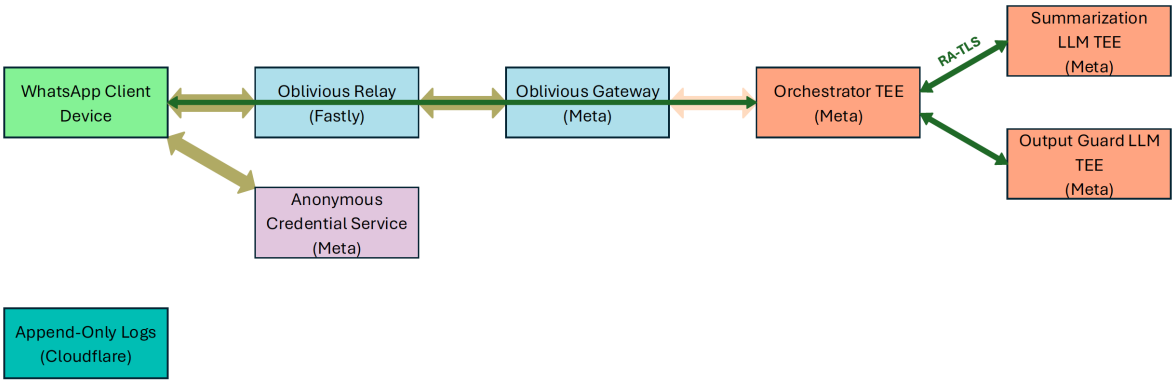

Figure 1: Simplified Architectural Diagram

While several basic security concerns are dealt with inline below, most cryptographic communication- and TEE-related details are deferred to the dedicated sections that follow.

User Content Lifecycle

Summarization will be supported on iOS and Android devices running WhatsApp, where the user is assured to be running the latest authentic version via standard app store practices. Users must trust Meta build processes to prevent the insertion of malicious behavior. While the client source code is not public, Meta considers it "publicly available" via the executable. Verifying the app's behavior would require significant proactive analysis.

The client application contains a number of hardcoded trust anchors, such as a certificate store, the Oblivious Relay URL, and TEE-related Certificate Revocation Lists. User messages are stored only on the local client, not on any remote systems (with the exception of encrypted backups[2]).

2. https://www.nccgroup.com/us/research-blog/public-report-whatsapp-end-to-end-encrypted-backups-security-assessment/

The WhatsApp client first connects to Meta's Anonymous Credential Service (ACS)[3] backend to authenticate and retrieve an ACS token used for authentication in subsequent outbound connections.

Next, the client assembles an outbound message batch containing (i) batch metadata, including a randomly generated UUID and the device locale (country), and (ii) a serialized oldest-to-newest set of messages, each of which include a 1-of-15 message type identifier, a UTF-8 text string, and the sender contact ("push name"). Regarding the various message types, the Android client currently serializes text messages along with captions for image, video, and animated GIF messages, while the iOS client currently only serializes text messages. Other information such as media content (e.g. images, audio, video), location information (other than locale), and version identifiers are not present in the outbound message batch. Disappeared messages are excluded from the message batch.

After the user initiates the summarization activity and the outbound message package is assembled, the WhatsApp client uses the ACS token to send the package to a Meta-operated Orchestrator service endpoint running inside of a TEE. This path involves several steps across several systems and will be described next, followed by the endpoint itself.

The client reaches the Orchestrator service using Oblivious HTTP[4], which "allows a client to make multiple requests to an origin server without that server being able to link those requests to the client or to identify the requests as having come from the same client, while placing only limited trust in the nodes used to forward the messages". The OHTTP protocol has been loosely referred to as "TLS inside TLS", and has similarities (though less complex and robust) with Tor[5]. Additionally, Remote Attestation over TLS[6,7] (RA-TLS) is used to verify the authenticity and integrity of the Orchestrator's TEE.

The outbound flow involves the WhatsApp client constructing an HTTP request intended for the Orchestrator from the message package. This request is in the "inner TLS" which is negotiated between the client and the Orchestrator. This "inner TLS" is encapsulated with Hybrid Public Key Encryption[8] (HPKE). The client sends the encapsulated records to an Oblivious Relay operated by Fastly (via the "outer TLS"). Fastly then forwards the encapsulated records to an Oblivious Gateway operated by Meta. The latter decapsulates and provides the original RA-TLS records using HTTPS to the target Orchestrator service endpoint. Connections are not reused.

The Meta-operated Orchestrator service can be envisioned as an eventual load-balancer of sorts with a safeguarding function. The Orchestrator oversees the forwarding of incoming requests to the Summarization LLM and Output Guard LLM endpoints, which are located within containers in separate TEEs. The Output Guard LLM is a model that checks whether message summaries comply with Meta's standards. These TEE-to-TEE connections also use RA-TLS. Each LLM calculation takes place on its own machine with the model on its own large GPU, currently NVIDIA GPUs in Confidential Computing mode.

The return path is essentially symmetric.

3. https://engineering.fb.com/2022/12/12/security/anonymous-credential-service-acs-open-source/
4. https://www.rfc-editor.org/rfc/rfc9458
5. https://svn-archive.torproject.org/svn/projects/design-paper/tor-design.pdf
6. https://arxiv.org/abs/1801.05863
7. https://datatracker.ietf.org/doc/draft-fossati-tls-attestation
8. https://www.rfc-editor.org/rfc/rfc9180

Inventory of Systems and Cryptographic Materials

This section elaborates the Message Summarization Service's systems and cryptographic materials.

1. **WhatsApp client** – the iOS or Android application with a user's chat messages.
   a. Public PKI certificate store (in OS)
   b. AMD Root public key
   c. AMD Certificate Revocation Lists
   d. Cloudflare public key (transparency)
   e. Fastly signature verification key (HPKE key configuration)
2. **Anonymous Credential Service** – a Meta-operated service to authenticate and provide connection-oriented tokens.
   a. TLS certificate (public PKI), key pair
   b. ACS private key to sign tokens
3. **Oblivious Relay** – a Fastly-operated system providing one half of the OHTTP path that relays messages between users and the Oblivious Gateway.
   a. TLS certificate (public PKI), key pair
   b. Signature key pair (HPKE key configuration)
   c. HPKE key configuration
4. **Oblivious Gateway** – a Meta-operated system providing the other half of the OHTTP path that relays messages between the Oblivious Relay and the Orchestrator.
   a. TLS certificate (public PKI), key pair
   b. HPKE private key
   c. ACS public key
5. **Orchestrator** – a Meta-operated service, running inside a TEE, that decrypts messages from the WhatsApp client and communicates with the Summarization and Output Guard LLMs.
   a. TLS certificate (self-signed), private key injected from TEE into container
   b. Versioned Chip Endorsement Key (VCEK) inside AMD SP
   c. RA-TLS attestation with endpoint in container in TEE
6. **Summarization LLM** – a Meta-operated service, running inside a TEE with NVIDIA GPU, that summarizes WhatsApp messages.
   a. TLS certificate (self-signed), private key injected from TEE into container
   b. Versioned Chip Endorsement Key (VCEK) inside AMD SP
   c. RA-TLS attestation with endpoint in container in TEE
7. **Output Guard LLM** – a Meta-operated service, running inside a TEE with NVIDIA GPU, that checks whether potential responses comply with Meta's standards.
   a. TLS certificate (self-signed), private key injected from TEE into container
   b. Versioned Chip Endorsement Key (VCEK) inside AMD SP
   c. RA-TLS attestation with endpoint in container in TEE
8. **NVIDIA GPU** – for Summarization and Output Guard LLMs, in Confidential Computing mode[9,10].
   a. Device Identity Certificate
   b. Device-unique private Identity Key (IK)
   c. Device-unique Attestation Key (AK)
9. **Cloudflare** – maintains logs of artifacts for transparency, distributes artifact revocation lists.
   a. Signing key pair
   b. Lists of signed transparency artifacts for different namespaces
10. **AMD** – root of trust for TEE certificate chain, issues certificates[11].
    a. AMD Root Key (ARK)
    b. AMD Signing Keys (ASK)
    c. VCEK Certificates
11. **NVIDIA** – similar to AMD, but less well documented.

9. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf
10. https://github.com/NVIDIA/nvtrust
11. https://www.amd.com/content/dam/amd/en/documents/developer/lss-snp-attestation.pdf

4 Evaluation of Meta's Desired Security and Privacy Assurances

This section presents NCC Group's evaluation of the seven desired security and privacy assurances relevant to user trust in the WhatsApp Message Summarization Service. Ideally, a violation of these assurances would require collusion between Meta and another entity. This evaluation is contingent upon the robust resolution of findings listed in the Table of Findings, which are also mentioned in this section.

1. **User Controlled**: Usage of the service is opt-in at the exclusive discretion of the user. Meta is unable to trigger sharing of any data to the summarization service without the user's control.
2. **Private**: Data is always encrypted in transit. User data is not accessible to Meta, not even to those with administrator privileges or physical access to the systems.
3. **Stateless and Forward Secure**: Compromise of the service at a single point in time must not reveal historically processed user data; the service maintains no access to user data after the request is returned to the user.
4. **Non-Targetability**: An attacker should not be able to attempt to compromise personal data that belongs to specific, targeted users without attempting a scaled compromise of the entire service.
5. **Guaranteed Purpose Limitation**: There must be no way to access or exfiltrate user data for purposes beyond the reasons for which the user provided it.
6. **Enforceable Guarantees**: It must be possible to constrain and analyze all the components that critically contribute to the guarantees of the overall service. Deviations from them must lead to user data not being shared with it.
7. **Verifiable Transparency**: Independent security researchers must be able to verify, with a high degree of confidence, that our privacy and security guarantees for the service match our public promises.

The relevant portions of the source code implementation for both the Android and iOS applications were reviewed to confirm that usage of the service is exclusively under **user control**.

Source code across each participating system was reviewed to confirm that data is kept **private**, in part by being encrypted in transit. The assurances provided by hardware, and rooted in the AMD Secure Processor (SP), are potentially vulnerable to glitch attacks. However, these attacks are considered of a lesser concern due to the requirements they involve, including physical access, custom tooling, advanced reverse engineering skills or detailed insider knowledge, time for experimentation, and possibly fortuitous circumstances. The implementation is deemed secure from traffic sniffing or injection over an unencrypted NVLink on a single GPU system, whereas a future multi-GPU LLM implementation could be vulnerable under similar constraints as glitch attacks. Both scenarios are considered to be improbable, under the assumption that device ID verification is applied to ensure that only Meta-owned devices are used (as mentioned in finding "OHAI Proxy Security Impacts TEE Remote Attestation").

Related key management best practices, e.g. for certificate lifespans and private key storage, were also reviewed (see finding "RA-TLS Key Stored in Cleartext Within the CVM" and finding "Excessive RA-TLS Certificate Lifespan"). The resistance to traffic analysis could be slightly improved (see finding "Traffic Analysis on Message Length"), although a practical

implementation will always involve some risk at the margins. Prompt (or unsafe message) injection attacks from malicious conversation participants may intentionally generate responses of recognizable lengths. Relatedly, injection attacks that result in displaying URLs could lead to a loss of privacy (see finding "Conversation Exfiltration Via Cross-User Prompt Injection").

An inspection of the participating systems confirmed there is no persistent user data maintained after processing. Proper use of TLS and RA-TLS ensures the use of ephemeral secrets in encryption keys. As such, the service is **stateless and forward secure**. In principle, the use of RA-TLS ensures the client is interacting with a known service endpoint running in an authentic CVM (Confidential Virtual Machine; see finding "OHAI Proxy Security Impacts TEE Remote Attestation"). The validity of the CVM firmware is guaranteed by SEV-SNP-provided verified-boot mechanism and verifiable via the RA-TLS certificates. The implementation has been reviewed and validated for proper adherence to the SEV-SNP specification.

The security model relies in part on two third-party providers (currently Fastly for unlinkability of user requests, and Cloudflare for transparency proofs), and in particular their non-collusion with any attempt at malicious activity by Meta. NCC Group notes that their contractual obligations are toward Meta, and not toward the WhatsApp users; the security assurances ultimately rely on the threat of reputation damage to these third parties, should they be discovered to be complicit to **targeted attacks** against users. Finding "HPKE Key Configuration Served From Meta" describes the now-remediated risk that Meta could, without colluding with another party, serve malicious key configurations to particular clients. The client application could perform additional anonymization of chat messages, e.g. sender user IDs, as a defense-in-depth measure to increase the difficulty of content attribution.

CVM configurations and logging functionality were scrutinized to ensure that user data cannot be exfiltrated, providing **purpose limitation** (see finding "CVM Initializes Unnecessary Hypervisor-Provided NICs" and finding "Non-ASCII Statements in Exported Logs"), and to ensure that security best practices are followed (see finding "Confidential Workload Executes with Root Privileges", finding "LLM Predictor Binaries Lack Security Hardening", finding "Opportunities for Enhanced CVM Kernel Hardening", finding "Overlay Filesystem Covering the Entire RootFS", and finding "RA-TLS Key Stored in Cleartext Within the CVM"). Remote attestation verification was reviewed to ensure that artifacts, such as CVM images, are thoroughly and correctly attested and verified (see finding "Missing Freshness Check on Transparency Proof for Attested Image", finding "Client Does Not Verify TCB Version in VCEK Certificate", finding "Transparency Proof Verification Bypass", finding "Insufficient VCEK Certificate Validation", finding "Missing Configuration Parameters in Attestation Report", and finding "Potential TEE Type Confusion Attacks"). Several of the assurances protecting the user from malicious actions by Meta primarily rely upon the robustness of the AMD SEV-SNP and NVIDIA Confidential Compute technologies. As a whole, these are relatively recent developments with an imperfect track record. With correct operation, the design will be capable of purpose limitation and **enforceable guarantees**. However, with respect to zero-day exploits, even a proven attestation design cannot fully cover unknown hardware- or firmware-level exploits. The revocation channels are less well developed, and the VCEK keys could be tracked to a finer granularity to ensure devices are under Meta control.

While an excellent position has been established to support **verifiable transparency**, full clarity on which materials are public, which are private behind a specific agreement, and which remain inaccessible, will be necessary to understand claims of reproducibility by third parties. This independent review has had sufficient access to establish high confidence.

There are multiple participants, such as Meta, Fastly, Cloudflare, AMD, NVIDIA and a variety of Internet service providers, that are each able to impact the availability of the WhatsApp Message Summarization Service (see finding "Use of PEM Encoding in Attestation is Fragile"). Users further depend upon assurances from the App Store to ensure the authenticity, integrity, and correct version of the client application. However, these availability- and distribution-related concerns primarily affect the overall WhatsApp application (similar to supply chain attacks), rather than introducing new attack surface or potential vulnerabilities specific to the summarization service. As such, they are outside this assessment's scope.

Overall, the WhatsApp Message Summarization Service has ambitious security and privacy targets; Meta has invested a significant amount of resources to achieve the leading edge of what is currently possible. The exceptions noted above effectively represent corner-conditions that require ongoing user trust similar to that placed in the Internet's public key infrastructure.

5 Review of Cryptography Primitives and Protocols

As noted in the assurances stated in the Section Evaluation of Meta's Desired Security and Privacy Assurances, the WhatsApp Message Summarization Service aims to maintain the security and privacy of user data as it transits various systems. This section provides an overview of several of the cryptographic primitives and protocols employed to support that objective. The implementation review element of the engagement checked for adherence to the relevant public specifications.

RFC 9458 – Oblivious HTTP

The Oblivious HTTP[12] (OHTTP) protocol is designed to decouple a client's network identity from the HTTP request–response exchange. It achieves this by encapsulating binary HTTP messages with Hybrid Public Key Encryption (HPKE) so that neither the Relay forwarding the message nor the Gateway processing it can link individual requests to a client. For the WhatsApp Message Summarization Service, the principal divergence from the specification is that the underlying message is effectively HTTPS rather than HTTP.

Participant Roles

OHTTP defines four principal roles that partition the process of encrypted message forwarding:

1. **Client**: As the originator of HTTP requests, the Client is responsible for constructing binary HTTP messages, selecting an authenticated key configuration, establishing a fresh HPKE static context for every request, and performing encryption (for requests) and decryption (for responses). The Client must also ensure that identifying information is not inadvertently included outside the encapsulated payload.
2. **Oblivious Relay**: Operating as an intermediary, the Relay accepts the client's POST request and forwards the Encapsulated Request to the Gateway. Its role is deliberately minimal—it should not inspect the encrypted payload nor add any metadata that might link subsequent requests.
3. **Oblivious Gateway**: The Gateway terminates the HPKE-protected layer by decapsulating the Client's request. It then forwards the resulting HTTP request to the target resource and, upon receiving a response, encapsulates it using data derived from the established HPKE context.
4. **Target Resource**: The Target Resource is a conventional HTTP server that processes HTTP requests. It receives requests from the Gateway without knowledge of the underlying OHTTP mechanisms. In essence, it remains agnostic to the encapsulation process that shields the Client's identity.

In the case of WhatsApp Message Summarization Service, the inner plaintext HTTP requests and responses that are conveyed through the Relay and Gateway are instead the inner RA-TLS records, which thus make for an extra encryption layer. From the point of view of the Relay and the Gateway, these records are opaque binary objects; they are meaningful only to the Client and the Orchestrator. For simplicity, in this subsection, we use the OHTTP terminology and consider the "HTTP payload".

HPKE Protection

Clients obtain an integrity-protected key configuration for the Gateway that includes a key identifier, the HPKE public key, and a list of acceptable cryptographic algorithm pairs.

OHTTP relies on HPKE to protect the HTTP payload in transit. Every request is protected using a newly generated ephemeral HPKE sender context. By sealing the binary HTTP

12. https://www.rfc-editor.org/rfc/rfc9458

message with this context, the protocol ensures that even repeated requests from the same Client remain unlinkable. On the response path, the Gateway uses the established HPKE context to export a secret value. Encrypting the binary HTTP response with parameters derived from the secret value ensures that only the Client, holding the corresponding decryption context, can recover the plaintext.

Transaction Example

Relative to more traditional HTTP(S) transactions, the protocol path between the requesting Client and the responding Target Resource contains several additional steps.

1. **Request Construction and Encapsulation**: The Client creates a traditional HTTP request for a target resource and encodes it as a binary message. It then builds a header containing a variety of identifiers, assembles an info string with the header, and initializes an HPKE sender context. The Client then derives both an encapsulated key and a sender context. The binary request is encrypted via the HPKE Seal operation, and the final Encapsulated Request comprises the header, the encapsulated key, and the ciphertext.
2. **Request Forwarding via the Relay**: The Client sends this Encapsulated Request as the payload of a POST message to a designated Relay. The Relay, purposefully constrained to simply pass through data, forwards the request to the Gateway without inspecting or altering its encrypted contents.
3. **Gateway Decapsulation and Target Request Transmission**: Upon receipt, the Gateway parses the header and uses the corresponding private key with its HPKE receiver context to decapsulate and decrypt the HTTP request. The plaintext HTTP request is then forwarded to the target resource as a conventional HTTP message.
4. **Response Encapsulation and Return**: After obtaining a response from the Target, the Gateway reuses the previously established HPKE context to export a secret, generates a fresh random nonce, and derives a unique AEAD key and nonce. The plaintext HTTP response is encrypted with these parameters, and the resulting Encapsulated Response is sent back to the Relay, then forwarded to the Client.
5. **Client Decapsulation**: Finally, the Client employs its HPKE sender context to decrypt the response, recovering the original HTTP response.

Adversarial Considerations

While the protocol protects the content of messages from intermediary observers, several adversarial vectors remain in its threat model including Relay misbehavior, traffic analysis, replay attacks, and key configuration.

As the Relay sees the exact boundaries of the encrypted exchanges, this is an ideal position to delay or rate-limit requests. Although the Relay cannot decrypt the payloads, it could perform statistical analysis, such as timing delays or differential treatment, that might reveal correlated patterns that reduce the anonymity of Clients.

Given that the Relay or a network adversary might replay Encapsulated Requests, the protocol relies on unique encapsulation to recognize and mitigate replays. However, replay prevention must be handled by the application layer or by careful windowing on the Gateway side.

Since the Client's key configuration is essential for constructing valid encapsulated messages, care must be taken that these configurations are not unique or infrequently shared among users. Unique configurations could be exploited to track individual Clients even if the actual requests remain encrypted.

Comparison with Tor and Additional Considerations

Unlike Tor, which employs a multi-hop onion routing scheme to obscure digital fingerprints and route traffic through several independently operated nodes, OHTTP relies on a two-hop separation (Client-Relay and Relay-Gateway) with explicit role responsibilities. The cryptographic encapsulation via HPKE ensures sufficient confidentiality of the exchanged data, though it does not hide the size of each encapsulated plaintext packet, which could be leveraged in some limited form of traffic analysis (see finding "Traffic Analysis on Message Length"). In general, this should be sufficient for unlinkability purposes, in scenarios where linking of requests could compromise privacy – such as anonymous DNS queries, telemetry submissions, or location-specific content requests. Deployment of OHTTP requires careful management of key rotation, rate limiting, and tolerances against network delays.

Divergence from RFC 9458

In the WhatsApp Message Summarization Service implementation, OHTTP is used to convey "plaintext" messages which really are the TLS records used by the inner RA-TLS connection. The mechanics of that inner TLS handshake imply that:

- Replay attacks cannot work. TLS already rejects any attempt at replaying individual records or even full handshakes.
- Even the Gateway cannot obtain the actual plaintext; all it sees is the RA-TLS records which negotiate an initial handshake, then convey all data in an encrypted format. Only the two endpoints of that inner TLS connection (the Client and the Orchestrator) can really access the data.

Rate-limiting is furthermore enforced through the ACS tokens. Each connection to the Relay requires usage by the Client of a token, which is obtained from the ACS. Cryptographic blinding is used to ensure that all received requests correspond to a valid token but without allowing Meta to track users through carefully crafted tokens. Since the ACS rate-limits emission of tokens, this inherently implies rate-limitation of Client requests to the WhatsApp Message Summarization Service, even though OHTTP does not provide that rate-limitation by itself.

Proxies and other intermediaries that do not interact with the contents of message payloads are not relevant to the security and privacy assurances.

RFC 9180 – Hybrid Public Key Encryption

Hybrid Public Key Encryption[13] (HPKE) integrates both asymmetric and symmetric cryptography to encrypt arbitrary-sized messages. It uses a key encapsulation mechanism (KEM) to "package" a freshly generated shared secret, derives a symmetric encryption key via a key derivation function (KDF), and uses the key in AEAD encryption.

Participant Roles

Participants in an HPKE transaction include:

1. **Recipient**: Possesses a static key pair sk_R, pk_R and is the sole party able to decapsulate the encapsulated key.
2. **Sender**: Uses the Recipient's public key to generate an ephemeral key pair and encapsulated key (enc). In modes supporting authentication, the Sender also contributes a static key pair sk_S, pk_S or a pre-shared key.

13. https://www.rfc-editor.org/rfc/rfc9180.html

Transaction Example

A typical HPKE transaction might proceed as follows:

1. The **Sender** retrieves the Recipient's public key pk_R and uses the KEM's `Encap` operation to generate an encapsulated value enc and an ephemeral secret value s. Exact details depend on the particular KEM in use. With Diffie-Hellman (DH), or an elliptic curve variant thereof, the sender generates an ephemeral DH key pair (sk_E, pk_E), enc is a serialization of that ephemeral DH public key (pk_E), and s is the result of combining the private part (sk_E) with the Recipient's public key (pk_R).
2. The **Sender** combines the shared secret s with protocol-specific information (including labels, version strings, and any application-specific info) and supplies it to a KDF to derive a shared secret. This secret is then used to instantiate an AEAD encryption context, complete with symmetric key and a base nonce.
3. **Sender** encrypts the plaintext message with the AEAD function `Seal`, incorporating any additional associated data (AAD), and sends the tuple $enc, ciphertext$ to Recipient.
4. **Recipient** uses its private key sk_R and the received encapsulated value enc with the `Decap` operation to derive the same shared secret s, builds its decryption context, and recovers the plaintext via AEAD decryption `Open`.

Adversarial Considerations

From an adversarial perspective, HPKE is designed to achieve Indistinguishability under Chosen Ciphertext Adaptive Attack (IND-CCA2) security provided that the underlying primitives are robust. However, several common considerations remain such as key compromise, the quality of randomness, and both replay and forward secrecy on recipient compromise are left to a higher layer.

Divergence from RFC 9180

Meta has no material divergence from the specification. The HPKE implementation utilizes *libsodium* with Curve25519, SHA256, and ChachaPoly1305.

Remote Attestation in TLS

The RA-TLS protocol extensions[14] augment the TLS 1.3 handshake with remote attestation capabilities. In a "normal" TLS connection, the client obtains the server's public key from the server's certificate, sent as part of the initial handshake, and gains some assurance that the key is correct by validating that certificate against a set of known trust anchors (hardcoded root CA certificates in the client). In RA-TLS as used in the WhatsApp Message Summarization Service, a somewhat different model is used: the server's certificate is self-signed, but it also comes with an *attestation* that the corresponding private key was generated and is held inside a Trusted Execution Environment (TEE) that runs a specific software image. In effect, the attestation binds the cryptographic identity used during the handshake to the underlying platform's security state. This binding makes it possible to confirm not just that the peer possesses the correct private key, but also that the key is managed by a TEE or other attestation-capable module.

The details of the attestation mechanism will be discussed in the next subsection; for now, we take a high-level view in which the attestation is a signed, serializable object called an *attestation report* that is verifiable by third parties.

14. https://www.ietf.org/archive/id/draft-fossati-tls-attestation-08.html

Participant Roles

Several principal roles are integral to the protocol's operation:

1. **Attester**: May be either the TLS client, or TLS server, or both. The Attester is responsible for obtaining the necessary evidence (the *attestation*) from its local attestation service (e.g. within a TEE) to prove the platform integrity and correct key management.
2. **Relying Party**: The Relying Party is the peer of the Attester; it receives the attestation from the Attester, and verifies it. The verification may involve the help of an external trusted verifier. If the TLS client and server are both Attesters, then each is also the Relying Party of the other one.
3. **Attestation Service**: Embedded within the Attester's platform, it provides the local API to generate the attestation tokens. This service ensures that the signing key's provenance is securely bound to the platform identity.
4. **Trusted Verifier**: An external entity that checks the validity and freshness of the attestation evidence. In scenarios where evidence is not sufficient on its own, the Relying Party consults the Trusted Verifier to obtain attestation results that can be cached and reused for subsequent handshakes.

Binding and Key Assurance

The Attester runs in a TEE; the Attestation Service is a secure element which is part of the TEE, and whose role is to validate that the TEE state is as expected, and in particular that it runs a specific software image. The attestation is cryptographically bound to the TLS connection in the following way:

1. At startup time, the Attester generates a new asymmetric key pair $(\boldsymbol{sk}_A, \boldsymbol{pk}_A)$. That key is ephemeral in the sense that a new one is produced at every startup. However, the key might exist and be in use for a long period of time, if the Attester is not often restarted.
2. The Attester requests from its Attestation Service a signed report, which will contain, among other information, a *measurement* of the Attester's state (conceptually, a cryptographic hash value computed over the entire software image that the Attester runs on), and a copy of the public key $\boldsymbol{pk}_A$ (or a cryptographic hash of that key).
3. The Attester produces an X.509 certificate containing $\boldsymbol{pk}_A$ as public key, and, in an appropriately tagged extension, a copy of the entire attestation report. The entire certificate is self-signed, though that self-signature does not really have any useful security property (such a signature, or at least signature-shaped value, is needed to make the resulting certificate processable by standard X.509 certificate decoders). This certificate, and the corresponding private key $\boldsymbol{sk}_A$, will be used as part of the TLS handshake process.

The Relying Party, upon receiving that certificate (as part of the handshake), will extract the attestation report and verify it, and then compare the public key in the certificate with its copy (or hash) in the signed report. If they match, then the Relying Party gains confidence that the peer with which it is setting up a TLS connection runs the correct software in an honest hardware system that behaves as expected.

The description above relates to how RA-TLS is used in the WhatsApp Message Summarization Service. This somewhat deviates from the RA-TLS draft standard, which specifies two modes:

- The "raw" public key ($\boldsymbol{pk}_A$) is sent in the TLS handshake, along with the attestation report, without any encoding as an X.509 certificate.

- The (non-ephemeral) public key is part of an X.509 certificate that the Attester obtained from a normal PKI (not self-signed), and meant to be validated in the usual way *in addition* to the verification of the attestation report; the report is conveyed in an appropriate extension in the relevant TLS handshake message. The contents of the attestation are bound to the TLS connection through cryptographic channel binding.

Meta diverged from this specification in order to ease integration with an existing TLS library; in effect, they follow the first mode (no PKI) but encode the public key and the attestation report into a certificate-shaped object so that the TLS implementation can be used without any modification, except for a custom certificate validation process provided as a callback. Notably, this avoids the need for extra channel blinding activities, and this makes the entire handshake compatible with TLS 1.2.

Usage and Divergence from the IETF Draft Specification

Meta diverged from the description above in the following ways:

- TLS 1.2 may be used, instead of TLS 1.3. Transition work to TLS 1.3 is ongoing. The security properties of TLS 1.2 are sufficient in the context of the use of RA-TLS in the envisioned architecture; the move to TLS 1.3 must rather be understood as a general alignment with future library versions.
- As noted above, Meta does not use specific RA-TLS extensions at the TLS level; instead, the (raw) public key is presented in a self-signed X.509 certificate, inside which are located the relevant attestation-related extensions.

TLS 1.3 is a newer protocol than TLS 1.2, but TLS 1.2 is not inherently vulnerable, and can provide sufficient security, if implemented correctly. Meta's code relies on existing, well-supported third-party libraries (OpenSSL, MbedTLS), and it can be assumed that they provide a solid TLS 1.2 implementation. One minor note is that contrary to TLS 1.3, TLS 1.2 does not offer an inherent support of data padding; padding *could* be used to deter some forms of traffic analysis that tries to infer information on users' identities or data from the lengths of the messages exchanged over RA-TLS. Meta does not currently employ padding in the WhatsApp Message Summarization Service.

Within the WhatsApp Message Summarization Service, RA-TLS is used between the Client and the Orchestrator, and also between TEEs (Orchestrator to Summarization LLM, Orchestrator to Output Guard LLM). For the RA-TLS connection from Client to Orchestrator, the Orchestrator is the Attester, and the Client is the Relying Party. In TEE-to-TEE connections, the connecting TEE (the "client" at the level of the network TCP connection; normally the Orchestrator) is the Relying Party, and the receiving TEE (the "server": Summarization or Output Guard TEE) is the Attester.

Adversarial Considerations

Since standard TLS 1.2[15] or 1.3[16] is used without any modification (except for the certificate validation), the usual security properties of TLS are ensured:

- Data confidentiality is ensured through strong encryption.
- The protocol is immune to replay attacks, and detects malicious alteration of data, including duplication, deletion or reordering of records.

On the other hand, TLS encryption does not by default hide the *size* of the data that it conveys. TLS 1.3 optionally includes provisions for *padding* individual plaintext records,

15. https://www.rfc-editor.org/rfc/rfc5246
16. https://www.rfc-editor.org/rfc/rfc8446

which can help with hiding the original sizes to some extent, though at the cost of increased bandwidth usage.

The main difference between RA-TLS and plain TLS is the change of security model with regard to the peer authentication. In "normal" TLS, the peer is authenticated by verifying the X.509 certificate against known roots, and assuming that all out-of-band processes for certificate issuance ensure that only the expected system, running the expected software, will have access to the private key corresponding to the public key in the certificate. The overall security assumption is that the PKI will not issue a certificate for a public key which is not under exclusive control of the entity whose name is specified in the certificate. In RA-TLS, the trust is moved onto the attestation service: the client obtains an assurance that the expected software, running on "honest hardware", is at the other end of the connection. The main security assumption is that all verifiable attestation services perform reliable measurements of TEE states, and all these TEEs ensure the confidentiality and integrity of the data they handle.

The RA-TLS model has the benefit of more closely binding the system's behavior (the software it executes) to the properties verifiable by the peer, whereas in the usual PKI model, the peer must assume that the system administrators of the authenticated system are running the correct software and prevent attacks from third parties. On the other hand, the RA-TLS model is looser with regard to the identity of the attester: the Relying Party learns that the Attester is running the expected software in the expected kind of hardware, but this hardware is not necessarily one of those maintained by Meta. Within the overall WhatsApp Message Summarization Service architecture, for remote clients (i.e. users' phones), some of that assurance is obtained from a different piece of the protocol, namely the OHTTP Gateway's public key for HPKE: since all RA-TLS records are sent encrypted with that public key, using the HPKE format, the client knows that if the RA-TLS handshake succeeds, then at least Meta's infrastructures were involved.

AMD SEV-SNP

The TEE is powered by AMD Secure Encrypted Virtualization (SEV), and in particular the Secure Nested Paging (SEV-SNP)[17] mechanism, introduced in 2020. The TEE runs as a virtual machine (guest); the hypervisor is responsible for managing the VM and its interactions with the physical world. The hypervisor is not part of the trusted environment; various technologies are employed to prevent the hypervisor from being able to inspect or alter the guest memory and operations.

Memory Protection

In SEV-SNP, the guest memory can be encrypted. This is performed on a per-page basis. The guest chooses which pages must be encrypted; in general, this will be all guest pages except for a few dedicated pages that serve as I/O buffers with the outside world. Encryption is applied when the data leaves the CPU; data is not encrypted when in the in-CPU caches, and the additional latency of encryption then applies only to accesses which are cache misses (and thus already have high latency). A performance-conscious guest may choose to keep some pages unencrypted, but the expected gain is slight in general, and removal of encryption should be done only after a careful analysis of the security consequences. The encryption uses a key generated by the CPU itself at boot time. The employed algorithm is not really documented, though information from the SEV Secure Nested Paging Firmware ABI Specification[18] seems to indicate that the method is based on

17. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf
18. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf

either AES128 in XEX mode[19], or AES256 in XTS mode[20]. XTS is an improvement over XEX, in that XEX uses the same "tweak" value for all blocks, while XTS generates distinct tweaks. Since older AMD documents only talk about "AES128 encryption", the AES256-XTS mode is probably available only in the most recent CPUs.

In addition to data confidentiality, the CPU enforces memory integrity through use of a dedicated table maintained in system memory (which is not directly accessible by the hypervisor), called the Reverse Map Table (RMT). The RMT keeps track of physical page owners (guest, hypervisor...) and blocks unauthorized accesses; it also remembers the page address (as seen by the guest) to prevent the hypervisor from moving pages in the guest space without the guest consent. The security mechanism relies on a two-level page mapping process between the hypervisor-maintained and guest-maintained page tables. The RMT does *not* use cryptography.

It is noteworthy that memory confidentiality (through encryption) and integrity (with the RMT) rely on slightly distinct threat models. The RMT works as long as the system memory is inaccessible to attackers; this is true for hypervisor software, but not for physical attackers interfacing directly with the DRAM chips. An industrious attacker would make their own DRAM chips and access the RMT directly, possibly modifying its contents. The integrity threat model employed by AMD implicitly assumes that attackers cannot access the DRAM contents directly, without help from the CPU; but the confidentiality threat model uses encryption precisely under the assumption that attackers *can* access the DRAM contents. The two models can be reconciled into an aggregate model in which attackers might read remnant DRAM contents after a hard reset (this is known as a cold boot attack[21]) but not on a live system; integrity is not a relevant concept for that scenario.

Secure Processor and Remote Attestation

Under the SEV-SNP model, the guest is protected from attacks by the hypervisor; remote attestation is used to convince a remote client that it is talking to a specific guest, running the expected software image and inside a proper CPU using SEV-SNP. An essential part is the Secure Processor (SP), which is a small autonomous CPU (ARM Cortex-A5 core) which has its own operating system. The SP performs the initial boot sequence, starting up the x86 cores; it also maintains an asymmetric key pair called the Versioned Chip Endorsement Key (VCEK). The SP uses the VCEK to sign an *attestation report* which includes a "measurement" of the software loaded into the guest (conceptually, a hash of the operating system and application image that the guest was started on). The attestation report includes a free field into which the guest can insert arbitrary data.

As described in the previous section, the WhatsApp Message Summarization Service architecture leverages the attestation in conjunction with RA-TLS by obtaining an attestation report that covers both the software image that the TEE runs, and a hash of the public key that the software generated at boot time and will use for RA-TLS handshakes. By verifying the attestation, the peer obtains a guarantee that the RA-TLS connection is established with the correct kind of hardware, in a VM with confidentiality and integrity protection (even from the hypervisor itself), and running exactly the software image that it should.

VCEK Derivation

The VCEK is held and used by the secure processor. It is *versioned*: it really is a derived key pair from an internal seed. The seed is a per-chip secret key, which is injected in the chip during its production, and stored as a series of write-once fuses. The VCEK is derived from

19. https://en.wikipedia.org/wiki/Xor%E2%80%93encrypt%E2%80%93xor
20. https://en.wikipedia.org/wiki/Disk_encryption_theory#XEX-based_tweaked-codebook_mode_with_ciphertext_stealing_(XTS)
21. https://en.wikipedia.org/wiki/Cold_boot_attack

the seed through a hash-based key derivation process, which is not documented; it has been reverse-engineered in 2021[22] but later CPU versions (or updates of the SP firmware) may have changed that process. The resulting key pair appears to employ the standard NIST elliptic curve P-384, to be used with the ECDSA signature algorithm.

The VCEK derivation uses as inputs not only the secret per-chip seed, but also the *TCB version*, which is a set of version numbers that qualify the upgradable components that the chip uses, namely the SP firmware, the microcode patch level of the x86 cores, and a few similar values. The point of this derivation is to be able to recover from SP firmware and microcode flaws; by requiring use of a specific VCEK when validating the attestation, the client can ensure that the TEE it is talking to is not using the known weak firmware and microcode versions. In more detail:

- When the CPU is powered, the SP starts, using an embedded ROM. This code makes the SP fetch its firmware from an external Flash chip. The firmware is signed (by AMD) and the SP verifies that signature before starting to execute the code in the firmware.
- Before starting up the x86 cores, the SP may also load some *microcode updates*. The x86 cores, by default, use the microcode which is included in the CPU hardware; the microcode updates can alter that microcode through hooks into an in-CPU *patch space*. Microcode updates are signed (again by AMD), and the SP will refuse to load an update without a valid signature.
- For both the SP firmware and the microcode updates, some anti-rollback features are included in the CPU, so that once newer versions have been loaded and committed to, they can no longer be reverted to an older version.
- The TCB version which is used in the VCEK derivation uses the version numbers (each over one byte) of the SP firmware and microcode updates that the SP loaded *before* starting the x86 cores.

The last property is important, in relation to a recently announced flaw[23]. Though the technical details are not fully described yet, the gist of the flaw is that signature forgeries on microcode updates were feasible. Crucially, these microcode updates appear to be doubly signed, with two distinct mechanisms: one used by the SP, in the start-up process described above; the other was used by the x86 cores themselves (in their microcode) to allow *runtime* load of microcode updates. Only the second mechanism was found vulnerable; the forgeries would not allow an attacker to make a malicious microcode update be loaded *by the SP*. Since only the pre-boot updates loaded by the SP have an impact on the TCB version, it follows that the VCEK derivation cannot be modified by such signature forgeries on fake microcode updates. By requiring a VCEK using a microcode patch level known to fix the issue, the Relying Party consuming the attestation report can make sure that it is talking to a TEE running on a CPU that is not using a malicious microcode update.

VCEK Validation

The whole process above assumes that the Relying Party knows the VCEK. Since these keys are chip-specific, they cannot be conveniently hardcoded in the Relying Party implementation. Instead, the VCEK is validated by the RP using X.509 certificates.

AMD maintains cloud systems that, given a CPU identifier and TCB version contents, can compute and return the VCEK (public part) of that CPU. The VCEK is returned as an X.509 certificate issued by a custom intermediate CA owned by AMD. The process is described in the VCEK interface specification[24]. The AMD intermediate and root CAs use RSA-4096 keys and PKCS#1 v2.1 signatures (RSASSA-PSS). The client is supposed to obtain a copy of these

22. https://github.com/PSPReverse/amd-sp-glitch/blob/main/payloads/bl_seed_to_vcek.py
23. https://github.com/google/security-research/security/advisories/GHSA-4xq7-4mgh-gp6w

certificates; in practice, the TEE fetches that chain and sends it to the client, as part of the RA-TLS handshake. The client validates the chain using standard X.509 procedures; only the relevant root CA keys need to be hardcoded in the client code.

The X.509 certificates are valid for 7 years, and can thus be cached for a long time. AMD also maintains revocation information through downloadable Certificate Revocation Lists (CRLs), though only for the intermediate CA level (i.e. the CRL is signed by the root and covers the revocation status of the intermediate CA itself); there is no revocation support for certificates containing the VCEKs themselves. These CRLs are thus empty and expected to remain so; moreover, they are valid for one year. Of course, such revocation works only for compromises that AMD has been made aware of. Fresh CRLs will have to be downloaded on a yearly basis.

Meta Divergence

Meta has no substantive divergence from the specification or standard practices described above.

Software Image Validation

Relying parties receive measurements (hash values) of the software images running in the TEE and container that they are connecting to. They must still verify those values against reference values. In order to smoothly support upgrades to the TEE software, a reference value is obtained as a *transparency proof* – a protobuf structure containing the hash value, signed by a third party (Cloudflare). The Relying Party validates that signature; on success, it accepts the proof. The importance of a freshness check was highlighted in finding "Missing Freshness Check on Transparency Proof for Attested Image", and the importance of type indication was pointed out in finding "Potential TEE Type Confusion Attacks". Transparency proofs now contain a timestamp that is checked for freshness, and a namespace, identifying the type of artifact that was measured (e.g. Orchestrator TEE image).

24. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/57230.pdf

6 Review of TEE and LLM Usage

Trusted Execution Environment (TEE) Usage

The security of the WhatsApp Message Summarization Service is built upon the security mechanisms provided by AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) technology. AMD SEV-SNP enables the TEE in which the guest VM, known as Confidential VM (CVM), runs, ensuring confidentiality and integrity for its memory and protecting it from the hypervisor and the host system.

In an SEV-SNP virtual machine, memory is encrypted by an encryption engine that operates within the memory controller, in a way that is transparent to the CVM. The keys for the encryption engine are managed by the AMD Secure Processor (ASP) and are unique to the guest VM. The ASP allows the guest VM to request an attestation report that includes platform's trusted computing base (TCB). This report can be extended with the VM's public key hash, which the product uses to authenticate the generated RA-TLS key.

The summarization service employs the SEV-SNP Identity Block[25] mechanism to verify the integrity of the firmware, including the bootloader (OVMF or Project Oak Stage 0), initrd, the Linux kernel, and command line arguments of the guest VM. Moreover, dm-verity is configured to ensure the integrity of the root filesystem.

The provided Identity Block also includes the guest policy, which disallows debugging and VM migration. These restrictions are enforced by the SEV-SNP firmware. The guest policy is likewise included in the attestation report, making it verifiable by the remote peer.

The NVIDIA H100 GPUs support Confidential Computing[26], allowing the AMD SEV-SNP provided TEE to be extended to include the GPU. The Secure Protocol and Data Model (SPDM) protocol is used to establish a secure channel between the NVIDIA driver running within the CVM kernel and the GPU hardware. Session keys are negotiated to encrypt and ensure the integrity of the communication over PCIe. Additionally, attestation of the GPU is provided over SPDM to verify its authenticity. However, the details of how the NVIDIA attestation works are not publicly documented. The verification is performed through an NVIDIA-provided SDK, which ultimately uses NVIDIA-maintained cloud systems as trusted verifier. Local caching may be employed to make external network calls infrequent.

All communication to and from the CVM goes through the virtio sockets and is directed to the loopback interface where their communication servers are listening. No other network access is made available to the CVMs.

The reviewed architecture documents call for a CVM running a hardened, minimal Linux kernel and that binary hardening should be applied to filesystem binaries. CVM kernel hardening was implemented during the review (see finding "Opportunities for Enhanced CVM Kernel Hardening"); however, binary hardening was not yet complete at the time of the review.

Vulnerability management for the application binaries running within the CVM is in place, and third-party dependency scans run continuously to identify vulnerabilities by matching them against known CVEs. Meta enforces strict policies that mandate addressing any issue with a CVSS score of 4 or higher.

Host hardening is developed using an in-house BPF-based jailing solution that manages roles and policies assigned to system binaries, defining access controls for system

25. SEV Secure Nested Paging Firmware ABI Specification, rev. 1.57, January 2025. Section 4.6: Identity Block. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf
26. https://developer.nvidia.com/blog/confidential-computing-on-h100-gpus-for-secure-and-trustworthy-ai/

resources. The solution also enables signature verification for binaries, libraries, and configuration files.

Large Language Model (LLM) Usage

LLMs provide the functionality of the WhatsApp Message Summarization Service: they take a message batch as input and produce a summary. They also perform checks on the message-summary output to see whether responses comply with Meta standards.

System Prompts

WhatsApp's prompting systems can be divided into three primary categories:

- Prompt for summarizing lengthy message chains
- Prompt for summarizing short message chains
- Prompt for safeguarding message summary

The first two systems contain a predefined prompt template with a single `messages` variable, which consumes the message history from the target WhatsApp conversation. Both summarizers are initialized with a template that instructs the model to perform a summarization task on the message history, including how to parse message contents and format its output. The prompt for the Output Guard LLM instructs the model to check whether generated message summaries comply with Meta's standards.

NCC Group evaluated the applicable system prompts at the time of the review and concluded that the content of all prompts was fit for their respective tasks. None of the supplied system prompts appeared susceptible to subtle manipulation or unstable behavior outside the bounds of typical LLM variance.

Model Selection

A "model" refers to the set of weights used to process inputs in a neural network. In short, the model is the core of how a machine learning system processes input and makes decisions.

Meta leverages the same model for both summarizer systems: the open-weight `Llama 3 8b Instruct` model[27]. The Output Guard LLM leverages an internal model private to Meta for identifying if summaries comply with Meta's standards. Both models appear fit for purpose and each resides on a single GPU, although NCC Group notes that a fine-tuned summarization model may offer better performance than the general-purpose Llama-series model. However, this attribute entirely impacts output quality and does not increase risk to end users.

Prompt Injection

Prompt injection is an attribute common to all Large Language Models. Because LLMs are trained to predict text tokens, the capacity to follow instructions is an emergent property resulting from the richness of language rather than an inherent element of these models' learned functionality. Instruction-tuned models are exposed to additional examples of instructions to increase the probability of alignment with system prompt commands, but as statistical systems, all LLMs as of Winter 2025 can be manipulated by untrusted input and produce output unwanted by the model owner.[28]

Consequently, prompt injection is not a traditional "vulnerability" insofar as it can be directly mitigated. Similarly, guardrails, safety mechanisms, and fine-tuning are at best defense-in-depth controls and not considered hard security boundaries. Instead, models must be integrated within application architectures that account for potential manipulation by

27. https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct
28. https://www.nccgroup.com/us/research-blog/analyzing-ai-application-threat-models/

untrusted input. For example, applications can dynamically reduce the permissions associated with a particular model at runtime, dependent on the trust classification of data that model consumes.[29]

Because Meta's LLM is granted no access to sensitive functionality, the maximum impact of prompt injection is limited to an incorrect summary provided to users. Additionally, due to the text nature of the platform, prompt injection attempts are likely to be exceedingly visible and detectable by other users. NCC Group does not consider prompt injection a serious risk to the security of the WhatsApp Message Summarization Service.

Attack Vector Analysis

NCC Group evaluated feasible attack vectors relevant to the WhatsApp Message Summarization Service according to two classifications: Attacks that can be conducted by Meta, and attacks that can be conducted by third parties.

Meta-Based Attack Vectors

Because LLMs consume text-based input, all instructions and user input are stored as a single block of text. Although NCC Group validated the contents of each system prompt, deliberate malicious input passed into the `messages` variable could alter the behavior of the summarizer systems. If Meta could prepend or append arbitrary data to a user's prompt, the organization could inject new instructions into the system prompt that induces the summarizer to behave maliciously, such as generating a link that, when clicked, exfiltrates the conversation history. For example, Meta could attempt to program the system to include malicious instructions whenever a trigger word is detected, converting the system's behavior to that of a threat actor.

However, the runtime images of both the Orchestrator and Inference systems are included in the attestation reports. The system has been architected such that any changes to the program within the Orchestrator or evaluators are evident in the output of the attestation report and could be detected by end users. Consequently, this attack path is inviable.

Meta may also attempt to change the model weights at runtime, impacting the behavior of the model in all situations. This attack would similarly be detectable in the attestation bundle via transparency proofs, and would be detected and thwarted at boot time before being made available for inference.

Third Party-Based Attack Vectors

Threat actors who aim to poison the output of the model would be forced to include prompt injections in exceedingly visible contexts, and would already have access to the target conversation. However, this attack vector may prove viable, if overly visible, for threat actors added to conversations later in the conversation chain, but not provided the conversation history. NCC Group created finding "Conversation Exfiltration Via Cross-User Prompt Injection" as a result of this attack vector.

Additionally, if the system were updated in the future to include multi-modal inputs, threat actors may be able to abuse multimodal prompt injection[30] to embed undetectable malicious data in otherwise-benign images. This attack could be exploited by threat actors who have no access to target chats by coercing unwitting users to send poisoned images (e.g. memes) to target chats that instruct the LLM to inject links to third party resources in its summary that exfiltrate conversation content. In its current state, the system is not vulnerable to this attack vector, but this risk should be considered in future updates to the platform.

29. https://www.nccgroup.com/us/research-blog/analyzing-secure-ai-architectures/
30. https://protectai.com/blog/hiding-in-plain-sight-prompt

7 Potential Service-Level Attacks and Mitigations

Many of the summarization service components are relatively standardized and straightforward to understand in isolation. This section describes a number of potential *service-level* attacks and their compensating mitigations. The intent is to better clarify the service's resilience and where gaps may require users to ultimately place their trust. This is not intended to be a comprehensive threat model, and the attacks are presented in no particular order. Present limitations and future opportunities are more tightly summarized in the next section.

Malicious or Altered WhatsApp Client

An attacker could build or distribute a doctored WhatsApp client that silently forwards a user's messages and summaries to a rogue destination. Because the user interface and code are not open sourced, determined adversaries or insiders might package a modified client capable of bypassing normal user controls (e.g. automatically pressing the button to initiate summarization or sending captured text via hidden channels). The potential for side-loading Android applications exacerbates this risk.

In defense, standard app store policies and code-signing practices are relied upon, as well as generally discouraging of the installation of unauthorized client forks. Beyond that, there is no explicit mechanism to certify or validate the client's integrity within the user's device at runtime beyond it being open to public inspection. Note that this risk is fundamental to the entire application, not the incrementally new summarization service.

Zero-Day Exploit in the TEE Firmware

By discovering and exploiting a vulnerability in AMD SEV-SNP firmware (or its supporting stack), an attacker could inject code at the Orchestrator or Summarization LLM layer. This would allow direct reading of messages in plaintext, violating the "Private" and "Stateless" assurances. Even short-lived data in memory would become visible to the attacker.

In defense, the system relies upon remote attestation to verify the authenticity of trusted software, but no "perfect forward-looking" guarantee can be provided that the software stack is free of exploitable flaws. AMD (and the industry's) track record must be considered. RA-TLS ensures only that some recognized firmware is in use, not that it is zero-defect. Hence, a successful TEE exploit remains feasible. Some partial mitigation against such attacks can be obtained with monitoring and revocation; see finding "OHAI Proxy Security Impacts TEE Remote Attestation".

Note that glitch attacks capable of extracting secret material remain a viable avenue of exploitation, and are unlikely to be fully removed[31].

Correlating Traffic to Identify Summarization Requests

The service depends on Oblivious HTTP (OHTTP) and an external Relay-Gateway split to mask the user identity from Meta's Orchestrator. However, a colluding or compromised Fastly-operated Relay and Meta-operated Gateway could analyze timing, volume, and frequency patterns of requests to link them to individual users' IP addresses. Similarly, Fastly serving different HPKE key configurations to individuals or groups could allow Meta to differentiate them at the Gateway. This would undermine the "Non-Targetability" assurance by permitting user identification or location correlation without requiring a broad compromise of the entire service.

31. https://i.blackhat.com/EU-21/Wednesday/EU-21-Buhren-One-Glitch-to-Rule-them-All-Fault-Injection-Attacks-Against-AMDs-Secure-Processor.pdf

In defense, note that this would require collusion between Fastly and Meta. Although OHTTP is advertised to decouple client identity from request content, this review notes that it cannot fully defend against traffic analysis in the presence of collusion. Therefore, an attacker with vantage across multiple network points can correlate traffic patterns despite OHTTP's cryptographic wrapping.

As discussed in finding "Traffic Analysis on Message Length", there are opportunities to further improve resistance against traffic analysis via additional padding, batching, or carefully randomized request timing.

Replay Attacks via Oblivious HTTP

Because an OHTTP message is packaged and forwarded through multiple hops, an attacker with an advantageous network position could record specific Encapsulated Requests and replay them later to the Orchestrator. If the Summarization LLM or Orchestrator processes repeated queries without robust duplication checks, the "Stateless" assurance may be violated. Replays can cause unintended repeated summarization, or allow differential analysis, potentially revealing derived information about the same set of messages.

In defense, the data conveyed in OHTTP consists of the RA-TLS records, and RA-TLS efficiently protects against replay attacks. A TLS handshake cannot be successfully replayed, and data within a given TLS connection cannot be altered in any way by network attackers without being detected by the receiving side.

GPU Memory Exposure

The Summarization and Output Guard LLMs operate on NVIDIA GPUs in a Confidential Computing mode that is a relatively recent development. Any oversights within the GPU attestation feature could leave memory buffers at risk if the GPU firmware or driver stack has exploitable vulnerabilities. Attackers with privileged access to the GPU could scrape ephemeral user data during summarization. Similarly, if the LLM models span multiple GPUs then internal data may be exposed or injected in plaintext.

In defense, while the architecture utilizes an NVIDIA H100 GPU with newly developed Confidential Computing support, the template for fully hardened GPU attestation has been well set, and in any event the user must trust the revocation processes (which are not currently online). Regarding exposed internal model values over the unencrypted NVlink, an attacker would require physical access, specialized equipment, knowledge of internal model layout, and advantageous plaintext in order to conceive of a result. Advantageous plaintext is data representing values either very early in the processing or very late in the processing – this situation is nearly unrealizable.

As with the Zero-Day Exploit in the TEE Firmware attack noted above, revocation capabilities are the primary defense.

Side-Channel Extraction from the Orchestrator

Even if the Orchestrator TEE is not directly compromised, hardware-based side channels (similar to Meltdown, Spectre, or new variants for AMD SEV-SNP) could allow an attacker co-resident on the same physical host to infer user data from cache or timing patterns. This violates the "Private" assurance if the attacker can glean partial or full conversation content from side-channel signals.

Regarding software side-channels, one of the classic objectives of cryptography code review is to ensure constant-time operation relative to secret data. Regarding hardware side-channels, AMD SEV-SNP memory encryption primarily protects data outside CPU caches, but does not itself guarantee the absence of microarchitectural side-channel

vectors. Due to the number and complexity of systems involved, this latter aspect is essentially an accepted risk.

As discussed in finding "Opportunities for Enhanced CVM Kernel Hardening", hardening the CVM kernel (which has been implemented by Meta) may make side-channel exploitation less feasible.

Prompt Injection by Conversation Participants

LLMs predict tokens based on seen inputs. An adversary already inside a conversation can insert maliciously crafted text that instructs the Summarization LLM to leak or highlight sensitive elements in the conversation. Even if only a subset of the text is normally expected in the final summary, a cunning injection can coax the LLM to output more details than intended.

The architecture acknowledges that prompt injection is inherent to LLMs and that the Summarization LLM is fully subject to user-supplied text. There is no explicit mechanism that ensures partial or sanitized summarization (beyond the Output Guard LLM) once the LLM receives the entire conversation. While this treads close to Meta's intended security and privacy assurances, it remains apart and does not violate them. However, as discussed in finding "Conversation Exfiltration Via Cross-User Prompt Injection", there are client-side mitigations that could increase the resistance to these attacks.

Plaintext Data at the Oblivious Gateway

Oblivious HTTP uses HPKE to encrypt data in transit, but the Gateway described in the architecture necessarily decapsulates that data to pass along the HTTPS request to the TEE-based Orchestrator. If an attacker gains control over the Gateway or logs memory from it, they directly obtain these user messages, which could undermine the "Private" and "Guaranteed Purpose Limitation" assurances.

In defense, the Gateway is an intermediary within a RA-TLS transaction and does not have access to plaintext. Nonetheless, this highlights that both the Gateway and Relay systems should be (at least) hardened to industry best practices. This is discussed further in finding "Conversation Exfiltration Via Cross-User Prompt Injection".

Undetected Debug Logging within the TEE

Even though the Summarization and Output Guard endpoints are "stateless," certain forms of error handling or debug logging (especially in HPC or ML frameworks) may temporarily write conversation fragments to disk under stress conditions. A misconfigured TEE or an extraneous physical exception in the model pipeline might produce logs that remain on the system. This would violate the "Stateless and Forward Secure" assurance if older logs persist.

In defense, the code review objectives specifically call out the implementation of logging, crash dump scrubbing, and/or the possibility of memory forensics in the event of a system crash.

Malicious Output Guard LLM Behavior, Supply Chain Risks

The Output Guard LLM runs in an lxc container in a separate TEE but, if it is internally reconfigured or replaced (e.g. perhaps through a supply chain attack within Meta), it might systematically record or leak the user's conversation data. Even ephemeral memory is accessible continuously while the LLM operates on it. An operator or attacker with control over that TEE or container could store or export data beyond the stated purpose. This undermines the "Guaranteed Purpose Limitation" and "Verifiable Transparency" assurances if changes in the Output Guard LLM go unreported or if no external verification is performed.

In defense, the Orchestrator communicates with both the Summarization and Output Guard LLM hosts over RA-TLS. This ensures validated CVM and container images. While it only verifies that known images are loaded, the importance of revocation is highlighted. Supply chain attacks are much broader than the summarization service, and apply to the entire application and the full suite of Meta applications. See the discussion on handling CVEs in the Section Review of TEE and LLM Usage.

Lack of Mutual Authentication in RA-TLS Connections

The implementation of RA-TLS verifies the integrity of the server, but the server does not verify the integrity of the client. In the normal operation flow, this is not an issue, as the RA-TLS connection between the WhatsApp client application and the Orchestrator verifies the integrity of the Orchestrator. The verified Orchestrator, in turn, verifies the integrity of the LLM TEEs, ensuring the client data is kept within verified TEEs at all times.

Since the TEEs do not verify the integrity of the Orchestrator, there is a possible attack in which the legitimate Orchestrator is temporarily replaced with a rogue client to attempt to exploit the Summarization and Output Guard TEEs. In this way, once the exploitation is complete, the attacker could restore the original system, leaving the legitimate Orchestrator unknowingly communicating with the compromised TEEs and exposing client data.

Since the predictor does not save any state coming from user requests, it cannot be “polluted” with rogue requests. However, to reduce the risk of the attack vector, it is necessary for the TEEs to be able to tolerate (i.e. properly reject) “hostile” requests.

8 Limitations and Opportunities

The code review elements of this assessment produced a number of findings listed in the Table of Findings. This section contains several broader issues worthy of discussion: availability, supply chain attacks, and transparency.

Availability

As noted in the Section Architectural Overview and Inventory of Systems, the WhatsApp Message Summarization Service involves a number of systems operated by several independent entities, so the overall availability of the system is outside of the assurances provided. This exception is in respect to an all-or-nothing scenario rather than an ability to target any individuals or groups.

Supply Chain Attacks

The code was reviewed for outdated dependencies, and the build flow for business logic errors that could result in security vulnerabilities, such as unintentionally enabling the presence of debug features in production. However, the risk of supply chain attacks, either from the existence of an insider threat able to push code to the build pipeline or from hidden backdoors in dependency code, has not been considered in depth. Likewise, opportunities for attacking the supply chain of the involved hardware and the consequences thereof have not been considered.

Transparency

In the current solution, four types of artifacts will be released under the binary transparency solution: the Confidential Virtual Machine (CVM) image, the container image and config file, the LLM Model, and the LLM System Prompts. The CVM and application container images are set to be publicly accessible such that anyone (pending the acceptance of some terms and conditions) should be able to verify them. The other items are private and will only be accessible to third parties under NDA. Their hashes, together with their respective transparency proofs, are public and will be verified during the client verification process.

NCC Group noted that some of the terminology used around the transparency solution is currently slightly misleading. The different reference documents describing the architecture of the Private Processing solution refer to *inclusion proofs* in the context of binary transparency. Some parts of the implementation similarly use this terminology. The conventional use of that term in the context of binary transparency implies that the artifact is included in some sort of append-only log, usually backed by one (or more) Merkle Trees. Verifying the inclusion proof means ensuring the hash of that artifact is included in the Merkle Tree, which is done by recomputing the tree root and comparing it with the currently published one. Currently, the solution does not appear to leverage such a transparency solution. The artifacts are signed and published on a third-party server (currently Cloudflare). When writing externally facing documentation, proper and universally accepted terminology should be used: the transparency solution not only relies on the ability to verify signatures on binaries, but also on public documentation detailing exactly what is being signed and what assurances are provided by signing these artifacts; precise documentation is a key part of the assurances the solution provides.

Ideally, a binary transparency solution would rely on open-source software and ensure the build process is reproducible. Researchers could then download the source code, build it following the given process, and ensure the binary they obtain is valid under the published signature. The CVM and container images portion of the transparency solution appear to somewhat follow that approach, at least with regard to open access. The benefits of the transparency solution for the private artifacts may be slightly more difficult to advocate for. A precise framework should be put in place and followed during third-party reviews, particularly for recurring reviews. This framework would help provide assurance that the

LLM Model and the System Prompts behave as advertised. Transparency around how the LLM Model and the System Prompts are injected into the CVM should also be provided, perhaps by open sourcing the relevant implementation and configuration files.

Additionally, a potential avenue to inspire more confidence could be to escrow the private artifacts with a third-party. If concerns around the non-public artifacts were ever raised, it would be possible to prove and verify exactly what the TEE was running.

Finally, another important aspect of the transparency solution relates to who has the ability to publish binaries (and obtain signatures on them) to the Cloudflare service. Best practices should be followed to restrict the number of actors that have such authorization.

9 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.

Title	Status	ID	Risk
HPKE Key Configuration Served From Meta	Fixed	PRD	Medium
Missing Freshness Check on Transparency Proof for Attested Image	Fixed	THQ	Medium
CVM Initializes Unnecessary Hypervisor-Provided NICs	Fixed	PWB	Medium
Confidential Workload Executes with Root Privileges	Fixed	PKA	Low
Missing Configuration Parameters in Attestation Report	Fixed	DKA	Low
RA-TLS Key Stored in Cleartext Within the CVM	Risk Accepted	D7D	Low
Opportunities for Enhanced CVM Kernel Hardening	Fixed	4CG	Low
LLM Predictor Binaries Lack Security Hardening	Risk Accepted	2QT	Low
OHAI Proxy Security Impacts TEE Remote Attestation	Not Fixed	BAR	Low
Conversation Exfiltration Via Cross-User Prompt Injection	Fixed	NRE	Low
Client Does Not Verify TCB Version in VCEK Certificate	Fixed	C6J	Low
Transparency Proof Verification Bypass	Fixed	PWU	Low
Insufficient VCEK Certificate Validation	Fixed	7DN	Low
Excessive RA-TLS Certificate Lifespan	Fixed	YF2	Low
Potential TEE Type Confusion Attacks	Fixed	H7H	Low
Overlay Filesystem Covering the Entire RootFS	Risk Accepted	JXF	Low
Non-ASCII Statements in Exported Logs	Fixed	FDV	Low
Traffic Analysis on Message Length	Risk Accepted	TR3	Info
Use of PEM Encoding in Attestation is Fragile	Fixed	7VK	Info
Missing Network Links for Remote Attestation Validation	Fixed	G3Q	Info
Fragile Filtering Approach for Logs	Fixed	VDT	Info

10 Finding Details

Medium

HPKE Key Configuration Served From Meta

Overall Risk	Medium	**Finding ID**	NCC-E021658-PRD
Impact	Medium	**Component**	OHTTP
Exploitability	Medium	**Category**	Cryptography
		Status	Fixed

Impact

The ability to serve malicious key configurations to the client allows Meta to (solely) impact the promised privacy assurances relating to client anonymity.

Description

Hybrid Public Key Encryption (HPKE) is an integral component to Oblivious HTTP, which in turn plays a critical role in assuring user privacy.

A precondition to an OHTTP transaction is for the client to obtain a trusted HPKE key configuration of the Meta-operated Gateway. The key configuration includes, among other things, the Gateway's public key and what KDF-AEAD algorithm pairs it supports. The client should retrieve the key configuration from a non-Meta party, such as Fastly, so as to remove Meta's ability to differentiate users (see Section 7 of RFC 9458[32]). However, as currently implemented, the key configuration is retrieved from a Meta endpoint via a GraphQL query (`OhaiKeyConfigQuery`).

This may allow, for example, the following scenarios:

1. Meta may decline to serve a specific source IP address or range, or serve incorrect key material, to deny service.
2. Meta may target a specific source IP address or range with a different working key configuration, e.g. a different supported algorithm, aimed at aiding external traffic analysis such as through a poorly performing algorithm, or allowing traffic 'tagging' at the Meta-operated Gateway since the encapsulated request contains algorithm information.

Similarly, the client must have a correct target configuration (such as Relay address) which should not be dynamically retrieved from Meta.

Recommendation

Ensure the WhatsApp client obtains key configuration material from the third-party (Fastly) Oblivious Relay. Target configuration may be hardcoded into the WhatsApp binary but should not be dynamically retrieved from Meta. The former is understood to be currently in development, while the latter is already correct.

Retest Results

2025-04-11 – Fixed

Since OHAI and ACS configuration cannot easily be covered by the image attestation and transparency proof, an alternative mechanism has been implemented: the configuration elements are now JSON files signed with Ed25519, and distributed by Fastly to clients, so that Meta cannot send client-specific configurations that could endanger privacy. Clients verify the signature against a hardcoded public key. Moreover, the configurations have tight expiration dates, which clients verify and enforce.

32. RFC 9458: Oblivious HTTP at https://www.rfc-editor.org/rfc/rfc9458.html#name-request-format

Medium

Missing Freshness Check on Transparency Proof for Attested Image

Overall Risk	Medium	**Finding ID**	NCC-E021658-THQ
Impact	High	**Component**	TEE attestation
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

Without the freshness check, any old Trusted Execution Environment (TEE) image with known vulnerabilities could be reused indefinitely by an active attacker.

Description

When the Client (or another TEE) connects to a TEE, it verifies the attestation. The attestation is a report on the software image running in the TEE, signed by the TEE private key. By verifying the signature, the Client thus obtains a guarantee that the peer is running in a true TEE with a specific software image, but it must still compare the *measurement* (the hash value computed by the TEE over the software image) with the expected value. This value is not hardcoded in the Client; instead, the Client learns the correct value by checking a *transparency proof*, which is here a structure that contains the hash value and is signed by a third party (Cloudflare).

As per the internal Meta architecture documentation (*Private LLM Inference on TEE*, section *Remote-Attested Transport Layer Security (Ra-TLS)*), the connecting party verifies that

> Transparency Inclusion Proof (TIP) is valid and signed by a known third-party public key and are fresh – i.e., not revoked.

The text then explains that the "simplest approach is here to check the timestamp included by Cloudflare to be recent". Indeed, the transparency proof is a protobuf structure which includes a timestamp field. However, nothing in the current implementation checks this timestamp value, nor applies any other form of revocation check. Right now, the transparency proof is accepted if the signature validation succeeds, regardless of freshness. The main consequence is that if a TEE image is found to have an exploitable vulnerability, then an active attacker can keep running it to respond to connecting clients, providing the relevant (old) "transparency proof", thus leveraging that vulnerability indefinitely (or, more realistically, until Meta deploys countermeasures by patching the client code, which can take up to three months to be propagated to user devices).

In the case of TEE-to-TEE connections, an additional hurdle is that any time-based validation policy relies on the validator knowing the current time and date. In the case of a TEE (such as the Orchestrator), the system clock is not necessarily trustworthy, since it ultimately is under control of the hypervisor, which, in the TEE security model, is a potential attacker. A compromised hypervisor could lie to the TEE about the current date, and make it accept an old transparency proof, thus evading freshness checks.

Recommendation

A freshness check must be implemented in the client code. Meta indicated that the relevant policy is currently being developed. Freshness can be merged into a revocation mechanism: issuing a new signed revocation list is equivalent (security-wise) to implicitly re-signing all

previous transparency proofs which are not specified as revoked by the new list. The freshness threshold (on the proofs themselves, or on any revocation list) should be set in such a way that obsolete software images become unusable in a relatively short time; in the similar context of X.509 certificates, freshness thresholds of about one week are common.

For TEE-to-TEE connections, one possibility for getting a reliable estimate of the current date and time is to have the Client (the user device) send its notion of the current time to the Orchestrator as part of its request: most Clients are network-synchronized, and since the Client is the ultimate source of the data which is to protect, the security model is not modified if the Client sets the validation time for all processing of the data that the same Client is sending.

Retest Results

2025-04-10 – Fixed

A new `isFreshProof()` function was added to attestation bundle verification. For each type of artifact present in the attestation bundle (CVM, TEE container, model, prompt, and/or revocation list), this function compares the connecting client's local trusted time against the signed timestamp in the proof. If the trusted time exceeds the timestamp by more than a certain validity period defined in the client's transparency policy, then the proof is rejected. Currently, all transparency policies in use stipulate 90-day validity periods for all artifact types.

For WhatsApp clients (iOS or Android users), the local trusted time is obtained with `std::time()`, while for TEEs, the local trusted time is computed as the maximum of the system time and the time extracted from the originating WA client's request. This effectively mitigates the use of stale proofs, even for TEEs whose notion of time comes from the (untrusted) host.

Medium

CVM Initializes Unnecessary Hypervisor-Provided NICs

Overall Risk	Medium	**Finding ID**	NCC-E021658-PWB
Impact	Medium	**Component**	CVM
Exploitability	Low	**Category**	Access Controls
		Status	Fixed

Impact

The hypervisor has the ability to assign network interfaces to the CVM, which are automatically initialized. Although these interfaces do not get assigned an IP address, they are fully functional and can be used, for example, to exfiltrate data by sending Layer 2 network packets or to attempt the exploitation of the network driver.

Description

The CVM virtual NIC driver enumerates and initializes the NIC provided by the hypervisor. The relevant kernel log entries and the output of the `ifconfig` command are shown below.

```
[root@localhost ~]# dmesg | grep eth0
[    7.788480] e1000e 0000:00:01.0 eth0: (PCI Express:2.5GT/s:Width x1) 52:54:00:12:34:56
[    7.789491] e1000e 0000:00:01.0 eth0: Intel(R) PRO/1000 Network Connection
[    7.790369] e1000e 0000:00:01.0 eth0: MAC: 3, PHY: 8, PBA No: 000000-000
[    8.864972] e1000e 0000:00:01.0 eth0: NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
↳ Rx/Tx
```

```
[root@localhost ~]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        ether 52:54:00:12:34:56  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 67536  bytes 11287126 (10.7 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 21  memory 0x80060000-80080000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 70564254  bytes 37001923394 (34.4 GiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 70564254  bytes 37001923394 (34.4 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Recommendation

Prevent the creation of network interfaces by applying specific udev rules. Additionally, ensure that no other unexpected devices, such as USB devices, can be created by the hypervisor.

Retest Results

2025-04-10 – Fixed

Specific udev rules have been added to prevent creation of all `eth*` devices within the CVM.

Low

Confidential Workload Executes with Root Privileges

Overall Risk	Low	**Finding ID**	NCC-E021658-PKA
Impact	Medium	**Component**	CVM
Exploitability	Low	**Category**	Access Controls
		Status	Fixed

Impact

The successful exploitation of the main service would grant the attacker full access privileges to the running container.

Description

The *systemd* unit file for the main service running within the container is defined by the `_start_main_systemd_unit_sidecar` function. In this unit file, the `User=` directive, which sets the UNIX user the process runs, is assigned the value defined in `start_main_config.command`.

```
# Generates a feature for systemd to start main service with Sidecar deployment model.
def _start_main_systemd_unit_sidecar(
        name: str,
        start_main_config: start_main_config_record,
        systemd_type: SystemDType,
        systemd_custom_setup: typing.Callable[[str, start_main_config_record],
        ↳ list[typing.Any]] | None = None):
# ...
    start_main_unit = """
[Unit]
Description=Start the main service binary

[Service]
Type=simple
User={run_as_user}
SyslogIdentifier=main_service
EnvironmentFile=/REDACTED/PATH/TO/FILE
{open_file_descriptors}
{exec_start_pre}
ExecStart={main_cmd}
TimeoutSec=1800

[Install]
WantedBy=multi-user.target
""".format(
# ...
        run_as_user = start_main_config.user,
# ...
    )
# ...
```

Figure 2: configs.bzl

The call trace to `_start_main_systemd_unit_sidecar` passes through the following functions in reverse order: `_sidecar_image_layer_features`, `_cvm_image_layer_features`, `cvm_fbpkg`, `cvm_container`, `cvm_gpu_container`, and `base_cvm_gpu_container`.

The user is initially assigned the string `root`, which is the default value for the `run_as_user` argument in the call to `cvm_container`. The `run_as_user` is then passed as an argument to the `cvm_fbpkg` function.

```
def cvm_container(
        *,
        name: str,
        vm_rootfs_image_layer: str | None = None,
        vm_fbpkg: str | None = None,
# ...
        run_as_user: str = "root",
# ...
        **container_kwargs):
        )
# ...
        container_features.append(
            tw2.build_fbpkg(
                cvm_fbpkg(
                    name = name,
# ...
                    run_as_user = run_as_user,
                    enable_svsm = enable_svsm,
                    systemd_type = systemd_type,
                    systemd_custom_setup = systemd_custom_setup,
                ),
```

Figure 3: cvm_helpers.bzl

The `cvm_fbpkg` function creates the `start_main_config` variable and assigns the `root` string to its `user` field, which is eventually passed to the `_start_main_systemd_unit_sidecar` function mentioned earlier.

```
def cvm_fbpkg(
# ...
        run_as_user: str = "root",
# ...
):
# ...
    start_main_config = config.start_main_config_record(
# ...
        user = run_as_user,
# ...
    )
# ...
    image.layer(
# ...
        features = features + vm_image_layer.features(
            name,
            start_main_config,
# ...
        ),
# ...
    )
# ...
```

Figure 4: cvm_helpers.bzl

Recommendation

Limit the privileges of the main service process by running as a non-privileged user and assigning only the necessary Linux capabilities. Alternatively, implement fine-grained access control using a Mandatory Access Control system such as SELinux.

Retest Results

2025-04-10 – Fixed

A layer of containerization is being added inside the CVM: the main service runs in an `lxc` container/sandbox in the CVM. This additional containerization is a suitable alternative to running the main application as a non-privileged user.

Low

Missing Configuration Parameters in Attestation Report

Overall Risk	Low	**Finding ID**	NCC-E021658-DKA
Impact	Low	**Component**	CVM
Exploitability	Low	**Category**	Access Controls
		Status	Fixed

Impact

Failing to include certain network configuration parameters in the attestation report could result in unauthorized modifications going undetected.

Description

The CVM includes a mechanism for sharing supplementary files such as the VCEK and ASK certificates, as well as transparency proofs with the container. However, among the supplied files there is one that configures a set of environment variables, with some related to IP addresses and port numbers.

These files are not included in the attestation report, therefore any changes to these IP addresses and port numbers cannot be verified through attestation.

Below is a build-system call to the `cvm_gpu_container`, passing a set of files to the array argument `files_to_copy_from_iso`. The highlighted `tw_start_main` file provides a set of environment variables, some of which will be used by the CVM.

```
    cvm_gpu_container(
        name = container_name,
        bt_artifacts = [BtArtifact(
            artifact_type = BtArtifactType("LlmConfig"),
            path = "/REDACTED/PATH",
        )],
# ...
        files_to_copy_from_iso = [
            "etc/fbwhoami",
            "etc/smc.tiers",
            "REDACTED/PATH/TO/tw_start_main",
        ],
```

Figure 5: base_cvm.bzl

The list of included environment variables is provided in the `pack_cvm_data.sh` script, as shown below, and exported into the `tw_start_main` file.

```
# List of env variables to pass from TW container CVM
ENV_VARS=(
    SMC_TIERS
    TEE_VSOCK_START_CID
    TW_JOB_CLUSTER
    TW_JOB_NAME
    TW_JOB_USER
    TW_ONCALL_TEAM
    TW_SERVICE_TAGS
    TW_SERVICE_TAGS_APP
    TW_TASK_ID
```


```
    TW_TASK_IP_ADDR
    TW_PORT
)
# ...
# Export env variables into a file
# ...
```

Figure 6: pack_cvm_data.sh

The variable `files_to_copy_from_iso` eventually reaches the `_sidecar_image_layer_features` through the `_cvm_image_layer_features` function.

At this point, the `files_to_copy_from_iso` list is appended to the ASK and VCEK certificates and transparency proofs and passed as an argument to the `configure_vm` function.

```
# Generates a list of features for CVM image read-only layer used in "Sidecar"
# deployment type..
def _sidecar_image_layer_features(
# ...
        files_to_copy_from_iso: list[str],
# ...
):
    return [
        config.configure_cvm(
# ...
            files_to_copy = [
                "REDACTED/PATH/TO/ask.pem",
                "REDACTED/PATH/TO/vcek.pem",
                "REDACTED/PATH/TO/cvm_transparency_proof.json",
                "REDACTED/PATH/TO/transparency_proofs/*.json",
            ] + files_to_copy_from_iso,
        ),
# ...
    ]
```

Figure 7: cvm.bzl

The call trace will finally arrive at the `_configure_vm` function, where the list of files is used in the created systemd unit file to set up the initial CVM configuration and is passed as argument to the `configure_cvm_.sh` shell script, specified in its `ExecStart=` directive.

```
# Generate a feature list for a "configure_cvm" systemd unit.
def _configure_cvm(
        name: str,
        iso_dev: str,
        files_to_copy: list[str]):
    configure_cvm_unit = """
[Unit]
Description=Initial CVM configuration
Wants=network-pre.target
Wants=cloud-init.service
Before=chronyd.service
After=cloud-init.service
ConditionPathExists=/usr/local/sbin/configure_cvm.sh
Before=start_main.service

[Service]
Type=oneshot
ExecStart=/usr/local/sbin/configure_cvm.sh -d {iso_dev} {files_to_copy}
```


```
TimeoutSec=120
# Output needs to appear in instance console output
StandardOutput=journal+console

[Install]
RequiredBy=start_main.service
WantedBy=multi-user.target
""".format(
        iso_dev = iso_dev,
        files_to_copy = " ".join(collections.uniq(files_to_copy)),
    )
```

Figure 8: configs.bzl

`configure_cvm.sh` copies the specified files to the filesystem from the given ISO device.

```
while [[ $# -gt 0 ]]
do
    case $1 in
        -d|--device)
            ISO_DEV="$2"
            shift
            shift
            ;;
        *)
            FILES_TO_COPY+=("$1")
            shift
            ;;
    esac
done
# ...
mount -o ro -t auto "$ISO_DEV" "$DATA_DIR"

MATCHING_FILES=()
for file_to_copy in "${FILES_TO_COPY[@]}"
do
    # Check if the file is a pattern (i.e., it contains a wildcard)
    if [[ $file_to_copy == *"*"* || $file_to_copy == *"?"* ]]; then
        for f in "$DATA_DIR"/$file_to_copy
        do
            # Strip "$DATA_DIR/" prefix from the file name.
            MATCHING_FILES+=("${f#"$DATA_DIR"/}")
        done
    else
        MATCHING_FILES+=("$file_to_copy")
    fi
done
```

Figure 9: configure_cvm.sh

`tw_start_main` is used by the systemd unit file shown below, where the provided `TW_TASK_ID` is used to calculate the VSOCK destination port.

```
[Unit]
Description=Listens on VSOCK address and forwards to local TCP Port inside VM
Before=start_main.service
RequiredBy=start_main.service
Wants=configure_cvm.service
After=configure_cvm.service
```


```
ConditionPathExists=/REDACTED/PATH/TO/tw_start_main

[Service]
Type=simple
EnvironmentFile=/REDACTED/PATH/TO/tw_start_main
ExecStart=/bin/sh -c "socat ... $((TW_TASK_ID ...))"

[Install]
WantedBy=multi-user.target
```

Figure 10: client-routing-tunnel.service

Recommendation

The complete configuration of the CVM should be verifiable through attestation, including all parameters passed during its initialization.

Retest Results

2025-04-11 – Fixed

The network configuration parameters cannot readily be part of the attestation, since they contain information about Meta's internal infrastructure that may be subject to quick change. An alternative mechanism has been implemented, in which the parameters are requested dynamically by the CVM early in its boot process, using the TEE Secure Analysis framework. The obtained parameters are furthermore validated, so that they can used in shell script and systemd configuration files:

- Only a fixed list of environment variable names is accepted.
- Each variable value is constrained to a small length (at most 50 characters).
- The variable contents may contain only "innocuous" characters (ASCII letters, digits, underscore, dash and dot) which do not have any special meaning in shells.

Systemd notifications are used to ensure that the variables have been retrieved and validated before executing any other startup script or service that uses them.

Low

RA-TLS Key Stored in Cleartext Within the CVM

Overall Risk	Low	**Finding ID**	NCC-E021658-D7D
Impact	High	**Component**	CVM
Exploitability	Low	**Category**	Cryptography
		Status	Risk Accepted

Impact

A successful exploitation of the CVM would provide the attacker access to the RA-TLS private keys and allow to them to supplant the container.

Description

The RA-TLS key and certificate are generated by the `attestation_agent` binary, which is executed by the `attestation_agent.service` running on the CVM. A snippet of its main function is shown below:

```
int main(int ac, char* av[]) {
  gflags::ParseCommandLineFlags(&ac, &av, true);

// ...

  std::unique_ptr<tazo::sev_snp::SNPCertificateChain> raCerts =
      ra_tls::readAmdSevSnpCertChainFromFiles(FLAGS_ra_certs_path);

  SignatureResponse transparencyProof =
      ra_tls::readTransparencyProofFromFile(FLAGS_transparency_proof_path);

  ra_tls::RaTls ra_tls(std::make_unique<ra_tls::RaExtensionFactory>(
      std::move(psp),
      /* vmpl */ 1,
      std::move(raCerts),
      transparencyProof));

  // Retrieve key pair and cert.
  auto [pkey, cert] = ra_tls.generateRaTlsCert();
  // Serialize both.
  std::string pemSerializedPrivateKey = tazo::encodePemPrivKey(pkey.get());
  std::string pemSerializedCert = tazo::encodePemCert(cert.get());
  // ...
  // We will write to a single file, concat the private key and cert.
  std::string keyAndCertPem(pemSerializedPrivateKey + pemSerializedCert);
  // ...
  return folly::writeFileAtomicNoThrow(
      FLAGS_key_and_cert_path, keyAndCertPem, options);
}
```

Figure 11: RaTlsMain.cpp

The highlighted `RaTls::generateRaTlsCert` function returns a tuple with the generated private key and certificate.

```
std::tuple<folly::ssl::EvpPkeyUniquePtr, folly::ssl::X509UniquePtr>
RaTls::generateRaTlsCert() {
  // Generate the key-pair for the self-siged TLS cert.
  folly::ssl::EvpPkeyUniquePtr pkey = generateRsaKeyPair(RaTls::kRsaKeyLength);
```


```
  // Read public key as PEM.
  std::string pemPublicKey = tazo::encodePemPubKey(pkey.get());

  // Generate the remote attestation extension data.
  std::string raExtensionData = raExtFact_->genRaExtension(pemPublicKey);

  // Create self-signed X509 cert with serialized attestation bundle as
  // extension.
  folly::ssl::X509UniquePtr cert =
      generateSelfSignedCert(pkey.get(), raExtensionData);

  return std::make_tuple(std::move(pkey), std::move(cert));
}
```

Figure 12: RaTls.cpp

At the end of the `main` function, the private key and certificate are concatenated and written to the filesystem path defined by the `FLAGS_key_and_cert_path` constant, which is a constant string storing the filesystem path as defined by the `DEFINE_string` macro:

```
DEFINE_string(
    key_and_cert_path,
    "ra_tls_key_and_cert.pem",
    "Path to write the concatenated RA-TLS private key and the cert to in PEM format.");
```

Figure 13: RaTlsMain.cpp

Recommendation

Implement a Secure VM Service Module (SVSM)[33], specifically Coconut-SVSM[34], to provide a virtual TPM (vTPM) service that is isolated from both the host and the guest.

Retest Results

2025-04-10 – Not Fixed

No particular action was taken since the whole CVM storage area is encrypted, and the RA-TLS private key is short-lived. Meta acknowledged this finding and accepted the risk.

Client Response

Given the RA TLS private key is short-lived and only resides in memory, Meta assesses this finding is an acceptable risk. Additionally, since the review date, Meta further reduced the lifetime of these keys by tying certificates to per-request, client-generated nonces. Longer term, we plan to explore solutions that can further isolate this key away from the guest CVM.

33. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf
34. https://github.com/coconut-svsm/svsm

Low

Opportunities for Enhanced CVM Kernel Hardening

Overall Risk	Low	**Finding ID**	NCC-E021658-4CG
Impact	Low	**Component**	CVM
Exploitability	Low	**Category**	Access Controls
		Status	Fixed

Impact

Relevant security hardening measures are missing from the CVM Linux kernel. Their absence facilitates the exploitation of the system from a compromised hypervisor or user application, potentially enabling a privilege escalation on the system.

The hardening options described in this finding are essential kernel modifications that raise the difficulty of successful exploitation.

Description

The Confidential VM (CVM) runs Linux version 6.12.0-rc6. To identify unused kernel hardening options, its configuration is compared to the recommended hardening settings published by the Kernel Self-Protection Project (KSPP)[35]. Below are the CVM kernel configurations that differ from KSPP recommendations and deserve attention.

Missing Exploit Mitigation Configurations

The following exploit mitigation configurations are available but not enabled in the CVM kernel.

- Introduce randomness in memory allocation to make memory allocation patterns less deterministic and harder to exploit.
  `CONFIG_SLAB_FREELIST_RANDOM=y`
  `CONFIG_RANDOM_KMALLOC_CACHES=y`
  `CONFIG_SHUFFLE_PAGE_ALLOCATOR=y`
- Randomize the kernel stack offset on syscall entry.
  `CONFIG_RANDOMIZE_KSTACK_OFFSET_DEFAULT=y`
- Disable DMA between EFI hand-off and the kernel's IOMMU setup.
  `CONFIG_EFI_DISABLE_PCI_DMA=y`
- Randomize the layout of system structures. Note that this setting has a significant impact on the system performance.
  `CONFIG_RANDSTRUCT_FULL=y`
- Wipe all caller-used registers on exit from the function.
  `CONFIG_ZERO_CALL_USED_REGS=y`
- Reboot devices immediately if kernel experiences an Oops.
  `CONFIG_PANIC_ON_OOPS=y`
  `CONFIG_PANIC_TIMEOUT=-1`
- Disallow allocating the first 64k of memory.
  `CONFIG_DEFAULT_MMAP_MIN_ADDR=65536`
- Initialize memory on allocation and free to mitigate information leaks and reduce the risk of exploitation.

35. https://kspp.github.io/Recommended_Settings.html

`CONFIG_INIT_ON_ALLOC_DEFAULT_ON=y`
`CONFIG_INIT_ON_FREE_DEFAULT_ON=y`

- Ensure uninitialized stack variables are zeroed.
  `CONFIG_INIT_STACK_ALL_ZERO=y`
- Validate reference counting and detect any reference count overflows.
  `CONFIG_REFCOUNT_FULL=y`
- Enable kernel address space layout randomization (KASLR) to make memory layout less predictable.
  `CONFIG_RANDOMIZE_BASE=y`
  `CONFIG_RANDOMIZE_MEMORY=y`
- Enable shadow stack for user-space applications to help mitigate return-oriented programming (ROP) exploits.
  `CONFIG_X86_USER_SHADOW_STACK=y`

Miscellaneous System Hardening Configurations

The following is an overview of the available configurations that contribute to hardening the kernel by reducing the exposed attack surface and limiting exploitation opportunities.

- Make module text and rodata memory read-only, and non-text memory will be made non-executable.
  `CONFIG_STRICT_MODULE_RWX=y`
- While seccomp is enabled, the following option allows for finer control over filtering.
  `CONFIG_SECCOMP_FILTER=y`

The options below enable various security mechanisms.

- Allows for the inclusion of security modules to enforce various security policies.
  `CONFIG_SECURITY=y`
- Introduces ptrace scope restrictions.
  `CONFIG_SECURITY_YAMA=y`
- Enables the Landlock security module which allows processes to secure themselves by defining rules that limiting access to the kernel.
  `CONFIG_SECURITY_LANDLOCK=y`
- Enable the kernel Lockdown feature which attempts to protect against unauthorized modification of the kernel.
  `CONFIG_SECURITY_LOCKDOWN_LSM=y`
  `CONFIG_SECURITY_LOCKDOWN_LSM_EARLY=y`
  `CONFIG_LOCK_DOWN_KERNEL_FORCE_CONFIDENTIALITY=y`
- Perform usercopy bounds checking.
  `CONFIG_HARDENED_USERCOPY=y`
- Enforce restrictions on unprivileged users reading the kernel syslog via dmesg.
  `CONFIG_SECURITY_DMESG_RESTRICT=y`
- Treat warnings as error during the kernel built.
  `CONFIG_WERROR=y`
- Force IOMMU TLB invalidation so devices will never be able to access stale data contents.
  `CONFIG_IOMMU_DEFAULT_DMA_STRICT=y`
- Disable vsyscall mapping and derived opportunities for exploitation.
  `CONFIG_LEGACY_VSYSCALL_NONE=y`

- Enforce the requirement for kernel modules to be signed in order to be allowed to load.
  `CONFIG_MODULE_SIG=y`
  `CONFIG_MODULE_SIG_FORCE=y`
  `CONFIG_MODULE_SIG_ALL=y`
  `CONFIG_MODULE_SIG_SHA512=y`
  `CONFIG_MODULE_SIG_HASH="sha512"`
  `CONFIG_MODULE_SIG_KEY="certs/signing_key.pem"`

Configurations that Weaken System Security

The following set of configurations are found enabled and increase the risk of exploitation by exposing a larger attack surface. Therefore, it is recommended to disable them if possible.

- Remove support for 32-bit binaries and syscalls.
  `CONFIG_COMPAT_32=n`
  `CONFIG_COMPAT=n`
- Prevent loading kernel modules on line discipline configuration requests.
  `CONFIG_LDISC_AUTOLOAD=n`
- Disable merging of adjacent slabs of the same size, making cross-slab heap attacks more difficult.
  `CONFIG_SLAB_MERGE_DEFAULT=n`
- Introduce structure layout randomization.
  `CONFIG_RANDSTRUCT_NONE=n`
- Prevent direct access to physical memory from userspace.
  `CONFIG_DEVMEM=n`
- Do not provide an ELF core file of the kernel.
  `CONFIG_PROC_KCORE=n`
- Disable support for additional non-standard binary formats.
  `CONFIG_BINFMT_MISC=n`
- Disable support for the `TIOCSTI` ioctl, which allows to push characters into a TTY's input buffer.
  `CONFIG_LEGACY_TIOCSTI=n`

Kernel Sanity Checks

For completeness, here is the list of options that introduce sanity checking for commonly targeted kernel structures. These could impact the system performance and, with few exceptions, are typically enabled only during development or testing. Some of them trigger a kernel panic if an issue is detected, while others simply log the information to the kernel's log without providing runtime protection.

- Generate a warning if any W+X mappings are found at boot.
  `CONFIG_DEBUG_WX=y`
- Debug checking for credential management.
  `CONFIG_DEBUG_CREDENTIALS=y`
- Enable this to turn on sanity checking for notifier call chains.
  `CONFIG_DEBUG_NOTIFIERS=y`
- Extended checks in the linked-list walking routines.
  `CONFIG_DEBUG_LIST=y`

- Checks on scatter-gather table.
  `CONFIG_DEBUG_SG=y`
- Sanity checks in virtual to page code.
  `CONFIG_DEBUG_VIRTUAL=y`
- Checks for a kernel stack overrun on calls to the `schedule` function.
  `CONFIG_SCHED_STACK_END_CHECK=y`
- Sanity check userspace page table mappings.
  `CONFIG_PAGE_TABLE_CHECK=y`
  `CONFIG_PAGE_TABLE_CHECK_ENFORCED=y`
- Detect undefined behavior at runtime.
  `CONFIG_UBSAN=y`
  `CONFIG_UBSAN_TRAP=y`
  `CONFIG_UBSAN_BOUNDS=y`
  `CONFIG_UBSAN_SANITIZE_ALL=y`
- Low-overhead detector of heap vulnerabilities
  `CONFIG_KFENCE=y`
  `CONFIG_KFENCE_SAMPLE_INTERVAL=100`

Recommendation

Consider improving the security of the CVM kernel by enabling the available hardening configurations, provided they do not significantly impact the system performance.

Retest Results

2025-04-10 – Fixed

All of the recommendations above have been applied to the kernel build system for the CVM image, save for those that would imply a substantial performance degradation and offer little to no added security in Meta's threat model.

Low

LLM Predictor Binaries Lack Security Hardening

Overall Risk	Low	**Finding ID**	NCC-E021658-2QT
Impact	Medium	**Component**	CVM
Exploitability	Low	**Category**	Access Controls
		Status	Risk Accepted

Impact

If a security vulnerability were to exist in a library that handles commands and processes user data before transferring it to the GPU, an attacker could exploit it. The absence of security mitigations could potentially allow for a privilege escalation attack in the CVM.

Description

Libraries deployed on the inference TEE lack security hardening. These libraries offer functionalites such as message handlers used by the Thrift server and other data manipulation functions provided by PyTorch, Pandas, SciPy, and others, which directly interact with the input data.

The output of the checksec[36] tool for some of the mentioned libraries is shown below. Highlighted in red are the insecure configurations. Most of the libraries lack stack canaries, include debugging symbols, are not fortified[37], are only partial RELRO, and include RPATH.

Filename	Canary	Fortify	NX	RELRO	RPATH	Symbols
liblogdevice_server_logdeviced_lib.so	no	no	yes	partial	yes	yes
liblogdevice_server_thrift_server.so	no	no	yes	partial	yes	yes

Table 1: Checksec output for shared libraries used by the server.

Filename	Canary	Fortify	NX	RELRO	RPATH	Symbols
_multiarray_tests.so	no	no	yes	partial	yes	yes
_simd.so	no	no	yes	partial	yes	yes
_umath_tests.so	no	no	yes	partial	yes	yes

Table 2: Checksec output for shared libraries in `numpy/core`.

Filename	Canary	Fortify	NX	RELRO	RPATH	Symbols
libai_codesign_gen_ai_marlin_marlin_ops.so	no	no	yes	partial	yes	yes
libdeeplearning_fbgemm_fbgemm_gpu_experimental_gen_ai_attention_ops.so	no	no	yes	partial	yes	yes

36. https://github.com/slimm609/checksec.sh
37. https://www.gnu.org/software/libc/manual/html_node/Source-Fortification.html

Filename	Canary	Fortify	NX	RELRO	RPATH	Symbols
libdeeplearning_fbgemm_fbgemm_gpu_experimental_gen_ai_comm_ops.so	no	no	yes	partial	yes	yes
libdeeplearning_fbgemm_fbgemm_gpu_experimental_gen_ai_gemm_ops.so	no	no	yes	partial	yes	yes
libdeeplearning_fbgemm_fbgemm_gpu_experimental_gen_ai_kv_cache_ops.so	no	no	yes	partial	yes	yes
libdeeplearning_fbgemm_fbgemm_gpu_experimental_gen_ai_quantize_ops.so	no	no	yes	partial	yes	yes
libevict-thrift-py3-extensions-configerator.so	no	no	yes	partial	yes	yes
libevict-thrift-py3-extensions-dsi.so	no	no	yes	partial	yes	yes
libevict-thrift-py3-extensions-smart.so	no	no	yes	partial	yes	yes
libevict-thrift-py3-extensions-tupperware.so	no	no	yes	partial	yes	yes
libfbcode_build_info.so	no	no	yes	partial	yes	yes
libgen_ai_llm_inference_fb_llm_csrc_kv_lru_cache.so	no	no	yes	partial	yes	yes
libgen_ai_llm_inference_fb_llm_csrc_llama_cpp.so	no	no	yes	partial	yes	yes
libgen_ai_llm_inference_fb_llm_sampling_radix_top_p_kernel.so	no	no	yes	partial	yes	yes
libippcore.so	no	no	yes	full	no	yes
libippi.so	yes	yes	yes	full	no	yes
libippie9.so	yes	no	yes	full	no	yes
libippil9.so	yes	no	yes	full	no	yes
libipps.so	yes	yes	yes	full	no	yes
libippse9.so	yes	no	yes	full	no	yes
libippsl9.so	yes	no	yes	full	no	yes
libippvm.so	yes	yes	yes	full	no	yes
libippvme9.so	no	no	yes	full	no	yes
libippvml9.so	no	no	yes	full	no	yes
liblazy_cuda.so	no	no	yes	partial	yes	yes
liblibfb_py_mkl_mkl_dep_handle_lp64.so	no	no	yes	partial	yes	yes
libmkl_avx2.so.2	no	no	yes	partial	no	yes
libmkl_avx512.so.2	no	no	yes	partial	no	yes

Filename	Canary	Fortify	NX	RELRO	RPATH	Symbols
libmkl_core.so.2	no	no	yes	partial	no	yes
libmkl_def.so.2	no	no	yes	partial	no	yes
libmkl_intel_lp64.so.2	no	no	yes	partial	no	yes
libmkl_intel_thread.so.2	no	no	yes	partial	no	yes
libmkl_mc3.so.2	no	no	yes	partial	no	yes
libmkl_vml_avx2.so.2	no	no	yes	full	no	yes
libmkl_vml_avx512.so.2	no	no	yes	full	no	yes
libmkl_vml_cmpt.so.2	no	no	yes	full	no	yes
libmkl_vml_def.so.2	no	no	yes	full	no	yes
libmkl_vml_mc3.so.2	no	no	yes	full	no	yes
libnuma.so.1	yes	yes	yes	partial	no	yes
libpytorch_vision__C.so	no	no	yes	partial	yes	yes
libpytorch_vision_image.so	no	no	yes	partial	yes	yes
libpytorch_vision_video_reader.so	no	no	yes	partial	yes	yes
libqhull_r.so.8.0	yes	yes	yes	partial	no	yes
libtinygemm_tinygemm.so	no	no	yes	partial	yes	yes

Table 3: Checksec output for shared libraries in `runtime/lib`.

Filename	Canary	Fortify	NX	RELRO	RPATH	Symbols
json.so	no	no	yes	partial	yes	yes
tslib.so	no	no	yes	partial	yes	yes
tslibs/conversion.so	no	no	yes	partial	yes	yes
tslibs/fields.so	no	no	yes	partial	yes	yes
tslibs/np_datetime.so	no	no	yes	partial	yes	yes
tslibs/offsets.so	no	no	yes	partial	yes	yes
tslibs/period.so	no	no	yes	partial	yes	yes
tslibs/strptime.so	no	no	yes	partial	yes	yes
tslibs/timedeltas.so	no	no	yes	partial	yes	yes
tslibs/timestamps.so	no	no	yes	partial	yes	yes
tslibs/tzconversion.so	no	no	yes	partial	yes	yes
tslibs/vectorized.so	no	no	yes	partial	yes	yes

Table 4: Checksec output for shared libraries in `pandas/_libs`.

Each of the compiler and linker defenses is explained below:

- The **Fortify Source** compiler option will mitigate memory corruption vulnerabilities and buffer overflows related to common misuses of unsafe memory (memcpy) and string operations (strcpy) in the standard C library. Without this protection, a memory corruption vulnerability may be easier to exploit.

- The **Stack Canary** mitigation prevents an overwritten return address on the stack from being loaded into the instruction pointer. The compiler accomplishes this by automatically inserting a random cookie on the stack, and checking the integrity of these cookies each time a stack frame is created or destroyed. This cookie is placed in between stack variables likely to be overwritten and the stored return address, so that any buffer overflow resulting in a return address being overwritten will also result in the cookie being overwritten.
- RELRO, which stands for **Relocation Read-Only**, is a technique that applies to the data sections of an ELF file. When RELRO is enabled, certain sections of binary are reordered and remapped as read-only during startup. This helps mitigate memory overwrite attacks that target the global offset table. For complete protection, full RELRO is required.
- **RPATH** specifies the locations the dynamic linker will search for binaries when linking the ELF executable with shared objects. Its presence can introduce security risks if it points to writable directories, as attackers could place malicious libraries in those directories.

Recommendation

Build binaries and libraries using compiler and linker options to add runtime mitigations for memory corruption vulnerabilities. This includes the inclusion of stack canaries, function fortification, full RELRO, stripping symbols, and disabling RPATH. The relevant compiler and linker options are presented below:

- Stack canaries are included using the compiler flag `-fstack-protector-strong`.
- The macro `_FORTIFY_SOURCE=1` fortifies calls to certain functions in the GNU C Library.
- Full RELRO is enabled with the linker option `-Wl,-z,relro,-znow`.
- Symbols can be stripped while building the binary by using GCC's `-s` option.
- To disable RPATH, do not include the linker option `-rpath`.

Retest Results

2025-04-10 – Not Fixed

An off-the-shelf vLLM binary is now used, thus benefitting from the bug-tracking and general maintenance inherent to widely used open-source software. The loaded models and weights use the safetensor format, which is resilient to maliciously crafted data. Meta acknowledged this finding and accepted the risk.

Client Response

Meta has minimized the impact of vLLM compromise by containerizing the workload such that any unknown application security issues will be limited in impact by limitations on communication beyond the VM. Additional work has been done to remove unnecessary dependencies from vLLM. Meta will continue to harden vLLM within CVMs.

Low

OHAI Proxy Security Impacts TEE Remote Attestation

Overall Risk	Low	**Finding ID**	NCC-E021658-BAR
Impact	High	**Component**	OHTTP
Exploitability	Low	**Category**	Cryptography
		Status	Not Fixed

Impact

A compromise of the OHAI Proxy servers may allow an attacker to leverage a remote attestation forgery attack without needing any access to the hardened hosts on which the TEEs run.

Description

Clients connect to the Orchestrator TEE using the RA-TLS protocol. All RA-TLS records are themselves wrapped into Oblivious HTTP (OHAI), going through the Relay (operated by Fastly) then the Proxy (maintained by Meta). The individual messages are encrypted by the client using the HPKE format, and decrypted on the Proxy. The main point of using OHAI is to prevent Meta from linking requests to individual users. It is tempting to assert that apart from this privacy role, the OHAI protocol plays no security role in the overall architecture: RA-TLS already ensures confidentiality and integrity of exchanges, and the HPKE layer seems redundant. The point of this finding is to document that this is not entirely true.

As explained in Section Architectural Overview and Inventory of Systems, clients trust that they are talking to a true TEE, running the expected software, through the process of *remote attestation*. This attestation relies on a signature by a chip-specific key (VCEK). An attacker trying to run a fake TEE with a forged attestation will need to obtain the private part of a VCEK. However, such extraction is probably feasible, in at least two different ways:

- The CPU vendor (AMD) keeps a copy of all VCEKs (specifically, of all per-chip seeds from which VCEKs are derived). This is done so that AMD can certify any VCEK as valid (i.e. belonging to an actual AMD CPU and not reported as compromised). A VCEK private key may thus be obtained from AMD through compromise or collusion.
- Extracting a VCEK from an existing CPU has been demonstrated at least once by academics, using power-based glitching, for a low cost. That particular key was reported as compromised to AMD, but the technique could be reproduced.

A crucial point here is that an attacker trying to run a fake TEE needs a VCEK, but not necessarily the VCEK of one of Meta's dedicated systems. It could be the VCEK corresponding to any other AMD CPU, of which millions are sold every year. From the point of view of the client, all VCEKs are interchangeable. Correspondingly, physical security measures on Meta's TEE systems provide no extra protection against this kind of attack.

For an attacker who obtained a VCEK, the only impediment to running an interception attack with a fake Orchestrator is the fact that the data (the RA-TLS records) is conveyed encrypted with HPKE, as per the OHAI protocol. This implies that this encryption layer is not, in fact, redundant with RA-TLS; it is an important component that ensures that interception attacks must involve a compromise of some of Meta's infrastructure.

Recommendation

The architecture documentation should be more explicit about the exact security role of the OHAI protocol in the system.

The risk of VCEK extraction can be lowered if a list of all Meta's CPU VCEKs (or at least CPU identifiers) is hardcoded in the client: if the client requires a specific VCEK, then an attacker can no longer extract the VCEK from any existing AMD CPU in the world, but must instead target the ones which are actually used by Meta and located in physically secure premises. Additionally, external monitoring would be used to report on any unexpected VCEK (as seen from a connecting client).

Retest Results

2025-04-11 – Not Fixed

Meta plans to enroll AMD chip IDs (identifying specific CPUs) to a transparency log maintained by Cloudflare using a separate Cloudflare namespace. The chip ID proofs will be incorporated into the attestation bundles, so clients will be able to verify that they are communicating with an authentic CPU in Meta's deployment that has not been revoked. If implemented as planned, these changes will fully address the finding.

Client Response

Meta now publishes Authorized Host Identities to an external log. Meta assesses that this provides sufficient protection against this issue, while providing external auditability for the use of publicly known compromised VCEKs.

Low

Conversation Exfiltration Via Cross-User Prompt Injection

Overall Risk	Low	**Finding ID**	NCC-E021658-NRE
Impact	High	**Component**	LLM
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

Threat actors may be able to exfiltrate earlier unreadable conversation contents by poisoning LLM summaries for other users.

Description

Prompt Injection is a technique where attackers influence the output of a Large Language Model (LLM) by abusing the lack of data-code segmentation between a model's instructions and a model's prompt. Self-Prompt Injection poses little risk to most application threat models. However, attackers who can influence the content of other users' prompts can often exploit Prompt Injection to influence the model's output within the context of another user's session. Cross-User Prompt Injection (also called Indirect Prompt Injection) may induce models to return malicious text, submit poisoned data to downstream systems, or execute state-changing operations on behalf of other users.

WhatsApp provides users functionality to join conversations later in the discussion chain and not receive the previous conversation history on entering. However, the content generated by these users is submitted to the LLM with the content generated by threat actors. Consequently, any functionality sink that consumes the output of the LLM can be abused by threat actors attempting to exfiltrate conversation data.

In the case of WhatsApp, threat actors can include prompt injections in their messages that induce the application to generate a corrupted summary, such as a link to external resources with conversation secrets embedded in URL parameters. Victims who click these links would inadvertently submit private conversation assets to attacker-controlled servers.

Recommendation

Ensure that no output from the WhatsApp summarizer is rendered as any format other than plain text. This control is largely client-side and should persist across future iterations of the application.

Retest Results

2025-04-22 – Fixed

Meta disabled link detection for summary messages on Android and iOS, so no output will be rendered as a clickable link.

Low

Client Does Not Verify TCB Version in VCEK Certificate

Overall Risk	Low	**Finding ID**	NCC-E021658-C6J
Impact	High	**Component**	Attestation
Exploitability	Low	**Category**	Authentication
		Status	Fixed

Impact

Vulnerabilities in older firmware or microcode versions could be leveraged to forge attestation reports in a manner undetected by the client, violating the guarantees provided by attestation.

Description

The Versioned Chip Endorsement Key (VCEK) is used to sign the SEV-SNP attestation report embedded in the RA-TLS certificate. It is derived from a per-chip secret and a specific TCB *version*, as its name implies. The TCB version (`SnpTcbVersion` in the attestation library) includes Security Version Numbers (SVNs)/Security Path Levels (SPLs) for 4 items: bootloader firmware, TEE firmware, SNP firmware, and CPU microcode.

With RA-TLS, an outer, self-signed certificate for the chip has an extension containing an attestation report and an AMD-issued VCEK certificate.

- The attestation report (`SnpAttestationReport` in the attestation library) comprises a lot of data, including the chip ID (64 bytes), the reported TCB, various policy flags, and a hash of the outer certificate's public key (as "report data").
- The inner, AMD-issued VCEK certificate[38] has an extension that includes the product name (e.g. Milan or Genoa), chip ID, and TCB. VCEK certificates have 7-year validity periods[39] and cannot be individually revoked[40] (only ASKs can be revoked). VCEK certificates are signed by a product-specific AMD SEV Key (ASK), whose certificate is in turn signed by a product-specific AMD Root Key (ARK). Notably, VCEK certificates are the only evidence that a VCEK corresponds to a certain TCB, because AMD re-derives each key given a chip ID (with which it can look up the chip secret) and TCB.

As part of establishing the TLS connection, the client calls `verifyReport()` on the received report and (inner) VCEK certificate (*whatsapp-cf/src/privacy_infra/attestation/MbedTLSBaseAttestationVerifier.cpp*). This function checks that the hash in the report matches the SHA-256 hash of the RSA key in the outer RA-TLS certificate, and verifies that the reported TCB in the report satisfies a certain minimum. It also verifies the signature on the report with respect to the VCEK in the inner AMD-signed certificate. Finally, it verifies the certificate chain (ARK and ASK).

This procedure *does not* include verification that the TCB in the inner VCEK certificate matches the reported TCB in the attestation report.

38. See Chapter 3 of "Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification" document, revision 1.00 at https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/57230.pdf.
39. See Table 9 of the above document.
40. This was stated during discussions with WhatsApp.

This missing check renders the minimum TCB check ineffective and enables exploiting vulnerabilities in previous TCBs. Suppose a previous firmware or microcode version was vulnerable to an attack that allows signing arbitrary data with the VCEK. Then, it could be used to forge an attestation report stating that its TCB is newer than it is. Given this forged attestation report and the VCEK's valid certificate (from sometime in the last 7 years), `verifyReport()` would report success, despite the TCB in the VCEK certificate differing from the TCB in the attestation report.

Recommendation

Verify that the TCB present in the VCEK's certificate matches the TCB in the attestation report, and fail report verification if not.

Location

- Attestation report verification in `verifyReport()` (line 78 of *whatsapp-cf/src/privacy_infra/attestation/MbedTLSBasedAttestationVerifier.cpp*).

Retest Results

2025-04-10 – Fixed

The implementation now explicitly checks that the TCB provided in the attestation report exactly matches the values encoded in the signing certificate.

Low

Transparency Proof Verification Bypass

Overall Risk	Low	**Finding ID**	NCC-E021658-PWU
Impact	High	**Component**	RA-TLS
Exploitability	Low	**Category**	Access Controls
		Status	Fixed

Impact

An attacker is able to run a system with any CVM that has modified firmware as long as the container does not add the transparency proof to the RA-TLS extension.

Description

The RA-TLS certificate extension data is verified by the `verifyBundle` function shown below. This function deserializes the extension data, extracts ASK and VCEK keys, and extracts and verifies the AMD SEV-SNP attestation report. At the end, the function checks if the deserialized data contains the transparency proof and, if present, proceeds with its verification.

However, if the received bundle does not contain a transparency proof, no verification is performed, and the caller is not notified about the missing verification.

It is noted that the block of code is preceded with a comment and task stating that the if clause needs to be removed, at which point the issue would be resolved.

```
attestation::ra_tls::RaTlsError
↳attestation::sev_snp::MbedTLSBasedAttestationBundleVerifier::verifyBundle(
  const std::vector<uint8_t> &serializedBundle,
  const std::vector<uint8_t> &reportData,
  const EnforcedPolicy &enforcedPolicy,
  transparency_report::TransparencyReport &transparencyReport)
{
// ...
  // Check if we have transparency proof.
  if (attestation->ra_amd_sev_snp->transparency_proof != NULL) {
    std::cout << "Transparency proof present in attestation bundle." << std::endl;
    const SignatureResponse *proof = attestation->ra_amd_sev_snp->transparency_proof;
    // Verify the transparency proof.
    if (!attestation::transparency::verifyTransparencyProof(*proof)) {
      errorCode = ra_tls::RaTlsError::TransparencyProofVerify;
      goto onComplete;
    }
    std::cout << "Transparency proof verified successfully." << std::endl;

    // Verify the CVM measurement against digest in the transparency proof.
    // Currently, the digest is derived from sha256(launch_digest | image_id).
    if (!checkProofDigest(proof->message->digest, output.measurement, output.image_id)) {
      errorCode = ra_tls::RaTlsError::AttestationTransparencyProofMismatch;
      goto onComplete;
    }
    std::cout << "Transparency proof matches measurement and image ID in attestation report."
    ↳<< std::endl;
  } else {
    std::cout << "No transparency proof provided in the attestation bundle." << std::endl;
  }
```


```
onComplete:
  cert_ra_ext__attestation_bundle__free_unpacked(unpackedExtension, &kProtoAllocator);

  // Set transparency report data if attestation bundle verification succeeded.
  if (errorCode == ra_tls::RaTlsError::None) {
    transparencyReport.setReportData(serializedBundle);
  }

  return errorCode;
}
```

Figure 14: MbedTLSBasedAttestationBundleVerifier.cpp

Recommendation

Ensure the transparency proof is always verified without exception.

Retest Results

2025-04-10 – Fixed

The code now verifies that the transparency proof is present, and returns an error otherwise, halting the establishment of the RA-TLS connection.

Low

Insufficient VCEK Certificate Validation

Overall Risk	Low	**Finding ID**	NCC-E021658-7DN
Impact	Undetermined	**Component**	Attestation
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

Insufficient validation could lead to undetected misconfigurations or be leveraged in attacks.

Description

The Versioned Chip Endorsement Key (VCEK) is used to sign the SEV-SNP attestation report embedded in the RA-TLS certificate. The VCEK certificate is issued by AMD and has an extension[41] that includes the product name (e.g. Milan or Genoa), chip ID, and TCB (4 version numbers).

As part of establishing the TLS connection, the client calls `verifyReport()` on the received report and VCEK certificate (*whatsapp-cf/src/privacy_infra/attestation/MbedTLSBasedAttestationVerifier.cpp*). This procedure performs minimal validation on the VCEK certificate. Additional validation could rule out certain attacks or reduce the impact of vulnerabilities or changes in AMD's certificate issuance.

- The subject public key, while verified to be an elliptic curve point[42], is not verified to be on any particular curve. It should be on curve P-384[43].
- The subject field should include the common name "CN=SEV-VCEK".
- The extension should include the same hardware ID as specified in the attestation report.

Recommendation

Perform additional validation on VCEK certificate fields as described.

Retest Results

2025-04-10 – Fixed

The recommended tests have been implemented.

41. See Chapter 3 of "Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification" document, revision 1.00 at https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/57230.pdf.
42. See line 174 of *whatsapp-cf/src/privacy_infra/attestation/MbedTLSBasedAttestationVerifier.cpp*.
43. See Table 9 of the above document.

Low

Excessive RA-TLS Certificate Lifespan

Overall Risk	Low	**Finding ID**	NCC-E021658-YF2
Impact	Medium	**Component**	RA-TLS
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

An attacker who can exfiltrate the private key from a CVM could attempt to exploit the system using that key for up to one year.

Description

The CVM creates a new key pair and associated public key certificate every 5 minutes for use in RA-TLS connections, ensuring private key and attestation report freshness. However, the public key certificate is issued with a validity period of one year.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Facebook, Inc., CN=Test Common Name
        Validity
            Not Before: Feb 21 19:59:37 2025 GMT
            Not After : Feb 21 19:59:37 2026 GMT
        Subject: C=US, O=Facebook, Inc., CN=Test Common Name
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (3072 bit)
                Modulus:
                    00:f3:a2:99:78:70:84:38:ee:54:10:3a:55:07:30:
                    cb:b0:63:37:99:b1:d9:16:ce:2f:50:35:8d:cd:15:
                    a7:ae:dd:a8:d5:45:1c:b5:b1:bb:e1:c9:4b:b7:65:
[...]
```

Figure 15: /etc/pki/tls/certs/server.pem

Recommendation

No path has been identified allowing the exfiltration of the private key; however, as an additional security measure to defend against the weakness described in finding "RA-TLS Key Stored in Cleartext Within the CVM", it is recommended to shorten the validity period of the public key certificate to more closely match its lifespan.

Retest Results

2025-04-10 – Fixed

Since private keys are short-lived (a new one is generated every 5 minutes), the certificate lifespan was reduced to 10 minutes.

Low

Potential TEE Type Confusion Attacks

Overall Risk	Low	**Finding ID**	NCC-E021658-H7H
Impact	High	**Component**	TEE attestation
Exploitability	Undetermined	**Category**	Data Validation
		Status	Fixed

Impact

By altering network routing, an attacker may redirect requests meant for a given TEE to another TEE that runs a distinct service with a different behavior, and may inadvertently leak secret user data when presented with the unexpected request.

Description

When a system connects to a TEE (either the user device, connecting to the Orchestrator, or the Orchestrator connecting to the Summarization or Output Guard TEEs), it uses the RA-TLS protocol to make sure that it connects to a real TEE running the expected software image. Once the attestation report has been verified, the client knows that it is talking to a real TEE, but it must still check that the software image is the right one. The hash of that image (the *measurement*) is part of the attestation report, and the client extracts it, in order to compare it with a reference value. That value is not known *a priori* by the client; instead, the server (the TEE, in this scenario) sends the value as part of a *transparency proof*, which is a protobuf object signed by a third party (Cloudflare). The signed data contains:

- a version number (`uint32`);
- a namespace (`string`);
- a timestamp (`uint64`);
- an epoch (wrapped `uint64`);
- the digest value (hash of the software image).

In the current implementation, the client verifies the signature and compares the included digest with the value extracted from the attestation report, but it does no other check. Finding "Missing Freshness Check on Transparency Proof for Attested Image" explains that the lack of check on the timestamp implies that an old, stale proof may be used by an attacker. In this finding, we note that the namespace is also ignored. This implies that *TEE type confusion attacks* are feasible: an active attacker with control over the network routing may redirect the connection from the client to a valid TEE (hence able to provide an attestation and a corresponding signed transparency proof) but running a different service than the client expects. The client will send the request with the user data to that TEE, and how the TEE reacts in that case depends on the details of the request encoding and what the contacted TEE service expects.

One potential scenario would be that, at a later date, a new TEE-based system is created, with transparency proofs, to fulfill a different service that expects requests containing only public data. That new system may legitimately log the contents of received requests, since it expects only public information. An active attacker, by redirecting the connection from the user's device to that new TEE (instead of the Orchestrator) will induce the new TEE to copy into its logs the contents of the user request, which contains the user's private data (identities, message contents...). This would constitute an efficient breach of the data privacy.

Recommendation

To avoid TEE type confusion attacks, the connecting client should verify that the TEE it is talking to is really the software that it expects. This can be done in several ways:

- A list of reference measurements for the expected TEEs may be hardcoded into the clients. This would even allow removing the need for a signed "transparency proof", but would raise issues on the distribution and upgrade of that list of reference measurements.
- The TEE type could be encoded in the "namespace" field of the transparency proof (e.g. as the string "WhatsApp Message Summarization - Orchestrator", for the Orchestrator TEE). Meta would need to maintain a registry of allocated namespaces to ensure that no string collision inadvertently occurs. The client must explicitly verify that the "namespace" field in the transparency proof it received has the correct value.
- The type check could also be done in the HTTP path. If *all* TEEs using the attestation/transparency mechanism only run HTTP-based services (inside the RA-TLS connection), then every request is attached to a given path value; paths may then encode the TEE type (e.g. the path for a request to the Orchestrator could be `/whatsapp-summarization/orchestrator`). If a TEE receives a request for an unexpected path, then it must consider it as potentially misrouted secret data and must drop it (it may log the faulty path value, but not the contents). Like the namespace solution, this method requires maintaining a registry of allocated paths. This method is slightly more restrictive since it requires all services (present and future) to be HTTP-based; on the other hand, it might be easier to apply in the existing code base, where attestation verification is deep inside a support library with no provided parameter to specify an expected namespace.

Retest Results

2025-04-10 – Fixed

Sets of trusted namespaces were added to client policies, which are checked when verifying an attestation bundle. When checking a transparency proof, the function `verifyTransparencyProof()` takes an `expectedNamespaceType` argument. This argument serves as a hint to the client, indicating which *group* of trusted namespaces (grouped by type in the client's chosen policy) it should search to find the namespace from the transparency proof. If the signed namespace present in the proof does not appear in the set of trusted namespaces for the expected type, then the proof is rejected. This check effectively stops TEE type-confusion attacks as long as the groups of trusted namespaces in the applicable policy are disjoint, which is the case for all policies currently in use.

Low

Overlay Filesystem Covering the Entire RootFS

Overall Risk	Low	**Finding ID**	NCC-E021658-JXF
Impact	Medium	**Component**	CVM
Exploitability	Low	**Category**	Access Controls
		Status	Risk Accepted

Impact

An attacker who obtained filesystem read/write capabilities would be able to modify any existing binaries and configuration files. These modifications, which are temporary and would last only for the lifespan of the CVM, could aid the attacker in exploiting the system.

Description

The rootfs of the CVM is entirely overlayed, potentially allowing an attacker to modify and/or replace files anywhere in the filesystem, although changes would not be persistent.

The overlay is configured during its initialization by the `cvm_init.sh` script. This script creates the dm-verity device mapping, mounts the root filesystem, and sets up the overlay mount, as the excerpt below shows.

```
# Overlay RO_ROOT
/bin/mount -t overlay -o \
rw,\
noatime,\
lowerdir="${RO_ROOT}",\
upperdir="${RW_ROOT}"/rw,\
workdir="${RW_ROOT}"/work \
none "${NEWROOT}" && echo "Overlay filesystem mount successful."
```

Figure 16: cvm_init.sh

Below is shown the output of the `mount` command executed on the Orchestrator, showing the filesystem type of the root filesystem to be OverlayFS.

```
[root@localhost ~]# mount | grep overlay
none on / type overlay (rw,noatime,lowerdir=/mnt/rootfs,upperdir=/mnt/rootwritefs/rw,workdir=/
↳ mnt/rootwritefs/work,uuid=on)
```

Recommendation

Although the VM is short-lived, a safer solution would be having a dedicated read/write partition that is not verified using dm-verity, while the rootfs partition would be dm-verity-verified and read-only. This would allow the removal of the overlay filesystem from the kernel, preventing an attacker from overlaying the filesystem at runtime. Additionally, it would be necessary to prevent the loading of kernel modules or require kernel module signing, as discussed in finding "Opportunities for Enhanced CVM Kernel Hardening".

Retest Results

2025-04-16 – Not Fixed

Meta does not plan to address this issue. Since the initial report of this finding, containerization of the main service within the CVM was added, which partially mitigates the potential impact.

Client Response

Meta has accepted this risk given it requires privileged execution within the CVM. The system already limits privileged access by containerizing the application workload on top of the TEE isolation controls. Further, we continue to explore adding filesystem protections and kernel hardening in the near future.

Low

Non-ASCII Statements in Exported Logs

Overall Risk	Low	**Finding ID**	NCC-E021658-FDV
Impact	Low	**Component**	CVM
Exploitability	Low	**Category**	Auditing and Logging
		Status	Fixed

Impact

Logging statements containing potentially sensitive data may be undetected by the filtering feature, and leave the boundary of the CVM.

Description

In order to preserve the confidentiality of data inside the CVM, the service logs are filtered prior to leaving the boundary of the CVM. The “CVM debuggability: logging, remote diagnostics and metrics collection” document summarizes some of the design principles for the filtering rules as follows:

> Considerations for applying filtering rules
>
> - Filters should be applied on the “allowelisted” basis. Meaning, unless the message is explicitly allowed in the rules, it should be denied from the export.
> - For now, the focus will be on basic allowlisted filters (e.g. static strings or what service owners are confident to be “safe”).
> - Exclude any log statements which have unicode (non-ascii) included.

In particular, the documentation notes that any log statements containing non-ASCII characters are to be filtered out, as the presence of non-ASCII characters could indicate that the data did not originate from a human, and thus may include data that should not leave the CVM.

This filtering feature is implemented in the `sanitize_log_line()` function. However, note that non-ASCII characters are not filtered out at any point within this function. This is of particular concern in the `allowed_tags` variant of filtering, which allows developers to mark log statements with an `allowed_tag` that identifies them as safe (as compared to the `allowed_patterns` approach, which explicitly lists allowed logging statements in the configuration file):

```
pub fn sanitize_log_line<'s>(
    line_msg: &'s str,
    config: &ExportConfig,
) -> Result<Option<Cow<'s, str>>> {
    for tag in config.allowed_tags.iter() {
        let tag_regex_literal = format!("((\\s*){}(\\s*)", regex::escape(tag));
        let tag_regex =. Regex::new(&tag_regex_literal)?;
        // only attempt to replace first instance
        let tag_removed_str = tag_regex.replace(line_msg, " ");
        // NOTE - Cow::Borrowed returned == no replacement occured
        if matches!(tag_removed_str, Cow::Owned(_)) {
            let trimmed_str = tag_removed_str.trim();
```


```
            return Ok(Some(Cow::Owned(trimmed_str.to_string())));
        }
    }
    // ...
```

Figure 17: rsyslog_exporter_filter.rs

Since rust strings support unicode-encoded characters by default, the presence of non-ASCII characters may remain undetected by the `sanitize_log_line()` function, and logging statements containing non-ASCII characters may thus leave the boundary of the CVM.

Recommendation

Filter out any log line containing non-ASCII characters within the `sanitize_log_lines()` function.

Retest Results

2025-04-17 – Fixed

The `sanitize_log_lines()` function was updated to exclude the log line if the `line_msg` variable contains any non-ASCII characters, thus matching the documentation.

Info

Traffic Analysis on Message Length

Overall Risk	Informational	**Finding ID**	NCC-E021658-TR3
Impact	Low	**Component**	RA-TLS
Exploitability	Low	**Category**	Cryptography
		Status	Risk Accepted

Impact

Observation of the sizes of the encrypted messages between client and TEE may reveal some partial information about the exchanged data. This low-bandwidth side-channel may help in the exfiltration of user data, following a compromise of the LLM weights or system prompts.

Description

RA-TLS and OHTTP apply several layers of encryption on the user data which is exchanged between the client and the orchestrator TEE, and between TEEs. Encryption hides the data *contents*, but preserves the data *length* (usually with byte precision in modern TLS cipher suites, which use padding-less modes such as AES/GCM). Outsiders observing the data exchanges may thus learn the respective sizes of the client requests and answers. This can impact user privacy in two ways:

- The data exchanged with OHTTP is really an inner TLS connection. TLS includes a handshake process; it starts with a `ClientHello` message from the client, which in particular lists the client's abilities (supported cryptographic algorithms and protocol options). Observing the size of that message may help with categorizing requests depending on the client-side TLS library version, which (partially) contradicts the promises of oblivious HTTP.
- The size of the response from the TEE may be used as a low-bandwidth channel for exfiltrating user data elements. This may help with the exploitation of a partial compromise of the LLM weights or system prompt in a TEE.

Observation of data sizes is commonly called *traffic analysis*. The main defense is use of *padding*, to normalize message sizes to a few fixed values, thus removing most of the information leak. At the oblivious HTTP level, this is described in RFC 9458 Section 6.2.3, which also covers timing attacks (which seem less of an issue in the envisioned architecture). Padding can also be added in TLS itself, as long as TLS 1.3 is used. Padding necessarily increases message sizes, which implies trade-offs between efficiency and security.

Recommendation

Padding should be added in both the OHTTP layer, and inside either the inner TLS (which terminates on the TEE orchestrator) or at the application message level (i.e. on the Protobuf messages that convey the user data back and forth). For the choice of the padding length, the Padmé method (section 4 of the paper) is recommended as providing a decent compromise, with a low size overhead (about 12% on average) and heuristically good information hiding characteristics.

Retest Results

2025-04-10 – Not Fixed

The impact of this side-channel leakage was considered too small to justify any investment in remediation. Meta thus acknowledged this finding and accepted the risk.

Client Response

Meta agrees that the impact of this finding is minimal and would require a sophisticated attack to leak minimal information about unidentified users. We will continue to harden this attack surface against this type of attack.

Info

Use of PEM Encoding in Attestation is Fragile

Overall Risk	Informational	**Finding ID**	NCC-E021658-7VK
Impact	Low	**Component**	RA-TLS
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Local operating system or library version variants in the client may break the attestation mechanism, leading to rejection of the connections and a loss of functionality.

Description

In RA-TLS, the server (TEE) generates at start-up time a new asymmetric key pair, which will be used to establish the TLS connection between client and server. To bind this key to the attestation mechanism, a hash of the public key is included in the (signed) attestation report. This hash is described in the architecture documentation as:

> With SHA256 of the PEM public key as the first 32 bytes of the report data

The choice of PEM is surprising and a bit fragile. PEM is, historically, a reference to a tentative standard for “privacy enhanced mail” from the early 1990s; PEM was never really used, but some elements of it were integrated into PGP, including the general structure of an ASCII-compatible encoding for binary objects, which was later incorporated into the well known OpenSSL library. Indeed, the current RA-TLS implementation calls the `PEM_write_bio_PUBKEY()` OpenSSL function. In practice, the PEM format is “what OpenSSL does”, which is not fully specified and is not guaranteed to never change.

In more details, the public key is first encoded into an ASN.1 structure of type `SubjectPublicKeyInfo`, which is also how it appears within an X.509 certificate. The structure is converted into a sequence of bytes by using DER encoding. PEM takes that DER-encoded ASN.1 structure, converts it to characters using Base64, and breaks the output into lines of text, with header and footer lines, yielding a result similar to the following:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAs77ZrEAlWycAVocFzPwV
wOGqDiZUsqsN19Is1+EqEPvMbTYNzx9R+9vzdXhlV5ooS+Qe6Lb37wNu5InR902S
Z38hms3hLL8o77bV/ZJ30bROrm0AqP2RCKtdfPRZR5rcwYD8yMHmLgkg/VvbMX6Y
mqFa7Mo/UKXJ9Lm96v6cF4ptg1fSukgRQr2rwqDfQtHUK84yK1QhrMDGIJ6m1uTz
3KqcfMtMdXSuvE0NCQd60MlDJBCOqNM7trAK7FAnlMqsCYPiB9dQUgfBgqOp6yCx
H3kMiAguPb51oOmd6n+UrOyFVKDm+7Iw6rf4iE3a7qLdLL1Oq/ZmiF0BXSYrwZfI
FwIDAQAB
-----END PUBLIC KEY-----
```

The PEM format was *designed* to tolerate various text-level alterations. In practice, the following variants are the most commonly encountered with PEM objects:

- End-of-line sequences may depend on the used operating system. Windows traditionally uses CR+LF line endings (two bytes, 0x0D then 0x0A) while Unix-derivatives such as Linux or macOS use LF line endings (one byte, 0x0A).
- Line length is arbitrary; 64 and 76 characters per line are the most common conventions.
- The end-of-line may be missing at the end of the footer line.

- An additional *byte order mark* (three bytes: 0xEF 0xBB 0xBF) may be present at the start of the string.

These variants imply no alteration to the public key itself, but modify the string of bytes that constitute the PEM string and thus impact the SHA-256 hash of that string. If the client does not use the same conventions as the server, then the hash value computed by the client will not match the value in the attestation report, and the client will reject the connection; the message summarization will not work for that client.

Recommendation

Instead of applying the SHA-256 hash on a PEM string, the DER-encoded `SubjectPublicKeyInfo` structure should be used directly. This would avoid any issue related to the PEM layer. In the OpenSSL API, the `EVP_PKEY_get1_encoded_public_key()` function can be used to obtained the DER-encoded `SubjectPublicKeyInfo` structure from an in-memory public key.

Retest Results

2025-04-10 – Fixed

Raw DER encoding (not PEM) is now used.

Info

Missing Network Links for Remote Attestation Validation

Overall Risk	Informational	**Finding ID**	NCC-E021658-G3Q
Impact	Low	**Component**	TEE attestation
Exploitability	Undetermined	**Category**	Other
		Status	Fixed

Impact

The architecture description lacks the description of some network connections originating from the TEEs and targeting external third-party servers for remote attestation validation purposes.

Description

The Summarization and Output Guard TEEs use NVIDIA GPUs, with Confidential Computing (CC) enabled, to prevent inspection of user data even by Meta themselves. CC relies on an attestation framework that is used by the TEE to ensure that it is sending the user data only to a real, uncompromised GPU running the expected software. The validation of the attestation is normally done with an NVIDIA-provided SDK, which itself delegates the operation to cloud systems maintained by NVIDIA. In the current architecture diagrams, these external links are not shown. Such network links should be clearly specified so that network-level monitoring and security controls can be set appropriately.

A similar case can be made for the TEEs' own attestations (AMD-SEV-SNP). In that mechanism, each attestation report is signed with a per-chip key (VCEK), and the public part of that key is wrapped in an X.509 certificate issued by AMD, that can be obtained by issuing an HTTP request to AMD's cloud systems. These certificates are normally fetched by the TEE itself, and sent to the validating client (which may be another TEE) as part of the RA-TLS protocol. These external connections to AMD's cloud systems can be made part of the deployment procedure of the host system running the TEE, because the certificates are valid for a long time (7 years). However, the X.509 validation should also involve revocation check, which entails downloading a fresh CRL from AMD's systems. If such a CRL check is implemented, then some external network connections originating from the TEEs will be involved, and will need to be properly documented.

Recommendation

Document the external network links required for NVIDIA CC attestation, and (if used) for AMD's CA revocation information.

Retest Results

2025-04-11 – Fixed

The whitepaper draft documents that NVIDIA's Remote Attestation Service (NRAS) is being used to validate attestations.

Info

Fragile Filtering Approach for Logs

Overall Risk	Informational	**Finding ID**	NCC-E021658-VDT
Impact	None	**Component**	CVM
Exploitability	None	**Category**	Auditing and Logging
		Status	Fixed

Description

In order to protect the confidentiality of data within the CVM, all logs in the CVM are passed through a filter prior to being output outside of the CVM in order to facilitate debugging of CVM operations. Two main approaches are provided for configuring this filter, `allowed_tags`, consisting of a small identifying string included within the logging statement at the source to mark it as "safe", and `allowed_patterns`, which provides an explicit list of allowed logging statements via regex patterns. The relevant *README.md* file notes that the `allowed_tags` approach is preferred, as it is more resilient to minor changes in the logging statements:

```
3. Define an explicit-allow filtering config to be applied to your systemd unit logs
    a. An established guidance from security XFN is: prefer using `allowed_tags` (which make
    ↪ log emissions explicit for analysis at the callsite) over `allowed_patterns` (which can
    ↪ fall out of sync with logging callsite contents).
    b. Allowing all log lines is not acceptable in general
```

Figure 18: README.md

The filtering feature for logs is implemented in the `sanitize_log_lines()` function. The current filtering approach for logs using `allowed_tags` may be a little fragile, depending on the particular choice of `allowed_tag` for a given service. In particular, `allowed_tags` consists of simple strings, which are not required to have any particular format, or to contain any concrete indicator characters. Additionally, `allowed_tags` can be contained at any point in the logging string, and do not need to be located at the start. As such, an unfortunately named `allowed_tag` may be accidentally included in the body of the logging statement due to collisions with other internal naming conventions, thus unintentionally allowing a potentially unsafe logging statement to leave the CVM boundary.

As a related issue, `allowed_tags` without clearly defined delimiter boundaries may also be subject to substring collisions that would allow logging statements to be output unintentionally. For instance, if safe logging strings were marked with the string `"ALLOWED_TAG"`, and unsafe logging strings were marked with the string `"DEFINITELY_NOT_ALLOWED_TAG"`, the `sanitize_log_lines` function would mark logging lines with the `"DEFINITELY_NOT_ALLOWED_TAG"` substring as safe (since they contain `"ALLOWED_TAG"` as a substring), and allow them to leave the boundary of the CVM. Note that current `allowed_tags` all follow the `"[XYZ_SERVICE_ALLOWED]"` format, which is safe from these substring attacks due to the presence of the `"["` and `"]"` delimiters, but this practice is not documented nor enforced within the code.

Finally, note that the `sanitize_log_line()` function removes the `allowed_tag` prior to returning the sanitized log line, but only attempts to replace the first instance of a tag found. As such, if a message is tagged with multiple `allowed_tags`, the resulting log message would still contain some `allowed_tag` values: the message `"tag1 tag2 log_message"` would get output as `"tag2 log_message"` instead of `"log_message"`. Note that this would not result in leaking any unsafe data, as `allowed_tag` values are contained within the source code and thus do not contain sensitive data, but may not be the desired behavior.

Recommendation

Consider formalizing and enforcing the location and format of `allowed_tags` in order to minimize the risk of substring attacks. Additionally, consider whether all `allowed_tag` values should be stripped from a logging message if multiple `allowed_tag` values are included.

Retest Results

2025-04-17 – Fixed

The *README.md* file was updated to note that allowed tags should consist of alphanumeric and underscore characters only, wrapped in square brackets, matching the existing practice. Additionally, the code was updated to validate that all `allowed_tags` match this pattern upon initialization of the filtering service.

Finally, Meta noted that stripping only the first occurrence of an `allowed_tag` was an intentional choice for simplicity. Additionally, they noted that logging messages should never contain multiple `allowed_tag` values, and thus accidentally including an `allowed_tag` in the code in the case of multiple tag values would allow developers to detect and remediate these instances. As such, this finding is considered *Fixed*.

11 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.