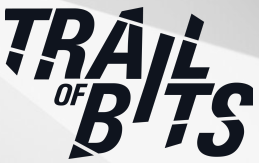

Automata DCAP Attestation and Onchain PCCS

Security Assessment

February 28, 2025

Prepared for:
Automata Network

Prepared by: **Joe Doyle and Jaime Iglesias**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jessica Nelson Project Manager
jessica.nelson@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Joe Doyle, Consultant
joe.doyle@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Project Timeline

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

Date	Event
November 27, 2024	Pre-project kickoff call
January 14, 2025	Status update meeting #1
January 21, 2025	Status update meeting #2
February 5, 2025	Delivery of report draft
February 5, 2025	Report readout meeting
February 28, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Automata Network engaged Trail of Bits to review the security of their Onchain PCCS and DCAP Attestation repositories.

A team of two consultants conducted the review from January 6 to January 30, 2025, for a total of eight engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the targets using automated and manual processes.

Observations and Impact

Overall, the codebase is well-structured, the APIs are clear, and the execution flow is relatively easy to follow. However, understanding the protocol requires extensive domain-specific knowledge (e.g., about Intel SGX, TEE, and X509 certificates), and the only available documentation is Intel's own reference documentation, the protocol's inline documentation, and some blog posts. This makes it difficult to quickly reason about certain design decisions or the protocol's implementation; we had to infer this information from reference documentation and from conversations with the client.

Most of the protocol's complexity stems from all of the data parsing that needs to be performed; this parsing not only involves multitude of steps, but also is a very delicate task, as any small mistake or divergence can lead to errors (such as the inability to parse some data) or bugs (validating incorrect data). Because of this, we strongly suggest spending development time building high-quality documentation and using it to drive additional testing.

The most severe issues we identified (TOB-AUT-5, TOB-AUT-6, and TOB-AUT-7) involve incorrect handling of certificate revocation and expiration. A comprehensive test suite should catch these issues, but the current test suite is extremely limited. The code maturity and automated testing sections provide suggestions for improving testing.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Automata Network take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Elaborate detailed documentation of the system.** The inline documentation, especially related to data structures and parsing functions, needs to be expanded.

Additionally, standalone, detailed documentation describing the system, the execution flows, and design decisions should be expanded.

- **Improve testing**. Unit tests need to be expanded significantly with both positive and negative test cases. In particular, existing test suites for common standards such as X.509 certificate verification should be ported to ensure coverage of common implementation errors. Finally, consider implementing fuzz tests; this can be done by differentially testing existing standard libraries against their smart contract counterparts and leveraging existing corpuses from previous open source fuzz work. Appendix C describes the differential fuzzing we performed.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	0
Low	1
Informational	8
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Data Validation	6
Error Reporting	1
Undefined Behavior	4

Project Targets

The engagement involved reviewing and testing the targets listed below.

Onchain PCCS

Repository	https://github.com/automata-network/automata-on-chain-pccs
Version	`352f4f69ae270823531a877e670ce7bffbf6a2c0`
Type	Solidity
Platform	Ethereum

DCAP Attestation

Repository	https://github.com/automata-network/automata-dcap-attestation
Version	`2d8b6b3dd35643081fa7fd98bbb1323a0601b2bb`
Type	Solidity
Platform	Ethereum

Project Coverage

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:

- While we did review some aspects of the `dcap-rs` library in the process of reviewing the DCAP contract's usage of zkVM-based attestation, we did not perform a thorough review of that library, nor of the RISC0 and SP1 programs that use it.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase is not math-heavy in nature; however, it does perform extensive byte manipulation when parsing data. While it often uses "battle-tested" third-party libraries (such as Solady's or ENS's), it also uses some "niche" libraries that have no tests (such as the Asn1Decoder) and implements others from scratch, such as the X.509-related libraries. These parsing operations generally lack documentation (e.g., about what type of inputs can be parsed and whether there are any unsupported cases, corner cases, or relevant nuances to the implementation). Additionally, although some testing is present, it needs to be expanded to include negative testing and, ideally, fuzzing. Finally, there already exist standard libraries written in other languages for the Asn1Decoder or the X.509 parser, so spending some time porting existing test cases from them would be very beneficial.	**Moderate**
Auditing	While most state-changing operations emit events, we identified several that do not (TOB-AUT-12).	**Moderate**
Authentication / Access Controls	Access controls are correctly implemented; however, there is no documentation indicating what the roles of the system are and who is meant to hold them. Finally, some of the functionality that manages permissions is implemented in a confusing way (TOB-AUT-3).	**Moderate**
Complexity	Both codebases are well-structured; they use libraries	**Moderate**

Management	and isolate functionality across different components with a clear intent of reusability, and they provide clear APIs with “generic primitives” that can be later expanded to support other use cases. However, there are instances in which attempts to create a generic base ultimately make the project unnecessarily complex. For example, the Onchain PCCS uses a generic storage layer for “data blobs,” which requires the creation of “middleware contracts” to encode and decode data whenever it is stored to or fetched from the generic storage (TOB-AUT-4). In contrast, whenever the data is parsed before being encoded and stored, it is parsed into a structured data type. We recommend a simplified design for storing data which has a consistent and unlikely-to-change format – e.g., Identity, Pck, Pcs, FmspcTcb. Currently, values in these formats are¡ stored as serialized “data blobs” via a generic data storage mechanism, although they could be stored directly as the data types. This would reduce the number of components and overall complexity and readability of the protocol.	
Cryptography and Key Management	ECDSA signature verification is done via existing libraries, and we found no issues in the codebase’s direct uses of cryptography. Key revocation is incorrectly checked in some cases (TOB-AUT-6), and we found issues that can cause the contract to check the wrong CRL or to accept certain expired certificates.	**Moderate**
Decentralization	The protocol uses a trusted setup.	**Not Applicable**
Documentation	The current documentation consists of several blog posts, inline documentation, and Intel’s own standard documentation and implementations. While this is enough to provide users an initial idea of what the protocol does, it is not enough to quickly get them up to speed with all the necessary domain-specific knowledge needed to navigate the protocol.	**Weak**

	Documentation for the expected execution flows of the system, documentation explaining how the attestation process works, and detailed documentation describing the data structures in use along with some examples is needed. This documentation would make it easier for developers to maintain the codebase and onboard engineers and users to it. This is especially important because extensive, domain-specific knowledge (e.g., about Intel SGX, X509 certificates, TEE, etc.) is needed to correctly understand the protocol.	
Low-Level Manipulation	All relevant assembly blocks are part of battle-tested third-party libraries (e.g., Solady). However, is it necessary to expand both the inline and standalone documentation to clearly identify what the code does and why whenever it invokes these libraries.	**Moderate**
Testing and Verification	While both codebases have positive test cases, these are not sufficiently extensive, and there is barely any negative testing if at all; unit tests need to be expanded. While creating effective testing for some of the components will require extensive engineering work, certain individual components can be isolated to help increase confidence in the protocol's behavior. With this in mind, isolating those components for which standard libraries already exist (e.g., X.509 parsing or ASN.1 encoding) and porting existing testing for them would be a good starting point for improving testing. Finally, we strongly recommend implementing fuzz tests whenever possible; there are corpuses available from existing fuzzing work for standard components (e.g., X.509 certificates) that can be leveraged to create differential testing between a standard library and a smart contract. See appendix C for the differential fuzzing we performed.	**Weak**
Transaction Ordering	The nature of the protocol prevents these types of issues.	**Not Applicable**

Summary of Findings

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

ID	Title	Type	Severity
1	Upsert functions allow inserting the same data	Data Validation	Informational
2	_onFetchDataFromResolver does not return an error	Error Reporting	Informational
3	Authorization mechanism is confusing	Undefined Behavior	Informational
4	Continuous encoding and decoding of data is confusing and error prone	Undefined Behavior	Informational
5	Certificate chain verification allows expired and irrelevant CRLs	Access Controls	High
6	Root CRL checks can be bypassed	Data Validation	High
7	Risc0- and sp1-based attestation may accept expired certificates	Data Validation	High
8	Constants have inconsistent endianness	Data Validation	Informational
9	CRL URI validation is inconsistent	Data Validation	Informational
10	Unclear definition and parsing of some header fields	Undefined Behavior	Low
11	No length checks can lead to panics	Data Validation	Informational

12	Some state-changing functions do not emit events	Auditing and Logging	**Informational**
13	Investigate failing differential fuzz tests	Undefined Behavior	**Undetermined**

Detailed Findings

1. Upsert functions allow inserting the same data

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AUT-1

Target:
`automata-pccs/src/bases/EnclaveIdentityDAO.sol`
`automata-pccs/src/bases/FmspcTcbDao.sol`
`automata-pccs/src/bases/PckDao.sol`
`automata-pccs/src/bases/PcsDao.sol`

Description
The upsert functions do not check whether the new identity is the same as the existing one.

For example, whenever the `upsertEnclaveIdentity` function is called, a number of checks are performed; for example, the identity string signature is checked against INTEL's TCB signing certificate.

```
function upsertEnclaveIdentity(uint256 id, uint256 version, EnclaveIdentityJsonObj
calldata enclaveIdentityObj)
    external
    returns (bytes32 attestationId)
{
    _validateQeIdentity(enclaveIdentityObj);
    (bytes32 key, bytes memory req) = _buildEnclaveIdentityAttestationRequest(id,
version, enclaveIdentityObj);
    bytes32 hash = sha256(bytes(enclaveIdentityObj.identityStr));
    attestationId = _attestEnclaveIdentity(req, hash, key);

    emit UpsertedEnclaveIdentity(id, version);
}
```

Figure 1.1: `upsertEnclaveIdentity` in `EnclaveIdentityDAO.sol#L88-L98`

Additionally, the data is parsed and validated in the `_buildEnclaveIdentityAttestationRequest` function. As shown in the figure below, some of the validation being performed involves checking the issuance date of the data or

its expiration date; however, another important check verifies that the new data is actually "newer" than the existing data.

```
function _buildEnclaveIdentityAttestationRequest(
    uint256 id,
    uint256 version,
    EnclaveIdentityJsonObj calldata enclaveIdentityObj
) private view returns (bytes32 key, bytes memory reqData) {
    IdentityObj memory identity =
EnclaveIdentityLib.parseIdentityString(enclaveIdentityObj.identityStr);
    if (id != uint256(identity.id)) {
        revert Enclave_Id_Mismatch();
    }

    if (id == uint256(EnclaveId.TD_QE) && version != 4 && version != 5) {
        revert Incorrect_Enclave_Id_Version();
    }

    if (block.timestamp < identity.issueDateTimestamp || block.timestamp >
identity.nextUpdateTimestamp) {
        revert Enclave_Id_Expired();
    }

    // make sure new collateral is "newer"
    key = ENCLAVE_ID_KEY(id, version);
    bytes memory existingData = _onFetchDataFromResolver(key, false);
    if (existingData.length > 0) {
        (IdentityObj memory existingIdentity, , ) =
            abi.decode(existingData, (IdentityObj, string, bytes));
        bool outOfDate = existingIdentity.tcbEvaluationDataNumber >
identity.tcbEvaluationDataNumber ||
            existingIdentity.issueDateTimestamp > identity.issueDateTimestamp;
        if (outOfDate) {
            revert Enclave_Id_Out_Of_Date();
        }
    }

    reqData = abi.encode(identity, enclaveIdentityObj.identityStr,
enclaveIdentityObj.signature);
}
```

Figure 1.2: `_buildEnclaveIdentityAttestationRequest` in `EnclaveIdentityDAO.sol#L127-L159`

However, as shown in the figure above, the new data is considered "out of date" (i.e., not "newer") only when the `tcbEvaluationDataNumber` or the `issueDateTimestamp` of the existing data is strictly higher than that of the new data, as this means that if the exact same data is posted again, the function will successfully update it.

In addition, if a certificate issuer at any point creates two certificates with the same issue timestamp, it would be possible to switch back and forth between those two certificates.

Note that while this does not lead to a security issue, it may lead to confusion in the case that multiple actors attempt to update the same data, and the emission of relevant events can lead to unnecessary gas expenditure.

Finally, note that this issue is also present in the other DAO contracts, particularly in the upsert functions that rely upon `_buildTcbAttestationRequest`, `_validatePck`, `_validatePcsCert`, and `_validatePcsCrl`.

Exploit Scenario

Alice and Bob each submit a transaction to update existing attestation data that is outdated. Alice's transaction is executed first, effectively updating the data; however, Bob's transaction is also executed successfully, leading to unnecessary gas expenditure and confusion in the case that some off-chain components are listening to attestation data update events.

Recommendations

Short term, add a check to prevent the same data from being inserted twice. This could be done by, for example, checking the new attestation data hash against the existing one. Note that it is best to perform this check as soon as possible (e.g., before the signature verification) to prevent unnecessary gas expenditure.

Long term, thoroughly document the intended behavior of the functions and in which cases the upsertion should fail.

2. _onFetchDataFromResolver does not return an error

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-AUT-2
Target: `automata-pccs/src/shared/AutomataDAOBase.sol`	

Description

The `_onFetchDataFromResolver` function does not return an error if the sender does not have reading rights; instead, it silently returns empty bytes.

```
function _onFetchDataFromResolver(bytes32 key, bool hash)
    internal
    view
    virtual
    override
    returns (bytes memory data)
{
    if (_callerIsAuthorized()) {
        data = super._onFetchDataFromResolver(key, hash);
    }
}
```

Figure 2.1: `_onFetchDataFromResolver` function in `AutomataDAOBase.sol#L17-L27`

Because of this behavior, a user calling an external function that relies on `_onFetchDataFromResolver`, such as the `getEnclaveIdentity` function, cannot tell the difference between a lack of available data available and lack of reading access to the data.

```
function getEnclaveIdentity(uint256 id, uint256 version)
    external
    view
    returns (EnclaveIdentityJsonObj memory enclaveIdObj)
{
    bytes memory attestedIdentityData = _onFetchDataFromResolver(ENCLAVE_ID_KEY(id,
version), false);
    if (attestedIdentityData.length > 0) {
        (, enclaveIdObj.identityStr, enclaveIdObj.signature) =
            abi.decode(attestedIdentityData, (IdentityObj, string, bytes));
    }
}
```

Figure 2.2: `getEnclaveIdentity` function in `EnclaveIdentityDAO.sol#L69-L79`

The example above shows that an empty struct will be returned in both cases, so the user cannot tell the difference between them.

Recommendations

Short term, revert when an unauthorized user attempts to access the system.

Long term, thoroughly document the intended behavior of function, and what should happen when an unauthorized user interacts with it or when no data is available.

3. Authorization mechanism is confusing

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-AUT-3
Target: `automata-pccs/src/shared/AutomataDAOStorage.sol`	

Description

The current implementation of the authorization mechanism is confusing, as it allows revoking individual writing permissions but only allows granting those permissions in bulk.

As we can see in the figures below, the owner has the ability to revoke and grant writing permissions.

```
function updateDao(address _pcsDao, address _pckDao, address _fmspcTcbDao, address
_enclaveIdDao)
    external
    onlyOwner
{
    _updateDao(_pcsDao, _pckDao, _fmspcTcbDao, _enclaveIdDao);
}

function revokeDao(address revoked) external onlyOwner {
    _authorized_writers[revoked] = false;
}
```

Figure 3.1: `updateDAO`, `revokeDAO` functions in `AutomataDAOStorage.sol#L48-L57`

However, when granting permissions the owner can only do so in bulk (i.e. to all DAOs at the same time), this flow is confusing as if individual DAOs may have their permissions revoked then it should also be possible to grant them individually.

```
function _updateDao(address _pcsDao, address _pckDao, address _fmspcTcbDao, address
_enclaveIdDao) private {
    _authorized_writers[_pcsDao] = true;
    _authorized_writers[_pckDao] = true;
    _authorized_writers[_fmspcTcbDao] = true;
    _authorized_writers[_enclaveIdDao] = true;
}
```

Figure 3.2: `_updateDAO` function in `AutomataDAOStorage.sol#L96-L101`

What makes this confusing is that, in the case an individual DAO has its permissions revoked and a new DAO has to be incorporated, then both the current DAOs and the new one will be granted permissions (even if it is a no-op for the existing ones).

Recommendations

Short term, make it possible to grant roles individually.

Long term, thoroughly document the intended behavior of the authorization flow.

4. Continuous encoding and decoding of data is confusing and error prone

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-AUT-4
Target: Across the protocol	

Description

Multiple structured representations of the collateral data exist within the system, such as the `IdentityObj`, `EnclaveIdentityJsonObj`, and `TcbInfoBasic` (figure 4.1). However, most of the data is encoded and stored as blobs of bytes (figure 4.2).

```
struct EnclaveIdentityJsonObj {
    string identityStr;
    bytes signature;
}

/// @dev Full Solidity Object representation of Identity.json
struct IdentityObj {
    EnclaveId id;
    uint32 version;
    uint64 issueDateTimestamp; // UNIX Epoch Timestamp in seconds
    uint64 nextUpdateTimestamp; // UNIX Epoch Timestamp in seconds
    uint32 tcbEvaluationDataNumber;
    bytes4 miscselect;
    bytes4 miscselectMask;
    bytes16 attributes;
    bytes16 attributesMask;
    bytes32 mrsigner;
    uint16 isvprodid;
    Tcb[] tcb;
}
```

Figure 4.1: Identity `structs in EnclaveIdentityHelper.sol#L20-L39`

```
function attest(bytes32 key, bytes calldata attData, bytes32 attDataHash)
    external
    override
    onlyDao(msg.sender)
    returns (bytes32 attestationId, bytes32 hashAttestationid)
{
    attestationId = _computeAttestationId(key, false);
    _db[attestationId] = attData;

    // this makes storing hash optional
    if (attDataHash != bytes32(0)) {
```

```
        hashAttestationid = _computeAttestationId(key, true);
        _db[hashAttestationid] = abi.encodePacked(attDataHash);
    }
}
```

Figure 4.2: attest function in AutomataDAOStorage.sol#L80-L94

In contrast, as shown in the figure below, whenever the data is validated before insertion, the aforementioned structured representations are used. For example, when upserting identity data during the parsing of the `IdentityJsonObj` in the `_buildEnclaveIdentityAttestationRequest` function, the `IdentityObj` data structure is populated by parsing the `EnclaveIdentityJsonObj`.

```
function _buildEnclaveIdentityAttestationRequest(
    uint256 id,
    uint256 version,
    EnclaveIdentityJsonObj calldata enclaveIdentityObj
) private view returns (bytes32 key, bytes memory reqData) {
    IdentityObj memory identity =
EnclaveIdentityLib.parseIdentityString(enclaveIdentityObj.identityStr);
    if (id != uint256(identity.id)) {
        revert Enclave_Id_Mismatch();
    }

    if (id == uint256(EnclaveId.TD_QE) && version != 4 && version != 5) {
        revert Incorrect_Enclave_Id_Version();
    }

    if (block.timestamp < identity.issueDateTimestamp || block.timestamp >
identity.nextUpdateTimestamp) {
        revert Enclave_Id_Expired();
    }

    // make sure new collateral is "newer"
    key = ENCLAVE_ID_KEY(id, version);
    bytes memory existingData = _onFetchDataFromResolver(key, false);
    if (existingData.length > 0) {
        (IdentityObj memory existingIdentity, , ) =
            abi.decode(existingData, (IdentityObj, string, bytes));
        bool outOfDate = existingIdentity.tcbEvaluationDataNumber >
identity.tcbEvaluationDataNumber ||
            existingIdentity.issueDateTimestamp > identity.issueDateTimestamp;
        if (outOfDate) {
            revert Enclave_Id_Out_Of_Date();
        }
    }

    reqData = abi.encode(identity, enclaveIdentityObj.identityStr,
enclaveIdentityObj.signature);
}
```

Figure 4.3: `_buildEnclaveIdentityAttestationRequest` in `EnclaveIdentityDAO.sol#L127-L159`

However, these data structures are encoded for storing while they are being used, and are then decoded whenever the information is accessed, as in the `getEnclaveIdentity` function. In fact, we can see that even though only some of the data is needed (the `identityStr` and `signature`), all of the data is accessed and decoded (figure 4.4).

```
function getEnclaveIdentity(uint256 id, uint256 version)
    external
    view
    returns (EnclaveIdentityJsonObj memory enclaveIdObj)
{
    bytes memory attestedIdentityData = _onFetchDataFromResolver(ENCLAVE_ID_KEY(id,
version), false);
    if (attestedIdentityData.length > 0) {
        (, enclaveIdObj.identityStr, enclaveIdObj.signature) =
            abi.decode(attestedIdentityData, (IdentityObj, string, bytes));
    }
}
```

Figure 4.4: `getEnclaveIdentity` function in `EnclaveIdentityDAO.sol#L69-L79`

This is done to create a very generic storage contract that allows storing arbitrary collateral and therefore supports multiple use-cases. However, since the collateral types currently appear well defined and understood, this approach increases the codebase's complexity, as it requires creating such "middleware" contracts to help encode and decode the data correctly before inserting into or reading from storage.

Note that this is being done across all DAOs.

Finally, because data is being encoded and decoded in multiple places, developers must have a strong understanding of the system to make modifications, as changes in encoding or decoding of the data could lead to future issues.

Recommendations

Short term, store data in a structured manner, at least for data types that are well understood, widely used, and unlikely to change.

Long term, thoroughly document the used data structures, what each data entry in the storage contract represents, and the reason for storing that particular data.

5. Certificate chain verification allows expired and irrelevant CRLs

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-AUT-5
Target: `automata-dcap-attestation/contracts/bases/X509ChainBase.sol`	

Description

When an X.509 certificate must be invalidated before its expiration time, its serial number is added to a Certificate Revocation List (CRL) associated with the issuing certificate. When validating a certificate chain, software must check each certificate in the chain for both expiration and for revocation. Failure to do so would allow an attacker who has compromised a certificate to continue using it even after the issuer revokes it.

Attestations are signed with an attestation key, which is included via a hash in the QE report data structure, which itself is signed by the first key in the certificate chain. The certificate chain is then verified through the `X509ChainBase.verifyCertChain` method. As shown in figure 5.1, to check that a certificate has not been revoked, this method retrieves corresponding CRLs from the PCCS contract, then checks the certificate's serial number against those CRLs.

```
bytes memory crl;
if (i == n - 2) {
    (, crl) = pccsRouter.getCrl(CA.ROOT);
} else if (i == 0) {
    string memory issuerName = certs[i].issuerCommonName;
    if (LibString.eq(issuerName, PLATFORM_ISSUER_NAME)) {
        (, crl) = pccsRouter.getCrl(CA.PLATFORM);
    } else if (LibString.eq(issuerName, PROCESSOR_ISSUER_NAME)) {
        (, crl) = pccsRouter.getCrl(CA.PROCESSOR);
    } else {
        return false;
    }
}
if (crl.length > 0) {
    certRevoked = crlHelper.serialNumberIsRevoked(certs[i].serialNumber, crl);
}
if (certRevoked) {
    break;
}
```

Figure 5.1: Fetching and checking CRLs during certificate chain verification (automata-dcap-attestation/contracts/bases/X509ChainBase.sol#78–96)

However, no additional validation of the CRL is performed, so this method relies on the PCCS contract to revert if `getCrl` is called for an expired CRL, or a CRL for a different issuing certificate. However, the `getCrl` function calls the method shown in figure 5.2, which performs no validation except ensuring that the returned bytes are nonempty.

```
function _getPcsAttestationData(CA ca, bool crl) private view returns (bool valid,
bytes memory ret) {
    PcsDao pcsDao = PcsDao(pcsDaoAddr);
    ret = pcsDao.getAttestedData(pcsDao.PCS_KEY(ca, crl));
    valid = ret.length > 0;
    if (!valid) {
        if (crl) {
            revert CrlNotFound(ca);
        } else {
            revert CertNotFound(ca);
        }
    }
}
```

Figure 5.2: The only validation performed when fetching a CRL is to ensure that the CRL data is nonempty (automata-dcap-attestation/contracts/PCCSRouter.sol#238–249)

In addition, the PCCS contract can return CRLs that correspond to previous certificates that are no longer in the PCCS contract. In the logic shown in figure 5.3, updating a certificate does not delete the corresponding CRL. This can cause certificate verification to use a CRL which was created for a previous version of the issuer's certificate, potentially allowing an attacker to use a revoked certificate in the process.

```
function upsertPcsCertificates(CA ca, bytes calldata cert) external returns (bytes32
attestationId) {
    (bytes32 hash, bytes32 key) = _validatePcsCert(ca, cert);
    attestationId = _attestPcs(cert, hash, key);

    emit UpsertedPCSCollateral(ca, false);
}
```

Figure 5.3: When upserting a certificate, the PCS entry corresponding to the certificate is updated via `_attestPcs`, but the CRL is not updated (automata-on-chain-pccs/src/bases/PcsDao.sol#108–113)

This also affects the RISC0- and sp1-based quote verification, but we have not evaluated whether additional checks performed by the `dcap-rs` library prevent exploitation.

Exploit Scenario

Alice compromises a certificate issued by the Intel SGX PCK Platform CA. Her compromise is detected, and Intel publishes a new CRL containing that certificate's serial number. The Automata Network team does not immediately update the CRL in the contract, and after that CRL has expired, Alice publishes an attestation which should be invalid.

Recommendations

Short term, add checks to ensure that CRLs are up to date and match to the issuer's certificate when fetching CRLs from the PCCS contract.

Long term, document the expected update procedures for time-sensitive data such as CRLs and certificates, and ensure that expired data cannot be used.

6. Root CRL checks can be bypassed

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AUT-6
Target: `automata-dcap-attestation/contracts/bases/X509ChainBase.sol`	

Description

Certificate chain verification is performed by the `verifyCertChain` method, shown below in figure 6.1.

```
function verifyCertChain(IPCCSRouter pccsRouter, address crlHelperAddr, X509CertObj[] memory
certs)
    internal
    view
    returns (bool)
{
    X509CRLHelper crlHelper = X509CRLHelper(crlHelperAddr);
    uint256 n = certs.length;
    bool certRevoked;
    bool certNotExpired;
    bool verified;
    bool certChainCanBeTrusted;
    for (uint256 i = 0; i < n; i++) {
        X509CertObj memory issuer;
        if (i == n - 1) {
            // rootCA
            issuer = certs[i];
        } else {
            issuer = certs[i + 1];
            bytes memory crl;
            if (i == n - 2) {
                (, crl) = pccsRouter.getCrl(CA.ROOT);
            } else if (i == 0) {
                string memory issuerName = certs[i].issuerCommonName;
                if (LibString.eq(issuerName, PLATFORM_ISSUER_NAME)) {
                    (, crl) = pccsRouter.getCrl(CA.PLATFORM);
                } else if (LibString.eq(issuerName, PROCESSOR_ISSUER_NAME)) {
                    (, crl) = pccsRouter.getCrl(CA.PROCESSOR);
                } else {
                    return false;
                }
            }
            if (crl.length > 0) {
                certRevoked = crlHelper.serialNumberIsRevoked(certs[i].serialNumber, crl);
            }
            if (certRevoked) {
                break;
            }
        }
```

```
        certNotExpired = block.timestamp > certs[i].validityNotBefore && block.timestamp <
certs[i].validityNotAfter;
        if (!certNotExpired) {
            break;
        }

        {
            verified = ecdsaVerify(sha256(certs[i].tbs), certs[i].signature,
issuer.subjectPublicKey);
            if (!verified) {
                break;
            }
        }

        bytes32 issuerPubKeyHash = keccak256(issuer.subjectPublicKey);

        if (issuerPubKeyHash == ROOTCA_PUBKEY_HASH) {
            certChainCanBeTrusted = true;
            break;
        }
    }
    return !certRevoked && certNotExpired && verified && certChainCanBeTrusted;
}
```

Figure 6.1: The `verifyCertChain` method (automata-dcap-attestation/contracts/bases/X509ChainBase.sol#60–119)

The two highlighted sections show two different ways of determining when the issuer is the root CA. In the first line, the issuer is the root CA when it is the last entry in the list. In the second, the issuer is the root CA when its public key matches the `ROOTCA_PUBKEY_HASH` constant. By constructing a certificate chain where the root CA certificate is not the last entry in the chain, it is possible to cause this method to exit early, skipping all validation of certificates past the root CA, and bypassing the root CRL check entirely.

To illustrate the failure, we can modify the implementation of `V4QuoteVerifier._verifyCommon` to modify the certificate chain and its serial numbers, as shown below in figure 6.2.

```
// Step 2: Fetch FMSPC TCB
X509CertObj[] memory parsedCerts_orig =
authData.qeReportCertData.certification.pck.pckChain;
X509CertObj[] memory parsedCerts = new X509CertObj[](parsedCerts_orig.length+1);

for(uint i = 0; i < parsedCerts_orig.length; i++) {
    parsedCerts[i] = parsedCerts_orig[i];
}

parsedCerts[parsedCerts_orig.length] = parsedCerts[parsedCerts_orig.length-1];

parsedCerts[parsedCerts_orig.length-2].serialNumber =
    283956722097140113082 89;
parsedCerts_orig[parsedCerts_orig.length-2].serialNumber =
```

```
    283956722097140113082 89;
parsedCerts = parsedCerts_orig;
```

Figure 6.2: A modified snippet which demonstrates the missing CRL validation (contracts/verifiers/V4QuoteVerifier.sol#160–174)

In the existing test cases, serial number `283956722097140113082 89` is in the root CRL, and so if the second-to-last certificate's serial number matches that, it should be rejected. If we copy the certificate chain into a new array, and duplicate the root certificate so that the signature checks still pass, the CRL check is never triggered. With the highlighted line present, the test `testTDXQuoteV4OnChainAttestation` fails, and when it is removed, that test incorrectly passes.

The early-exit logic when encountering the end of the loop also skips checking the expiration and self-signature of the root CA. We do not consider this to be a serious security concern, since the current certificate will not expire until 2049; however, we still encourage checking this data.

Exploit Scenario

Alice gains access to a previously compromised platform certificate issued by the Intel SGX Root CA, which is present on the root CA's CRL. Using this certificate, she signs a fraudulent quote, and submits it to the DCAP contract, bypassing the CRL checks using the method described above.

Recommendations

Short term, improve the implementation of `verifyCertChain` to consistently handle the root certificate, for example by checking that the root certificate appears exactly in the final entry in the array.

Long term, develop tests to cover all cryptographic verification functions. Especially focus on negative tests that cover edge cases that should fail, such as certificate chains with extraneous certificates and certificate chains with revoked certificates.

7. Risc0- and sp1-based attestation may accept expired certificates

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AUT-7
Target: `contracts/verifiers/{V3QuoteVerifier.sol,V4QuoteVerifier.sol}`	

Description

In addition to Solidity-based DCAP verification, the `AttestationEntrypointBase` contract can verify quotes by running Rust programs using the `dcap-rs` library within the RISC0 and Succinct SP1 zkVM proof systems. These proofs include public data, such as the hashes of various certificates and CRLs, which must be checked to ensure that proofs generated using old data are not accepted.

This validation occurs in the `verifyZkOutput` methods of the V3 and V4 quote verifiers, shown in figures 7.1 and 7.2.

```
function verifyZkOutput(bytes calldata outputBytes)
    external
    view
    override
    returns (bool success, bytes memory output)
{
    uint256 offset = 2 + uint16(bytes2(outputBytes[0:2]));
    success = checkCollateralHashes(offset + 72, outputBytes);
    if (success) {
        output = outputBytes[2:offset];
    } else {
        output = bytes("Found one or more collaterals mismatch");
    }
}
```

Figure 7.1: V3 quote zkVM output verification (automata-dcap-attestation/contracts/verifiers/V3QuoteVerifier.sol#15–28)

```
function verifyZkOutput(bytes calldata outputBytes)
    external
    view
    override
    returns (bool success, bytes memory output)
{
    bytes4 teeType = bytes4(outputBytes[4:8]);
    if (teeType != SGX_TEE && teeType != TDX_TEE) {
        return (false, bytes("Unknown TEE type"));
    }
```

```
    uint256 offset = 2 + uint16(bytes2(outputBytes[0:2]));

    success = checkCollateralHashes(offset + 72, outputBytes);
    if (success) {
        output = outputBytes[2:offset];
    } else {
        output = bytes("Found one or more collaterals mismatch");
    }
}
```

Figure 7.2: V4 quote zkVM output verification (automata-dcap-attestation/contracts/verifiers/V4QuoteVerifier.sol#30–49)

However, the timestamp used for checking these attestations, which appears in the 72 bytes between the "output" and "collateral hashes" portions of the `outputBytes` array, is not externally validated. If an attacker compromises a certificate that has been expired for long enough that it has been removed from the issuer's CRL, then it would be possible to verify quotes signed by that compromised certificate by using an old timestamp for the proof generation.

This would be mitigated by a check in the zkVM program that confirms that the timestamp is after the issue date of the CRL, since that would require the timestamp used to occur after the certificate has already expired. It does not appear that the `dcap-rs` library performs this check. However, since the `dcap-rs` library is not in scope for this review, we have not performed an exhaustive investigation, and we have not ruled out the possibility that other checks performed by that library suffice to prevent this attack.

Exploit Scenario

Alice compromises a certificate issued by the Intel SGX PCK Platform CA. Her attack is detected, and that certificate is revoked. However, after that certificate has expired, Intel publishes a CRL that no longer includes the compromised certificate. Alice then crafts a zkVM-based attestation that uses a timestamp from before the expiration, and publishes it to the Automata Network DCAP contract.

Recommendations

Short term, implement checks to ensure that zkVM-based attestations are accepted only while their inputs remain valid. For example, `dcap-rs` could be modified to check a window of potential timestamps rather than a single timestamp, and the current timestamp could be compared to that window in the DCAP contract.

Long term, review and test both the on-chain and zkVM-based attestation checks to ensure that they behave identically and that maliciously crafted inputs will be rejected.

8. Constants have inconsistent endianness

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-AUT-8
Target: `automata-dcap-attestation/contracts/types/Constants.sol`	

Description

Several fields in the quote header data structure are expected to match constants defined in `Constants.sol`, shown below in figure 8.1. The byte ordering of these is not consistent between constants, which may lead to confusion and errors as development progresses.

```
bytes2 constant SUPPORTED_ATTESTATION_KEY_TYPE = 0x0200; // ECDSA_256_WITH_P256_CURVE (LE)
bytes4 constant SGX_TEE = 0x00000000;
bytes4 constant TDX_TEE = 0x00000081;
bytes16 constant VALID_QE_VENDOR_ID = 0x939a7233f79c4ca9940a0db3957f0607;
```

Figure 8.1: header-related constants with different endianness (automata-dcap-attestation/contracts/types/Constants.sol#6–9)

All of these constants are specified using Solidity's hexadecimal number notation, and then converted to byte-arrays in big-endian format. For the constants `SUPPORTED_ATTESTATION_KEY_TYPE` and `VALID_QE_VENDOR_ID`, these are directly compared against the bytes present in the header when being parsed. The `SGX_TEE` and `TDX_TEE` constants are options for the `teeType` field, but the bytes are in the opposite order in which the underlying data will appear. These constants are not directly compared against the header byte sequence, but instead compared against the byte array converted by the line shown below in figure 8.2; first they are parsed through the `leBytesToBeUint` function, then converted to a `uint32`, then converted to a `bytes4`. The effect of this parsing is to reverse the order of bytes, thus matching the reversed byte order of the constants.

```
teeType: bytes4(uint32(BELE.leBytesToBeUint(rawQuote[4:8]))),
```

Figure 8.2: The `teeType` field is set by byte-reversing bytes 4 through 8 of `rawQuote`. (automata-dcap-attestation/contracts/AttestationEntrypointBase.sol#170)

Recommendations

Short term, choose a consistent byte order for constants.

Long term, document the expected data formats explicitly, and consider bypassing endianness concerns by representing integer fields with integer types instead of byte arrays.

9. CRL URI validation is inconsistent

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-AUT-9
Target: `automata-dcap-attestation/contracts/bases/X509ChainBase.sol`	

Description

X.509 certificates can specify the URIs from which their CRLs can be retrieved via the `CRLDistributionPoints` extension field. In the `dcap-rs` implementation of certificate chain verification, each certificate is checked for revocation via the `IntelSgxCrls::is_cert_revoked` and `check_pck_issuer_and_crl` functions, shown below in figures 9.1 and 9.2.

```
pub fn is_cert_revoked(&self, cert: &X509Certificate) -> bool {
    let crl = match get_crl_uri(cert) {
        Some(crl_uri) => {
            if
crl_uri.contains("https://api.trustedservices.intel.com/sgx/certification/v3/pckcrl?
ca=platform")
                ||
crl_uri.contains("https://api.trustedservices.intel.com/sgx/certification/v4/pckcrl?
ca=platform") {
                self.sgx_pck_platform_crl.as_ref()
            } else if
crl_uri.contains("https://api.trustedservices.intel.com/sgx/certification/v3/pckcrl?
ca=processor")
                ||
crl_uri.contains("https://api.trustedservices.intel.com/sgx/certification/v4/pckcrl?
ca=processor") {
                self.sgx_pck_processor_crl.as_ref()
            } else if
crl_uri.contains("https://certificates.trustedservices.intel.com/IntelSGXRootCA.der"
) {
                self.sgx_root_ca_crl.as_ref()
            } else {
                panic!("Unknown CRL URI: {}", crl_uri);
            }
        },
        None => {
            panic!("No CRL URI found in certificate");
        }
    }.unwrap();

    // check if the cert is revoked given the crl
    is_cert_revoked(cert, crl)
```

```
}
```

Figure 9.1: One certificate revocation checks in `dcap-rs` (`dcap-rs/src/types/cert.rs#74–96`)

```
fn check_pck_issuer_and_crl(
    pck_cert: &X509Certificate,
    pck_issuer_cert: &X509Certificate,
    intel_crls: &IntelSgxCrls,
) -> bool {
    // we'll check what kind of cert is it, and validate the appropriate CRL
    let pck_cert_subject_cn = get_x509_issuer_cn(pck_cert);
    let pck_cert_issuer_cn = get_x509_subject_cn(pck_issuer_cert);

    assert!(
        pck_cert_issuer_cn == pck_cert_subject_cn,
        "PCK Issuer CN does not match with PCK Intermediate Subject CN"
    );

    match pck_cert_issuer_cn.as_str() {
        "Intel SGX PCK Platform CA" => verify_crl(
            intel_crls.sgx_pck_platform_crl.as_ref().unwrap(),
            pck_issuer_cert,
        ),
        "Intel SGX PCK Processor CA" => verify_crl(
            &intel_crls.sgx_pck_processor_crl.as_ref().unwrap(),
            pck_issuer_cert,
        ),
        _ => {
            panic!("Unknown PCK Cert Subject CN: {}", pck_cert_subject_cn);
        }
    }
}
```

Figure 9.2: Another certificate revocation check in `dcap-rs`, which is only applied to the first certificate (`dcap-rs/src/utils/quotes/mod.rs#185–212`)

Any certificate that does not match one of the specified URI patterns will be rejected in zkVM-based attestation. However, in the smart contract setting, the CRL is selected based on the issuer name and the position in the certificate chain, but no URI checking is performed.

If the CRL URIs change in the future, the zkVM-based attestation would stop working. This would lead to a partial denial of service, and may cause confusion to users.

Recommendations

Short term, decide whether the CRL URIs should be checked, and modify both verification methods so that they behave the same way.

Long term, clearly specify the verification requirements for attestations, and build tests that exercise each major check, with both success and failure cases.

10. Unclear definition and parsing of some header fields

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-AUT-10
Target: `contracts/bases/AttestationEntryPointBase.sol`	

Description

Intel's SGX reference documentation for "Data Center Attestation Primitives (DCAP): ECDSA Quote Library API" and "Intel's TDX DCAP: Quote Generation Library and Quote Verification library" define the APIs and data structures used when performing attestations.

These documents describe, amongst other things, the expected header structure, data types, and expected values of each one of the fields.

A.3.1. TD Quote Header

Name	Size (bytes)	Type	Description
Version	2	Integer	Version of the *Quote* data structure. • *Value*: 4
Attestation Key Type	2	Integer	Type of the *Attestation Key* used by the *Quoting Enclave*. • *Supported values*: 2 (ECDSA-256-with-P-256 curve) 3 (ECDSA-384-with-P-384 curve) (*Note: currently not supported*) (Note: 0 and 1 are reserved, for when EPID is moved to version 4 quotes.)
TEE Type	4	Integer	TEE for this Attestation 0x00000000: SGX

Intel® Trust Domain Extensions Data Center Attestation Primitives (Intel® TDX DCAP): Quote Generation Library and Quote Verification Library

Figure 10.1: TD Quote Header structured in defined in Intel's documentation (A.3.1).

Table 2: High-Level Quote Structure

Name	Size (bytes)	Type	Description
Version	2	Integer	Version of the *Quote* data structure. • *Value*: 3
Attestation Key Type	2	Integer	Type of the *Attestation Key* used by the *Quoting Enclave*. • *Supported values*: - 2 (ECDSA-256-with-P-256 curve) - 3 (ECDSA-384-with-P-384 curve) (*Note: currently not supported*) (Note: 0 and 1 are reserved, EPID is moved to version 3 quotes.)
Reserved	4	Byte Array	Reserved field. • *Value*: 0

Figure 10.2: Quote Header structured in defined in Intel's documentation (A.4).

In the figures above, we can see some of the fields—particularly the Attestation Key Type and TEE Type fields, which are both of type Integer and of size 2 and 4 bytes, respectively. Note that the second image does not specify the TEE Type field; however, it reserves 4 bytes that can be used for the TEE Type.

Now, if we look at the Header structure defined in the DCAP Attestation repository, we can see that both fields are defined as bytes.

```
struct Header {
    uint16 version;
    bytes2 attestationKeyType;
    bytes4 teeType;
    bytes2 qeSvn;
    bytes2 pceSvn;
    bytes16 qeVendorId;
    bytes20 userData;
}
```

Figure 10.3: Header struct in `CommonStruct.sol#L8-L16`

This discrepancy is not particularly important, but there is a nuance: per Intel's documentation, the Integers in the header (and other structs) are little-endian encoded (for reference, Solidity integers are big-endian encoded), so it is important to be aware of this "endianness" when dealing with values like these.

The issue does not stem from the fact that the values are being treated as bytes, since as long as the "endianness" is taken into account when dealing with these values (e.g., when comparing them to Solidity Integers), it will not be a problem; rather, the problem comes when the raw header is parsed:

```
function _parseQuoteHeader(bytes calldata rawQuote) private pure returns (Header
memory header) {
    bytes2 attestationKeyType = bytes2(rawQuote[2:4]); // 2bytes
    bytes2 qeSvn = bytes2(rawQuote[8:10]); // 2 bytes
    bytes2 pceSvn = bytes2(rawQuote[10:12]); // 2 bytes
    bytes16 qeVendorId = bytes16(rawQuote[12:28]); // 16 bytes

    header = Header({
        version: uint16(BELE.leBytesToBeUint(rawQuote[0:2])), // 2bytes
        attestationKeyType: attestationKeyType,
        teeType: bytes4(uint32(BELE.leBytesToBeUint(rawQuote[4:8]))), // 4bytes
        qeSvn: qeSvn,
        pceSvn: pceSvn,
        qeVendorId: qeVendorId,
        userData: bytes20(rawQuote[28:48])
    });
}
```

Figure 10.4: `_parseQuoteHeader` struct in `AttestationEntrypointBase.sol#L159-L174`

As shown above, even though both fields are defined as bytes, the TEE Type has its endianness reversed by the call to `BELE.leBytesToBeUint`, which effectively creates a header that has certain integer values little-endian encoded and others big-endian encoded.

This is not only confusing, but also error prone: a data structure is being created with different encodings, and this data structure is then logged with an event and could lead to errors if off-chain components are listening to these events.

Note that these are not the only fields defined as Integers in the documentation.

Exploit Scenario

A third-party protocol integrates with Automata Network to perform quote verifications through the DCAP Attestation contracts. Whenever the protocol needs to verify a quote, they store off-chain the raw quote and the output of the verification.

As a final verification step, the protocol parses the raw quote to extract the header and compare it to Automata Network's verification outcome; this leads to an error, as the headers are different.

Recommendations

Short term, consider whether reversing the endianness of the Integer fields is needed and why; additionally, consider how this will impact users and protocol consuming parsed data from Automata Network.

Long term, thoroughly document the data structures being used and how and why they should be parsed.

11. No length checks can lead to panics

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AUT-11
Target: contracts/bases/`AttestationEntryPointBase.sol`	

Description
There are some instances in which arrays are accessed without length checks, which can lead to an out-of-bounds panic. Instead, a length check should be performed, and the contract should throw an error.

```
function _parseQuoteHeader(bytes calldata rawQuote) private pure returns (Header
memory header) {
    bytes2 attestationKeyType = bytes2(rawQuote[2:4]); // 2bytes
    bytes2 qeSvn = bytes2(rawQuote[8:10]); // 2 bytes
    bytes2 pceSvn = bytes2(rawQuote[10:12]); // 2 bytes
    bytes16 qeVendorId = bytes16(rawQuote[12:28]); // 16 bytes

    header = Header({
        version: uint16(BELE.leBytesToBeUint(rawQuote[0:2])), // 2bytes
        attestationKeyType: attestationKeyType,
        teeType: bytes4(uint32(BELE.leBytesToBeUint(rawQuote[4:8]))), // 4bytes
        qeSvn: qeSvn,
        pceSvn: pceSvn,
        qeVendorId: qeVendorId,
        userData: bytes20(rawQuote[28:48])
    });
}
```

Figure 11.1: `_parseQuoteHeader` struct in `AttestationEntrypointBase.sol#L159-L174`

As shown in the figure above, the `_parseQuoteHeader` function parses a raw quote without performing any length checks, which will lead to an out-of-bounds panic in the case that the length of the quote is lower than the position in the array being accessed.

Exploit Scenario
A bug in a protocol integrating with Automata Network leads to the submission of an empty quote. During the verification process, the DCAP Attestation contract will panic; however, because Automata Network does not return an explicit error, debugging the error takes more time than necessary.

Recommendations
Short term, perform length checks before accessing arbitrary array members.

12. Some state-changing functions do not emit events

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-AUT-12
Target: Across the protocol	

Description
Contracts should emit events for state-changing operations to aid off-chain code (e.g., to identify whether important protocol-parameters have been changed).

We identified a number of state-changing functions that do not emit events:

- `upsertPlatformTcbs`
- `setQuoteVerifier`
- `setZKConfiguration`
- `setAuthorized`
- `enableCallerRestriction`
- `disableCallerRestriction`
- `setConfig`

Recommendations
Short term, implement events for state-changing operations.

Long term, thoroughly document all state-changing operations.

13. Investigate failing differential fuzz tests

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-AUT-13
Target: `src/helpers/X509Helper.sol`	

Description

As mentioned in appendix C, during the review we developed a preliminary differential fuzzing harness between Automata Network's X509Helper library and Golang's X509 standard library to try and identify implementation errors or edge-cases. While running it, we identified some cases for which the results offered by both implementations diverged; however, we were not able to fully triage these results and attribute them to an actual bug in Automata Network's implementation.

These cases need to be further triaged to identify whether or not they are a result of intended behavior. We have provided the source code for this harness to the Automata Network team to assist them in improved testing.

Finally, note that the reason this finding is of "Undetermined" severity is because further triaging is needed to confirm whether this is due to an implementation bug.

Recommendations

Short term, triage the aforementioned cases to identify whether they are due to an implementation bug.

Long term, continue running the differential fuzzer and triaging any failing cases to identify possible bugs in the implementation. Additionally, if possible, consider extending the differential fuzzing to other parts of the codebase.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity, 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 Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeded industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category does not apply to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Differential Fuzzing

During the review, we developed preliminary differential fuzzing between an external X509 certificate parsing library and Automata Network's X509Helper library used to parse X509 certificates. Since X509 certificates are widely adopted, it is relatively simple to find reference implementations in different languages.

For this differential fuzzing, we chose to use Golang's X509 standard library, primarily because it allowed us to leverage Golang's native fuzzing (go-fuzz) and go-ethereum's abigen to easily test the contract against the Go implementation. We also leveraged some open-source go-fuzz corpuses to seed the fuzzer.

The main idea behind the differential testing is to compare how each of the parsing library behaves when parsing the same certificate:

- Are there certificates that one implementation can parse and the other cannot?
- Are there any divergences in the outputs of each one of the implementations?

However, there are some obstacles to overcome derived from the difference between both implementations:

- Automata Network's implementation does not support certain certificates; for example, it only supports ECDSA signatures and p256 key algorithms, and it does not support compressed public keys.
- Golang's reference implementation does not output the parsed certificate exactly how Automata Network's library does. For example, Golang's implementation will return a ASN1 DER Encoded signature from the parsing, while Automata Network's will return the decoded signature.

To avoid these issues, we restricted the input certificates to ones that Automata Network's implementation should be able to parse, and we manually converted Golang's outputs to match those expected by Automata Network's implementation:

```
// Fuzz function
f.Fuzz(func(t *testing.T, data []byte) {
	// parse certificate using standard library
	referenceParsing, err := x509.ParseCertificate(data)
	[...]

	// Parsing contract only supports ECDSA signature algorithm and p256 key
algorithm.
	if referenceParsing.PublicKeyAlgorithm != x509.ECDSA ||
referenceParsing.SignatureAlgorithm != x509.ECDSAWithSHA256 {
		return // invalid input, not supported by the contract
```

```
	}

	// The signature is still ASN1 encoded, we need to decode it
	var signature ECDSASignature
	_, err = asn1.Unmarshal(referenceParsing.Signature, &signature)
	if err != nil {
		t.Log(err)
		panic("this should never happen")
	}
```

Figure C.1: Part of the X509Helper harness

While we were not able to confirm issues, we did uncover failing cases—in particular, we identified certificates that could not be parsed by the standard library but could be by Automata Network's, and we identified discrepancies in the ways both implementations parse the Issuer and Subject's "Common Name" fields.

These inputs need to be triaged to identify whether there is an actual bug in Automata Network's implementation.

D. Non-Security-Related Recommendations

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

- The following comment should be rephrased to clarify that this function is retrieving data, not storing it:

```
/**
 * @dev SHOULD store the hash of a collateral (e.g. X509 Cert, TCBInfo JSON etc) in
the attestation registry
 * as a separate attestation from the collateral data itself
 */
function getCollateralHash(bytes32 key) external view returns (bytes32
collateralHash) {
```

Figure D.1: The `getCollateralHash` function in `src/bases/DAOBase.sol#L27-L31`

- Consider making two separate functions (one for the hash and one for the pointer) instead of using the flag in figure D.2. This would make the code more readable, even if at the cost of a bit of code duplication.

```
function _fetchDataFromResolver(bytes32 key, bool hash) internal view returns (bytes
memory) {
    bytes32 attestationId;
    if (hash) {
        attestationId = resolver.collateralHashPointer(key);
    } else {
        attestationId = resolver.collateralPointer(key);
    }
    return resolver.readAttestation(attestationId);
}
```

Figure D.2: The `_fetchDataFromResolved` function in `src/bases/DAOBase.sol#L42-L50`

- It is unclear why both of these functions need to exist, as they do exactly the same thing (one calls the other). Consider using only one function.

```
function _fetchDataFromResolver(bytes32 key, bool hash) internal view returns (bytes
memory) {
    bytes32 attestationId;
    if (hash) {
        attestationId = resolver.collateralHashPointer(key);
    } else {
        attestationId = resolver.collateralPointer(key);
    }
    return resolver.readAttestation(attestationId);
```

```
}

function _onFetchDataFromResolver(bytes32 key, bool hash) internal view virtual
returns (bytes memory) {
    return _fetchDataFromResolver(key, hash);
}
```

Figure D.3: The `_fetchDataFromResolver` and `_onFetchDataFromResolver` functions in `src/bases/DAOBase.sol#L42-L59`

- The following data could be decoded into two structs (`IdentityObj`, `enclaveIdObj`) instead of one struct and the members of the other. (Note that, if changed, the encoding also needs to be changed.)

```
function getEnclaveIdentity(uint256 id, uint256 version)
    external
    view
    returns (EnclaveIdentityJsonObj memory enclaveIdObj)
{
    bytes memory attestedIdentityData = _onFetchDataFromResolver(ENCLAVE_ID_KEY(id,
version), false);
    if (attestedIdentityData.length > 0) {
        (, enclaveIdObj.identityStr, enclaveIdObj.signature) =
            abi.decode(attestedIdentityData, (IdentityObj, string, bytes));
    }
}
```

Figure D.4: The `getEnclaveIdentity` function in `src/bases/EnclaveIdentityDAO.sol#L69-L79`

- Consider using static types when the length of a field is known. Per the reference implementation, the length of the reserved fields is known beforehand, so there is no need to use a dynamic field (e.g., `bytes`).

```
struct EnclaveReport {
    bytes16 cpuSvn;
    bytes4 miscSelect;
    bytes28 reserved1;
    bytes16 attributes;
    bytes32 mrEnclave;
    bytes32 reserved2;
    bytes32 mrSigner;
    bytes reserved3; // 96 bytes
    uint16 isvProdId;
    uint16 isvSvn;
    bytes reserved4; // 60 bytes
    bytes reportData; // 64 bytes - For QEReports, this contains the hash of the
concatenation of attestation key and QEAuthData
}
```

Figure D.5: The `EnclaveReport` struct in `contracts/types/CommonStruct.sol#L19-L32`

- Follow a consistent coding style. In the example below, the struct is built “in-place,” while some fields are calculated outside of the struct initialization.

```
function _parseQuoteHeader(bytes calldata rawQuote) private pure returns (Header
memory header) {
    bytes2 attestationKeyType = bytes2(rawQuote[2:4]);
    bytes2 qeSvn = bytes2(rawQuote[8:10]);
    bytes2 pceSvn = bytes2(rawQuote[10:12]);
    bytes16 qeVendorId = bytes16(rawQuote[12:28]);

    header = Header({
        version: uint16(BELE.leBytesToBeUint(rawQuote[0:2])),
        attestationKeyType: attestationKeyType,
        teeType: bytes4(uint32(BELE.leBytesToBeUint(rawQuote[4:8]))),
        qeSvn: qeSvn,
        pceSvn: pceSvn,
        qeVendorId: qeVendorId,
        userData: bytes20(rawQuote[28:48])
    });
}
```

Figure D.6: The `_parseQuoteHeader` function in `contracts/AttestationEntrypointBase.sol#159-L174`

E. Fix Review Results

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

From February 18 to February 21, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Automata Network team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 13 issues described in this report, Automata Network has resolved ten issues, has partially resolved one issue, and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Upsert functions allow inserting the same data	Informational	Resolved
2	_onFetchDataFromResolver does not return an error	Informational	Resolved
3	Authorization mechanism is confusing	Informational	Resolved
4	Continuous encoding and decoding of data is confusing and error prone	Informational	Unresolved
5	Certificate chain verification allows expired and irrelevant CRLs	High	Resolved
6	Root CRL checks can be bypassed	High	Resolved
7	Risc0- and sp1-based attestation may accept expired certificates	High	Resolved
8	Constants have inconsistent endianness	Informational	Resolved
9	CRL URI validation is inconsistent	Informational	Unresolved
10	Unclear definition and parsing of some header fields	Low	Resolved

11	No length checks can lead to panics	Informational	Resolved
12	Some state-changing functions do not emit events	Informational	Resolved
13	Investigate failing differential fuzz tests	Undetermined	Partially Resolved

Detailed Fix Review Results

TOB-AUT-1: Upsert functions allow inserting the same data
Resolved in PCCS PR #18. Upsertion now tests for duplicate entries and additionally requires that the new entry's validity window begins strictly later than the previous entry's validity window.

TOB-AUT-2: _onFetchDataFromResolver does not return an error
Resolved in PCCS PR #17. Unauthorized callers now cause `_onFetchDataFromResolver` to revert.

TOB-AUT-3: Authorization mechanism is confusing
Resolved in PCCS PR #19. The authorization mechanism has been refactored, allowing revocation and authorization to occur separately for each DAO.

TOB-AUT-4: Continuous encoding and decoding of data is confusing and error prone
Unresolved.

TOB-AUT-5: Certificate chain verification allows expired and irrelevant CRLs
Resolved in PCCS PR #22, PCCS PR #23, DCAP PR #19, and DCAP PR #20. Validity windows are now recorded for several types of data, and the access methods in `PCCSRouter` fetch and check the corresponding validity window, preventing the improper use of expired data. In addition, certificate chain verification now checks the authority key identifier of each CRL against the corresponding certificate's subject key identifier, preventing the use of a CRL with the wrong certificate. As an additional layer of validation, commit c08b2c34904c08952a4134ea3e05f656d709bb3e modifies the upsert logic to prevent the PCCS contract from accepting collaterals signed by expired certificates.

TOB-AUT-6: Root CRL checks can be bypassed
Resolved DCAP PR #18. The early exit allowing this exploit has been removed, and the certificate chain length has been asserted to equal 3. This version of certificate chain validation still potentially allows for a similar bypass to occur if a valid certificate chain of length 2 is created, but we believe that this is prevented by the CRL key-identifier logic added in DCAP PR #20. However, we would encourage the Automata Network team to confirm this and to add an additional check to reject any certificate chain where the root CA certificate is in a position other than the final position.

TOB-AUT-7: Risc0- and sp1-based attestation may accept expired certificates
Resolved in `dcap-rs` PR #13. Certificate verification in `dcap-rs` now explicitly checks the expiration of CRLs and certificates.

TOB-AUT-8: Constants have inconsistent endianness
Resolved in DCAP PR #23. The `teeType` constants now are big-endian `bytes4` constants and no byte-reversal occurs, and several data fields have been converted from `bytes8` values to `uint64` constants so that their types match their logical usage.

TOB-AUT-9: CRL URI validation is inconsistent
Unresolved.

TOB-AUT-10: Unclear definition and parsing of some header fields
Resolved in DCAP PR #23 and commit c5e9d15037c2bc5a7f49a3c08859598d2908ba3e. These fields are now parsed more consistently, and documentation comments have been added that directly tie the data structure definitions to sections of the Intel reference documentation. The Automata Network team should continue to develop improved documentation to ensure that the intended and required behavior is clear to users of their codebase.

TOB-AUT-11: No length checks can lead to panics
Resolved in DCAP PR #21. Length checks have been added.

TOB-AUT-12: Some state-changing functions do not emit events
Resolved in PCCS PR #24 and DCAP PR #22. All the functions identified in this finding now emit events.

TOB-AUT-13: Investigate failing differential fuzz tests
Partially resolved in PCCS PR #26. Several additional test values have been added, and additional validation logic has been added to the X.509 parsing implementation. For example, fetching the common name field no longer looks at a fixed position in the input, and instead searches for the common name object identifier when parsing. The Automata Network team has reported to us that they continue to perform additional fuzzing; we encourage them to continue this, to extend the fuzzing harness, and to continue addressing issues as they are uncovered.

F. Fix Review Status Categories

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

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

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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.