Calyx Institute HSM Provisioning Ceremony Scripts

Security Assessment

January 23, 2026

Prepared for:
Aysha Habbaba
Calyx Institute

Prepared by: **Joop van de Pol**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Kimberly Espinoza, Project Manager
kimberly.espinoza@trailofbits.com

The following engineering director was associated with this project:

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

The following consultant was associated with this project:

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Project Timeline

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

Date	Event
January 2, 2026	Pre-project kickoff call
January 12, 2026	Delivery of report draft and readout meeting
January 23, 2026	Completion of fix review
January 23, 2026	Delivery of final comprehensive report

Executive Summary

Engagement Overview

The Calyx Institute engaged Trail of Bits to review the security of its HSM provisioning ceremony scripts.

The Calyx HSM provisioning ceremony scripts will be used to set up a secure machine and provision two YubiHSM 2 devices (a primary and a backup device) with a wrapping key and three authentication keys corresponding to the roles of admin, signer, and auditor. The wrapping key is split into five shards with a threshold of three shards required for recombination. Each shard and authentication key is encrypted using one of five public keys that are part of the ceremony scripts. In a later operational phase, the admin or signer can generate CalyxOS device signing keys on the HSM, attest to them using the preloaded certificate, and wrap them using the wrapping key for external storage.

One consultant conducted the review from January 5 to January 12, 2026, for a total of one engineer-week of effort. Our testing efforts focused on finding any flaws that would lead to the exposure of sensitive data. With full access to source code, documentation, ceremony playbooks, and a testing HSM, we performed static and dynamic testing of the provisioning ceremony scripts, using automated and manual processes.

Observations and Impact

We did not observe any flaws in the ceremony scripts that would allow an attacker to obtain the wrapping key or legitimately generated signing keys. We identified one attack path where a holder of the HSM could insert a malicious admin key in a disjoint domain before the ceremony and use it to mount a denial-of-service attack or generate new signing keys that are attested by the HSM (TOB-CHP-1). Additionally, the password for the authentication key of the signer role is too short (TOB-CHP-2).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that the Calyx Institute take the following steps before executing the ceremony:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes in the scripts or by updating the ceremony steps.
- **Test the updated ceremony scripts.** Because the remediation of some findings requires updates to the ceremony scripts, we recommend rerunning all tests, including the corresponding signing scripts, to ensure that the system still works.
- **Consider the feedback in the appendices.** Specifically, consider the feedback on ceremony steps in appendix C, the observations regarding attestation and

endorsement of public keys in appendix D.3, and the known vulnerabilities of YubiHSM 2 with old firmware versions in appendix E.2.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Cryptography	1
Data Validation	1

Project Goals

The engagement was scoped to provide a security assessment of the Calyx Institute HSM provisioning ceremony scripts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do the provisioning scripts ensure that both HSMs are configured securely, and that it is not possible to exfiltrate or otherwise obtain signing keys?
- Do the provisioning scripts ensure that all necessary artifacts are stored in the correct location for saving after the ceremony?
- Are there any known vulnerabilities in the included packages that could affect the security of the provisioning?

Project Targets

The engagement involved reviewing and testing the following target during the initial review. In order to resolve the findings in this report, Calyx updated three of these scripts; the updated hashes are displayed in bold beneath the originally reviewed hashes.

ceremony.zip

SHA-256: `bd1d7f5c958d5a53a017b13ad0a390e1df53e6f3b273b22385191784c0ed5031`

`638ff9c5cc5637c208d1753c89b5ef97ef09bf24300d5777ed64b37115947635`

Directory	File	SHA-256
/	start.sh	44e74845a6c4d4cad2b8c0b67a0de211 a9bb7359571c0d2f45e7b9aec9eee257
packages/	age_1.2.1-1+b5_amd64.deb	a87ab5ba69a99bc6d8c95935ae467fa8 77daef50d11d9dabe0520ae01411f85c
packages/	libcurl4t64_8.14.1-2+deb13u2_amd64 .deb	d73aa7c6e5293edce3d25d814d662ab4 b127cf697895ebdd7ea5c0a54548616e
packages/	libeac3_1.1.2+ds+git20220117+453c3 d6b03a0-1.1+b3_amd64.deb	089366fb2b71cadca53fb844cfabb0d0 798623463247f5b8fa6ad1cc3ab382f2
packages/	libengine-pkcs11-openssl_0.4.13-1_ amd64.deb	6c7767e16e785a4752fb07c1e57bc4a6 0174486118204ae450fd47a838e114a5
packages/	libpcsclite1_2.3.3-1_amd64.deb	a8bc4725549060d1e696d358e32a3699 d947d5381871553a290389830bebccbd
packages/	libusb-1.0-0_1.0.28-1_amd64.deb	bb6467a002915c4ac464aa1b22204907 951a30e43bcb86596ebd3539e84458e9
packages/	opensc-pkcs11_0.26.1-2_amd64.deb	6f278a3a506530d0675c112bc23ffe37 a5716300d3eb26304c75b67544de5506
packages/	opensc_0.26.1-2_amd64.deb	2797d3f18b026f325a8901b91692618f 105c3d50d60a5f0263054725a13ea9e5
packages/	p11-kit_0.25.5-3_amd64.deb	3b048929b011eab66ad30cf888060456 281b12a0e2fce2e17dc62906947d0835

packages/	python3-yubihsm_3.1.1-1_all.deb	211b2bb012aca1fbc91602d02813c9cb a6006f49533dab622c6812fc516b01fa
scripts/	audit.py	a28bc3ebb797e59f6cb24cba1af44936 dd813cd6e2f09a897bafb628c787a3e8
		6c329831d0a6f23857af58fed820a0c7 a5cd00763c76b8d6e7a5a0d1bf223de9
scripts/	calyx-shamir-split	c2f25e8c5530de6f534618c8cceaca47 67456f6e027499f3b6d90385c8feb7b4
scripts/	provision.py	0b9b73c2ca89b868a1da511f5bdd5977 722e3b96a22af42930c81b95b186d49e
		5a70adc520015bf14ab96575e3873cb1 bded204d0273946b3a9e949da996cc03
scripts/	setup.py	238aa41f203e644322e529ee2b30c1da 94cef30aa4f1cb38ee0af2a35f84ab39
		33642f79e4c4ffe8175f0e103f95d25c 83b4426694476efc5e46458eae528c43
scripts/ keys/	1.pub	e77a30c4fd1c9df295d0f534b09f327d f050aaab247321fbc0ef80d4f8de9da3
scripts/ keys/	2.pub	46a4e9b8e9597b8b3d95351cf398cdd1 1ad4fde5d5514ec2316a724a41f53340
scripts/ keys/	3.pub	d526ace52a36c2c41b8a2f36e0cee7e6 2c14c358dace99e07e1e7f961280e909
scripts/ keys/	4.pub	517c9eb9b84c63aefa299cd02a7c9354 3bbe880b2f1a540fdb6ae2965ef003dd
scripts/ keys/	5.pub	dbd1dcecbec8245775927d45220683ac b462eb77f42471675741137ee671d2a8
sdk/	yubihsm2-sdk-2025-06-debian12-amd6 4.tar.gz	99501dced7e04ff5bb9150503a2e0ab7 031714bfd1f6c08b997ea49467d8713a

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:

- Manual review of the ceremony scripts
- Static analysis of the ceremony scripts using Semgrep and CodeQL
- Testing on a local YubiHSM 2
- Review of the ceremony documentation
- Hash verification of included packages

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:

- **Third-party packages.** For the third-party packages, we verified their hash values and investigated known vulnerabilities affecting them. However, we did not perform a review of the implementation of these packages.

Summary of Findings

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

ID	Title	Type	Severity
1	Factory reset verifications are insufficient to prevent rogue admin keys	Data Validation	**Medium**
2	The signing authentication key password entropy is below 80 bits	Cryptography	**Informational**
3	Some keys have unnecessary capabilities	Access Controls	**Informational**

Detailed Findings

1. Factory reset verifications are insufficient to prevent rogue admin keys	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CHP-1
Target: `scripts/setup.py`	

Description

When provisioning each HSM, the setup script performs checks to ensure that the HSM has default values corresponding to a factory reset state, but these checks still allow for malicious behavior. Specifically, the script checks that it can authenticate using the authentication key with ID 1 and the default password (`password`); that listing all objects in the resulting session shows there is only one object (which must necessarily be the authentication key); and that this object has the label `"DEFAULT AUTHKEY CHANGE THIS ASAP"`.

```
213     try:
214         # establish session (using factory default credentials)
215         session = hsm.create_session_derived(0x0001, "password")
216     except yubihsm.exceptions.YubiHsmAuthenticationError as e:
217         # if we can't authenticate with default credentials, the HSM isn't
factory reset
218         on_not_factory_reset()
219         raise e
```

Figure 1.1: The first check comprises authentication with key ID 1 and the default password. (ceremony/scripts/setup.py#L213–L219)

```
241     # get all keys on device and check if this is factory default
242     keys = session.list_objects()
243     if len(keys) != 1 or keys[0].get_info().label != "DEFAULT AUTHKEY CHANGE
THIS ASAP":
244         on_not_factory_reset()
245     else:
246         print("\nDevice seems to be factory reset, continuing...\n")
```

Figure 1.2: The second check verifies that there is only one object and that it has the default key label. (audit-calyx/scripts/setup.py#L241–L246)

However, it is possible to create keys on the device while still passing these checks. The YubiHSM 2 supports different domains, and authentication keys limited to specific domains

cannot access keys outside those domains. Therefore, it is possible to replace the default key with an identically configured key that does not have access to domain 16, while creating a key with all capabilities inside domain 16. This will not appear when listing the objects using the replaced default key, and therefore, all checks will pass. Since the factory default logging does not log commands, the only evidence visible in the logs would be an additional boot event.

Exploit Scenario

Someone holding the HSM before the ceremony authenticates using the default password and creates two authentication keys with all (delegated) capabilities. The first gets ID `0xFFFE` and domain 16, and the second gets ID `0xFFFD` and all domains. They use key `0xFFFD` to delete the default key and generate a new authentication key with all (delegated) capabilities, ID 1, and domains 1 through 15. Finally, they delete the key with ID `0xFFFD`. This leaves the keys with IDs 1 and `0xFFFE` on the HSM, which cannot see each other since they are in disjoint domains.

Since it is necessary to work in different domains to hide the malicious authentication key, it is not possible to exfiltrate any sensitive keys using this key. Still, the malicious key can be used to factory reset the HSM or configure additional logging settings to soft-lock the HSM, which requires a physical reset, both of which cause a denial of service. Alternatively, it can be used to generate signing keys and attest to them using the preloaded attestation certificate, which appear to be legitimate signing keys to outside observers.

Recommendations

Short term, update the ceremony procedures to include a physical reset of each HSM on first use, and consider updating the scripts to include a logical reset after the first authentication. When listing the objects, you can also check that the sequence number of the first key is zero, which means it has not been replaced. You can also verify that the log does not contain more than one boot event.

Long term, carefully design the procedures around key selection for image signing. See appendix D.3 for more information.

2. The signing authentication key password entropy is below 80 bits	
Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-CHP-2
Target: `scripts/setup.py`	

Description

The password of the signing authentication key is randomly generated as 12 characters from an alphabet of 94 characters, corresponding to approximately 79 bits of entropy. While the password would still be challenging to brute force, it is not considered sufficiently strong by cryptographic standards (e.g., see NIST SP 800-57, table 4).

```
171     def create_auth_password(auth_key_dir, name, public_key_id):
172         # generate password
173         alphabet = string.ascii_letters + string.digits + string.punctuation
174         password = ''.join(secrets.choice(alphabet) for _ in range(12))
175
176         # encrypt password with age
177         public_key_path = os.path.join(script_dir, "keys",
f"{public_key_id}.pub")
178         auth_key_path = os.path.join(auth_key_dir, f"{name}.key")
179         age_command = ['age', '-R', public_key_path, '-o', auth_key_path]
180         subprocess.run(age_command, input=password, text=True, check=True)
181
182         return password
```

Figure 2.1: The password consists of 12 characters. (ceremony/scripts/setup.py#L171–L182)

Recommendations

Short term, update the password length to 20 characters for 128 bits of entropy or to 40 characters for 256 bits of entropy.

Long term, store the password in a secrets manager after the ceremony and use the secrets manager to authenticate to the HSM.

3. Some keys have unnecessary capabilities

Severity: **Informational**	Difficulty: **N/A**
Type: Access Controls	Finding ID: TOB-CHP-3
Target: `scripts/setup.py`	

Description

The wrap key receives unnecessary delegated capabilities to generate asymmetric keys and to import/export wrapped keys, and the three authentication keys receive the unnecessary capability to be exportable under wrap.

```
33    WRAP_KEY_DEL_CAPS = (
34            CAPABILITY.DECRYPT_PKCS |
35            CAPABILITY.DECRYPT_OAEP |
36            CAPABILITY.GENERATE_ASYMMETRIC_KEY |
37            CAPABILITY.SIGN_PKCS |
38            CAPABILITY.SIGN_PSS |
39            CAPABILITY.SIGN_ECDSA |
40            CAPABILITY.SIGN_EDDSA |
41            CAPABILITY.DERIVE_ECDH |
42            CAPABILITY.IMPORT_WRAPPED |
43            CAPABILITY.EXPORT_WRAPPED |
44            CAPABILITY.EXPORTABLE_UNDER_WRAP |
45            CAPABILITY.GET_LOG_ENTRIES |
46            CAPABILITY.GET_OPTION
47    )
```

Figure 3.1: Delegated capabilities assigned to the wrap key (ceremony/scripts/setup.py#L33–L47)

The capability for generating asymmetric keys applies to only authentication keys, and the capabilities for importing/exporting wrapped keys apply only to authentication keys and wrap keys. Therefore, they are needed only if authentication keys and wrap keys are planned to be wrapped. However, the ceremony ensures that only one wrap key will be created, and there is no need to wrap authentication keys, as they can be directly registered on other devices if necessary.

```
132    AUDIT_AUTHKEY_CAPS = (
133            CAPABILITY.GET_LOG_ENTRIES |
134            CAPABILITY.EXPORTABLE_UNDER_WRAP |
135            CAPABILITY.GET_OPAQUE
136    )
```

Figure 3.2: Capabilities assigned to the audit authentication key (`ceremony/scripts/setup.py#L132–L136`)

The authentication keys do not need to be exported under wrap, and the only wrap key that will remain after the ceremony cannot wrap them, because that wrap key lacks the delegated capability for getting opaque objects.

None of these capabilities can be used to mount any attacks on their own; however, granting them violates the principle of least privilege.

Recommendations

Short term, remove these capabilities. Additionally, grant the auditor user the capability to change their authentication key. Finally, add the signer's delegated capabilities to the administrator's delegated capabilities.

Long term, create tests that remove any capability and run through the provisioning ceremony and signing flows to determine whether those capabilities are currently strictly needed for operational use.

A. Vulnerability Categories

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

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

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

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

B. Code Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

Since this code is to be used in an upcoming ceremony, and any additional code changes require further testing to ensure the implementation will not break during the ceremony, we recommend planning to address these issues in future development after the upcoming ceremony.

- **Audit configuration can be performed in a single command.** Each command to be audited is configured using a separate `put_option` call. However, it is possible to create a byte array of each command followed by the byte `b"\x02"` and configure everything in one call.

```
340     for command in commands_to_audit:
341         session.put_option(OPTION.COMMAND_AUDIT, command.to_bytes() +
b"\x02")
342     print("Successfully provisioned auditing options.")
```

Figure B.1: Each command to be audited is configured in a separate operation. (ceremony/scripts/setup.py#L340–L342)

- **The default key is deleted without authenticating the administrator key.** The current script authenticates using the default key, creates an admin authentication key, verifies that the admin authentication key object exists, deletes the default authentication key, and authenticates a new session using the admin key, without verifying that authentication using the admin authentication key works. It would be more robust to generate the admin authentication key, open a new session using the admin authentication key, and only then delete the default authentication key (which can be done in either session). This will ensure that the default key is deleted only when the administrator key works.

```
402     def delete_default_auth_key(hsm, session, admin_private_key):
403         # first double check that new admin auth key really exists, or we
lock ourselves out
404         admin_key = session.get_object(ADMIN_AUTHKEY_ID,
OBJECT.AUTHENTICATION_KEY)
405         admin_key.get_info()
406         try:
407             default_auth_key = session.get_object(0x0001,
OBJECT.AUTHENTICATION_KEY)
408             default_auth_key.get_info()  # causes OBJECT_NOT_FOUND if key
doesn't exist
409             default_auth_key.delete()
```

```
410             print("Deleted default auth key")
    411             return hsm.create_session_asymmetric(ADMIN_AUTHKEY_ID,
   admin_private_key)
```

Figure B.2: Authentication using the administrator key happens only after the default key is deleted. (`ceremony/scripts/setup.py#L402–L411`)

C. Ceremony Quality Issues

This appendix contains findings in the ceremony descriptions that do not have immediate or obvious security implications. However, addressing them may enhance the ceremony's executability while maintaining a sufficient security level.

- **Generating a securely random TailsOS administrator password is unnecessary.** During TailsOS startup, the ceremony description prescribes generating a random administrator password and reading it aloud to all participants. This password is not persistent and is used only for sudo access to install packages and start the YubiHSM connector when running the provisioning process. Knowing the password will help an attacker only if they are allowed to interact with the provisioning machine during the ceremony. However, anyone allowed to interact with the provisioning machine can set up extraction of the wrap key without sudo access by making targeted modifications to the provisioning scripts. Since all interactions with the ceremony computer are recorded, any such interactions should be detected. Therefore, it should be fine to choose a simple password for easier entry during provisioning.

- **Placing HSMs in tamper-evident bags during the ceremony is unnecessary.** During HSM prep, the ceremony description prescribes connecting the HSMs and verifying their status. Afterward, the HSMs are to be placed inside tamper-evident bags until they need to be used. However, this does not serve any purpose, since it is not possible to interact with them unless they are plugged in. It is sufficient to have them present on the ceremony table in full view of the participants (both local and remote). The only exception to this is if the ceremony needs to be interrupted; in this case, the description for breaks already states that all materials are to be put into tamper-evident bags.

  Note that it is essential to place the HSMs in tamper-evident bags at the end of the ceremony, not between HSM preparation and actual provisioning. The purpose of bagging at this point is to protect against tampering from the end of the ceremony until the HSMs are in the trusted custody of the holder at their intended location. But this protection is meaningful only when the ceremony proceeds without significant irregularities; if the recording shows numerous exceptions, emergencies, or participants leaving the ceremony room, observers will not trust the resulting HSMs regardless of whether they were bagged during the ceremony.

D. YubiHSM 2 Limitations

While the YubiHSM 2 offers many of the functionalities of an HSM at a competitive price, it has certain limitations that affect the design of the CalyxOS device signing system.

D.1 Storage

The YubiHSM 2 provides only 256 slots for objects, so it is never possible to store more than 255 keys on the device (as one slot is required to store an authentication key at a minimum). Furthermore, due to the 126 KB storage capacity, it can store only 127 RSA 2048 keys, 93 RSA 3072 keys, or 68 RSA 4096 keys.

CalyxOS requires more keys for the various devices that need to be signed, so not all keys will fit inside the HSM. As the YubiHSM does not provide extensive key derivation functionality that would allow all device keys to be derived from a master seed, the signing keys must be stored persistently outside of the HSM. To this end, the ceremony creates a wrap key that wraps all signing keys for secure storage outside of the HSM.

As a result, no one should have access to this wrap key, as it would allow them to decrypt signing keys. However, if the wrap key is lost (e.g., when the HSMs break), the encrypted signing keys can no longer be decrypted. That is why the wrap key is split into five shares, three of which are required for recombination. This allows a recombination ceremony where a new HSM is provisioned with the same wrap key.

D.2 Audit Logging

The YubiHSM 2 offers very limited audit logging capabilities. The log entries themselves are very succinct, describing only the global entry number, the command that was used, the length of the command data, the keys that were involved (session key, target key, and second key), the command response code, the number of ticks since the previous reset, and the truncated hash computed over the aforementioned data and the previous truncated hash.

This means that it is almost impossible to determine what happened by examining the logs. In the CalyxOS signing system, for example, the log will not explain which signing key was unwrapped (beyond the assigned ID), which value was signed, or what the resulting signature was.

Furthermore, there is room for only up to 62 different log entries. This means that logs, like keys, have to be stored externally for persistence. However, once they are removed from the HSM, it is very difficult to prove that they are authentic. The person reading the log directly from the HSM cannot prove to anyone else that the logs are authentic. There is no signature from the HSM over the audit log (although there is a feature request on this topic), and anyone can compute a correct hash chain of events given an arbitrary starting

point. Fake audit logs can be detected only by an honest auditor who interacts directly with the HSM and compares the latest audit logs with the externally stored logs.

Therefore, it is important that multiple parties periodically authenticate to the HSM and perform an operation that is logged (even if they do not have the permissions for this operation, because unsuccessful attempts also get logged), and read the log to get the latest entry. They can then compute the hash chain from the entries captured in the ceremony up to this latest entry and verify that there is an unbroken chain, which proves that the logs are authentic. If there is even a single gap, it is no longer possible to connect these logs, and the new entry becomes a trust root for verifying future entries, although it cannot be verified itself.

We strongly recommend that auditors use asymmetric authentication keys to authenticate to the HSM, because symmetric keys do not protect against MitM attacks by attackers who also possess this symmetric key.

We further recommend that signers be required to store all signatures produced by the HSMs in addition to the log entries. This will allow auditors to account for all signatures as they can count the number of signature operations performed in the logs. Since signers are responsible for keeping the logs up to date when executing device signing, it is essential that this process handles failures as robustly as possible to prevent gaps in the resulting logs. Nonetheless, system administrators and auditors should be prepared for the scenario where intermediate log entries are lost due to device crashes or other mishaps. Such events should be treated as incidents and thoroughly investigated to determine whether and how many unauthorized signatures could have been generated as part of this gap.

Note that, for robust auditing, it is necessary to configure the HSMs to force logging, which means that the HSMs will not allow the execution of commands requiring logging until the log is cleared using the `SET LOG ENTRIES` command. However, if the configuration requires logging of commands such as `GET LOG ENTRIES`, `SET LOG ENTRIES`, or `SESSION MESSAGE`, it will no longer be possible to read or reset the logs when they are full, resulting in a soft-lock of the HSM. Resolving this situation would require performing a physical factory reset of the HSM.

D.3 Attestation

While the YubiHSM 2 stores all private signing keys, at some point, the corresponding public keys must be endorsed as signing keys for devices in the CalyxOS ecosystem. This is a very critical operation that requires a careful process to ensure that only legitimately generated signing keys are endorsed.

One part of this is to ensure that the signing key gets an attestation from the YubiHSM 2, using the preloaded attestation key, which has a certificate chain tracing back to a Yubico root certificate. The attestation will state whether the key was generated on a YubiHSM 2 device, whether it was imported into the attesting device, and whether wrapping was used

for this import. It is necessary, but not sufficient, to require attestation certificates showing that keys were generated directly on the primary or backup HSM before endorsing them for use in the CalyxOS ecosystem. Therefore, when generating new signing keys, the HSM operator must generate attestations before wrapping the signing keys and deleting them from the HSM.

To understand why it is not sufficient to endorse any key that has a correct attestation, it is essential to recognize that the preloaded attestation key is burned into the YubiHSM 2 and will persist across factory resets. So, any user who holds the HSM before the initial ceremony can generate as many signing keys as they want, generate attestations from the HSM, and exfiltrate the keys from the device, as there are no security controls to prevent this yet.

Now, to protect against such an attack, it may seem sufficient to require that each key receive some data in its key label upon generation that proves freshness (such as a hash value appearing in a recent blockchain block). However, note that the signer has physical access to the HSM, which allows them to perform a physical reset at any time. After this reset, they possess the same capabilities as the attacker in the aforementioned scenario and can generate attestations at will for newly generated keys. As this will eventually be detected by auditors, the malicious signer can claim that the HSM stopped working and had to be reset.

As long as the wrap key shard holders do not collude, the wrap key can exist only in a legitimate HSM that has not been reset. Therefore, to prevent these kinds of attacks based on rogue attestations, the endorsement process should include a step where one or more parties (such as the administrator) verify that the signing key is accompanied by a wrapped version for persistent storage (in addition to the attestation certificate). This verification must include unwrapping the wrapped key and exporting its public key to ensure that it matches the attestation.

E. Third-Party Vulnerabilities

There are known vulnerabilities in both the third-party `curl` package version that is used in the ceremony and in the YubiHSM 2 itself.

E.1 curl

The used version of `curl` has a number of CVEs, as shown in the following table:

CVE	Description
CVE-2025-9086	Out of bounds read for cookie path
CVE-2025-10148	Predictable WebSocket mask
CVE-2025-10966	Missing SFTP host verification
CVE-2025-13034	Lack of pinning (`CURLOPT_PINNEDPUBLICKEY`)
CVE-2025-14017	Broken TLS options for threaded LDAPS
CVE-2025-14524	OAuth2 bearer token exposure
CVE-2025-14819	OpenSSL partial chain store policy bypass
CVE-2025-15079	libssh global `known_hosts` override
CVE-2025-15224	libssh key passphrase bypass

The provisioning scripts use `curl` only when connecting to the YubiHSM connector on `localhost`. Therefore, none of these vulnerabilities apply to the usage of `curl` in the provisioning ceremony.

E.2 YubiHSM 2

YubiHSM 2 devices with firmware versions before 2.4.0 are susceptible to CVE-2024-45678, also known as EUCLEAK. While this is a difficult attack that requires specialized equipment to measure electromagnetic signals emanating from the chip of the HSM, the attack scenario theoretically applies to the CalyxOS use case. This attack would allow a signer to extract ECDSA signing keys from the device by observing many ECDSA signing executions.

Since each signing execution would be logged, it is not possible to perform this attack without detection. Still, a malicious signer can perform many operations, causing the log entries to wrap around. It is not possible to craft a correct hash chain while hiding the number of operations in this case; however, the signer can create a small gap in the chain and claim that their machine crashed before the log could be saved. Therefore, it is highly recommended to ensure that YubiHSM 2 devices have firmware version 2.4.0 or later to prevent the possibility of this attack.

F. Fix Review Results

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

On January 23, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the Calyx Institute team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

The Calyx Institute updated three of the in-scope scripts to fix the reported issues. We reviewed the updates and added the hashes to the Project Targets section.

In summary, the Calyx Institute has resolved all three issues described in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Factory reset verifications are insufficient to prevent rogue admin keys	Medium	Resolved
2	The signing authentication key password entropy is below 80 bits	Informational	Resolved
3	Some keys have unnecessary capabilities	Informational	Resolved

Detailed Fix Review Results

TOB-CHP-1: Factory reset verifications are insufficient to prevent rogue admin keys
Resolved in commit `2c08aea`. Instead of performing factory reset verifications, the setup script now resets each HSM via the `RESET DEVICE` command.

TOB-CHP-2: The signing authentication key password entropy is below 80 bits
Resolved in commit `527a165`. The password is now 40 characters long, corresponding to at least 256 bits of entropy.

TOB-CHP-3: Some keys have unnecessary capabilities
Resolved in commit `b3b7eb8`. The unnecessary capabilities have been removed and the auditor role can now change their own authentication key.

G. Fix Review Status Categories

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

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

About Trail of Bits

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

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

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

We specialize in software testing and code review 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.