Amazon Bee Private Compute

Security Assessment with Fix Review

September 9, 2026

Prepared for:
Amazon Bee

Prepared by: **Artem Dinaburg, Tjaden Hess, Keegan Ryan, and Artur Cygan**

Table of Contents

Table of Contents **1**
Project Summary **3**
Executive Summary **4**
Project Goals **6**
Project Targets **8**
Threat Model **9**
Threat Model Scope 9
Sensitive Assets 9
Data Flow 9
Components and Trust Zones 11
Threat Actors 14
Threat Scenarios 15
Project Coverage **19**
Summary of Findings **20**
Detailed Findings **22**
1. Legacy boot path executes code outside measurement context 22
2. Sigstore context is never policy-enforced 24
3. Application data disks are mounted by device path without measured or CVM-bound protection 27
4. Android RFC 3161 timestamp verification can fail open and accept attacker-chosen verification time 30
5. AMI Rekor entries are never signature-verified or bound to the transparency team signing certificate 33
6. Go proxy accepts NitroTPM COSE_Sign1 payloads without signature verification 37
7. Shared LLM prefix caches are not isolated across users 41
8. Shared CVM Redis access is not scoped by service identity 43
9. Bee Private Compute server derives deterministic CTR keys for BLE with insufficient checks 46
10. Client attestation failures do not block key release 48
11. Paired-app developer tokens lack scoped permissions 52
12. Conversation summaries are stored unencrypted in the search index 53
13. Risk of container environment-variable injection via AWS Secrets Manager that can bypass the attestation chain 55
14. Confidential VM image includes unnecessary packages 58

15. APNs proxy Lambda logs notification content and request payloads 60
16. Internal networking policy is overly lax for the Confidential VM tier 62
17. Internal security controls for user data are overly lax 65
18. Integration APIs receive user data without a sharing-consent check 70
19. Push notifications routed to other users are sent in plaintext 73
20. Consent proof is generated server-side, not by the client as documented 75
21. Ambiguous measurement serialization lets distinct configurations collide 77
A. Vulnerability Categories 79
B. Threat Model Maintenance 81
When to Update Your Threat Model 81
Updating Your Threat Model 82
C. Fix Review Results 84
Detailed Fix Review Results 86
D. Fix Review Status Categories 89
About Trail of Bits 90
Notices and Remarks 91

Project Summary

Contact Information

The following project manager was associated with this project:

Emily Doucette, Project Manager
emily.doucette@trailofbits.com

The following engineering director was associated with this project:

Tjaden Hess, Engineering Director, ML Security
tjaden.hess@trailofbits.com

The following consultants were associated with this project:

Artem Dinaburg, Consultant
artem.dinaburg@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Keegan Ryan, Consultant
keegan.ryan@trailofbits.com

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Project Timeline

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

Date	Event
May 11, 2026	Pre-project kickoff call
May 18, 2026	Status update meeting #1
June 1, 2026	Status update meeting #2
June 8, 2026	Status update meeting #3
June 10, 2026	Delivery of report draft
June 15, 2026	Report readout meeting
July 1, 2026	Completion of fix review
July 23, 2026	Delivery of final comprehensive report
September 9, 2026	Delivery of updated report with fix review

Executive Summary

Engagement Overview

Amazon engaged Trail of Bits to review the security of its Bee wearable and Bee Private Compute system.

Bee Private Compute is Amazon's platform for a personal AI assistant that continuously captures audio through a wearable device (or supported Apple Watch models) and analyzes it with frontier AI models. Because the system ingests highly sensitive personal data, privacy is enforced architecturally rather than by policy: user devices generate root secrets that never persist on Amazon systems, encrypt data before it leaves the device, and release scoped processing keys only after cryptographically verifying the runtime that will receive them. The platform comprises client apps that manage keys and verify attestations, the wearable hardware, and Confidential VMs that decrypt and process user data only after proving their software identity—rooted in NitroTPM measured boot and Sigstore transparency logs.

A team of four consultants conducted the review from May 11 to June 10, 2026, for a total of eight engineer-weeks of effort. Our testing efforts focused on evaluating the robustness of the transparency claims and threats from external actors that could compromise sensitive personal data. With full access to source code and documentation, we performed static and dynamic testing of the Bee Private Compute servers, Bee Private Compute clients, and AMI builder using automated and manual processes. From June 30 to July 1, 2026, one consultant performed a review of the fixes provided by Amazon, as described in appendix C.

Observations and Impact

Over the course of this engagement and the fix review that concluded on July 1, 2026, we reviewed the Bee Private Compute system to establish the following properties:

- Root secrets are generated on the user's device and do not persist on Amazon systems; server-side processing keys are scoped, time-limited, and held only in confidential VM memory.
- Production clients refuse to release key material to a runtime that fails attestation, and they verify the owner, environment, role, and service identity carried in the signed attestation context.
- Only code registered in the public transparency log can receive processing keys, and clients verify inclusion in the transparency log.
- Operators cannot inject or modify code at runtime.

Several of these properties did not hold at the time of the initial review; all findings reported during the review have since been resolved or mitigated.

The engagement uncovered issues in multiple critical components of the Bee system, including multiple avenues for a malicious Amazon operator to inject arbitrary code in attested instances (TOB-BEE-1, TOB-BEE-3, TOB-BEE-13) and missing verification of signatures and attestations in clients and proxies (TOB-BEE-2, TOB-BEE-4, TOB-BEE-5, TOB-BEE-6, TOB-BEE-10). These issues have been resolved.

In addition, the design document described privacy features that were not yet implemented at the time of the review. These include context-policy enforcement, signature verification on transparency-log entries and attestation documents, fail-closed client attestation, encryption of all user content, and a cryptographic consent gate on outbound plaintext, none of which the code provided at the time of the review. These features have since been implemented in the Bee Private Compute system.

Recommendations

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

- **Conduct targeted feature-specific audits.** The agent sandbox and integrations are currently experimental and gated from regular users, but once they are enabled, they will present a substantial new attack surface.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	7
Medium	7
Low	2
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	7
Configuration	4
Cryptography	5
Data Exposure	4
Data Validation	1

Project Goals

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

Security Model Validation

- Can AWS Nitro attestation ensure that the mobile app sends encrypted audio only to servers running authorized AMIs?
- Does the proposed key architecture (i.e., that keys stay on the mobile app and are ephemeral in TEE memory only) prevent operator access to plaintext data?
- Can the stateful PostgreSQL design (i.e., data encrypted on-disk and keys stored in TEE memory only) allow infrastructure administrators to tamper with database contents or access user data?

Cryptographic Implementation

- Does client-side encryption (by the mobile app) use secure key derivation and storage?
- Can server-processing keys held only in Confidential VM memory be extracted through the application or infrastructure layers?
- Are envelope keys properly protected during TLS transport to the TEEs?
- Is cryptographic forward secrecy present and sufficient to prevent compromise of historical data?

Attestation and Trust Chain

- Does the AWS Nitro root of trust provide verifiable TEE integrity before data transmission?
- Can malicious AMIs be served without inclusion in the transparency log or detection by monitors?
- Can unmeasured code or configuration impact the execution of inference servers?
- Is client-side attestation verification robust against MitM or rollback attacks?

Infrastructure Isolation

- Do CDK constructs properly isolate the TEEs from standard AWS infrastructure?
- Can the AMI builder service be compromised to inject malicious code?

- Are AMIs strongly linked to build provenance? Are the resulting binaries reproducible and auditable?
- Are network policies and ACLs sufficient to prevent lateral movement?
- Can AWS operators influence the execution of approved AMIs after deployment?

Phased Deployment Risks

- What security gaps exist in ready components (AMI builder, encryption, RDS) that could impact the initial launch?
- How should the in-progress attestation implementation be architected to minimize vulnerabilities?

Project Targets

The engagement involved reviewing and testing the following targets.

Bee Private Compute iOS App

Type	Mobile
Platform	iOS

Bee Private Compute Android App

Type	Mobile
Platform	Android

Bee Private Compute AWS CDK

Type	IaC
Platform	AWS

Bee Private Compute Server

Type	Web Service
Platform	Docker, AWS Nitro

AMI Builder Service

Type	IaC, AWS Service
Platform	AWS

Threat Model

As part of the audit, Trail of Bits conducted a lightweight threat model, using our TRAIL, or Threat and Risk Analysis Informed Lifecycle, process, the threat modeling methodology for exhaustively following the implications of flawed trust assumptions throughout a modeled system. Our TRAIL process 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 assessed the design of the confidential inference system built on Amazon EC2 with NitroTPM attestation for the Bee wearable AI assistant product by reviewing documentation provided by the client and interviewing client engineers.

Threat Model Scope

This threat model covers the holistic privacy and security of the wearable device and private inference system, with a focus on transparency and zero-operator-access principles. This review does not include a full analysis of the AWS Nitro System itself.

Sensitive Assets

The following are the primary assets under protection in the private inference system:

- Audio data recorded by wearable devices
- Textual transcripts and derived content produced by the LLM inference system
- Server-side stored user data related to asynchronous and scheduled tasks
- Data derived from third-party integrations
- Credentials for third-party integrations

Data Flow

Below, we depict known connections between system components of the Amazon Bee Private Compute product. These diagrams are intended to convey our understanding of the system as a whole. Further details are discussed in the Components and Trust Zones report subsection.

The colored regions indicate trust boundaries separating zones, where the system enforces (or should enforce) interstitial controls and access policies. Where additional message-specific authentication or authorization information is needed, we label the connection with it.

The basic flow involves user data flowing from a wearable device to a user mobile device via Bluetooth Low Energy (BLE). The data then travels via a TLS connection to an attested API service and then via mTLS to other cached and internal services. TLS connections are

authenticated by certificates issued by the AWS Private Certificate Authority (CA) instance and also carry NitroTPM attestation documents that are verified during certificate validation.

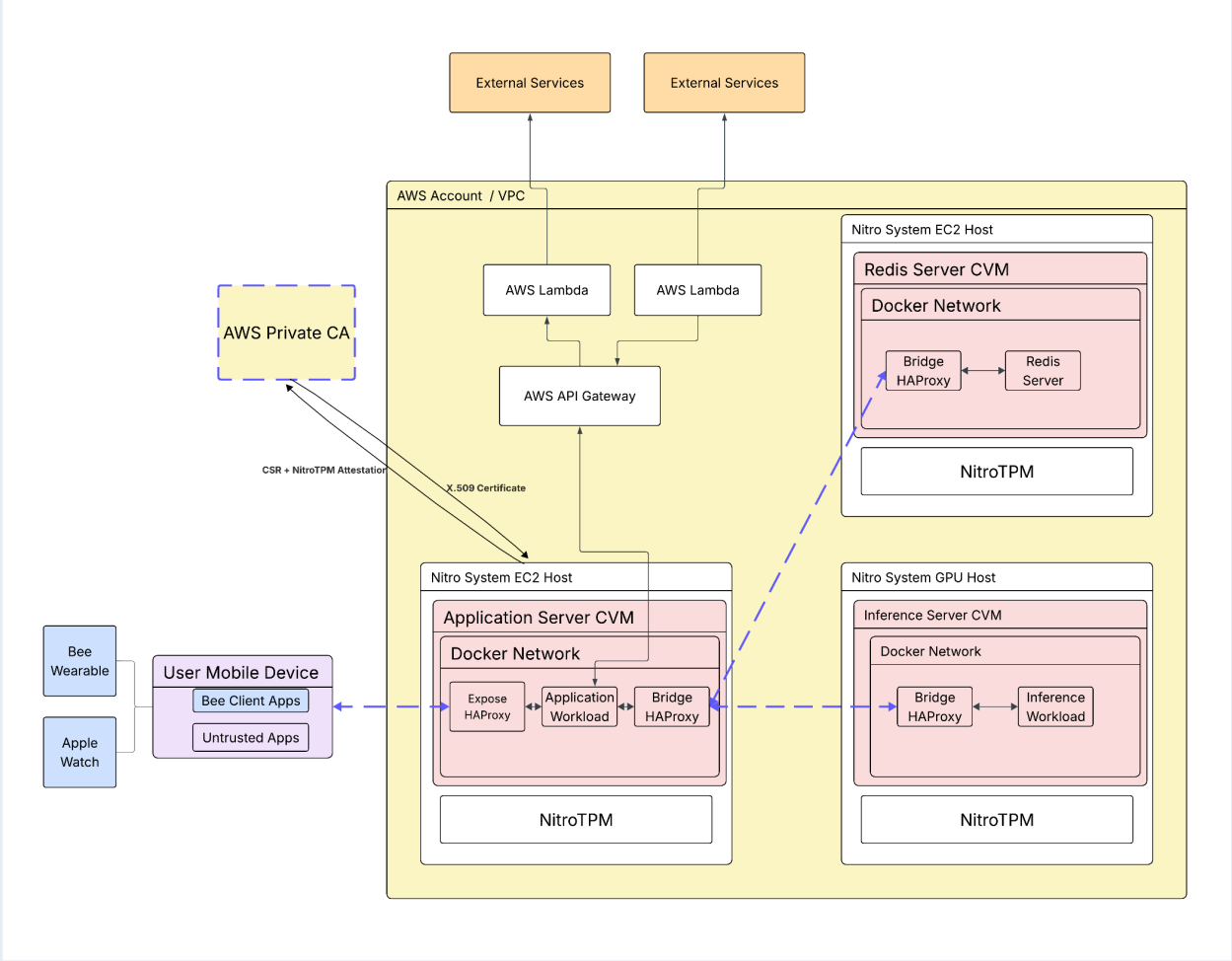

Figure 1: Simplified high-level data flow of the Bee Private Compute service

Attested services are identified by NitroTPM Platform Configuration Register (PCR) values that incorporate both the base AMI hash and a manifest of Docker images and bridge configurations. Figure 2 shows an overview of the build pipeline, whereby components built by Bee Private Compute-team CI/CD are stored, signed, and logged by the transparency team. TLS clients verify the PCR values in an attestation report against a Sigstore bundle. Clients also check the inclusion proof for the Sigstore bundle in the public-good Rekor log.

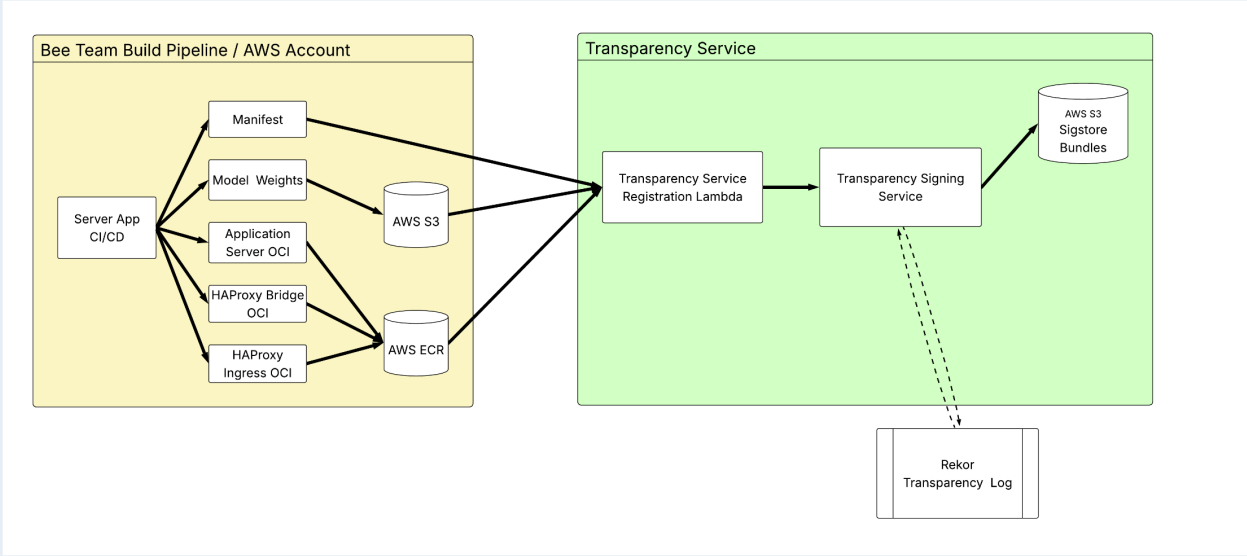

Figure 2: Simplified view of the AMI and container build process, with transparency logging

Components and Trust Zones

The following table describes the components that make up the Amazon Bee Private Compute system, as well as the external dependencies on which they rely. These system elements are further classified into *trust zones*—logical clusters of system components and dependencies, between which the system enforces (or should enforce) interstitial controls and access policies. A *trust boundary* delineates the edge of a trust zone.

Component	Description
Bee Private Compute Service Production AWS Account	Amazon Bee Private Compute services are operated from the Bee Private Compute team root AWS account.
Amazon EC2 Control Plane	Amazon EC2 instance lifecycle, monitoring, user-data scripts, and similar infrastructure are under the control of the root AWS account.
Persistent Storage	Raw disk images, snapshots, and persistent databases are available to the Bee Private Compute AWS account and may be read or modified arbitrarily. These raw disks may be encrypted, however, by the attested services.
AWS Private CA	Client-server and service-to-service bridge connections are protected by server authenticated TLS and mTLS respectively. The Bee Private Compute team maintains an AWS Private CA instance that validates attestation documents produced by the Amazon EC2 CVM and issues certificates to the

	HAProxy containers.
AWS Secrets Manager	Production secrets for the Bee Private Compute services are stored in AWS Secrets Manager. These secrets include but not limited to the following: • OAuth client secrets • Apple Push Notification service (APNs) secrets • Database connection configuration
Amazon S3 Buckets	Amazon S3 is used for storage of objects including the following: • LLM and TTS model weights • Transparency artifacts • Encrypted Lorekeeper user-memory / search index storage. • User file storage • Monitoring and log storage
Transparency Service	The transparency service is managed under an independent AWS organization and by a separate team of engineers from the core Bee Private Compute services. The transparency service is responsible for managing the signing keys for Sigstore publishing. The transparency service independently recomputes PCR values and hashes, ensuring that all artifacts to be published on the transparency log are properly retained for later auditing.
Transparency CA	The transparency organization maintains a CA used to sign bundles for publication to the Rekor transparency log. This is an entirely separate CA from that used by the Bee Private Compute system to issue TLS certificates.
Amazon S3 Buckets Replica	The transparency service manages a replica of the buckets used for artifacts uploaded to the transparency log, to ensure indefinite retention.
Attested Services and CVM	The attested services and CVM run on EC2 instances built on the Nitro system and should be protected from tampering even from the Bee Private Compute AWS account root. The code and configuration for these services is committed in the Rekor transparency log.
Confidential VMs	Confidential VMs are virtual hosts based on the AWS Nitro System. These include CPU and GPU instances that serve the application logic, Redis caches, and GPU LLM inference. Amazon EC2 instances consume a manifest via the user-data injection mechanism and are responsible for measuring and booting the Docker containers indicated in the manifest, as well as establishing network bridges.

Application Services	The primary API and agent services run as Docker containers within the attested Amazon EC2 hosts. Application services do not communicate directly with outside services but instead use HAProxy bridge containers to manage incoming and outgoing TLS sessions.
HAProxy Bridges	Application containers use HAProxy bridges and TLS proxies to terminate incoming client connections and to establish mTLS-protected inter-service sessions. Bridge connections and exposed services are generated by the Amazon EC2 CVM using a statically declared manifest.
Redis Caches	Redis caches retain server processing keys and sensitive user data in-memory, with persistence disabled, with a seven-day expiry.
Bee Private Compute Client	These components include the Bee Private Compute client iOS/Android applications and the Bee wearable or Apple Watch apps.
Mobile Application	The mobile application streams audio from the wearable device via Bluetooth and manages user data encryption keys. The mobile app is a user's primary interface to the Bee Private Compute system.
Wearable Device	The wearable Bluetooth LE device pairs with the mobile device in order to stream audio data to the Bee Private Compute service.
User Mobile Device	This zone includes the Android or iOS device, along with any on-device enclaves or backup systems.
Third-Party Services	Third-party services include CLI applications or other integrators with which the end-user establishes a pairing session, as well as Amazon Alexa integrations and other non-Bee Private Compute Amazon services.

Threat Actors

When conducting a threat model, we define the types of actors that could threaten the security of the system. We also define other users of the system 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 who is 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 in descriptions of the security findings that we uncovered through the threat modeling exercise.

Actor	Description
Bee Private Compute Team Engineer	An engineer on the Bee Private Compute team with repository write access or an actor with access to credentials for such an engineer.
Bee Private Compute Team Administrator	An actor with access to the AWS IAM resources on the Bee Private Compute AWS account, up to and including root account admin permissions.
Transparency Team Engineer	An engineer with repository write access on the transparency team or an actor with access to credentials for such an engineer.
Transparency Team Administrator	An actor with access to the AWS IAM resources on the transparency AWS account, up to and including root account admin permissions.
Application User	A user with a registered account for the Amazon Bee Private Compute service.
External Attacker	An attacker with access to externally visible APIs or to general web resources consumed by Bee Private Compute agents.
Device-Local Attacker	An attacker with physical access to an end user's Bee wearable or arbitrary access to applications resident on the end user's mobile device.

Threat Scenarios

The following table describes possible threat scenarios given the design, architecture, and risk profile of Amazon Bee Private Compute. These scenarios are designed to enumerate possible scenarios that the Bee Private Compute system may face and is designed to mitigate. For the purposes of the implementation reviews, we take these scenarios as starting points for exploit scenarios.

#	Scenario	Actor(s)	Component(s)
1	**Publication of malicious Bee Private Compute server code.** A Bee Private Compute engineer or actor with equivalent deployment access may publish source changes, container images, AMIs, or startup inputs that cause approved CVMs to run malicious application code. The attacker's goal is to access, retain, or alter user data during otherwise legitimate processing. The system is expected to resist or expose this through measured deployments, PCR publication, Sigstore/Rekor transparency evidence, and artifact retention.	• Bee Private Compute team engineer	• Attested services and CVM • Confidential VMs • Application services • Transparency service
2	**Replacement of measured persistent artifacts.** A Bee Private Compute engineer may substitute model weights, runtime files, Amazon S3 objects, container image contents, or other persistent artifacts referenced by a production manifest. The attacker's goal is to change model or service behavior while preserving the appearance of normal deployment. The system is expected to resist or expose this through manifest measurement, Amazon ECR/S3 hashes, PCR validation, and cross-team replication.	• Bee Private Compute team engineer	• Confidential VMs • Amazon S3 buckets • Application services • Transparency service

3	**Mis-issuance of Bee Private Compute TLS certificates.** A Bee Private Compute engineer or administrator may attempt to obtain a Bee Private Compute-trusted TLS certificate for infrastructure that should not receive user keys or internal service traffic. The attacker's goal is to impersonate a production endpoint or internal service without running the expected measured workload. The system is expected to resist this through NitroTPM-bound issuance, public-key binding, hostname binding, and client-side attestation checks.	• Bee Private Compute team engineer • Bee Private Compute team administrator	• AWS Private CA • Confidential VMs • HAProxy bridges • Bee Private Compute client
4	**Internal service identity confusion.** A compromised or malicious Bee Private Compute service with a valid Bee Private Compute-issued certificate may attempt to receive traffic intended for a different internal service. The attacker's goal is to confuse service identity across internal bridge connections. The system is expected to resist this through hostname verification, manifest bridge targets, and service identity policy.	• Bee Private Compute team engineer • Bee Private Compute team administrator	• HAProxy bridges • AWS Private CA • Application services • Confidential VMs
5	**Incorrect transparency evidence publication or retention.** A transparency engineer or administrator may publish misleading transparency artifacts, omit expected artifacts, or fail to retain artifacts needed for later audit. The attacker's goal is to weaken the user's ability to verify which measured software and artifacts were approved for production. The system is expected to resist this through	• Transparency team engineer • Transparency team administrator	• Transparency service • Transparency CA • Amazon S3 buckets replica

	Bee Private Compute AWS Private CA identity management, Rekor inclusion, independent artifact replication, and durable retention.		
6	**Account-key enrollment or recovery manipulation.** A registered user may attempt to enroll, restore, replay, or swap account-key material to bypass authentication or confidentiality mechanisms. The attacker's goal is to gain access to protected data or cause protected data to be encrypted under the wrong authority. The system is expected to resist this through key possession proofs, enrollment validation, and user ID binding.	• Application user	• Bee Private Compute client • Mobile application • Application services • Persistent storage
7	**Device-local recovery of mobile-side key material.** An attacker with access to a user's mobile device or backup surfaces may attempt to recover mnemonic-derived keys, account client bundles, or BLE sealed keys. The attacker's goal is to recover credentials that allow future server-processing key release or local decryption. The system is expected to resist this through platform key stores, recovery envelope encryption, and BLE transport key signing.	• Device-local attacker	• User mobile device • Mobile application • Bee Private Compute client • Wearable device
8	**Consent overreach by a paired third-party app or integration.** A third-party integration, or an actor controlling one, may attempt to access user-derived data after consent expiry, after revocation, or outside the intended consent purpose. The attacker's goal is to turn a legitimate pairing or integration into broader or	• External attacker • Application user	• Third-party services • Application services • Persistent storage

	longer-lived access. The system is expected to resist this through signed consent records, expiry checks, and per-CLI encrypted secrets.		
9	**External influence over private processing.** An external attacker may provide inputs, web resources, integration data, or API traffic that is later consumed by private processing or agent workflows. The attacker's goal is to cause unauthorized disclosure through tools, notifications, integrations, generated outputs, or logs. The system is expected to resist this through authentication, consent checks, tool policy, egress restrictions, and prompt/data boundary controls.	• External attacker	• Application services • Third-party services • Bee Private Compute service production AWS account
10	**Sensitive behavior inference from allowed metadata.** An actor with access to logs, metrics, timing information, traffic volume, dashboards, or operational metadata may attempt to infer user behavior without directly accessing plaintext. The attacker's goal is to learn recording activity, integration use, model invocation patterns, or sensitive workflow state. The system is expected to reduce this risk through metadata minimization, aggregation, log redaction, and operational access controls.	• Bee Private Compute team engineer • Bee Private Compute team administrator • External attacker	• Bee Private Compute service production AWS account • HAProxy bridges • Application services • Amazon S3 buckets

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:

- **Documentation review.** We reviewed documentation for the Bee wearable and AMI builder service to familiarize ourselves with the product goals. We also reviewed the threat model.
- **Manual review.** We manually reviewed the client codebase for critical-path code such as early boot and signature verification, focusing on correctness, robustness, and input validation.
- **Automated analysis.** We used automated LLM-based methods to surface issues across the codebase before manually triaging and verifying the issues.

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 review the internals of the AWS Nitro System or the Nitro Hypervisor.
- We did not review the Amazon-internal transparency and AMI signing service beyond the registration interface exercised by the AMI builder.
- We did not review wearable firmware or Bluetooth link-layer security beyond the server-side key derivation (TOB-BEE-9).
- We did not perform physical side-channel or hardware tampering analysis.
- The AMI builder and base images were reviewed in the second half of the engagement and received correspondingly less dynamic testing time.

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Legacy boot path executes code outside measurement context	Data Validation	High
2	Sigstore context is never policy-enforced	Access Controls	Medium
3	Application data disks are mounted by device path without measured or CVM-bound protection	Configuration	High
4	Android RFC3161 timestamp verification can fail open and accept attacker-chosen verification time	Cryptography	Medium
5	AMI Rekor entries are never signature-verified or bound to the transparency team signing certificate	Cryptography	High
6	Go proxy accepts NitroTPM COSE_Sign1 payloads without signature verification	Cryptography	High
7	Shared LLM prefix caches are not isolated across users	Data Exposure	High
8	Shared CVM Redis access is not scoped by service identity	Access Controls	Informational
9	Bee server derives deterministic CTR keys for BLE with insufficient checks	Cryptography	Medium
10	Client attestation failures do not block key release	Access Controls	High
11	Paired-app developer tokens lack scoped permissions	Access Controls	Low

12	Conversation summaries are stored unencrypted	Data Exposure	**Medium**
13	Container environment-variable injection via AWS Secrets Manager bypasses the attestation chain	Configuration	**High**
14	Confidential VM image includes unnecessary packages	Configuration	**Informational**
15	APNS proxy Lambda logs notification content and request payloads	Data Exposure	**Medium**
16	Internal networking policy is overly lax for the Confidential VM tier	Configuration	**Informational**
17	Internal security controls for user data are overly lax	Access Controls	**Medium**
18	Integration APIs receive user data without a sharing-consent check	Access Controls	**Medium**
19	Push notifications routed to other users are sent in plaintext	Data Exposure	**Low**
20	Consent proof is generated server-side, not by the client as documented	Access Controls	**Informational**
21	Ambiguous measurement serialization lets distinct configurations collide	Cryptography	**Informational**

Detailed Findings

1. Legacy boot path executes code outside measurement context

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-BEE-1
Target: `ami-cdk/overlay/base/var/lib/cloud/scripts/per-boot/boot.sh`	
Fix Status: **Resolved**	

Description

System administrators can deploy malicious AMIs without affecting the client-verifiable measurement. In the typical boot flow, the boot script fetches a YAML manifest and launches a secondary script that fetches data using the manifest and extends PCR 14 in the TPM with a hash of the data. However, there is a legacy boot flow that bypasses this secondary script and allows arbitrary code execution (figure 1.1).

```
 24    # Fetch UserData
 25    USER_DATA=$(curl -sf -H "X-aws-ec2-metadata-token: $TOKEN"
http://169.254.169.254/latest/user-data || true)
       [...]
 41    # Detect mode by first line
 42    FIRST_LINE=$(echo "$USER_DATA" | head -1)
 43    echo "UserData first line: $FIRST_LINE"
 44
 45    if [[ "$FIRST_LINE" == "#!/bin/bash"* ]]; then
 46        # OLD MODE: UserData is bash script, execute it
 47        echo "MODE: BASH (legacy)"
 48        echo "$USER_DATA" | bash
 49    elif [[ "$FIRST_LINE" == "#!/bin/cat"* ]]; then
 50        # NEW MODE: UserData is YAML manifest with #!/bin/cat shebang
 51        # cloud-init executes cat (prints and exits 0), no schema validation
 52        echo "MODE: PROPOLIS (yaml)"
```

Figure 1.1: `ami-cdk/overlay/base/var/lib/cloud/scripts/per-boot/boot.sh`

This script runs early in the boot process. It dynamically loads `UserData` from the instance metadata service; this value is set by Amazon operators when launching the Amazon EC2 instance. A malicious Amazon employee can set the user data to a Bash script, and the boot process would automatically execute the code. This malicious script can extend TPM PCRs with the `tpm2_pcrextend` command, leaving all PCRs in the exact same state they would be if the legitimate boot process were followed. Note that the remaining PCRs do not

include the Amazon EC2 user data value. The attacker's script has full control over the instance and can run arbitrary code without client detection.

Exploit Scenario

An Amazon employee launches the legitimate AMI with a malicious bash script in the `UserData` field. The resulting image passes all PCR checks and is indistinguishable from a legitimate instance, but it is running malicious code. The employee uses this access to extract customer data.

Recommendations

Short term, fully remove the legacy boot path from the boot scripts.

Long term, redesign the measurement flow to immediately extend PCR 15 with a hash of the manifest before the data is parsed. `tpm2_pcrextend` may be called again later once the other measurements are available (including Docker images and other variables), but immediately extending the PCR will prevent an attacker from modifying the manifest without modifying the PCR values in the AWS Nitro attestation.

2. Sigstore context is never policy-enforced

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BEE-2

Target: `propolis-proxy/verifier/sigstore/verify.go`, `propolis-proxy/verifier/verifier.go`, `propolis-proxy/verifier/sigstore/ami.go`, `propolis-proxy/cmd/verify/main.go`, `bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt`, `bee-ios/Shared/Crypto/AttestationVerifier.swift`

Fix Status: **Resolved**

Description

All attestation verifiers ignore the `context` field of the Sigstore object, so an attestation for any service and environment (test, production, etc.) can be used to connect to any other service. The `context` object carries fields such as environment, owner, and service name, but none of the runtime verifier entrypoints accepts an expected production policy to compare against. A valid, internally consistent attestation for a staging, canary, development, or wrong-service build satisfies every cryptographic check as long as the NitroTPM PCRs and SPKI binding match that bundle.

In the Go proxy, the runtime entrypoint `Verify` receives only certificate bytes, and `validateCMW` checks structural fields but no service identity (figure 2.1).

```
func (v *Verifier) Verify(certDER []byte) (*VerificationResult, error) {
```

Figure 2.1: The Go runtime verifier entrypoint accepts only certificate bytes, with no expected-service policy parameter. (`propolis-proxy/verifier/verifier.go`)

The Sigstore `Validate` function checks the bundle version, expiry, CA chain, inclusion proofs, timestamp, and PCR map, but it does not compare any `context` field to an expected service identity (figure 2.2).

```
func (v *Verifier) Validate(bundle *Bundle, nitroPCRs map[int][]byte) error {
	if bundle == nil {
		return fmt.Errorf("nil sigstore bundle")
	}
	if bundle.Version != 1 {
```

```
		return fmt.Errorf("unexpected sigstore version: %d", bundle.Version)
	}
	if bundle.Expiry <= time.Now().Unix() {
		return fmt.Errorf("sigstore payload expired at %d", bundle.Expiry)
	}
```

Figure 2.2: `Validate` enforces structural and freshness checks but never compares the context to an expected policy. (`propolis-proxy/verifier/sigstore/verify.go`)

The context is consumed only to reconstruct the AMI digest in `buildAMIDigest`, which requires the context to be non-empty but never compares it to an expected value (figure 2.3).

```
context := map[string]any(bundle.Context)
if len(context) == 0 {
	return nil, fmt.Errorf("missing Sigstore context")
}
```

Figure 2.3: The context is required only to be non-empty for digest reconstruction. (`propolis-proxy/verifier/sigstore/ami.go`)

The capability to enforce context exists in the stand-alone CLI tool, which accepts a `-context key=value` flag and calls `enforceContext`, but this enforcement is absent from the runtime proxy and mobile paths (figure 2.4).

```
if err := enforceContext(requiredContext, contextMap); err != nil {
	fmt.Fprintf(stderr, "attestation.json context check failed: %v\n", err)
```

Figure 2.4: Context enforcement exists only in the stand-alone CLI tool, not the runtime verifier. (`propolis-proxy/cmd/verify/main.go`)

The mobile verifiers behave the same way. Android's `verify(certChain, context)` function receives only an Android resource context, and iOS's `verify(serverTrust:)` function receives only a `SecTrust` object (figure 2.5).

```
static func verify(serverTrust: SecTrust) async throws -> AttestationResult {
```

Figure 2.5: The iOS runtime entrypoint accepts only `SecTrust`, with no expected host, environment, or service policy input. (`bee-ios/Shared/Crypto/AttestationVerifier.swift`)

Because the iOS `AttestationResult` struct omits the Sigstore context, callers cannot enforce the context after verification either (figure 2.6).

```
struct AttestationResult {
    let moduleId: String
    let timestamp: Date
    let coseVerified: Bool
```

```
    let caChainVerified: Bool
    let publicKeyBound: Bool
    let sigstoreVerified: Bool
    let pcrConsistent: Bool
}
```

Figure 2.6: The iOS result omits the Sigstore context, preventing caller-side enforcement after verification. (bee-ios/Shared/Crypto/AttestationVerifier.swift)

Exploit Scenario

A Bee Private Compute engineer creates a malicious CVM and submits it to the DSPE registration endpoint with the "staging" environment flag. The Bee Private Compute attacker issues a production TLS certificate for the service that incorporates the "staging" attestation bundle. End-client verifiers check that the bundle is internally consistent and not expired, but none compares `context.env`, `context.owner`, or `context.serviceName` to the intended production service. The connection is treated as an approved attested target, so a client trusts a non-production enclave as though it were the intended production service.

Later auditors may not detect or inspect the malicious service binary in the DSPE retention buckets or transparency log, since it is mislabeled and seemingly unrelated to core infrastructure.

Recommendations

Short term, extend the runtime verifier entrypoints on the Go proxy, Android, and iOS to accept an expected context policy, and compare the authenticated `context` fields against that policy during verification.

Long term, use mTLS client identity information for application-level access controls, such as by limiting Redis cache access to specific authenticated services.

3. Application data disks are mounted by device path without measured or CVM-bound protection

Severity: **High**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-BEE-3

Target: `bee-cdk-production/lib/vm/ConfidentialASG.ts`, `bee-cdk-production/lib/apps/bee.ts`, `bee-cdk-production/lib/dsl/stack.ts`, `bee-cdk-production/lib/dsl/manifest.ts`, `ami-cdk/overlay/base/opt/propolis/boot/scripts/format_disks.py`

Fix Status: **Resolved**

Description

The confidential-VM services mount `/app` from a configured block-device path such as `/dev/nvme1n1`, format it as plain XFS, and do not encrypt it with a key that exists only inside the CVM. The launch template explicitly defines only the root EBS block device, while application services assume that `/dev/nvme1n1` is suitable ephemeral storage. That assumption is not verified by the mount script and is not bound to the measured state that clients attest before releasing keys.

The launch template declares the root EBS volume but no application data volume mapping or instance-store mapping (figure 3.1).

```
blockDevices: [
    {
        deviceName: '/dev/xvda',
        volume: BlockDeviceVolume.ebs(props.volumeGB, {
            encrypted: true,
            deleteOnTermination: true,
        }),
    },
],
```

Figure 3.1: The launch template defines only the root EBS block device. (bee-cdk-production/lib/vm/ConfidentialASG.ts#L171–L179)

Services pass a configurable device path through to the manifest, commonly mapping `/dev/nvme1n1` to `/app` (figures 3.2 through 3.4).

```
disks: opts.appDisk ? { mount: { [opts.appDisk]: '/app' } } : undefined,
```

Figure 3.2: Application services mount the configured app disk at /app. (bee-cdk-production/lib/apps/bee.ts#L154)

```
for (const [device, mountPoint] of Object.entries(opts.disks.mount)) {
    disk(device, mountPoint);
}
```

Figure 3.3: The manifest records only the device path and mount point. (bee-cdk-production/lib/dsl/stack.ts#L249–L252)

```
export function disk(device: string, mount: string): void {
    if (!current) {
        throw new Error('disk() must be called within manifest() context');
    }
    current.disks.push({ device, mount });
}
```

Figure 3.4: The measured manifest stores a device string rather than an authenticated device identity. (bee-cdk-production/lib/dsl/manifest.ts#L207–L212)

The boot-time disk formatter checks only that the path exists and is a block device. It then formats the device with XFS and mounts it directly. The individual-disk path does not verify instance-store provenance, enforce an expected device model, detect Multi-Attach semantics, or apply dm-crypt or dm-integrity before mounting (figure 3.5).

```
def format_and_mount(device: str, mount: str) -> None:
    """Format a device with XFS and mount it."""
    print(f"Processing disk: {device} -> {mount}")

    if not device_exists(device):
        print(f"  Skipping: device {device} does not exist")
        return
# ...
    print(f"  Formatting {device} with XFS...")
    run(["mkfs.xfs", "-f", device])

    print(f"  Creating mount point {mount}...")
    os.makedirs(mount, exist_ok=True)

    print(f"  Mounting {device} to {mount}...")
    run(["mount", device, mount])
```

Figure 3.5: The individual-disk path formats and mounts whichever block device appears at the configured path. (ami-cdk/overlay/base/opt/propolis/boot/scripts/format_disks.py#L121–L140)

Because the measured state covers the manifest string, not the Amazon EC2 block topology or a CVM-bound disk key, an attacker with control-plane access can move `/app` to snapshot-capable or concurrently attached storage while preserving the same device path

in the manifest. Secrets, Redis snapshots, certificates, temporary files, and other runtime state written to `/app` would then be recoverable through storage APIs without changing the PCR values that clients verify.

Exploit Scenario

An attacker with Amazon EC2 control-plane access changes the Auto Scaling group or launch-template environment so that a snapshottable EBS data volume appears as `/dev/nvme1n1`, or the attacker selects storage with Multi-Attach semantics for the application disk. The manifest still says `/dev/nvme1n1` maps to `/app`, so PCR measurement over the manifest remains unchanged. During boot, the formatter creates a plain XFS filesystem on that device and mounts it. Clients verify the same attested software state and release server-processing keys, while the attacker snapshots or concurrently attaches the `/app` backing volume and reads runtime artifacts such as keystore persistence files, environment secrets, and TLS key bundles.

Recommendations

Short term, update `format_and_mount` so that, before mounting `/app`, the function encrypts it with a per-boot key that is generated inside the CVM and is never written to the mounted device, using plain dm-crypt for confidentiality or detached-header LUKS with dm-integrity when active block tampering is in scope. Ensure detached headers are used for any LUKS encrypted disks. This addresses the issue because relocating `/app` to EBS or another operator-readable block device would expose only ciphertext under a key unavailable through storage APIs.

Long term, bind the expected storage topology, instance type, and block-device configuration into the attested policy, and deny unexpected EBS data volumes or Multi-Attach-capable volumes for CVM application disks through deployment policy. This will further prevent the issue because clients and deployment controls will both reject a CVM whose key-bearing runtime storage differs from the approved configuration.

4. Android RFC 3161 timestamp verification can fail open and accept attacker-chosen verification time

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BEE-4
Target: `bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt`	
Fix Status: **Resolved**	

Description

The Android verifier's RFC 3161 timestamp handling can accept a timestamp response without proving that a trusted timestamp authority (TSA) signed the TSTInfo that supplies `genTime`. Several code paths fail open, and the returned `genTime` is later used as the verification time for the Sigstore certificate chain. An attacker who controls the bundle timestamp bytes can therefore weaken the supply-chain time proof and cause certificate material to be evaluated at an attacker-selected historical time, undermining the freshness guarantee of the attestation verification.

After matching the message imprint against the reconstructed AMI payload bytes, the verifier attempts to locate the CMS `signerInfos`. When `findSignerInfos` returns null, the code skips signature verification entirely and proceeds (figure 4.1).

```
val signerInfosBytes = findSignerInfos(signedDataBytes)
if (signerInfosBytes != null) {
    verifyRFC3161CmsSignature(signerInfosBytes, tstInfoContent, tsaLeaf)
}

// Verify TSA cert chains to TSA root
verifyCertAgainstPinnedRoot(tsaLeaf, tsaRoot, emptyList(), "TSA leaf", genTime)

return genTime
```

Figure 4.1: When `signerInfos` is absent, CMS signature verification is skipped and `genTime` is still returned. (`bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt`)

Inside `verifyRFC3161CmsSignature`, empty `signerInfos`, a short signer info sequence, and missing signature bytes all return successfully rather than throwing an error (figure

4.2). Parser and signature exceptions other than an explicit `AttestationError` are logged as a warning and swallowed, so a malformed token does not fail verification (figure 4.3).

```
val signerInfos = parseDERSet(signerInfoBytes)
if (signerInfos.isEmpty()) return

val siSeq = parseDERSequence(signerInfos[0])
// SignerInfo: version, sid, digestAlgorithm, [signedAttrs], signatureAlgorithm,
signature, [unsignedAttrs]
if (siSeq.size < 5) return
```

Figure 4.2: Empty or short signer info returns successfully instead of failing. (bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt)

```
} catch (e: AttestationError) {
    throw e
} catch (e: Exception) {
    Log.w(TAG, "RFC3161 CMS signature verification warning: ${e.message}")
}
```

Figure 4.3: Non-`AttestationError` exceptions are logged as warnings and swallowed. (bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt)

Even when signed attributes are present, the implementation verifies the signature over the re-encoded attributes but never validates a CMS `messageDigest` signed attribute against the attached TSTInfo content. As a result, the signed attributes are not bound to the TSTInfo that drives `genTime`. The parsed `genTime` is then returned and used as the Sigstore certificate-chain verification time (figure 4.4).

```
verifyTimestampMessageImprint(hashAlgOid, hashedMessage, payloads)
```

Figure 4.4: The returned `genTime` later drives the Sigstore certificate-chain verification time. (bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt)

Exploit Scenario

An attacker builds a timestamp response whose TSTInfo contains an attacker-chosen `genTime` and a message imprint matching the reconstructed AMI payload. The attacker omits `signerInfos`, supplies malformed signer info, omits the signature bytes, or triggers a parser exception. The Android verifier logs a warning or silently returns instead of failing timestamp authentication, and it returns the attacker-chosen `genTime`. The verifier then uses that time to validate the Sigstore signing certificate chain, so expired or not-yet-valid supply-chain certificate material is evaluated at the attacker-selected time and accepted.

Recommendations

Short term, update the Android verifier's RFC 3161 timestamp handling to treat a missing or malformed `signerInfos`, missing signature bytes, and any parser or verification exception as a hard verification failure, and to verify the CMS `messageDigest` signed attribute against the TSTInfo content before trusting `genTime`. This will address the issue because a timestamp token that is not properly signed by the trusted TSA over the attested content will no longer be able to set the verification time.

Long term, add tests that submit timestamp responses with absent, malformed, and unsigned signer information to confirm they are rejected. Do this for all verification libraries to prevent issues like TOB-BEE-6.

5. AMI Rekor entries are never signature-verified or bound to the transparency team signing certificate

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BEE-5

Target: `propolis-proxy/verifier/sigstore/ami.go`, `propolis-proxy/verifier/sigstore/certchain.go`, `propolis-proxy/verifier/sigstore/verify.go`, `bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt`, `bee-ios/Shared/Crypto/AttestationVerifier.swift`

Fix Status: **Resolved**

Description

The Sigstore verifiers never check the AMI Rekor `signature.content` over the digest or bind that signature to the leaf signing certificate or public key that chains to the pinned root. The consequence is that the verifiers prove that an entry for some digest was logged in Rekor, but they do not prove that the trusted AMI signer authorized the PCR digest. A valid Rekor entry produced with attacker-controlled signing material can substitute for trusted AMI authorization. This attack would not be detected by third-party log monitors.

Each Sigstore bundle is of the form in figure 5.1. The issue is that while `amiInclusionProofs` and `signingCertificateEntries` are each independently validated, the validator does not link the two together by checking that the AMI `hashedrekord` entries are signed using the leaf signing certificates in `signingCertificateEntries`.

```
{
  "version": 1,
  "expiry": 1735689600,
  "context": {
    "ami": "ami-abc123",
    "manifest": "..."
  },
  "pcrMap": {
    "PCR4": "a1b2c3...",
    "PCR7": "d4e5f6...",
    "PCR12": "789abc...",
    "PCR14": "def012..."
  },
```

```
  "caChain": "-----BEGIN CERTIFICATE-----\n...",
  "amiInclusionProofs": [
    {
      "logIndex": "12345",
      "logId": {
        "keyId": "base64..."
      },
      "inclusionProof": {
        "logIndex": "12345",
        "rootHash": "base64...",
        "treeSize": "67890",
        "hashes": [
          "base64...",
          "..."
        ],
        "checkpoint": {
          "envelope": "rekor.sigstore.dev\n..."
        }
      },
      "canonicalizedBody": "base64..."
    }
  ],
  "signingCertificateEntries": [
    {
      "certificateBytes": "base64...",
      "inclusionProofs": []
    }
  ],
  "timestamp": [
    "base64-rfc3161-response..."
  ]
}
```

Figure 5.1: Sigstore bundle JSON schema

In the Go proxy, `Verify` orchestrates the Sigstore checks. It runs inclusion proof verification, AMI digest verification, signing certificate proof digest verification, and certificate chain validation, and then returns success. There is no AMI signature verification step (figure 5.2).

```
func (v *Verifier) Verify(data []byte, nitroPCRs map[int][]byte) (bool, error) {
	bundle, err := v.Parse(data)
	if err != nil {
		return false, err
	}
	if err := v.Validate(bundle, nitroPCRs); err != nil {
		return false, err
	}
	if err := v.verifyInclusionProofs(bundle); err != nil {
		return false, err
	}
	if err := v.verifyAMIProofDigests(bundle, nitroPCRs); err != nil {
```

```
                return false, err
        }
        if err := v.verifySigningCertificateProofDigests(bundle); err != nil {
                return false, err
        }
        if err := v.verifyCertificateChain(bundle, nitroPCRs); err != nil {
                return false, err
        }
        return true, nil
}
```

Figure 5.2: The Sigstore `Verify` pipeline contains no AMI signature verification step (`propolis-proxy/verifier/sigstore/verify.go`)

The AMI entry validation that `verifyAMIProofDigests` performs checks only the Rekor `kind`, `apiVersion`, and digest algorithm, and then compares the body digest to the reconstructed digest. The `signature.content` field present in the body is never verified, and it is never compared against the leaf signing certificates.

```
if body.Spec.HashedRekordV002.Data.Algorithm != "SHA2_256" {
        return fmt.Errorf("unexpected AMI Rekor digest algorithm: %s",
body.Spec.HashedRekordV002.Data.Algorithm)
}

gotDigest, err :=
base64.StdEncoding.DecodeString(body.Spec.HashedRekordV002.Data.Digest)
if err != nil {
        return fmt.Errorf("invalid AMI Rekor digest: %w", err)
}
if !bytes.Equal(gotDigest, expectedDigest) {
        return fmt.Errorf("AMI Rekor digest does not match reconstructed
PCR/context/expiry payload")
}
```

Figure 5.3: `verifyAMIEntryDigest` checks only the kind, version, algorithm, and digest equality. (`propolis-proxy/verifier/sigstore/ami.go`)

The Android and iOS verifiers exhibit the same gap, as shown in figures 5.4 and 5.5. They copy the AMI Rekor `signature.content` and `publicKey` fields into the timestamp payload, so the RFC 3161 timestamp covers those bytes, but they do not verify the signature over the digest or bind it to a trusted signing certificate.

```
val payload = JSONObject()
payload.put("digest", body.spec.hashedRekordV002.data.digest)
payload.put("digestAlgorithm", body.spec.hashedRekordV002.data.algorithm)
payload.put("signature", signature.content)
payload.put("publicKey", publicKey.rawBytes)
payload.put("keyDetails", verifier.keyDetails)
```

Figure 5.4: Android copies the AMI Rekor signature and public key into the timestamp payload without verifying the signature. (bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationVerifier.kt)

```
let payload: [String: Any] = [
    "digest": body.spec.hashedRekordV002.data.digest,
    "digestAlgorithm": body.spec.hashedRekordV002.data.algorithm,
    "signature": signature.content,
    "publicKey": publicKey.rawBytes,
    "keyDetails": verifier.keyDetails,
]
```

Figure 5.5: iOS includes the AMI Rekor signature and public key in the timestamp payload without binding them to a trusted signing certificate. (bee-ios/Shared/Crypto/AttestationVerifier.swift)

Exploit Scenario

An attacker controls an AWS Nitro–attested VM and chooses its PCRs. The attacker builds a Sigstore bundle whose `pcrMap` matches those PCRs and whose context, expiry, and version reconstruct to an AMI digest the attacker selects. Using attacker-controlled signing material, the attacker logs a Rekor `hashedrekord` for that digest and assembles valid inclusion and timestamp material, along with any signing certificate that independently chains to the pinned root. The attacker presents the bundle to the Android, iOS, or Go verifier. Each verifier confirms that the digest appears in a logged entry and that the chains are well formed, but none proves that the trusted AMI signer signed the digest, so the attacker-controlled measurements pass as a trusted AMI authorization.

Recommendations

Short term, update the Sigstore verifiers to check the AMI Rekor `signature.content` over the reconstructed digest using a public key confirmed to chain to the pinned signing root, and to reject the bundle if the signature fails or the key does not chain.

Long term, add conformance tests across the Go, Android, and iOS verifiers that reject bundles with valid inclusion proofs but invalid or untrusted AMI signatures. Wherever a signature is validated, add a rejection test that ensures mis-signed objects are rejected.

6. Go proxy accepts NitroTPM COSE_Sign1 payloads without signature verification

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BEE-6

Target: `propolis-proxy/verifier/nitrotpm/verify.go`, `propolis-proxy/verifier/verifier.go`, `propolis-proxy/main.go`, `propolis-proxy/config/haproxy-reverse.cfg`

Fix Status: **Resolved**

Description

The Go `propolis-proxy` verifier trusts the contents of a NitroTPM `COSE_Sign1` attestation document without ever verifying the COSE signature over the payload. A correct verifier must reconstruct the COSE `Sig_structure` (`["Signature1", protected, external_aad, payload]`) and validate the ECDSA signature in the fourth array element before trusting any signed payload field, such as the PCRs, `user_data`, certificate, or CA bundle. The proxy omits this step entirely, so the binding between the AWS Nitro Attestation PKI signing key and the attested measurements is never established. As a result, a malicious AWS service in the VPC that can acquire an mTLS certificate is able to impersonate a trusted service.

`DecodeAttestation` checks only that the input is a four-element `COSE_Sign1` array and decodes the third element as the payload. The first element (the protected header), the second element (the unprotected header), and the fourth element (the signature) are read past or ignored, and no later code in the file consumes them (figure 6.1).

```
coseArr, ok := item.([]interface{})
if !ok {
	return nil, fmt.Errorf("attestation is not a COSE_Sign1 array")
}
if len(coseArr) != 4 {
	return nil, fmt.Errorf("COSE_Sign1 must have 4 elements, got %d",
len(coseArr))
}

payloadBytes, ok := coseArr[2].([]byte)
if !ok {
	return nil, fmt.Errorf("COSE_Sign1 payload is not bytes")
}
```

Figure 6.1: `DecodeAttestation` reads the payload from `coseArr[2]` and never uses the signature in `coseArr[3]`. (`propolis-proxy/verifier/nitrotpm/verify.go`)

`Verify` decodes the attestation and then validates only the certificate chain embedded inside that already-trusted payload through `verifyAttestationChain` (figure 6.2). There is no call that verifies the COSE signature over the payload, so the embedded certificate, PCRs, and `user_data` are accepted on the strength of an unverified payload.

```
func (v *Verifier) Verify(data []byte) (*VerificationResult, error) {
	attestation, err := DecodeAttestation(data)
	if err != nil {
		return nil, fmt.Errorf("failed to decode attestation document: %w",
err)
	}

	if err := v.verifyAttestationChain(attestation); err != nil {
		return nil, fmt.Errorf("attestation chain verification failed: %w",
err)
	}

	return &VerificationResult{
		ModuleID:   attestation.ModuleID,
		InstanceID: extractInstanceID(attestation.ModuleID),
		PCRs:       attestation.PCRs,
		UserData:   attestation.UserData,
	}, nil
}
```

Figure 6.2: `Verify` validates only the embedded certificate chain and never verifies the COSE signature over the payload. (`propolis-proxy/verifier/nitrotpm/verify.go`)

The verification of the embedded certificate chain to the baked-in AWS Nitro Attestation PKI root still occurs, so the attacker must present a payload whose embedded certificate chains to the real AWS Nitro Attestation PKI root. However, the signature-over-payload binding is precisely what is missing: nothing forces the PCRs and `user_data` fields in the payload to be the ones the NitroTPM actually signed. The Android and iOS verifiers do call `verifyCOSESign1` before trusting the payload, so this gap is specific to the Go mesh path.

The top-level verifier consumes the unauthenticated `nitroResult.PCRs` for the Sigstore PCR comparison and the unauthenticated `nitroResult.UserData` for the leaf certificate SPKI binding check (figure 6.3).

```
nitroResult, err := v.verifyNitroTPM(cmw.NitroTPM)
if err != nil {
	return nil, err
}

sigstoreVerified, err := v.verifySigstore(cmw.Sigstore, nitroResult.PCRs)
```

```
if err != nil {
	return nil, err
}

if err := v.verifyLeafCertificateBinding(cert, cmw.PubkeyHash,
nitroResult.UserData); err != nil {
	return nil, err
}
```

Figure 6.3: The top-level verifier uses the unauthenticated PCRs and `user_data` returned from the NitroTPM step. (`propolis-proxy/verifier/verifier.go`)

On verification success, the SPOE handler sets the `nitro_verified` transaction variable to `true`, which is the HAProxy security decision variable. In reverse-proxy mode, HAProxy explicitly disables normal backend certificate verification (`verify none`) and relies on this custom verifier before allowing any HTTP request to reach the backend (figure 6.4).

```
# TLS to backend - verify none because we do custom verification via SPOE
server server1 "${SERVER_DNS}:443" ssl verify none check

# CRITICAL: Check verification BEFORE sending any HTTP request to backend
http-request deny deny_status 502 unless { var(txn.nitro.nitro_verified) -m bool }
```

Figure 6.4: HAProxy disables backend certificate verification and gates traffic on `nitro_verified`. (`propolis-proxy/config/haproxy-reverse.cfg`)

Exploit Scenario

A Bee Private Compute engineer obtains a legitimate, unexpired NitroTPM attestation certificate and CA bundle from an approved service artifact, along with a matching Sigstore bundle. The attacker generates a fresh TLS key pair and uses the mTLS CA to issue a certificate that contains the attestation OID `2.23.133.5.4.9`, with the CMW `pubkey-hash` set to the SHA-256 of the attacker certificate's SPKI. The attacker then edits the embedded NitroTPM COSE payload so that `user_data` is the hex SHA-256 of the attacker SPKI and the PCR fields match the values in the copied Sigstore `pcrMap`, while leaving the original COSE signature in place and invalid.

The attacker serves this certificate from a backend that HAProxy is configured to reach and triggers the proxy with an ordinary request such as `GET /health HTTP/1.1`. The proxy decodes the forged payload, validates only the embedded AWS Nitro Attestation PKI certificate chain, compares Sigstore against the attacker-controlled PCRs, passes the SPKI binding against the attacker-controlled `user_data`, and sets `nitro_verified` to `true`. HAProxy forwards traffic to an endpoint whose measurements were never authenticated by AWS Nitro hardware.

Recommendations

Short term, reconstruct the COSE `Sig_structure` from the protected header and payload, and verify the ECDSA signature in `coseArr[3]` against the public key of the leaf attestation certificate before `Verify` returns any payload field. This will address the issue because the proxy will reject any attestation document whose PCRs and `user_data` were not signed by the AWS Nitro Attestation PKI signing key.

Long term, consider using a dedicated, audited COSE verification library such as go-cose.

7. Shared LLM prefix caches are not isolated across users

Severity: **High**	Difficulty: **Medium**
Type: Data Exposure	Finding ID: TOB-BEE-7
Target: `bee-cdk-production/lib/apps/bee.ts`, `bee-cdk-production/lib/service-*.ts`, `bee-server/sources/modules/ai/inference.ts`, `bee-server/sources/hive/utils/convertSystemPrompt.ts`, `bee-server/sources/routing.ts`	
Fix Status: **Resolved**	

Description

The CVM-hosted LLM inference stack sends multi-user traffic to SGLang backends that keep prefix caches enabled, while the calling application does not add a random per-request or per-conversation cache prefix before private prompt content. An attacker who can send requests to the same serving pool can use latency differences to test whether another user's request recently populated a guessed prompt prefix.

The production CVM application sets `IS_ENCLAVE` to `true` and routes LiteLLM traffic to `http://bridge-llm:6383/`. In enclave mode, the application selects the shared chat inference backends for assistant and product inference. LiteLLM maps these model names to the internal model bridges.

The SGLang service definitions for the five shared chat inference backends do not pass `--disable-radix-cache`. The embedding backends explicitly pass `--disable-radix-cache`, so cache isolation is not applied consistently across the inference stack.

The common inference wrapper sends `model`, sampling settings, `messages`, tool options, and provider options, but it does not send `cache_salt`, `user`, `extra_body`, or another cache-isolation field in either non-streaming or streaming calls. The existing cache-control path does not isolate the SGLang backend cache.

The CVM caller also does not prepend a random cache-isolation prefix before private prompt material. Assistant prompts include the assistant name, user name, current date, reminders, user guidelines, facts, utterances, conversation summaries, and journal entries loaded from the authenticated user's records. Product flows include facts, order history,

user needs, product context, recent conversations, journals, to-dos, and suggestions. These plaintext-derived values can therefore affect a shared backend cache key.

Exploit Scenario

An attacker submits prompts to the same shared chat serving pool and measures response latency for carefully chosen prefixes that match the application's stable prompt templates. The attacker varies guessed user-specific fragments, such as a fact, order-history item, journal phrase, conversation summary, assistant guideline, or product signal. Faster responses identify prefixes that were already cached by another user's recent request, allowing the attacker to confirm whether the victim's decrypted CVM-side prompt context contained the guessed private text.

Recommendations

Short term, disable prefix caching for every privacy-sensitive SGLang chat backend that receives multi-user prompts by passing `--disable-radix-cache`, or prepend a high-entropy per-conversation or per-request cache isolation value before any stable or private prompt material. This will address the issue because requests from different users and conversations will not share the same backend prefix-cache keys.

Long term, add a deployment guardrail that fails when a multi-user SGLang chat service is exposed without either `--disable-radix-cache` or an explicit application-level cache-isolation mechanism. This will prevent the issue from recurring because newly added inference backends will not silently reintroduce shared prefix caches.

References

- SGLang server arguments, `--disable-radix-cache`: https://docs.sglang.io/advanced_features/server_arguments.html
- vLLM prefix caching, `cache_salt`, and timing side-channel mitigation: https://docs.vllm.ai/en/stable/design/prefix_caching/

8. Shared CVM Redis access is not scoped by service identity

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BEE-8
Target: `bee-cdk-production/lib/service-redis-new.ts`, `bee-cdk-production/lib/apps/bee.ts`, `propolis-init/configs/haproxy-expose-redis-mtls.yaml`, `bee-server/sources/apps/encryption/keystore.ts`	
Fix Status: **Resolved**	

Description

The CVM service mesh grants Redis access to clients that pass generic CVM attestation, but it does not authorize Redis clients by service identity, user identity, or Redis key namespace. Therefore, a workload that can reach a shared Redis bridge can read secrets for users and services unrelated to the workload it compromised.

This affects Redis instances that hold live server-processing keys, agent key material, and plaintext user caches. Redis key names include user IDs, but key naming does not prevent an authorized Redis client from scanning or fetching other users' entries.

The Redis service configuration disables persistence, but it does not configure Redis authentication, ACL users, ACL key patterns, or command restrictions (figure 8.1).

```
files: {
    ['/app/redis-' + serviceName + '.conf']: trimIndent(`
        maxmemory ${opts.memoryGB}gb
        maxmemory-policy ${opts.eviction}
        save ""
        appendonly no
        tcp-backlog 511
        timeout 0
        tcp-keepalive 300
    `)
},
```

Figure 8.1: Redis is configured without authentication or ACLs. (bee-cdk-production/lib/service-redis-new.ts#L47–L57)

Application CVMs receive credential-free Redis bridge URLs for shared cache and key-store services (figure 8.2).

```
REDIS_URL: 'redis://bridge-redis:6379',
REDIS_KEYSTORE_URL: 'redis://bridge-redis-keys-a:6380',
REDIS_KEYSTORE_URL_2: 'redis://bridge-redis-keys-b:6380',
REDIS_CACHE_URL: 'redis://bridge-redis-cache:6381',
REDIS_CACHE_LARGE_URL: 'redis://bridge-redis-cache-large:6387',
REDIS_SPEECH_URL: 'redis://bridge-redis-speech:6379',
```

Figure 8.2: Application CVMs use shared Redis bridge URLs without Redis credentials. (bee-cdk-production/lib/apps/bee.ts#L74–L79)

The Redis expose accepts a private-CA client certificate when the generic NitroTPM attestation verification result is true (figure 8.3). The policy does not check that the caller's attested service identity is allowed to access that Redis instance.

```
frontend tcp_front
    bind *:{{ .PublicPort }} ssl crt /etc/ssl/private/combined.pem ca-file
/etc/ssl/private/rootCert.pem verify required ssl-min-ver TLSv1.2 no-tls-tickets
ciphers
ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA25
6:ECDHE-ECDSA-AES128-GCM-SHA256
    filter spoe engine nitro-verify config /var/run/propolis/spoe-nitro-client.cfg
    acl nitro_ok var(txn.nitro.nitro_verified) -m bool
    tcp-request content reject unless nitro_ok
    default_backend app_back
```

Figure 8.3: The Redis expose gates access on generic NitroTPM attestation verification. (propolis-init/configs/haproxy-expose-redis-mtls.yaml#L25–L30)

The keystore writes server-processing keys under predictable user-ID-derived keys and also mirrors derived agent keys (figure 8.4). These names organize the keyspace, but they do not restrict what another Redis client can read after it reaches the service.

```
if (keymaster && redisInstances.length > 0) {
    const value = encodeKeyValue(key, newExpiresAt);
    const redisKey = `${TTL_KEYSPACE_PREFIX}${uid}`;

    for (const r of redisInstances) {
        backoff(async () => {
            await r.setIfExpirationGreater(redisKey, value, newExpiresAt, nowMs());
        });
    }
}

if (keymaster) {
    reportAgentKey(uid, new ContentKeys(key).agentMainKey());
}
```

Figure 8.4: The keystore writes live key material under user-ID-derived Redis keys. (bee-server/sources/apps/encryption/keystore.ts#L291–L304)

Exploit Scenario

An attacker compromises a workload running in an attested CVM that has access to a Redis bridge. The attacker connects to a shared Redis endpoint such as a cache, keystore, or agent-keystore bridge, and the mesh accepts the connection because the CVM passes generic NitroTPM attestation verification. The attacker then uses Redis commands such as `SCAN` and `GET` to read entries including `keyv2:<uid>` and `agent:keyv1:<uid>` for unrelated users. The attacker uses recovered server-processing or agent keys to decrypt protected content outside the request and service context that originally released those keys.

Recommendations

Short term, enforce access controls to Redis instances based on mTLS certificate identities. Bind each service identity to attested CVM manifest metadata and Sigstore context, and require attestation verifiers to check that the authenticated manifest identity matches the certificate identity, such as the SAN, CN, or another explicit certificate field.

Long term, generate service identities, certificate identities, mesh policy, Redis ACLs, and auditor-facing identity inventories from the CVM deployment manifest. The inventory should let auditors quickly identify every CVM issued with a particular identity and compare that identity against the attested metadata that verifiers enforce.

9. Bee Private Compute server derives deterministic CTR keys for BLE with insufficient checks

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BEE-9
Target: `bee-server/sources/apps/api/routes/ble.ts`, `bee-server/sources/apps/api/startApi.ts`	
Fix Status: **Resolved**	

Description

User phones communicate with Bee devices over BLE, and sensitive audio data from the device is encrypted using a per-device AES-CTR key known to both the phone and Bee device. This key is deterministically derived from the Bee device's hardware identifier using a secret key on the server. There is no check that the client querying the server is the authentic owner of a particular hardware device, so an attacker with access to hardware identifier and encrypted audio packets can recover the decrypted audio data.

The POST `/ble/secure-key` route derives a per-device AES-CTR key as a deterministic HMAC of the caller-supplied `device_id` and a global secret, returns the raw 16-byte key to the caller, and never checks that the authenticated principal owns or has paired the requested device. Any authenticated user who knows or guesses a wearable's 8-byte device ID can therefore obtain the BLE audio key for that device.

The route accepts a JSON body whose only validation is that `device_id` is 16 hexadecimal characters, as shown in Figure 9.1.

```
10    app.post('/ble/secure-key', {
11        schema: {
12            body: z.object({
13                // 8-byte device_id in hex (16 hex chars)
14                device_id: z.string().regex(/^[0-9a-fA-F]{16}$/)
15            })
16        }
17    }, async (request, reply) => {
```

Figure 3.1: The route validates only the hex shape of `device_id`. (`bee-server/sources/apps/api/routes/ble.ts#L10–L17`)

The route then deterministically derives the CTR key directly from the attacker-controlled device ID bytes and the global `BLE_CTR_KEY_SECRET`. This CTR key is eventually returned to the caller.

Multiple factors affect the difficulty of exploiting this issue. It is unknown how difficult it is for an attacker to obtain the device ID. If the 8 bytes have a predictable prefix, an attacker may be able to learn the encryption keys for every device. It is also unclear whether the device broadcasts the identifier frequently, or if an attacker needs to interact with the device to obtain the identifier.

In the current design, there is a separation between the AES-encrypted BLE packets and the `BLE_CTR_KEY_SECRET` value known to Amazon. An attacker can target these two elements separately. They can target a specific user's BLE connection, storing the encrypted audio data for later decryption. Separately, a compromised Amazon employee could reveal the root secret, enabling a retroactive wiretap of unlimited duration on the targeted Bee Private Compute customer.

Exploit Scenario

A Bee Private Compute customer downloads a malicious app and grants access to Bluetooth devices. The app queries the Bee Private Compute device for its hardware identifier and records the AES-CTR encrypted audio packets. Separately, the malicious app creator queries the Bee Private Compute service from their own account, spoofing the device ID in the call to `/ble/secure-key`. The attacker learns the AES-CTR key, decrypts the audio packets, and has access to the customer's sensitive data.

Recommendations

Short term, update the `/ble/secure-key` route so that it consults the table of registered devices and refuses to release the key if the device is registered to a different user.

Long term, negotiate the BLE transport key between the app on the user's mobile device and the Bee device itself, so that the Bee Private Compute server never holds or can derive the transport key. Depending on product requirements, there are stronger cryptographic designs for creating a secure channel between the Bee device and phone, including frequently rotated ephemeral keys or provisioned device certificates.

10. Client attestation failures do not block key release

Severity: **High**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BEE-10

Target: `bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationGate.kt`, `bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestingTrustManager.kt`, `bee-ios/Shared/Crypto/AttestationGate.swift`, `bee-ios/Shared/CertificatePinning.swift`, `bee-android/modules/bee-background/android/src/main/java/com/bee/background/notifications/EncryptedNotificationDecryptor.kt`

Fix Status: **Resolved**

Description

The mobile clients release account key material even when attestation verification fails. Android hard-codes the attestation gate to non-enforcing mode (figure 10.1), and iOS enables blocking enforcement only for test users. In the non-enforcing paths, attestation runs asynchronously after ordinary TLS succeeds, and failures are logged without canceling the connection.

```
/** Hardcoded enforcement mode. Change here to switch behavior globally. */
val mode: EnforcementMode = EnforcementMode.NON_ENFORCING

fun shouldEnforce(context: Context): Boolean {
    return when (mode) {
        EnforcementMode.ALL_USERS -> true
        EnforcementMode.NON_ENFORCING -> false
        EnforcementMode.TEST_USERS -> prefs(context).getBoolean(KEY_IS_TEST_USER,
false)
    }
}
```

Figure 10.1: Android disables blocking attestation for all users. (bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestationGate.kt#L40–L64)

The Android trust manager contains a fail-closed path, but that path is reached only when `AttestationGate.shouldEnforce` returns true (figure 10.2). In the deployed

configuration, the non-enforcing branch starts a detached thread and catches verification exceptions without throwing a `CertificateException` into the TLS handshake (figure 10.3).

```
val enforcing = AttestationGate.shouldEnforce(context)

if (enforcing) {
    runAttestationEnforcing(chain, url)
} else {
    runAttestationAsync(chain, url)
}
```

Figure 10.2: The trust manager selects the asynchronous log-only path when enforcement is disabled.
(bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestingTrustManager.kt#L127–L138)

```
try {
    val result = AttestationVerifier.verify(chain, context)
    val elapsedMs = System.currentTimeMillis() - startMs
    AttestationGate.clearFailureRateLimit(url)
    Log.i(TAG, "Attestation PASSED for $url " +
        "[source=$sourceTag, module=${result.moduleId}, elapsed=${elapsedMs}ms, " +
        "cose=${result.coseVerified}, caChain=${result.caChainVerified}, " +
        "pubKeyBound=${result.publicKeyBound}, sigstore=${result.sigstoreVerified},
" +
        "pcrConsistent=${result.pcrConsistent}]")
} catch (e: Exception) {
    val elapsedMs = System.currentTimeMillis() - startMs
    val errorMsg = e.message ?: "unknown error"
    Log.e(TAG, "Attestation FAILED for $url [source=$sourceTag,
elapsed=${elapsedMs}ms]: $errorMsg")
    AttestationGate.logFailure(url, errorMsg, false, sourceTag)
}
```

Figure 10.3: Android non-enforcing attestation catches failures without blocking the handshake.
(bee-android/modules/bee-background/android/src/main/java/com/bee/background/attestation/AttestingTrustManager.kt#L165–L179)

The iOS flow has the same shape. `AuthManager` sets enforcement from the user's `isTest` flag, as shown in figure 10.4, while the certificate delegate returns `.useCredential` after starting detached verification for non-test users, as shown in figure 10.5.

```
// Enforce attestation for test users
let isTest = currentUser.isTest == true
Task { await AttestationGate.shared.setEnforcing(isTest) }
```

Figure 10.4: iOS enables blocking attestation only for test users.
(bee-ios/Owl/Services/AuthManager.swift#L32–L34)

```
} else {
    [...]
    Task.detached(priority: .userInitiated) {
        do {
            let att = try await AttestationVerifier.verify(serverTrust: serverTrust)
            LoggingDebug.trace("✅ Attestation: \(att.moduleId)")
        } catch {
            LoggingDebug.trace("❌ Attestation: \(error)")
            await AttestationGate.shared.logFailure(host: host, error: "\(error)",
enforcing: false)
        }
    }
}
return (.useCredential, URLCredential(trust: serverTrust))
```

Figure 10.5: iOS returns the TLS credential even when attestation runs in a detached task. (bee-ios/Shared/CertificatePinning.swift#L123–L135)

A native Android recovery path also constructs a plain `OkHttpClient` rather than using the attesting trust-manager wiring (figure 10.6). That path can load or recover the account authentication state for notification decryption without the same attestation gate that protects the normal client.

```
val recoveryClient = OkHttpClient.Builder()
    .connectTimeout(10, TimeUnit.SECONDS)
    .writeTimeout(10, TimeUnit.SECONDS)
    .readTimeout(10, TimeUnit.SECONDS)
    .build()

val recovery = BackgroundSecureAuth.loadOrRecover(
    context = context,
    client = recoveryClient,
    authEndpoint = defaultAuthEndpoint(context)
)
```

Figure 10.6: A recovery path uses a plain `OkHttpClient` without attestation wiring. (bee-android/modules/bee-background/android/src/main/java/com/bee/background/notifications/EncryptedNotificationDecryptor.kt#L149–L159)

Exploit Scenario

An attacker with control over endpoint routing or a valid TLS certificate for a non-attested backend steers a production client to the attacker's service. Ordinary TLS validation succeeds, but the attestation bundle is absent, malformed, or for the wrong measurement. Because Android and iOS run attestation in log-only mode for ordinary users, the connection remains open, and normal authentication code sends `x-bee-secret`, `x-bee-proof`, device-backed ephemeral tokens, or recovered key material across that channel. The attacker obtains the user's server-processing key and account proof and uses them to decrypt protected content or to refresh backend key availability.

Recommendations

Short term, update the Android and iOS clients to enforce fail-closed attestation for every production user and every network path that can release or recover account key material, and to reject key-bearing requests unless they use an attesting client configured for the intended enclave hosts. This will address the issue because attestation failures will terminate the TLS handshake before key material leaves the device.

Long term, centralize account-key release behind a single attestation-enforcing transport abstraction on each platform and add regression tests that simulate missing, malformed, and wrong-measurement attestations for every key-bearing flow. This will prevent the issue from recurring because new network paths will inherit the same fail-closed policy instead of deciding enforcement locally.

11. Paired-app developer tokens lack scoped permissions

Severity: **Low**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BEE-11
Target: `bee-server/sources/apps/apps/appPairingAccept.ts`, `bee-server/sources/apps/encryption/developerTokens.ts`, `bee-server/sources/apps/api/auth.ts`, `bee-server/sources/apps/api/developer/index.ts`, `bee-server/sources/apps/api/developer/devConversations.ts`	
Fix Status: **Unresolved**	

Description
The documentation states that third-party app pairing tokens are “scoped” and detail “requested permissions.” This capability-based access is not present in the code, and all developer API endpoints share the same permission level.

This means that even the simplest apps have full access to the user’s information. This is acknowledged in the apps, which state that “[third-party apps] will be able to access all your Bee data.”

Exploit Scenario
A user wishes to use a third-party app that uses the name information at the `/v1/me` endpoint to construct a personalized message. Unbeknownst to them, the app calls `/v1/conversations/batch` to bulk-download all recorded conversations. The user must decide between not using third-party apps and risking a total loss of privacy.

Recommendations
Short term, ensure public-facing documentation does not claim the existence of a scoped permission system.

Long term, implement the scoped permission system.

12. Conversation summaries are stored unencrypted in the search index

Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-BEE-12
Target: `bee-server/sources/apps/search/denseIndexingWorker.ts`	
Fix Status: **Partially Resolved**	

Description

The documentation asserts that "[a]ll user content is encrypted using symmetric key cryptography" using client-derived keys to ensure that customers retain control over how their data is used. This is false for the conversation summaries and vector embeddings stored in the server-side search index database.

The server-side search index is a database technology for storing dense vectors and performing nearest-neighbor searches. This is used to store embedding representations of conversation topics, and nearest-neighbor search retrieves conversations with similar semantic meaning. There is also a separate PostgreSQL database that stores an encrypted textual summary of a user's conversations. Periodically, the search index collection is synchronized with the PostgreSQL database. The basic steps are detailed in a comment in `denseIndexingWorker.ts`, shown in figure 12.1.

```
 45    /**
 46    * Two-Stage Indexing Worker
 47    *
 48    * This worker handles both stages of search indexing for conversations:
 49    * - Stage 1 (BM25): Creates collection and indexes text for keyword search
 50    * - Stage 2 (Dense): Generates embeddings for hybrid semantic search
 51    *
 52    * Flow:
 53    * 1. Run BM25 indexing (Stage 1) - enables immediate keyword search
 54    * 2. Fetch recent conversations from PostgreSQL
 55    * 3. Decrypt summaries
 56    * 4. Get embeddings (from cache or compute new ones)
 57    * 5. Upsert into [the search index] with real embeddings (replacing zero
vectors from Stage 1)
 58    * 6. Track dense indexing completion for adaptive search
 59    */
```

Figure 12.1: Description of the reindexing process (bee-server/sources/apps/search/denseIndexingWorker.ts#L45–L59)

Note that there is no mention of re-encrypting summaries before upserting, nor could re-encryption be identified in the code.

Both the `SUMMARY` and `SUMMARY_DENSE` fields in the search index database, which correspond to the BM25 and dense vector embedding search, contain sensitive information about the user's conversations. Normally, revocation of access to this sensitive information is achieved by expiring the user's symmetric key. However, the search index database is unencrypted and not protected by these cryptographic access controls.

Exploit Scenario

A user has a sensitive conversation about their financial or medical status, which is captured by the Bee device and uploaded to the Amazon server. An unencrypted summary of the conversation is stored in the search index database. After 7 days, the user's server-processing key expires and cannot be used to decrypt PostgreSQL entries. However, the search index database remains available.

Recommendations

Short term, disable the search index or revise the documentation to reflect that summaries of a user's conversations are not encrypted at rest. If the search index database remains enabled, ensure the cache retention time is small, ensure that all operations that expire the user's symmetric key immediately expire their entries in the search index database, and ensure that all paths that query the search index database check the caller has the cryptographic permissions to access the content.

Long term, encrypt the summary text and vectors in the search index so that retrieval requires the caller to demonstrate possession of the relevant key.

13. Risk of container environment-variable injection via AWS Secrets Manager that can bypass the attestation chain

Severity: **High**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-BEE-13
Target: `propolis-init/secrets.go`, `propolis-init/docker.go`, `propolis-init/tpm.go`, `bee-cdk-production/lib/env/secrets.ts`, `bee-cdk-production/lib/apps/bee.ts`, `bee-cdk-production/lib/agents-main.ts`	
Fix Status: **Resolved**	

Description

The code that fetches environment variables from the AWS Secrets Manager does not handle special characters properly, and a maliciously inserted newline in the environment variable value can add a new KEY=VALUE pair to the environment of Docker containers without affecting the attestation.

The design document states that any change to the runtime configuration of a Confidential VM is reflected in the attestation. The implementation of the prepare phase contradicts this. The manifest YAML hashed into PCR14 commits only to the names of the AWS Secrets Manager secrets the VM will read, such as `bee/production`, `inngest/production`, and `agents/inference`, not to the values resolved against those names at boot. The resolved values are fetched after the measurement is taken, written to `/app/.env`, and injected wholesale into every workload container's process environment. Any principal that holds `secretsmanager:PutSecretValue` on a listed secret ARN can add arbitrary KEY=VALUE lines that will be present in every workload container's environment, without changing PCR14, the Sigstore inclusion proof, or the certificate the VM presents at TLS handshake time. An external party verifying the attestation chain receives no signal that the environment seen by the workload differs from the environment seen by an auditor. The post-measurement secret resolution appears in figure 13.1.

```
result, err := smClient.GetSecretValue(ctx, &secretsmanager.GetSecretValueInput{
    SecretId: &secretID,
})
if err != nil {
    return fmt.Errorf("failed to fetch secret %s: %w", secretID, err)
}

if result.SecretString == nil {
```

```
    return fmt.Errorf("secret %s has no SecretString value", secretID)
}

// Parse JSON secret into KEY=VALUE format
envVars, err := parseSecretToEnv(*result.SecretString)
```

Figure 13.1: `GetSecretValue` results are parsed into `KEY=VALUE` `.env` lines after the manifest is measured. (`propolis-init/secrets.go#L44–L56`)

The same .env file is injected into every workload container, with no per-container filter, as shown in figure 13.2.

```
envFromFile, err := loadEnvFile(appEnvFilePath)
if err != nil {
    return fmt.Errorf("failed to read %s: %w", appEnvFilePath, err)
}
log.Printf("Container env source: name=%s env_file=%s entries=%d explicit_env=%d",
name, appEnvFilePath, len(envFromFile), len(cfg.Env))
containerConfig.Env = append(containerConfig.Env, envFromFile...)
```

Figure 13.2: Every workload container inherits the full `/app/.env` file. (`propolis-init/docker.go#L213–L218`)

The measurement chain that produces PCR14 commits to the manifest YAML, the container image digests, and Amazon S3 content hashes; secret values are not inputs. The Sigstore inclusion proof verified at TLS handshake time was generated at AMI publication and cannot reflect post-publication secret rotation. The certificate presented by the VM is bound to the measured boot state, not to the secret resolution.

Exploit Scenario

An attacker compromises a principal that holds `secretsmanager:PutSecretValue` on `bee/production`, such as through a phishing campaign against an SRE, a vulnerability in a CI worker that holds the deployment role, or a federated identity-provider misconfiguration. The attacker calls `PutSecretValue` once to add an attacker-controlled value for an environment variable the workload consumes, such as `DATABASE_URL`, an `AWS_ENDPOINT_URL_*` override, a JWT or encryption key, or an OAuth client secret. Every Confidential VM launched after the call, across scale-outs, deployments, and AMI rolls, picks up the malicious value during the prepare phase; PCR14 is unchanged, the Sigstore inclusion proof is unchanged, and the certificate the VM presents at handshake is unchanged. The workload executes against the attacker-chosen secret value, leaking user data or accepting forged artifacts as authoritative, while every external verifier of the attestation chain continues to see a green light.

Recommendations

Short term, limit environment keys and values to an allowlist of characters. In addition, separate the environment variables whose value must remain secret from those that can be included in the attestation. Attest to all environment variable keys and public values,

and understand all the ways private environment variables are used. In addition, pin secret versions in the manifest. Extend the manifest schema to carry a `VersionId`, or a `VersionStage` together with a monotonic version-ID check, for every entry in the secrets array, include that field in the measurement chain, and refuse to start if the version resolved at boot does not match the manifest. This will address the issue because the secret version will become part of the attested boot identity and a `PutSecretValue` call will rotate the version away from the one the manifest commits to.

Long term, separate the trust placement of secret material from the trust placement of attestation. Move signing keys, encryption keys, and other secrets whose substitution defeats the confidentiality and integrity properties of the system out of Secrets Manager values and into KMS-resident keys whose key policy is bound to the attested VM identity. The workload references key ARNs in the manifest rather than key material in a mutable secret value, and KMS controls whether the workload can use the key at all. This will further prevent the issue because a `PutSecretValue` call will no longer reach material that the workload trusts as authoritative.

References

- AWS Secrets Manager: Version staging labels
- AWS KMS: Key policy conditions for attested workloads

14. Confidential VM image includes unnecessary packages

Severity: **Informational**	Difficulty: **Undetermined**
Type: Configuration	Finding ID: TOB-BEE-14
Target: `ami-cdk/ami-config.json`	
Fix Status: **Resolved**	

Description

The design document represents the Confidential VM image as a stripped-down Linux distribution containing only the hardened kernel, networking, and GPU drivers necessary for operation. The AMI build configuration in scope describes an Amazon Linux base image extended with a list of additional packages rather than reduced from one. The discrepancy enlarges the attested base image, increases the maintenance surface of the trusted boot path, and weakens a property that the design document presents as load-bearing. The packages added on top of the base appear in figure 14.1.

```
"dependencies": [
  {
    "packageName": "aws-cfn-bootstrap"
  },
  {
    "packageName": "nvme-cli"
  },
  {
    "packageName": "vim-common"
  },
  {
    "packageName": "nfs-utils"
  },
  {
    "packageName": "amazon-efs-utils"
  },
  {
    "packageName": "nvidia-fabricmanager"
  },
  {
    "packageName": "chronicle"
  }
],
```

Figure 14.1: Packages added to the Confidential VM base image, with no corresponding stripping step (`ami-cdk/ami-config.json#L3–L25`)

Some of these packages are necessary, such as to enable inference, but others, like `vim-common`, do not need to be on production deployments that no engineer has access to.

The “stripped down” AMI is constructed according to the YAML files in `AmbientComputeAmiBuilderCDK`.

Exploit Scenario

There is a vulnerability in common Linux packages that are not necessary for Confidential VM operation; reviewers conclude the package must not exist on the machine even though it does.

Recommendations

Short term, remove unnecessary packages like `vim-common`.

Long term, ensure that the remaining packages are the minimal packages necessary to achieve the desired functionality.

15. APNs proxy Lambda logs notification content and request payloads

Severity: **Medium**	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-BEE-15
Target: `bee-cdk-production/lib/lambdas/apns-proxy/apns-proxy-handler.js`	
Fix Status: **Resolved**	

Description

The Lambda function that proxies push notifications to the APNs writes the request body, a prefix of the body, and per-device tokens to Amazon CloudWatch Logs on every invocation. The request body contains the notification title, body, and payload that the Bee Private Compute server intends to deliver to a user's device. The logging is not gated by a debug flag, is not redacted, and is not encrypted. Any principal that holds read access to the Amazon CloudWatch Logs group, and any downstream system the log group is exported to, reads the same content the user's device receives, without holding any client-side encryption material. The unconditional content logging appears in figure 15.1.

```
exports.handler = async (event) => {
  console.log('APNS Batch Proxy request:', {
    path: event.path,
    method: event.httpMethod,
    queryParams: event.queryStringParameters,
    headers: Object.keys(event.headers || {}),
    bodyLength: event.body ? event.body.length : 0,
  });

  // ...

  try {
    // Debug: Log what we received
    console.log('Event body received:', event.body);
    console.log('Event body type:', typeof event.body);
    console.log('Is base64 encoded:', event.isBase64Encoded);
    console.log('First 100 chars of body:', event.body ? event.body.substring(0,
100) : 'empty');
```

Figure 15.1: The APNs proxy logs the full request body and a prefix on every invocation. (bee-cdk-production/lib/lambdas/apns-proxy/apns-proxy-handler.js#L215–L235)

A subsequent log entry emits the decoded body for Base64-encoded inputs, and the per-device call site emits a device-token prefix and status code on every send.

```
if (event.isBase64Encoded && bodyToParse) {
  bodyToParse = Buffer.from(bodyToParse, 'base64').toString('utf8');
  console.log('Decoded base64 body:', bodyToParse);
}
```

Figure 15.3: The decoded request body is also logged when the input is Base64-encoded. (bee-cdk-production/lib/lambdas/apns-proxy/apns-proxy-handler.js#L240–L243)

Exploit Scenario

An attacker obtains any principal with read access to the APNs proxy log group, such as a compromised on-call account, a CloudWatch reader role granted to a development environment, a third-party log-forwarder consumer, or an AWS support employee acting on a misrouted ticket. The attacker reads every notification routed through the proxy, organized chronologically by device token, correlates the content with the recipient by matching tokens to user records, and exfiltrates a stream of user-facing data that the rest of the system is designed to protect under encryption.

Recommendations

Short term, remove the `console.log` calls that emit `event.body`, the decoded body, and the body prefix from the Lambda handler, and restrict the remaining structured log entries to non-content fields such as path, method, query keys, header names, and body length. This will address the issue because the log group will no longer contain user-visible notification content.

Long term, introduce a lint or CI rule that fails the build when a request handler in this Lambda directory references `event.body`, `event.headers`, or any decoded request payload inside a logging call. This will prevent future modifications from reintroducing the same exposure without an explicit suppression.

16. Internal networking policy is overly lax for the Confidential VM tier

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Configuration	Finding ID: TOB-BEE-16
Target: `bee-cdk-production/lib/agents-main.ts`, `bee-cdk-production/lib/stacks/stack-networking-agents.ts`, `bee-cdk-production/lib/vm/operations.ts`	
Fix Status: **Resolved**	

Description

The Confidential VM tier maintains a closed-by-default networking posture: services land in isolated subnets, security groups disallow outbound traffic by default, and AWS-service access is provided through interface and gateway endpoints rather than wildcard egress. Four call sites and helpers depart from this posture in ways that consume defense-in-depth margin without an observable need. The `agents` service in the agents VPC opens unbounded IPv4 egress on ports 53, 80, and 443 even though every dependency it declares is reachable through an internal bridge or a VPC endpoint. The agents VPC exposes the SSM, SSM-messages, and EC2-messages endpoints that the main Bee Private Compute VPC explicitly omits. The `allowDNS` and `allowPublicDNS` helpers grant outbound DNS to every host in the VPC or to the entire IPv4 internet rather than to a defined set of resolvers. An `allowPublicHTTP` helper grants outbound plaintext HTTP that no in-scope production service requires. Each item is informational-severity on its own; together, they enlarge the agents-tier attack surface beyond what the design document presents.

The `agents` service requests wildcard internet egress while declaring only internal bridges, as shown in figure 16.1.

```
service({
            name: 'agents',
            image: 'server/agent:v243',
            metricsPort: 3000,
            allowLogging: true,
            allowInternet: true,
            [REDACTED]
            instanceType: InstanceType.of(InstanceClass.R6ID, InstanceSize.LARGE),
            bridges: {
                'inngest-api': { host: 'agents-inngest.bee.internal', port: 8288 },
                'inngest-connect': { host: 'agents-inngest.bee.internal', port: 8289
},
                'agent-keystore-a': { host: 'agent-keystore-a.bee.internal', port:
```

```
6379 },
                'agent-keystore-b': { host: 'agent-keystore-b.bee.internal', port:
6379 },
                'agents-proxy': { host: 'agents-proxy.bee.internal', port: 8080 },
            },
```

Figure 16.1: The agents service opens wildcard IPv4 egress despite reaching every declared dependency through internal bridges. (bee-cdk-production/lib/agents-main.ts#L9–L23)

The agents VPC exposes the SSM endpoint trio that the main Bee Private Compute VPC removes for defense-in-depth, as shown in figure 16.2.

```
// Remote Access
        this.exposeEndpoint(InterfaceVpcEndpointAwsService.SSM);
        this.exposeEndpoint(InterfaceVpcEndpointAwsService.SSM_MESSAGES);
        this.exposeEndpoint(InterfaceVpcEndpointAwsService.EC2_MESSAGES);
```

Figure 16.2: The agents VPC's networking stack enables the SSM endpoint trio. (bee-cdk-production/lib/stacks/stack-networking-agents.ts#L36–L39)

The DNS and HTTP egress helpers admit traffic to broad peer sets, as shown in figure 16.3.

```
export const allowDNS = (sg: SecurityGroup, vpc: IVpc) => {
    sg.addEgressRule(Peer.ipv4(vpc.vpcCidrBlock), Port.tcp(53), 'Allow outgoing DNS
traffic (TCP)');
    sg.addEgressRule(Peer.ipv4(vpc.vpcCidrBlock), Port.udp(53), 'Allow outgoing DNS
traffic (UDP)');
};

// ...

export const allowPublicHTTP = (sg: SecurityGroup) => {
    sg.addEgressRule(Peer.anyIpv4(), Port.tcp(80), 'Allow outgoing HTTP traffic for
any HTTP');
};

export const allowPublicDNS = (sg: SecurityGroup) => {
    sg.addEgressRule(Peer.anyIpv4(), Port.tcp(53), 'Allow outgoing Public DNS
traffic (TCP)');
    sg.addEgressRule(Peer.anyIpv4(), Port.udp(53), 'Allow outgoing Public DNS
traffic (UDP)');
};
```

Figure 16.3: The DNS and HTTP egress helpers admit traffic to broad peer sets. (bee-cdk-production/lib/vm/operations.ts#L214–L242)

If the goal is to ultimately provide agents with the ability to search the web, allowing services to freely access the web significantly increases the attack surface for exfiltration. In addition, note that agents are not currently included, limiting the extent of this issue.

Exploit Scenario

An attacker reaches code execution inside the `agents` service, such as through a vulnerability in the agent runtime's interpretation of an LLM-generated tool call. The attacker finds outbound IPv4 reachable on ports 53, 80, and 443. The attacker tunnels exfiltrated user content over UDP/53 to an attacker-controlled resolver, a path the closed-by-default posture of the main Bee Private Compute VPC would have blocked, and then uses the SSM endpoint trio reachable from the agents VPC as a hint to enumerate operator paths into the same subnet rather than receiving the empty result that the main Bee Private Compute VPC's posture would return. The attestation chain continues to verify the agents service's code identity, but the design document's representation of the Confidential VM tier as closed-by-default does not match the agents VPC's effective posture.

Recommendations

Short term, replace `allowInternet: true` on the `agents` service with the targeted egress helpers required by its declared dependencies, including the inngest, keystore, and proxy bridges and the existing VPC endpoint, and remove the SSM, SSM-messages, and EC2-messages endpoint declarations from the agents VPC networking stack. Narrow `allowDNS` to the AmazonProvidedDNS resolver at the VPC base address plus two, remove `allowPublicHTTP` from the helper module, and remove `allowPublicDNS` or restrict it to a defined set of provider resolver addresses. This will address the issue because the agents tier will then match the closed-by-default posture the main Bee Private Compute VPC already enforces.

Long term, expose, in the CDK, the union of effective egress reachable by every Confidential VM service as a derived value, and assert in synth-time tests that no service in the Confidential VM tier reaches more peer sets than its declared dependencies require. This will prevent the issue from recurring because future service additions will not be able to quietly enlarge the tier's effective egress without an explicit, reviewed change.

17. Internal security controls for user data are overly lax

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BEE-17
Target: `bee-server/sources/apps/cache/populateConversationCache.ts`, `bee-server/sources/apps/cache/queries/queryConversationsById.ts`, `bee-server/prisma/schema.prisma`, `bee-cdk-production/lib/standalone-services/outlook-sync-stack.ts`, `bee-cdk-production/lib/standalone-services/integrations-ecs-construct.ts`, `bee-cdk-production/lib/standalone-services/integrations-elasticache-construct.ts`	
Fix Status: **Partially Resolved**	

Description

Several internal security controls around the storage and movement of user data are weaker than the design document represents the system as providing, and together they constitute a defense-in-depth regression on the user-data path. The user-data store applies column-level encryption to conversations, chat content, and many other records via `sec_*` columns, but the same encryption is not applied to `Email` and `CalendarEvent` rows. Note that the lack of encryption here may reflect an accepted design decision rather than an oversight and should be confirmed with the team before remediation. The cache tier that fronts the user-data store holds decrypted conversation content in plaintext after the records are decrypted from the at-rest store. The email-ingestion ElastiCache backing the Gmail and Outlook ingestors runs without transit or at-rest encryption. The email-ingestion containers hold both the RDS connection and `BEE_ENCRYPTION_KEY` simultaneously, collapsing the separation between data access and key access that the rest of the architecture maintains.

The conversation-cache loop reads decrypted records from the user-data store and writes them straight to Redis, as shown in figure 17.1.

```
for (let i = 0; i < missingIds.length; i += BATCH_SIZE) {
        const batchIds = missingIds.slice(i, i + BATCH_SIZE);
        const endBatch = startPipelineStep(ctx, PIPELINE, 'batch');

        const conversations = await queryConversationsById(ctx, batchIds);
```

```
        // Cache batch of conversations (updates both keys and tracking SET)
        const endRedisWrite = startPipelineStep(ctx, PIPELINE, 'redis-write');
        await setConversations(ctx.uid, conversations);
        endRedisWrite();
```

Figure 17.1: Decrypted conversations are written directly to Redis. (bee-server/sources/apps/cache/populateConversationCache.ts#L136–L145)

The decryption step that produces the cached objects appears in figure 17.2.

```
if (decryptableUtterances.length > 0 && Object.keys(conversationKeys).length > 0) {
        const endDecrypt = startPipelineStep(ctx, PIPELINE, 'decrypt');
        try {
            const grpcResults = await decryptUtterancesGrpc(
                ctx,
                Buffer.from(masterKey),
                decryptableUtterances,
                conversationKeys,
                // ...
            );
            decryptedUtterances = normalizeGrpcUtterances(grpcResults);
```

Figure 17.2: `queryConversationsById` returns plaintext utterances after a gRPC decryption step. (bee-server/sources/apps/cache/queries/queryConversationsById.ts#L154–L168)

The `Email` model defines plaintext columns for `from`, `to`, `cc`, `bcc`, `subject`, `snippet`, `plain_text_body`, and `html_body`, with no `sec_*` siblings, as shown in figure 17.3.

```
model Email {
  id                    Int             @id @default(autoincrement())
  user_id               Int
  gmail_id              String
  thread_id             String
  from                  String?
  to                    String?
  cc                    String?
  bcc                   String?
  subject               String
  date                  DateTime        @db.Timestamptz(6)
  snippet               String?
  plain_text_body       String?
  html_body             String?
```

Figure 17.3: `Email` rows are stored without the column-level encryption applied to other tables. (bee-server/prisma/schema.prisma#L687–L700)

The `CalendarEvent` model follows the same pattern, with `summary`, `description`, `location`, `creator_email`, `organizer_email`, and `attendees` stored as plaintext columns.

The ElastiCache cluster used by the email ingestors is configured without `transitEncryptionEnabled` or `atRestEncryptionEnabled`, as shown in figure 17.4.

```
this.cacheCluster = new CfnCacheCluster(this, 'ElastiCacheCluster', {
      cacheNodeType: props.nodeType,
      cacheSubnetGroupName: this.subnetGroup.ref,
      clusterName: `${props.stageName}-integrations-redis`,
      engine: 'redis',
      engineVersion: props.redisVersion,
      numCacheNodes: props.numNodes,
      preferredMaintenanceWindow: 'sun:05:00-sun:06:00',
      snapshotRetentionLimit: 5,
      snapshotWindow: '03:00-04:00',
      vpcSecurityGroupIds: [this.securityGroup.securityGroupId],
    });
```

Figure 17.4: The integrations ElastiCache cluster omits transit and at-rest encryption. (bee-cdk-production/lib/standalone-services/integrations-elasticache-construct.ts#L36–L47)

The Outlook ingestor receives both the database connection string and the master encryption key in its container environment, as shown in figure 17.5.

```
DATABASE_URL: EcsSecret.fromSecretsManager(databaseSecret, 'DATABASE_URL'),
        BEE_ENCRYPTION_KEY: EcsSecret.fromSecretsManager(encryptionSecret,
'BEE_ENCRYPTION_KEY'),
        BEE_EPHEMERAL_SEED: EcsSecret.fromSecretsManager(encryptionSecret,
'BEE_EPHEMERAL_SEED'),
```

Figure 17.5: The Outlook ingestor receives the master encryption key alongside the database URL. (bee-cdk-production/lib/standalone-services/outlook-sync-stack.ts#L153–L155)

The Gmail integrations ingestor follows the same pattern, and the Redis endpoint URL it consumes is constructed as `redis://`, not `rediss://`, so the connection is unencrypted in transit. The `google_tokens` and `microsoft_tokens` tables hold per-user OAuth refresh tokens encrypted with the same `BEE_ENCRYPTION_KEY` the ingestor process possesses, so a compromise of one ingestor yields decryption of every user's OAuth refresh tokens in addition to read and write access to the rest of the database.

Exploit Scenario

An attacker achieves code execution inside one of the Gmail or Outlook ingestor Fargate tasks, such as by exploiting a parser bug in the inbound webhook handler reachable

through the public-facing ALB. From the ingestor, the attacker reads `BEE_ENCRYPTION_KEY` out of the process environment, opens the RDS connection that the same container already holds, and decrypts every user's `google_tokens` and `microsoft_tokens` rows. The attacker also reads every user's `Email` and `CalendarEvent` rows directly, with no decryption step required. Separately, an operator with access to ElastiCache snapshots reads queries and responses against the integrations cluster from the snapshot's on-disk form, and a principal able to observe in-VPC traffic to the cache reads the same content from the wire. A follow-on compromise of the Confidential VPC's Redis tier returns plaintext conversation content without further key material.

Recommendations

Short term, separate the data-access and key-access surfaces of the ingestor containers. Replace the ingestors' raw RDS access with calls to an internal API in the Confidential VPC that writes ingested email and calendar rows on the ingestor's behalf, and replace the ingestors' possession of `BEE_ENCRYPTION_KEY` with a token-exchange endpoint that decrypts OAuth tokens only for the duration of an upstream provider call. Enable `transitEncryptionEnabled` and `atRestEncryptionEnabled` on the integrations ElastiCache cluster and switch the consumer URLs from `redis://` to `rediss://`. Confirm with the team whether the absence of `sec_*` columns on `Email` and `CalendarEvent` is an accepted design decision; if not, apply the column-level encryption already used by other user-data tables. This will address the issue because an ingestor compromise will no longer yield the encryption key, the cached and at-rest user data will no longer accumulate plaintext at intermediate hops, and the documented confidentiality property will hold at every tier.

Long term, treat the user-data path as a single trust surface whose tiers, including the Confidential VM, the ingestor, the cache, and the database, each hold the minimum data, keys, and connections needed for their role. Each new tier added to the user-data path should require an explicit declaration of which records it holds in plaintext, which keys it holds, and which connections it terminates, reviewed against the design document's confidentiality and integrity claims. This will prevent the issue from recurring because future tier additions will not be able to accidentally collapse the separations the rest of the architecture maintains.

18. Integration APIs receive user data without a sharing-consent check

Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BEE-18
Target: `bee-server/sources/apps/product/shoppingApi.ts`, `bee-cdk-production/lib/lambdas/amazon-shopping-proxy/amazon-shopping-handler.js`, `bee-cdk-production/lib/lambdas/amazon-shopping-list-proxy/amazon-shopping-list-handler.js`	
Fix Status: **Resolved**	

Description

The integrations send user-generated content from the Bee server to external Amazon retail services without a sharing-consent check at the boundary. The Bee Private Compute server's integration module invokes the proxy with the user's Amazon customer ID, the unredacted keyword string, and additional product context; the proxy Lambdas forward the request to the downstream retail Lambda without verifying that the user has authorized the data flow. The audited tree contains a sharing-consent mechanism that the design document presents as the cryptographic gate on outbound plaintext, yet the integration path does not call into it before sending user content out of the Confidential VPC. The Bee Private Compute server's outbound integration invocation is shown in figure 18.1.

```
const operation = {
            type: 'OperationInvocation',
            serviceName: 'ShoppingSearchProductsApi',
            name: 'searchProducts',
            arguments: {
                keywords,
                store,
                sortRoutine: sortRoutine ?? 'BEST_SELLERS',
                isLiveNotesRequest: 'true',
            },
        };

        const label = sanitizeLabel(keywords);
        const data = await invoke(amazonCustomerId, 'searchProducts', label,
operation);
```

Figure 18.1: Search keywords and the customer identifier are forwarded to the proxy with no consent step. (`bee-server/sources/apps/product/shoppingApi.ts#L444–L457`)

The integration create operation forwards a plaintext title unless a separately supplied `encryptedTitle` is present, as shown in figure 18.2.

```
const title = nonEmptyString(requestData.title) || nonEmptyString(requestData.name);
    const encryptedTitle = nonEmptyString(requestData.encryptedTitle) ||
nonEmptyString(requestData.encrypted_title);
    if (!title) {
      return { error: 'Missing title in request body' };
    }

    return {
      payload: {
        ...baseAapiPayload({ ...requestData, customerId }, action),
        title,
        ...(encryptedTitle ? { encryptedTitle } : { encryptTitle: true }),
```

Figure 18.2: The integration proxy admits a plaintext title at the unattested boundary. (bee-cdk-production/lib/lambdas/amazon-shopping-list-proxy/amazon-shopping-list-handler.js#L142–L152)

Other integration operations, including order search, order status, cart additions, and product detail lookups, follow the same pattern: they construct an operation payload from user input, the Amazon customer identifier, and product titles or identifiers, and invoke the proxy without consulting the consent store.

This feature is currently not available; the consent checks should be added before the feature is enabled.

Exploit Scenario

A user's Bee Private Compute installation calls an integration method as part of an LLM-driven flow. The Bee Private Compute server packages the user's search keywords, product titles, or list titles into the proxy call; the proxy Lambda forwards them to the retail backend. The retail backend stores the call, the user's customer ID, and the request payload in its own infrastructure, where they are no longer covered by the Confidential VM's attestation, encryption, or operator-restriction properties. A retail-backend operator, a retail-backend support principal, or any downstream consumer of retail logs reads the user's queries and reconstructs the user's integration intent, including any sensitive items the user named in the keywords. An external auditor verifying the design document's consent property finds the integration path materially in conflict with the documented gate.

Recommendations

Short term, gate every outbound integration call on the same sharing-consent verification used by other integrations that release plaintext content. The Bee Private Compute server's integration module refuses to construct an operation payload unless the consent store returns an active record for the current user and the integration data flow. This will address

the issue because the integration path will no longer send user content out of the attested VPC unless the user has authorized the data flow.

Long term, audit every outbound integration in the Bee Private Compute server for the same property: outbound calls that release user data must verify sharing consent at the boundary, while outbound calls that release only opaque identifiers must explicitly justify why no consent step is required, in code and in the design document. This will further prevent the issue because the design document's consent claim will then match every plaintext-releasing code path rather than a subset of them.

19. Push notifications routed to other users are sent in plaintext

Severity: **Low**	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-BEE-19
Target: `bee-server/sources/apps/push/sendPushToUser.ts`	
Fix Status: **Resolved**	

Description

The Bee Private Compute server function that delivers a push notification to a user other than the request's current user explicitly disables encryption for the notification payload. The function accepts a title, body, and payload, and forwards them unchanged to the APNs, Expo, and FCM provider clients. A comment in the function acknowledges the choice and cites the absence of an encryption context for the target user as the reason. The function is invoked in flows where one user notifies another, such as the join-request creation and acceptance handlers; in those cases, the title, body, and payload may carry the name of the requesting user, the space being joined, or other context that a passive observer of the provider's traffic, or a principal with read access to the provider's logs, can correlate to user identities. The plaintext branch appears in figure 19.1.

```
const payload = opts.payload ?? {};

    // Note: We don't support encryption for cross-user notifications
    // since we don't have the target user's encryption context.
    // These notifications will be sent as plain text.
    const notificationTitle = opts.title;
    const notificationBody = opts.body;
    const notificationPayload = payload;
```

Figure 19.1: Cross-user notifications skip the encryption step that applies to self-notifications. (`bee-server/sources/apps/push/sendPushToUser.ts#L17–L24`)

The same plaintext title and body are then passed to the APNs, Expo, and FCM providers without further transformation. The APNs path traverses the proxy Lambda covered in the adjacent finding on notification-content logging, which broadens the surface on which the plaintext is exposed.

This lack of encryption is not the case for all cross-user notifications. The `sendPushToSpaceMembers` implementation encrypts to the other user's key if available; otherwise, it sends a fallback notification.

Exploit Scenario

A user requests to join a space owned by another user. The Bee Private Compute server invokes the cross-user push function to notify the space owner of the request; the title, body, and payload include the requesting user's display name and the space's title in plaintext. An attacker with read access to the APNs proxy log group, to the FCM project's delivery logs, or to a network position between the provider and the device observes the notification content and reconstructs the social graph of who is requesting to join whose spaces, without holding any client-side encryption material.

Recommendations

Short term, fix the comment in the code to reflect the fact that encryption can be supported for cross-user notifications in some scenarios.

Long term, adapt the push notifications in this flow to use similar logic as in `sendPushToSpaceMembers`.

20. Consent proof is generated server-side, not by the client as documented

Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BEE-20
Target: `bee-server/sources/apps/user/consentRecord.ts`, `bee-server/sources/apps/user/consentGet.ts`, `bee-server/sources/apps/api/routes/user.ts`	
Fix Status: **Resolved**	

Description

The design document states that the consent proof is produced by the client: “the client generates a cryptographic proof and records it in the database” and “the user’s device submits a fresh consent signature” (section 8, “Cryptographic Consent”). The implementation does neither. The client submits only a plaintext intent: the `/users/me/consent/record` handler reads the consent type from `request.params.consent` and an optional `request.body.metadata`, with no signature in the request, and the server generates the signature itself inside the Confidential VM (figure 20.1).

```
const consentId = await randomCUID2();
const signed = consentSignature({
    consentId,
    uid: ctx.uid,
    consent: opts.consent,
    expiresAt,
    metadata,
});
const signature = ctx.encryption!.ed25519Sign(signed.path, signed.payload);

const created = await ctx.db.signedConsent.create({
    data: {
        id: consentId,
        user_id: ctx.uid,
        consent: opts.consent,
        expires_at: expiresAt,
        signature: Buffer.from(signature),
        metadata: metadata ?? undefined,
    },
});
```

Figure 20.1: Inside `consentRecord`, the server builds the payload, signs it with the user's VM-resident key, and stores the signature. (`bee-server/sources/apps/user/consentRecord.ts#L50–L69`)

Recommendations

Short term, clarify section 8 ("Cryptographic Consent") claims and document the current system behavior.

Long term, sign consent on the user's device with a key the VM never holds.

21. Ambiguous measurement serialization lets distinct configurations collide

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BEE-21
Target: `propolis-init/tpm.go`, `bee-cdk-production/lib/lambdas/amazon-dspe-registration/sources/measurement.ts`	
Fix Status: **Unresolved**	

Description

The enclave measurement chain serializes Amazon S3 entries into the NitroTPM attestation value using colon-delimited fields that are neither escaped nor length-prefixed. The serialization is therefore not injective on its own, but the measurement is not forgeable today because Finalize first mixes in `configHash`, so the finding is a latent fragility rather than an exploitable flaw.

The ambiguity is in the Amazon S3 serializer, which joins fields with a colon:

```
// formatS3Measurement formats an S3 measurement string
func formatS3Measurement(source, target, hash string) string {
	return fmt.Sprintf("%s%s:%s:%s", measurementPrefixS3, source, target, hash)
}
```

Figure 21.1: The colon-delimited Amazon S3 serializer (`propolis-init/tpm.go#L122–L124`)

A source is an `s3://bucket/key` URI that always carries the scheme colon, and a target is a local path in which colons are legal, so for any hash H the entries (source="`s3://b/x`", target="`y:z`") and (source="`s3://b/x:y`", target="`z`") both serialize to `@s3://b/x:y:z:H`. The container serializer is saved by validation—`ParseImageRef` and `parseImageRef` require exactly two colon-separated parts and Docker names cannot contain : or @—so it is unambiguous unless that validation is relaxed.

Because `configHash` covers every image, source, and target verbatim, any change that shifts a field boundary also changes `configHash`, and the only party controlling these fields is the trusted manifest author.

```
configBytes, _ := hex.DecodeString(mc.configHash)
h.Write(configBytes)
// ... build one @container:<image>:<digest> string per container ...
joinedContainers := strings.Join(containerStrings, "")
```

```
containerHash := sha256.Sum256([]byte(joinedContainers))
h.Write(containerHash[:])
```

Figure 21.2: Finalize mixes in `configHash`, then combines the entry serializations. (`propolis-init/tpm.go#L85–L96`)

Exploit Scenario

A maintainer restructures Finalize and removes or reorders the `configHash` term so it no longer covers the fields that follow it. An attacker who controls the source or target of an Amazon S3 entry, both of which may contain colons, chooses values whose embedded colons re-frame the field boundaries, so a manifest that fetches different content serializes to the same measured and NitroTPM value as an approved one. The enclave attests under the approved measurement, the registration service accepts it, and the workload-integrity guarantee no longer holds. The container list becomes vulnerable to the same confusion if image-reference validation is ever relaxed to permit a registry port or digest pin, because the entries are joined without a separator.

Recommendations

Short term, replace the colon-delimited serialization in both the Go producer and the TypeScript verifier with a self-delimiting encoding such as PASETO's pre-authentication encoding (PAE): prefix the count of items, length-prefix each field by its byte length, and include a format-version tag. This will prevent the issue because the encoding will become injective—neither the field boundaries within an entry nor the number of entries can be reframed—so distinct configurations will no longer be able to produce identical bytes and measurement integrity stops depending on the `configHash` term.

Long term, add shared test vectors, including delimiter-bearing and non-ASCII fields, that assert the Go and TypeScript implementations produce identical output, so that any future serialization divergence is caught in continuous integration.

References

- Canonicalization Attacks Against MACs and Signatures

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**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
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. Threat Model Maintenance

This report describes the threat model of the Amazon Bee Private Compute architecture at a single point in time. If the system's design or implementation significantly changes in a way that invalidates the underlying assumptions of this threat model, it will be necessary to investigate and document the implications of these changes so that the threat model remains in sync with the actual system. Keeping this threat model up to date will ensure that it remains a useful tool for informing security-related decisions in ongoing development. Access to make an editable copy of this threat model report is available upon request.

You are also welcome to re-engage with Trail of Bits to assist with evaluating the security of your new architecture and iterating on your threat model accordingly; please reach out to your current contact or use our contact form.

When to Update Your Threat Model

The threat model should be updated whenever changes are made to the set of components, trust zones, or actors; the connections between components; the access privileges of actors within the system; or the means by which users may acquire those privileges. Such changes affect the placement of the trust boundaries in the model. Examples of changes that could invalidate a previous threat model include the following:

- Putting a web server behind a load balancer and/or reverse proxy
- Adding features that rely on new supporting infrastructure or third-party services
- Changing network ingress/egress rules
- Changing how users authenticate to the system (e.g., adding a second factor, adding a "remember this device" option)
- Adding a new privilege level or authorization check for application users
- Reorganizing company structure in a way that affects which employee roles have access to source code, key material, signing devices, the CI pipeline, and so on

Changes that swap out a component in a like-for-like manner (i.e., those that refactor internal implementation details of a component but do not change its external behavior) generally do not affect the high-level architecture of the system nor result in adding or changing trust boundaries, so they will not invalidate the existing threat model. Examples of such changes include the following:

- Changing how a microservice is implemented internally, but retaining its original API

- Adding a new way to interact with a microservice that has the same security properties and authentication requirements as an existing method of access (e.g., making existing functionality callable over a new protocol, authenticated against an existing set of users)
- Switching to different, but backward-compatible, database software
- Horizontally scaling an existing service

Updating Your Threat Model

When the system's design changes in a way that could invalidate the existing threat model, consider the following questions to decide appropriate next steps:

Does this change add a new system component (e.g., microservice, module, major feature, or third-party integration)?

- Add it to the Components table under the appropriate trust zone, or create a new trust zone if none apply. A component falls within an existing trust zone if an actor gaining access to any one component within the existing trust zone would also implicitly gain access to the new component in question.
- Identify the components that will send data to, or retrieve data from, the new component; the means by which they do so; and the trust zones in which they reside. Add one entry to the Trust Zone Connections table per data flow.
- Iterate through all actors noted in the Threat Actors table. Consider whether each type of actor has access to the new component and by what means they have that access. Keep in mind that any given actor may have multiple ways to access a single component or trust zone. For any threat actors that have access, document each case in the Threat Actor Paths table.

Does this change add a new trust zone (e.g., by adding a new network segment)?

- Reclassify any system components accessed in different ways into the new trust zone as appropriate. If a new authentication check or privilege level was added, this likely indicates that a new trust zone has been added. If services were moved between existing network segments or existing access roles were reassigned, it may be sufficient to simply move the components in question to different, existing trust zones.
- Update the Trust Zone Connections and Threat Actor Paths tables as noted above.

Does this change add a new threat actor (e.g., a new user role)?

- Add the new actor to the Threat Actors table.
- Identify which trust zones and components the new threat actor should have privileges to access or interact with, and update the Threat Actor Paths table.

Does this change add a new connection between system components that crosses a boundary between trust zones (e.g., a new application service on an existing server instance that can be called by a service in a different zone)?

- Update the Trust Zone Connections table to account for the new connection. Be sure to note when a connection is encrypted, the type of encryption in use, and any authentication or authorization data needed for the connection to successfully occur.
- Identify which threat actors have direct access to the trust zones from which this new connection originates, and add corresponding entries to the Threat Actor Paths table.

When making changes to any of the above-noted tables, remember to also update the system diagram accordingly.

C. 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 June 30 to July 1, 2026 and from September 8 to September 9, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the Amazon Bee Private Compute team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 21 issues described in this report, the Amazon Bee Private Compute team has resolved 17 issues, has partially resolved two issues, and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Legacy boot path executes code outside measurement context	High	Resolved
2	Sigstore context is never policy-enforced	Medium	Resolved
3	Application data disks are mounted by device path without measured or CVM-bound protection	High	Resolved
4	Android RFC 3161 timestamp verification can fail open and accept attacker-chosen verification time	Medium	Resolved
5	AMI Rekor entries are never signature-verified or bound to the transparency team signing certificate	High	Resolved
6	Go proxy accepts NitroTPM COSE_Sign1 payloads without signature verification	High	Resolved
7	Shared LLM prefix caches are not isolated across users	High	Resolved
8	Shared CVM Redis access is not scoped by service identity	Informational	Resolved

9	Bee Private Compute server derives deterministic CTR keys for BLE with insufficient checks	Medium	Resolved
10	Client attestation failures do not block key release	High	Resolved
11	Paired-app developer tokens lack scoped permissions	Low	Unresolved
12	Conversation summaries are stored unencrypted in the search index	Medium	Partially Resolved
13	Risk of container environment-variable injection via AWS Secrets Manager that can bypass the attestation chain	High	Resolved
14	Confidential VM image includes unnecessary packages	Informational	Resolved
15	APNs proxy Lambda logs notification content and request payloads	Medium	Resolved
16	Internal networking policy is overly lax for the Confidential VM tier	Informational	Resolved
17	Internal security controls for user data are overly lax	Medium	Partially Resolved
18	Integration APIs receive user data without a sharing-consent check	Medium	Resolved
19	Push notifications routed to other users are sent in plaintext	Low	Resolved
20	Consent proof is generated server-side, not by the client as documented	Informational	Resolved
21	Ambiguous measurement serialization lets distinct configurations collide	Informational	Unresolved

Detailed Fix Review Results

TOB-BEE-1: Legacy boot path executes code outside measurement context
Resolved in the latest code. The patch fully removes the legacy boot path.

TOB-BEE-2: Sigstore context is never policy-enforced
Resolved in the latest code. The revised code introduces checks of context values for the Go proxy, Android, and iOS. Clients now ensure that the CN contained in a certificate matches the Sigstore context so that auditors can determine what service identities have used a given set of measurements.

TOB-BEE-3: Application data disks are mounted by device path without measured or CVM-bound protection
Resolved in the latest code. `format_disks.py` adds LUKS encryption with a unique, randomly generated, and in-memory key.

TOB-BEE-4: Android RFC3161 timestamp verification can fail open and accept attacker-chosen verification time
Resolved in the latest code. The revision verifies the existence of signer information and throws on errors rather than failing open.

TOB-BEE-5: AMI Rekor entries are never signature-verified or bound to the transparency team signing certificate
Resolved in the latest code. The Go proxy, Android code, and iOS code now verify signatures on AMI Rekor entries.

TOB-BEE-6: Go proxy accepts NitroTPM COSE_Sign1 payloads without signature verification
Resolved in the latest code. The verification routines in `nitrotpm/verify.go` now verify the COSE signature payload.

TOB-BEE-7: Shared LLM prefix caches are not isolated across users
Resolved in the latest code. Both embedding backends now pass `--disable-radix-cache`, extending the cache mitigation to the remaining services. Together with the existing changes to the shared chat backends, this disables cross-request prefix caching on the affected inference services.

TOB-BEE-8: Shared CVM Redis access is not scoped by service identity

Resolved in the latest code. `service-redis-new.ts` now accepts an incoming context for each Redis instance that gates access based on role.

TOB-BEE-9: Bee Private Compute server derives deterministic CTR keys for BLE with insufficient checks
Resolved in the latest code. The `/ble/secure-key` route now checks the device is owned and active before releasing the key.

TOB-BEE-10: Client attestation failures do not block key release
Resolved in the latest code. The Android and iOS client applications now enforce attestation for production builds.

TOB-BEE-11: Paired-app developer tokens lack scoped permissions
Unresolved in the latest code. The client provided the following context for this finding's fix status:

> *Developer tokens are unscoped by design, as they're only available to the owner and the owner has full access to their data.*

TOB-BEE-12: Conversation summaries are stored unencrypted in the search index
Partially resolved in the latest code. The revised code attempts to delete unencrypted search-index entries when a user's processing key expires or their account is deleted. However, deletion can be delayed or fail, leaving the summaries available after the key expires.

TOB-BEE-13: Risk of container environment-variable injection via AWS Secrets Manager that can bypass the attestation chain
Resolved in the latest code. `secrets.go` ensures that keys and values do not contain newlines, and secrets are versioned.

TOB-BEE-14: Confidential VM image includes unnecessary packages
Resolved in the latest code. The patch removes unnecessary packages from `ami-config.json` files.

TOB-BEE-15: APNs proxy Lambda logs notification content and request payloads
Resolved in the latest code. The console logging now only includes non-sensitive metadata.

TOB-BEE-16: Internal networking policy is overly lax for the Confidential VM tier
Resolved in the latest code. The revision removes unrestricted internet access from the agents service.

TOB-BEE-17: Internal security controls for user data are overly lax
Partially resolved in the latest code. The ingestion caches now enable encryption in transit and at rest, and their consumers use TLS connections. However, Email and CalendarEvent columns remain unencrypted, ingestor containers retain both database access and the shared encryption key, and the conversation cache continues to hold decrypted content.

The patch addresses the ingestion-cache encryption issue but leaves the other identified controls unchanged. The client provided the following context for this finding's fix status:

> *The ingestion cache is now encrypted in transit and at rest. The remaining items concern the third-party integrations service, which currently runs outside the confidential-VM boundary and holds only data from accounts the user has explicitly connected. Bee requests the status Partially Resolved.*

TOB-BEE-18: Integration APIs receive user data without a sharing-consent check
Resolved in the latest code. All outbound integration paths now require verified `amazon_shopping` consent before forwarding user data, including API handlers and background workers. The client chose to leave Amazon-related consents exempt from the seven-day expiry as an intentional design decision.

TOB-BEE-19: Push notifications routed to other users are sent in plaintext
Resolved in the latest code. The approval notification now uses a fixed, generic message, and the space name has been removed from the unencrypted notification type and its caller. This eliminates the plaintext space-name disclosure in the approval notification.

TOB-BEE-20: Consent proof is generated server-side, not by the client as documented
Resolved in the latest code. The recommendation for this finding was to update the documentation. The client provided the following context for this finding's fix status:

> *[T]he whitepaper will be clarified to describe the actual consent-proof flow: the server constructs the consent payload and signs it with `ctx.encryption.ed25519Sign` during `consentRecord`, then `consentGet` verifies and returns the signed consent record.*

TOB-BEE-21: Ambiguous measurement serialization lets distinct configurations collide
Unresolved in the latest code. No change was made. The client provided the following context for this finding's fix status:

> *Bee agrees with Trail of Bits' analysis that the serialization is not exploitable because the manifest hash covers every field before the per-entry measurements are folded in. We have chosen not to change the measurement format in this release, since doing so would invalidate every currently registered image and require coordinated re-registration. The finding is tracked for the next measurement-format revision. Please mark as Risk Accepted.*

D. 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 assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn 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

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.