IOActive
Research-fueled Security Services

Signing Transparency Service Security Assessment

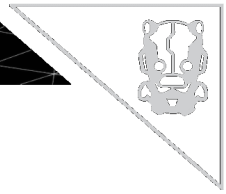
Microsoft Corporation

IOActive, Inc.
1426 Elliott Ave W
Seattle, WA 98119

Toll free: (866) 760-0222
Office: (206) 784-4313
Fax: (206) 784-4367

Approved for public release by Microsoft and IOActive on 01-Oct-2025

Contents



Executive Summary

Microsoft Corporation (Microsoft) engaged IOActive, Inc. (IOActive) to assess the security threats and risks associated with the Signing Transparency (ST) service, an open platform built on the open-source Confidential Consortium Framework (CCF). The ST service is intended to be implemented within Azure to allow third parties to validate the provenance of the deployed software.

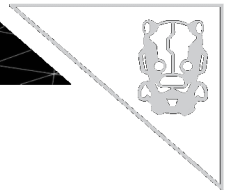
CCF uses trusted execution environments (TEEs) along with decentralized computing and strong cryptography to allow the implementation of secure multi-party computing systems. The ST service extends CCF to implement a ledger that can be used to provide provenance for artefacts in digital supply chains—verifiable signed claims are stored in an unmodifiable way within a ledger managed through the ST service. The ST service does this through a web service API that allows verifiable claims to be submitted to and information about specific claims to be retrieved from the underlying ledger. Claims are constructed and verified as Concise Binary Object Representation (CBOR) Object Signing and Encryption (COSE) requests. CBOR itself is essentially a restricted binary representation of a JSON object.

The underlying ledger application runs on three separate nodes, as a Kubernetes workload within Azure, with the ledger being managed in a distributed manner. The underlying compute hardware is intended to be the trusted environment provided by the AMD SEV-SNP platform.

Core system functionality, such as HTTP processing, deserialization, and cryptographic operations, are implemented within CCF itself. Beyond that, several of the underlying implementations are free and open-source software (FOSS), such as OpenSSL for cryptography and QCBOR for CBOR processing.

The ST service requires specific operational security requirements, such as storage for the ledger being write-once (so it cannot be modified), as well as specific requirements for system bootstrap. The implementation and operation of the ST service is intended to meet the following claims:

1. The ST service is transparent, attestable, and auditable.
2. The ST service ledger is immutable, tamper-proof, tamper-evident, and auditable.
3. The ST service provides non-repudiable and cryptographically verifiable receipts that guarantee inclusion of data on the ledger.
4. The ST service ledger and policy engine are open-source and reproducible.
5. The ST service runs in a TEE, backed by confidential computing hardware.
6. The ST service enforces registration policies on relying parties before inclusion of data hashes on the ledger.



7. The ST service satisfies predefined registration policies associated with its code upgrades, and the ST service TEE code is auditable.
8. Customers can query the ST service to view the production deployment history for both the ST service and the relying parties.
9. All code upgrades to relying parties are transparent and always logged on the immutable ledger.

Project Description

From the 28th of April to the 13th of June 2025, IOActive performed a code review, dynamic testing, and targeted fuzzing of the ST service and underlying CCF. The primary security concerns for the engagement were:

- The security of the implementation of the ST service
- The security of components used from and exposed by the usage of CCF
- Adherence to the security claims listed above

The source code for CCF was retrieved from the following URL on the 14th of April 2025 (commit 363cd4b4965de54ad2102cfa84df052e95464192):

`https://github.com/microsoft/CCF`

The source code for the ST service was retrieved from the following URL on the 14th of April 2025 (commit e4e7473fd3f7b732f608d2e74ec9c5de62d9602a):

`https://github.com/microsoft/scitt-ccf-ledger`

An environment was set up for dynamic testing within Azure at the following URL:

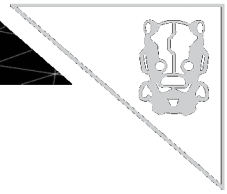
`https://ioactive-dynamic-tests.confidential-ledger.azure.com/`

Key Takeaways

IOActive found the security posture of the ST service and the underlying CCF to be robust. Only four security findings were reported, three of which were only reported for informational purposes.

The only finding with a security impact was a coding error that meant that attestation data may not be validated correctly. This issue was verified as being remediated after a fix had been implemented.

The first informational finding was a potential supply chain issue with libraries being used for development without being cryptographically verified. The second informational issue is related to the use of a third-party dependency by CCF that has a known command injection vulnerability. However, a custom build of this dependency is used that is not affected by this vulnerability, and the code is not compiled; and there are security controls in place that



would prevent the vulnerable code from accidentally or deliberately being included in a production build of CCF.

The final informational finding related to the architecture of the overall system and the choice to implement the ST service API on top of the CCF APIs. In principle, this exposes the CCF APIs (notably the Public Node API used to manage the network of trusted compute nodes, as well as the Governance API used to implement node consensus) along with the ST service API. In principle, the use of the CCF APIs and the ST service API are separate, whereby the Public Node API could be considered the underlying control plane, for example, and should be used by separate users or user groups in separate use cases. There is no need for a user with access to the ST service API to have access to the Public Node API or Governance APIs.

Having said that, no security issues were identified in the CCF APIs and the information they disclosed would only be of use to an attacker who had already compromised the internal node network (i.e. of the containing Kubernetes workload if deployed through Kubernetes). Since no direct security issues were identified, restricting or removing access to APIs that do not need to be publicly exposed should be considered as a means of reducing the attack surface; however, the cost of doing this should be balanced against the potential impact. It should be noted that, due to the use of a shared underlying implementation, functionality such as CBOR/COSE or HTTP processing will still be available and will not be affected by any attack surface reduction measures.

The ID of a node could be returned in specific error messages from CCF through the ST service; however, the node ID is a digest of the node's public key, as such is considered public, and so does not present a security risk and was not reported as such.

Regarding the security claims outlined above, since the implementation and architecture were in scope, claims affected by these aspects of the system were validated. Claims that rely on specific deployment or operational choices require validation based on an actual production deployment and were not evaluated. IOActive made the following observations regarding the claims:

1. The service is transparent, auditable, and attestable based on the analysis performed. Specifically, the service relies on well-established cryptographic principles.

2. The ST service ledger is immutable tamper-proof, tamper-evident, and auditable based on access through the ST service API. Actual physical tamper-proofness or tamper-evidence depends on the underlying hardware used to store the ledger, which is intended to be storage that is essentially write-once.

3. The ST service uses the cryptographic functionality implemented within and exposed through CCF to provide non-repudiable and cryptographically verifiable receipts that guarantee inclusion of data on the ledger.

4. The implementations of both the ST service and CCF are open source and were audited as part of this assessment. The ST service itself can be verified within a deployment as it is added to the ledger as part of the bootstrap process.

5. The underlying hardware was out of scope for this assessment and would need to be verified per-deployment; however, the use of a TEE, such as AMD SEV-SNP, provides the necessary secure environment to support the aims of the ST service.

6. The ST service requires user registration and relies on strong cryptographic principles for request validation; however, user access must be set up as part of the bootstrap process.

7. Registration policies are configurable for an ST service deployment, and TEE code is auditable.

8. Deployment history is recorded in a non-modifiable manner in the underlying ledger.

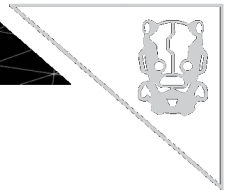
9. Code upgrades/modifications must be recorded in the ledger before they can be executed.

As noted in the comments on the security claims and elsewhere within this report, aspects of the system's security and security claims are reliant on appropriate operational security measures. This applies in particular to the underlying hardware and the bootstrap process. These areas were out of scope for this assessment, but in terms of providing assurance to third parties, it would be important to provide direct assurance regarding the underlying system hardware actually used and that it meets the relevant requirements, as well as to provide assurance about the bootstrap of a given deployment.

Since bootstrap will include the provision of system secrets, system users, and initial system code, this inherently creates the root-of-trust for an ST service deployment without necessarily relying on a root-of-trust—one of the fundamental aims of the bootstrap process is to move from untrusted to trusted. Ideally, the initial bootstrap itself should be repeatable and verifiable, allowing the bootstrapped state to be verified. Although of limited scope as an attack, bootstrapping with a compromised TEE and compromised ST service could allow malicious changes to be transparently made later. In essence, this is no different from underlying cryptographic material being compromised, allowing an attacker to post malicious updates to the ledger.

Consequently, aspects of the operational security of an ST service deployment should also be made transparent to provide the maximum level of assurance possible. Equally, some level of trust is necessary, as absolute assurance cannot reasonably be gained.

Notwithstanding these aspects of the system, IOActive validated the ST service security claims where appropriate and no meaningful implementation or architectural security issues were identified. As such, the ST service implements an appropriate solution as designed and implemented for its intended use.



IOActive would like to highlight the Temporal Logic of Actions (TLA+) formal specification used to describe and model the consistency and consensus algorithms for CCF. This specification provides a formal validation of these algorithms against attacks, such as race conditions, that may impact the implementation of something like a distributed ledger. The TLA+ modelling is openly available as part of the CCF open-source implementation. Additionally, IOActive would like to highlight the threat model provided at the beginning of the assessment, which suitably covered the potential threats that an ST service deployment could face. Along with the implementation itself and discussions regarding hardware and deployment, this indicates an appropriate level of concern and attention to the security of the ST service, considering its intended use.

Analysis of Findings

Figure 1 shows the distribution of findings by risk rating.

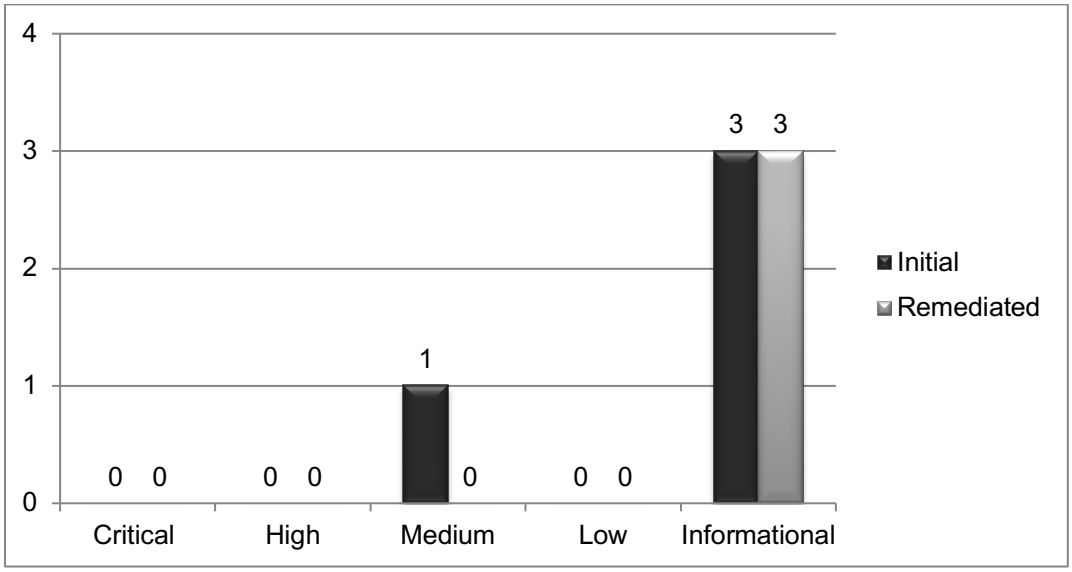
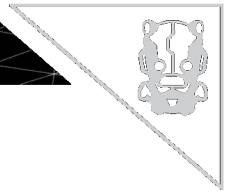


Figure 1. Distribution of Findings

IOActive identified one medium-risk vulnerability in Microsoft's in-scope assets, as well as three informational findings.

The medium-risk issue was a typographical error that means that attestation data is not completely correctly validated by CCF and the use of third-party libraries without verification. This issue has been verified as being fixed.

The informational findings were the ST service and CCF exposing information about the deployment and a command injection vulnerability in the QuickJS engine. IOActive did not identify a way to exploit these issues during the time allocated for the assessment.



Next Steps

IOActive recommends considering fixing the issues presented in this report to improve the security posture of the in-scope assets. Once Microsoft has addressed the findings, IOActive further recommends performing remediation validation testing to confirm that the findings are properly fixed.

IOActive believes the most advantageous, efficient, and effective way to accomplish remediation is to start by focusing on vulnerabilities that are high-risk and low effort to fix. After these are fixed, the organization should focus on the remaining high-risk and more complex vulnerabilities.

Table 1. Remediation status

Finding ID	Title	Total Risk	Effort to Fix	Status as of Report Date
#ST-01	[CCF] Incorrect Verification in verify_snp_attestation_report()	Medium	Low	Fixed
#ST-05	[CCF] Insufficient Checks for Third-party Dependencies	Informational	Low	
#ST-04	[QuickJS] Command Injection	Informational	Low	
#ST-03	[ST] CCF Internal Configuration Disclosed via CCF Public API	Informational	Medium	

Important The effort to address vulnerabilities is an estimate reflecting the assessment team's experience; actual remediation effort may vary based on numerous factors including skill sets, process efficiency, and available resources.

Technical Summary

Scope

The in-scope systems comprise the ST service, which implements a ledger that can be used to provide provenance for artefacts in digital supply chains—verifiable signed claims are stored in an unmodifiable way within a ledger managed through the ST service. The ST service is intended to be used within Azure by the Azure Resource Manager (ARM) to provide a level of assurance around the components built and used on top of a trusted computing platform.

The underlying implementation has unified processing of HTTP requests that provides the APIs exposed from the ST service and CCF:

- The ST service API for querying and adding authenticated entries to the artefact ledger
- CCF Public Node API for trusted compute node management
- CCF Governance API for management of consensus and node network constitution

These implementations are open source and intended to be publicly verifiable.

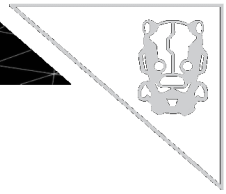
CCF provides the data serialization, ledger, and cryptographic functionality used by the ST service, with low-level data processing functionality in turn being implemented using third-party components such as OpenSSL and QCBOR.

The aim of the assessment was to look for potential vulnerabilities within the ST service and the underlying CCF system that would impact the security of the ST service and its ability to function as intended, particularly those that could be exploited to compromise the ST service or CCF service and add unauthorized ledger entries or modify existing ledger entries.

Project Approach

The consultants aimed to identify issues within the following broad areas based on the specific abstractions split across the ST service/CCF system as a complete unit:

- Implementation, which includes hardware, language, and low-level platform features where appropriate and addresses baseline platform security as well as code correctness and data processing
- System logic, which includes the higher-level behavior of the system and addresses behavior that is relevant from a cybersecurity perspective (e.g. authorization or auditing)
- System architecture, which includes the definition and enforcement of security contexts and the trust boundaries between them



- System configuration, which can be the storage, use, or management of any security-relevant configuration of hardware, first- or third-party code, or deployment and management mechanisms

The broad, high-level, security-related areas of concern for a given system are as follows:

- Authentication/authorization
- Input/data/request validation
- Data/information security (both data-at-rest and data-in-transit)
- System auditing

The assessment was divided into three phases:

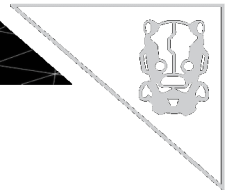
- Initial project ramp-up
- Source code review
- Deployed instance dynamic assessment

The initial project ramp-up provided the consultants with the information necessary to broadly understand the in-scope systems and code and was provided as an initial meeting as well as access to internal documentation including the security claims as well as a threat model.

The source code for the ST service and CCF was primarily C++ with some JavaScript for CCF applications as well as Python for testing and scripting. Consequently, the primary concern regarding implementation security was the correctness of data processing and input validation. When auditing data-processing code, the aim is to determine edge cases whereby the data processing implementation is not correctly defined for the input. This may be due to a programming error or assumptions about the data received. For C++ code, the implementation not being correctly defined typically leads to undefined behavior at the programming-language level. Often the practical manifestation of undefined behavior is memory corruption, which in turn may be exploitable for arbitrary code execution by corrupting execution metadata (e.g. function or return pointers).

Input validation should involve checking any flags or enumerations have valid values, including logically valid for their intended usage, as well as checking data sizes and array access to ensure that only valid data is read and written. This is of particular importance for data deserialization. The primary concerns related to input validation for the in-scope APIs since state-altering requests are authenticated as signed messages containing data in a binary format.

Regarding system logic, the primary concern is the correct logical behavior of security-relevant functionality. Logic flaws are not language-specific and may relate to assumptions about the sequence in which specific APIs may be called to enact a given operation or about the state of a system when a specific operation is requested. The primary concern



from a logical perspective for the in-scope systems is the use of the functionality exposed by the CCF APIs and whether this is appropriate for the use of the ST service.

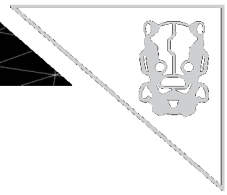
Finally, regarding system architecture, the primary concern is the implementation of security boundaries between different system components: the compartmentalization of functionality based on user or component privileges. A rigorous security architecture will require a root-of-trust that can then be used to establish identity for system components and system users, from which the correct compartmentalization can be derived. The failure to implement these boundaries, or the failure to implement them correctly, can lead to significant and costly security flaws. The system uses and implements a strong definition of identity, both user as well as software, based on strong cryptographic principles. As such, any architectural concerns relate to the separation between the services exposed by the ST service/CCF, as well as the establishment of the root-of-trust itself. The establishment of a root-of-trust inherently touches on implementation details as well, such as underlying hardware, and was beyond the scope of this assessment.

Bearing the above concepts and abstractions in mind, the source code was analyzed from the primary perspective of the ST service, but reached into areas of the CCF codebase that directly supported the ST service functionality to capture complete attack or data-processing paths.

Regarding the correctness of the underlying CCF implementations for consensus and consistency, which essentially implement the ledger, the consultants noted that formal specifications were created using TLA+. IOActive did not independently verify either the correctness of these specifications or that TLA+ itself validated these specifications; however, the use of a formal language such as TLA+ is an excellent and commendable way to demonstrate the robustness of the design of the consensus and consistency algorithms, particularly regarding operational sequencing that may lead to race conditions (which would be of concern for a multi-party ledger).

The dynamic assessment focused on manually confirming understanding of the ST service deployed in an environment specifically set up for this security assessment. The API endpoints were enumerated based on the source code, as well as by interrogating the API itself (since endpoints can be registered to be discoverable through a dedicated query API). Manual API testing used a combination of command line tools (e.g. cURL) and web application testing software (e.g. Burp Suite). Manual testing focused on the logical functionality implemented within the API, such as authentication and authorization of requests (e.g. testing data signed by an untrusted authority), as well as well-known malformed test cases for requests such as excessively large requests.

Processing of complex binary data was performed using a basic bit-flipping fuzzer. Given the time constraints and the fact that the majority of binary data processing was performed at the level of CBOR/COSE processing, the consultants judged simple manipulation of the COSE message to be suitable for this purpose—the message must be deserialized, and partly processed, correctly before it can be validated. The actual signed payload can be arbitrary, so was not considered during the assessment.



The following Python implementation was used to generate 100,000 test cases, which were sent to the ST service API at a rate that would not risk creating a denial-of-service (DoS) attack. As noted above, the HTTP and CBOR/COSE processing is common to all the ST service and CCF APIs, and so this testing is applicable to CCF as well as the ST service even though the ST service API was not specifically targeted.

```
import sys;
import random;

def main():
    args = sys.argv[1:]

    if len(args) != 2:
        print("Usage: python bitflip.py [infile] [tests]")
        return

    file = args[0]
    test = int(args[1])

    f = open(file, 'rb')
    inbin = f.read()
    binlen = len(inbin)
    outbin = bytearray(binlen)

    for x in range (0, test):
        outbin[:] = inbin

        flips = random.randrange(1,5)

        for flip in range(flips):
            flipbyte = random.randrange(binlen)
            flipbit = 1 << random.randrange(8)

            outbin[flipbyte] = outbin[flipbyte] ^ flipbit

        outfile = open(".\\out\\" + file + "-" + str(x), "wb")
        outfile.write(outbin)
        outfile.close()

if __name__=="__main__":
    main()
```

The consultants also ran an instance of the Fuzzilli JS engine fuzzer on the QuickJS engine. The intent was to identify any vulnerabilities that could be exploited through ballot or policy scripts. No reproducible crashes were observed in the timeframe of the assessment.

Finally, the CCF APIs were manually inspected to determine what functionality was available as well as what information was retrievable, with a view to determining the security impact on the ST service API itself.

Detailed Findings

#ST-01 - [CCF] Incorrect Verification in verify_snp_attestation_report() [FIXED]

Host(s) / File(s)	CCF-main\src\js\extensions\snp_attestation.cpp
Category	CWE-697: Incorrect Comparison
Testing Method	White Box
Tools Used	VS Code
Likelihood	High (4)
Impact	Low (2)
Total Risk Rating	Medium (8)
Effort to Fix	Low
CVSS	5.3 (Medium) - CVSS:3.1/AV:N/AC:H/PR:L/UI:N/S:U/C:N/I:H/A:N

Threat and Impact

The `verify_snp_attestation_report()` function does not correctly verify all the attestation data. The function uses data that is valid but in an unspecified state due to repeated calls to `std:move()` on the same variable.

`verify_snp_attestation_report()`:186

```
auto attestation_host_data =
      jsctx.new_array_buffer_copy(attestation.host_data);
    JS_CHECK_EXC(attestation_host_data);
    JS_CHECK_SET(a.set("host_data",
std::move(attestation_host_data)));

    auto attestation_id_key_digest =
      jsctx.new_array_buffer_copy(attestation.id_key_digest);
    JS_CHECK_EXC(attestation_id_key_digest);
    JS_CHECK_SET(a.set("id_key_digest",
std::move(attestation_id_key_digest)));

    auto attestation_author_key_digest =

jsctx.new_array_buffer_copy(attestation.author_key_digest);
    JS_CHECK_EXC(attestation_author_key_digest);
    JS_CHECK_SET(
```

```
      a.set("author_key_digest",
std::move(attestation_id_key_digest)));

    auto attestation_report_id =
      jsctx.new_array_buffer_copy(attestation.report_id);
    JS_CHECK_EXC(attestation_report_id);
    JS_CHECK_SET(a.set("report_id",
std::move(attestation_id_key_digest)));

    auto attestation_report_id_ma =
      jsctx.new_array_buffer_copy(attestation.report_id_ma);
    JS_CHECK_EXC(attestation_report_id_ma);
    JS_CHECK_SET(a.set("report_id_ma",
std::move(attestation_report_id_ma)));
```

There is one correct attestation of `attestation_id_key_digest` and a call to `std::move()` followed by two further attestations that reuse `attestation_id_key_digest` rather than the variable that seems appropriate.

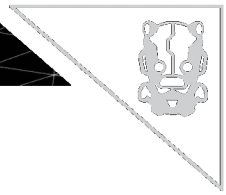
Additionally calling `std::move()` leaves the argument in a valid but unspecified state, meaning that the repeated attestations are using unspecified data, which could lead to unexpected behavior.

Recommendations

The issue appears to be a copy/paste issue where the repeated pattern has been copied, pasted, but not appropriately corrected. The code should be modified to attest the appropriate data field for that particular check in question; this will also prevent the use of unspecified data.

Additional Information

https://en.cppreference.com/w/cpp/utility/move



#ST-05 - [CCF] Insufficient Checks for Third-party Dependencies

Host(s) / File(s)	tla/install_deps.py
Category	CWE-325: Missing Cryptographic Step
Testing Method	Manual
Tools Used	Sublime
Likelihood	Informational (1)
Impact	Informational (1)
Total Risk Rating	Informational (1)
Effort to Fix	Low
CVSS	0.0 (Informational) - CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

Threat and Impact

The CCF framework uses various third-party libraries and tooling. Some of the dependencies, like LLVM for cross compilation, are cryptographically validated before being installed in a dev environment.

The framework uses TLA+ (a formal specification language) as a dependency. TLA+ is downloaded as a JAR file or compressed binaries without being cryptographically checked. An attacker who compromises the TLA+ repositories or takes over the TLA domain would be able to get code execution on systems running CCF.

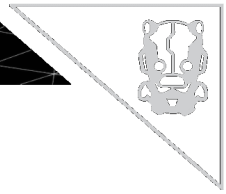
The following code is from `tla/install_deps.py`:

```
def fetch_latest(url: str, dest: str = "."):
    subprocess.Popen(f"wget -N {url} -P /tmp".split()).wait()
    file_name = url.split("/")[-1]
    file_path = f"/tmp/{file_name}"
    assert os.path.exists(file_path)
    bin_path = None

    if file_name.endswith(".bin"):
        os.chmod(file_path, os.stat(file_path).st_mode |
stat.S_IEXEC)
        subprocess.Popen(f"{file_path} -d
{dest}".split()).wait()
        bin_path = f"{dest}/bin"

    elif file_name.endswith(".tgz"):
        with tarfile.open(f"/tmp/{file_name}") as tar:
```

```
            tar.extractall(dest)
            rel_bin_path = next(
                member.name for member in tar.getmembers() if
"bin" in member.name
            )
            bin_path = os.path.join(dest, rel_bin_path)

    elif file_name.endswith(".jar"):
        shutil.copyfile(file_path, os.path.join(dest,
file_name))

    if bin_path is not None:
        append_bashrc(f"export PATH:$PATH:{bin_path}")

fetch_latest(
        url="https://nightly.tlapl.us/dist/tla2tools.jar",
        dest=TLA_DIR,
    )
```

Recommendations

The use of TLA+ is provided as an example within the assessed codebase, is not used in the CCF/ST service build process and is not deployed as part of a production system. As such, this issue is raised for informational purposes.



#ST-04 - [QuickJS] Command Injection

Host(s) / File(s)	QuickJS
Category	CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Testing Method	Manual
Tools Used	Manual
Likelihood	Informational (1)
Impact	Informational (1)
Total Risk Rating	Informational (1)
Effort to Fix	Low
CVSS	0.0 (Informational) - CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

Threat and Impact

The CCF framework uses QuickJS as the JavaScript engine to interpret ballot and policy scripts. Each of the scripts would execute with separation provided by individual executions of the engine per script. Information between scripts is shared using key-value pairs; however, the QuickJS engine has a command injection vulnerability in the `std.urlGet` function. A URL that contains a command injection payload will be executed as an OS command.

Additionally, an attacker could also use the `std.popen` function to execute OS commands. This would breach the security boundary provided by the engine. This vulnerability is not applicable for CCF or the ST service ledger as the QuickJS contexts within these applications do not allow the execution of std functions. This was tested by the consultants.

The following was run on a regular QuickJS interpreter:

```
localhost:~# qjs
QuickJS - Type "\h" for help
qjs > std.urlGet(";touch /tmp/ioactive;")
sh: : Permission denied
null
qjs >
(Press Ctrl-C again to quit)
qjs >
localhost:~# ls /tmp/
ioactive
```

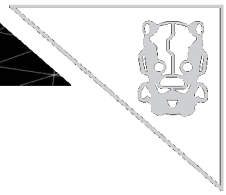
The consultants also verified that the `std` module cannot be accessed from within the policy scripts by creating a policy script that attempted to access `std.urlGet`:

```
COSE_CLAIMS_PATH="demo-poc/payload.sig.cose"
OUTPUT_FOLDER="demo-poc" ./demo/cts_poc/3-client-demo.sh

Setting up environment

Getting service parameters
  % Total    % Received % Xferd  Average Speed   Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100   776  100   776    0     0  42970      0 --:--:-- --:--:-
- --:--:-- 43111

Submitting claim to the ledger and getting receipt for the
committed transaction
2025-06-13 14:05:54.288 | DEBUG    |
pyscitt.client:request:402 - POST /entries 400 PolicyError
2025-06-13 14:05:54.288 | ERROR    |
pyscitt.client:request:432 - Request failed: PolicyError Error
while applying policy: ReferenceError: 'std' is not defined
    at apply (configured_policy)

Traceback (most recent call last):
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/venv/bin/scitt", line 8, in <module>
    sys.exit(main())
             ~~~~^^
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/pyscitt/pyscitt/cli/main.py", line 47, in main
    args.func(args)
    ~~~~~~~~~^^^^^^
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/pyscitt/pyscitt/cli/register.py", line 61, in cmd
    register_signed_statement(
    ~~~~~~~~~~~~~~~~~~~~~~~~~^
        client,
        ^^^^^^^
    ...<2 lines>...
        args.skip_confirmation,
        ^^^^^^^^^^^^^^^^^^^^^^^
    )
    ^
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/pyscitt/pyscitt/cli/register.py", line 34, in
register_signed_statement
    submission =
client.submit_signed_statement_and_wait(signed_statement)
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/pyscitt/pyscitt/client.py", line 558, in
submit_signed_statement_and_wait
    resp = self.post(
        "/entries",
        headers=headers,
```

```
        content=signed_statement,
    )
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/pyscitt/pyscitt/client.py", line 462, in post
    return self.request("POST", *args, **kwargs)
           ~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/<redacted>/projects/MS/cts/scitt-ccf-
ledger/pyscitt/pyscitt/client.py", line 433, in request
    raise response_error
pyscitt.client.ServiceError: PolicyError: Error while applying
policy: ReferenceError: 'std' is not defined
    at apply (configured_policy)
```

Recommendations

The custom build of QuickJS used by CCF does not currently use this functionality - as such, this issue is raised for informational purposes. Security controls are in place to ensure that this functionality should not be re-enabled in a production build.



#ST-03 - [ST] CCF Internal Configuration Disclosed via CCF Public API

Host(s) / File(s)	CCF
Category	CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
Testing Method	Manual
Tools Used	Burp Suite
Likelihood	Informational (1)
Impact	Informational (1)
Total Risk Rating	Informational (1)
Effort to Fix	Medium
CVSS	0.0 (Informational) - CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N

Threat and Impact

The CCF provides two APIs for programmatic control and governance functionality of the nodes compromising a confidential computing network: the CCF Public Node API and the CCF Governance API. The ST service provides additional API endpoints on top of the underlying framework that then implement the specific secure ledger containing signed claims about digital artifacts to be executed on a given compute node. The Public Node API implements the control plane for nodes, with request authentication implemented using signed requests, as well as multiple read-only endpoints that provide information about the system. The CCF Governance API implements the underlying system that allows node behavior to be configured as well as nodes to submit proposals to the node network, with request authentication also implemented using signed requests.

Both APIs provide a significant amount of information about a given the ST service deployment, the majority of which may only be of use during system bootstrap. This includes information about the Kubernetes deployment, as well as network configuration. This information itself does not currently facilitate compromise of a deployment, and no security flaws were found within the ST service or CCF API; consequently, this issue has been rated as informational.

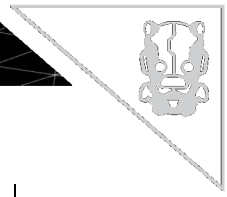
For example, the following request was sent to the demo environment used during the assessment:

```
GET /node/network/nodes HTTP/1.1
Host: 134.33.167.10
Content-Length: 0
```

The response to this request was as follows:

```
HTTP/1.1 200 OK
content-length: 2729
content-type: application/json
```

```
x-ms-ccf-transaction-id: 6.1280

{
  "nodes": [
    {
      "last_written": 660,
      "node_data": {
        "containerImage":
"confidentialledgeracrprod.azurecr.io/scitt-
snp:0.14.0_1.0.030401-efd79a15",
        "kubernetesNamespace": "00000000-0000-0000-0000-
c90288598eec",
        "ledgerName": "ioactive-dynamic-tests",
        "nodeName": "accledger-0",
        "vmName": "vn2-zone-3-virtualnode-0"
      },
      "node_id":
"8c25fc51bbaa97d63e23a005dd31f216bb2fe40e7bdb615262ec1d6b3fa24
7e7",
      "primary": true,
      "rpc_interfaces": {
        "node": {
          "bind_address": "0.0.0.0:16386",
          "endorsement": {
            "authority": "Node"
          },
          "published_address": "10.2.0.7:16386"
        },
        "operator": {
          "bind_address": "0.0.0.0:16387",
          "endorsement": {
            "authority": "Service"
          },
          "published_address": "10.2.0.7:16387"
        },
        "primary": {
          "bind_address": "0.0.0.0:16385",
          "endorsement": {
            "authority": "Service"
          },
          "http_configuration": {
            "max_body_size": "1MB"
          },
          "published_address": "10.2.0.7:16385"
        }
      },
      "status": "Trusted"
    },
    {
      "last_written": 25,
      "node_data": {
        "containerImage":
"sha256:08670dc8b7ea1520381d5742c7b21c78deeb095de0d85db368e4f0
b0357fb576",
        "containerImageId":
"confidentialledgeracrstaging.azurecr.io/scitt-
```




```
snp@sha256:c6a119e92e381a94c058c96e5adb080968bb4db1fc4944db4ab
3c7389f15804c",
        "kubernetesNamespace": "00000000-0000-0000-0000-
c90288598eec",
        "ledgerName": "ioactive-dynamic-tests",
        "nodeName": "accledger-2",
        "vmName": "vn2-virtualnode-0"
      },
      "node_id":
"6e90ee3e86dceaae014223fa2ad464df50eb2a49eebcd513e7efc3b3b7dd6
84b",
      "primary": false,
      "rpc_interfaces": {
        "node": {
          "bind_address": "0.0.0.0:16386",
          "endorsement": {
            "authority": "Node"
          },
          "published_address": "10.2.0.14:16386"
        },
        "operator": {
          "bind_address": "0.0.0.0:16387",
          "endorsement": {
            "authority": "Service"
          },
          "published_address": "10.2.0.14:16387"
        },
        "primary": {
          "bind_address": "0.0.0.0:16385",
          "endorsement": {
            "authority": "Service"
          },
          "http_configuration": {
            "max_body_size": "1MB"
          },
          "published_address": "10.2.0.14:16385"
        }
      },
      "status": "Trusted"
    },
    {
      "last_written": 675,
      "node_data": {
        "containerImage":
"sha256:08670dc8b7ea1520381d5742c7b21c78deeb095de0d85db368e4f0
b0357fb576",
        "containerImageId":
"confidentialledgeracrstaging.azurecr.io/scitt-
snp@sha256:c6a119e92e381a94c058c96e5adb080968bb4db1fc4944db4ab
3c7389f15804c",
        "kubernetesNamespace": "00000000-0000-0000-0000-
c90288598eec",
        "ledgerName": "ioactive-dynamic-tests",
        "nodeName": "accledger-1",
        "vmName": "vn2-zone-2-virtualnode-0"
      },
```

```
      "node_id":
"913b99df07bffbc58ae8acbcbd9e868a09861437920a94db95675e925ee52
62a",
      "primary": false,
      "rpc_interfaces": {
        "node": {
          "bind_address": "0.0.0.0:16386",
          "endorsement": {
            "authority": "Node"
          },
          "published_address": "10.2.0.19:16386"
        },
        "operator": {
          "bind_address": "0.0.0.0:16387",
          "endorsement": {
            "authority": "Service"
          },
          "published_address": "10.2.0.19:16387"
        },
        "primary": {
          "bind_address": "0.0.0.0:16385",
          "endorsement": {
            "authority": "Service"
          },
          "http_configuration": {
            "max_body_size": "1MB"
          },
          "published_address": "10.2.0.19:16385"
        }
      },
      "status": "Trusted"
    }
  ]
}
```

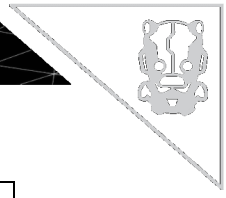

A limited amount of information has been highlighted, but as can be seen, details such as Kubernetes namespace, internal IP addresses and ports, as well as VM information, are provided.

Recommendations

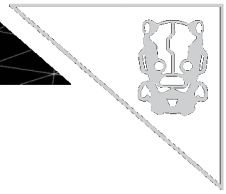
Although the assessment of the ST service and CCF APIs did not discover any direct security weaknesses, and the information provided does not weaken the security of the system as it is, reducing the amount of information provided to arbitrary third-parties as well as restricting access to these APIs in order to reduce the available attack surface means that restricting access to the Node and Governance APIs (where practical) may be beneficial from a security perspective.

As is often the case, there is a partial conflict between security and usability. APIs should only be restricted where it has no practical impact on the use of an ST service deployment. Additionally, since the system is intended to provide a verifiable way of assessing the provenance of systems deployed on top of the ST service within Azure, it may be desirable to keep all APIs open and accessible for the sake of transparency

Finally, it is worth highlighting that in terms of attack surface reduction, this will only impact higher-level logical functionality within the Node and Governance endpoints; functionality such as COSE



processing, CBOR processing, and low-level HTTP-processing is shared between all APIs and as such will still be exposed through the ST service API endpoints.



Appendix A: Overview of Detailed Findings

Host(s) / File(s)

This section includes a list of the assets affected by the finding.

Category

IOActive uses Common Weakness Enumeration (CWE™)[1] identifiers to categorize each finding. CWE is a community-developed list of software and hardware weakness types that have security ramifications. This software assurance strategic initiative is sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security and published by The MITRE Corporation.

Testing Method

The testing method captures the approach that the consultants used to discover the finding.

Table 2. Examples of testing methods

Method	Description
Black Box	The consultants had no internal knowledge of the target and were not provided with any information that was not publicly available.
Grey Box	The consultants had access and knowledge levels comparable to a user, potentially with elevated privileges. The consultants may also have been provided documentation, accounts, or other information.
White Box	The consultants had full access to the target's source code, documentation, etc.

Tools Used

The section lists the specific tools the consultants used to discover the finding.

[1] https://cwe.mitre.org/

Likelihood and Impact

IOActive assigns two ratings for each finding: one for likelihood and another for impact. Each rating corresponds to a numeric score ranging from 5 (critical) to 1 (informational).

Table 3. Description of likelihood and impact

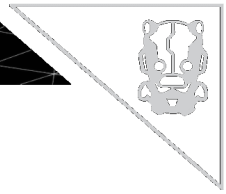
Rating (Score)	Likelihood	Impact
Critical (5)	The finding is almost certain to be exploited; knowledge of the issue and how to exploit it are in the public domain	Extreme impact to the entire organization if exploited
High (4)	The finding is relatively easy to detect and exploit by an attacker with low skills	Major impact to the entire organization or a single line of business if exploited
Medium (3)	A knowledgeable insider or expert attacker could exploit the finding without much difficulty	Noticeable impact to a line of business if exploited
Low (2)	Exploiting the finding would require considerable expertise and resources	Minor damage if exploited or could be exploited in conjunction with other vulnerabilities as part of a more serious attack
Informational (1)	The finding is not likely to be exploited on its own but may be used to gain information for launching another attack	Does not represent an immediate threat but may have security implications if combined with other vulnerabilities

Total Risk Rating

IOActive then calculates a total risk score by multiplying likelihood and impact.

Table 4. Total risk rating and corresponding aggregate risk scores

Total Risk Rating	Total Risk Score Range (Likelihood × Impact)
Critical	20–25
High	12–19
Medium	6–11
Low	2–5
Informational	1



Effort to Fix

IOActive estimates the effort it will take to fix the finding based on our consultants' experience. An organization's actual effort may vary based on factors such as skill sets, process efficiency, and available resources.

CVSS

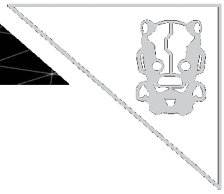
IOActive may also use the Common Vulnerability Scoring System (CVSS)[2] to capture the principal characteristics of a finding and produce a numerical score reflecting its severity. CVSS is used by organizations worldwide to supply a qualitative measure of severity; however, CVSS is not a measure of risk.

IOActive assigns a value to each metric of the scoring system.

Table 5. CVSS metrics and selectable values

Metric	List of Values
Attack Vector (AV)	Network (N) Adjacent (A) Local (L) Physical (P)
Attack Complexity (AC)	Low (L) High (H)
Privileges Required (PR)	None (N) Low (L) High (H)
User Interaction (UI)	None (N) Required (R)
Scope (S)	Unchanged (U) Changed (C)
Confidentiality (C)	None (N) Low (L) High (H)
Integrity (I)	None (N) Low (L) High (H)
Availability (A)	None (N) Low (L) High (H)

[2] https://www.first.org/cvss/



These values translate to a base score[3] and severity rating.

Table 6. CVSS 3.1 base score and associated rating

Severity Rating	Base Score Range
Informational	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

[3] https://www.first.org/cvss/calculator/3.1