Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

puffer

RAVe

Veridise Inc.

July 18, 2023

► **Prepared For:**

Puffer Finance
https://puffer.fi/

► **Prepared By:**

Xiangan He
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Jul. 18, 2023 V1

Contents

1 Executive Summary

From Jul. 5, 2023 to Jul. 10, 2023, Puffer Finance engaged Veridise to review the security of their Remote Attestation VErification (RAVe) library. The review covered Solidity smart contract implementations of: an ASN.1 DER format parser, some X.509 certificate validation logic, and an implementation of a PKCS#1 v1.5 signature validation scheme. Additionally, the review covered a smart contract that uses the above components to validate cryptographically-signed remote attestation reports generated by the Intel Attestation Service (IAS). Veridise conducted the assessment over 8 person-days, with 2 engineers reviewing code over 4 days on commit `664d724`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The RAVe developers provided the source code of the RAVe contracts for review. To facilitate the Veridise auditors' understanding of the code, the RAVe developers provided documentation that describes the high-level concept of RAVe*. The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables. The source code contained a test suite, which the Veridise auditors noted provides partial test coverage of the key functions as well as a basic fuzzing infrastructure based on Foundry.

Summary of issues detected. The audit uncovered 13 issues, 2 of which are assessed to be of high severity by the Veridise auditors. Specifically, several flaws in the implementation of the PKCS#1 v1.5 signature validation scheme can reduce the difficulty of forging valid signatures (V-RAV-VUL-001), and a business logic bug causes valid attestation reports with the status OK to always be rejected (V-RAV-VUL-002). The Veridise auditors also identified 3 medium-severity issues, including incorrect handling of optional fields leading to valid reports being rejected (V-RAV-VUL-003), some missing X.509 certificate validation logic (V-RAV-VUL-004), and incorrect ASN.1 length parsing logic (V-RAV-VUL-005). Furthermore, the auditors identified 5 low-severity issues as well as a number of warnings.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve RAVe. First, the auditors noted that only the leaf certificate of the signing certificate chain is provided to the RAVe smart contracts, which means that the full certificate chain cannot be validated within RAVe. We recommend that the developers incorporate logic to validate the full certificate signing chain into any other smart contracts that use RAVe.

Next, while the auditors did point out some X.509 certificate validation issues such as V-RAV-VUL-004 and V-RAV-VUL-007, the list should not be considered exhaustive. To improve security, we recommend that the developers incorporate as many validation features† as is feasible.

* `https://docs.puffer.fi/tech/rave` - the developers noted that the documentation is slightly out-of-date, however.

† The developers may want to check Sections 3 and 4 of https://datatracker.ietf.org/doc/html/rfc5280

Lastly, the auditors noted that issues related to the attestation report and ASN.1 logic such as V-RAV-VUL-002 and V-RAV-VUL-010, respectively, could have been caught with more extensive unit testing. We recommend that the developers carefully read relevant documentation linked to various issues (such as the Intel Attestation Service API documentation and ITU-T Rec. X.690) and augment their test suite with both positive and negative examples.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
RAVe	`664d724`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 5 - Jul. 10, 2023	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	0
Medium-Severity Issues	3	0
Low-Severity Issues	5	0
Warning-Severity Issues	3	0
Informational-Severity Issues	0	0
TOTAL	13	0

Table 2.4: Category Breakdown.

Name	Number
Data Validation	8
Logic Error	4
Usability Issue	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of RAVe's smart contracts. In our audit, we sought to answer the following questions:

- Does the ASN.1 parser correctly handle invalid data?
- Does the ASN.1 parser correctly parse the data values that are used by X.509 certificates?
- Are there any bugs in the ASN.1 parser, such as buffer overflow or indexing errors, that can be exploited by attackers to defeat the X.509 certificate validation mechanisms?
- Are there any flaws in the implementation of the signature verification scheme used for the X.509 certificates returned by IAS?
- Does the X.509 verification logic sufficiently validate signing certificates?
- Are there obvious ways for an attacker to forge a valid attestation report signature?
- Are there cases in which valid attestation report signatures will be falsely rejected by RAVe?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the `src` folder of the source code provided by the RAVe developers, which contains the smart contract implementation of the RAVe. During the audit, the Veridise auditors referred to third-party code used by the RAVe that are out-of-scope of the audit, but the auditors assumed that they have been implemented correctly.

Methodology. Veridise auditors inspected the provided tests, read the RAVe documentation*, and reviewed documents related to the X.509 certificate format†. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the RAVe developers to ask questions about the code.

* The Puffer Finance developers indicated that the documentation can be found at `https://docs.puffer.fi/tech/rave/`, but noted that the documentation is out-of-date with respect to the commit that was audited.
† The auditors found IETF RFC 5280 and ITU-T Rec. X.690 to be particularly helpful for this audit.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RAV-VUL-001	PKCS#1 v1.5 signature verification implementati...	High	Open
V-RAV-VUL-002	Valid reports with status OK will always be rej...	High	Open
V-RAV-VUL-003	Incorrect handling of optional fields may cause...	Medium	Open
V-RAV-VUL-004	X.509 certificate Issuer and Subject not checked	Medium	Open
V-RAV-VUL-005	readNodeLength does not handle length 0 data va...	Medium	Open
V-RAV-VUL-006	Unchecked assumption that leaf signature algor...	Low	Open
V-RAV-VUL-007	Missing check for X.509 KeyUsage extension	Low	Open
V-RAV-VUL-008	getNodeLength does not handle out-of-bounds int	Low	Open
V-RAV-VUL-009	Unchecked assumption that X.509 leaf public key...	Low	Open
V-RAV-VUL-010	ASN.1 data values with more than one identifier...	Low	Open
V-RAV-VUL-011	BadReportSignature error thrown if reconstructe...	Warning	Open
V-RAV-VUL-012	No logic to check certificate revocation	Warning	Open
V-RAV-VUL-013	bitstringAt() reverts on bitstrings that have l...	Warning	Open

4.1 Detailed Description of Issues

4.1.1 V-RAV-VUL-001: PKCS#1 v1.5 signature verification implementation is flawed

Severity	High	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	`X509Verifier.sol`, `RAVEBase.sol`		
Location(s)	verifyChildCert(), verifyReportSignature()		

The `verifyChildCert()` function in `X509Verifier` verifies that an X509 certificate was signed by a supplied public key. First, the function uses the parent's RSA public key to decrypt the signature, resulting in a byte buffer. The last 32 bytes are then matched against the SHA256 hash of the body of the given `childCertBody`.

Based on a sample signing certificate chain provided by the developer, it appears that `verifyChildCert` is a flawed implementation of the PKCS#1 v1.5 signature verification scheme with RSA-SHA256. Compared to the actual algorithm specified in section 8.2.2 of IETF RFC 8017, we have identified at least two flaws in RAVe's implementation.

First, the actual algorithm requires that the number of octets (bytes) of the signature matches the number of octets (bytes) of the modulus. However, there is no such check in the implementation.

Second, the decrypted signature is an "encoded message" consisting of some padding concatenated with an identifier of the signature algorithm used and the SHA256 hash of the certificate. The actual algorithm reconstructs the encoded message and then compares it to the decrypted version to perform the verification. RAVe's implementation, however, only checks the SHA256 hash of the encoded signature; the padding prepended to the hash is ignored. An identical implementation is in the `RAVEBase.verifyReportSignature()` function, except that a report returned by the Intel Attestation Service is verified instead of an X.509 certificate. This implementation is similarly flawed in that it does not check the padding.

Impact The difficulty of forging a valid signature is lower than it should be due to the missing checks. In the current implementation, an attacker can forge a signature by finding a signature such that the last 32 bytes of the decrypted signature will match the SHA256 hash of the report or certificate. In contrast, the actual algorithm would require all bytes of the decrypted signature to match the padding and the SHA256 hash. This may be made easier by the missing signature length check, where an attacker could potentially insert extra bytes into the signature.

There are many well-known, similar vulnerabilities related to flawed PKCS#1 v1.5 implementations that do not validate all parameters correctly. These vulnerabilities have been exploited in the past to forge signatures, as can found be in Bleichenbacher's e=3 attack and similar variants such as BERserk.

Recommendation The developers should change `verifyChildCert()` and `verifyReportSignature()` to check the full decrypted message; we refer the developers to section 9.2 of IETF RFC 8017 for the details. Note that the padding also includes the algorithm identifier, which is likely to be `sha256WithRsaEncryption` (for both `verifyChildCert()` and `verifyReportSignature()`) based on

```
/*
 * @dev Verifies an x509 certificate was signed (RSASHA256) by the supplied public
    key.
 * @param childCertBody The DER-encoded body (preimage) of the x509   child
    certificate
 * @param certSig The RSASHA256 signature of the childCertBody
 * @param parentMod The modulus of the parent certificate's public RSA key
 * @param parentExp The exponent of the parent certificate's public RSA key
 * @return Returns true if this childCertBody was signed by the parent's RSA private
    key
 */
function verifyChildCert(
  bytes memory childCertBody,
  bytes memory certSig,
  bytes memory parentMod,
  bytes memory parentExp
) public view returns (bool) {
  // Recover the digest using parent's public key
  (bool success, bytes memory res) = RSAVerify.rsarecover(parentMod, parentExp,
    certSig);
  // Digest is last 32 bytes of res
  bytes32 recovered = res.readBytes32(res.length - 32);
  return success && recovered == sha256(childCertBody);
}
```

Snippet 4.1: Definition of `verifyChildCert`

the the sample signing certificate chain and the IAS documentation (respectively). Also, the functions should return `false` if `sig.length != signingMod.length`.


```
/**
 * @inheritdoc IRave
 */
function verifyReportSignature(
  bytes memory report,
  bytes calldata sig,
  bytes memory signingMod,
  bytes memory signingExp
) public view returns (bool) {
  // Use signingPK to verify sig is the RSA signature over sha256(report)
  (bool success, bytes memory got) = RSAVerify.rsarecover(signingMod, signingExp, sig
    );
  // Last 32 bytes is recovered signed digest
  bytes32 recovered = got.readBytes32(got.length - 32);
  return success && recovered == sha256(report);
}
```

Snippet 4.2: Definition of `verifyReportSignature`

4.1.2 V-RAV-VUL-002: Valid reports with status OK will always be rejected

Severity	High	**Commit**	664d724
Type	Logic Error	**Status**	Open
File(s)	`JSONBuilder.sol`		
Location(s)	buildJSON()		

In order to verify the signature on a report returned by Intel's Attestation Service, the RAVe smart contracts will be provided with (1) an ABI-encoded representation of the fields of the attestation report JSON object; (2) the cryptographic signature of the report; (3) the ASN.1 DER-encoded X.509 leaf certificate; and (4) some other parameters. The signature will be verified by reconstructing the original JSON object and then checking the signature against the reconstructed JSON and the certificate.

Specifically, the fields of the report will be decoded into a `Values` struct, which will then be passed into the `buildJSON()` method to reconstruct the JSON report from the fields listed in `Values`. Currently, the logic will unconditionally include the `advisoryURL` and `advisoryIDs` in the reconstructed JSON. This will be inconsistent with the actual report when the `isvEnclaveQuoteStatus` is `OK`, as the `advisoryURL` and `advisoryIDs` fields will be omitted from the actual report.

```
1  function buildJSON(Values memory values) public pure returns (string memory json) {
2      json = string(/*...*/);
3      json = string(
4          abi.encodePacked(
5              json,
6              '","advisoryURL":"',
7              values.advisoryURL,
8              '","advisoryIDs":',
9              values.advisoryIDs,
10             ',"isvEnclaveQuoteStatus":"',
11             values.isvEnclaveQuoteStatus,
12             '","isvEnclaveQuoteBody":"',
13             values.isvEnclaveQuoteBody,
14             '"}'
15         )
16     );
17 }
```

Snippet 4.3: Relevant lines in `buildJSON()`

Impact When the status is `OK`, the reconstructed report will always be different from the actual report due to the extra `advisoryURL` and `advisoryIDs` fields. Thus, the signature verification will almost certainly fail as the SHA-256 digest of the reconstructed report will be different from that of the actual report. This may incentivize users to worsen their security status to `SW_HARDENING_NEEDED` in order to pass RAVe's verification logic.

Recommendation The reconstructed report should only include `advisoryURL` and `advisoryIDs` when the status matches one of the prerequisite statuses indicated in the Intel Attestation Service documentation.

4.1.3 V-RAV-VUL-003: Incorrect handling of optional fields may cause rejection of valid reports

Severity	Medium	**Commit**	664d724
Type	Logic Error	**Status**	Open
File(s)	`JSONBuilder.sol`		
Location(s)	buildJSON()		

Similar to V-RAV-VUL-002, the `buildJSON()` function does not correctly handle optional fields that may or may not be present under certain conditions. This may cause valid reports to be rejected. Below are a few examples according to the Intel Attestation Service API Documentation. Note that `_verifyReportContents()` only allows reports with status `OK` or `SW_HARDENING_NEEDED`, so we only discuss the examples with relation to those two statuses.

- `epidPseudonym` is always present in the reconstructed JSON, but it is only present in the actual JSON if the `isvEnclaveQuoteBody` has the `SIGNATURE_TYPE` bit set to 1 (linkable EPID signature).

```
function buildJSON(Values memory values) public pure returns (string memory json) {
    json = string(
        abi.encodePacked(
            '{"id":"',
            values.id,
            '","timestamp":"',
            values.timestamp,
            '","version":',
            values.version,
            ',"epidPseudonym":"',
            values.epidPseudonym
        )
    );
```

Snippet 4.4: Lines in `buildJSON()` where `epidPseudonym` is unconditionally included

- `pseManifestStatus` and `pseManifestHash` are optional fields that will be present in the actual report if the original attestation evidence payload has a `pseManifest` field. In particular, `pseManifestStatus` will only be included if `isvEnclaveQuoteStatus` is also set to `OK` or `SW_HARDENING_NEEDED`. However, `buildJSON()` does not include these fields if they exist.
- `platformInfoBlob` is an optional field that may be included if `pseManifestStatus` is set to `OUT_OF_DATE`, `REVOKED`, or `RL_VERSION_MISMATCH`.
- `nonce` is an optional field that will be present in the actual report if and only if the original attestation evidence payload also provides a `nonce`.

Impact The reconstructed report will be different from the actual report due to the extra optional fields. Thus, the signature verification will almost certainly fail as the SHA-256 digest of the reconstructed report will be different from that of the actual report.

Recommendation The developers should extend `buildJSON()` to handle the above cases, and they should carefully read the Intel Attestation Service API documentation to ensure that all fields that may be present in a report whose status is accepted by RAVe (`OK` and `SW_HARDENING_NEEDED`) are correctly handled.

4.1.4 V-RAV-VUL-004: X.509 certificate Issuer and Subject not checked

Severity	Medium	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	`X509Verifier.sol`		
Location(s)	verifySignedX509()		

The `verifySignedX509()` function is used to check whether an X.509 leaf certificate is valid. The certificates are assumed to be issued by the Intel Attestation Service. According to section 4.2.3 of the IAS documentation, the signing chain will have certificates matching two specific distinguished names:

1. CN=Intel SGX Attestation Report Signing CA, O=Intel Corporation, L=Santa Clara, ST=CA, C=US
2. CN=Intel SGX Attestation Report Signing, O=Intel Corporation, L=Santa Clara, ST=CA, C=US

The second distinguished name corresponds to the issuer of the `cert` argument to `verifySignedX509()`. However, the function does not check that the `Subject` (if it exists) and `Issuer`'s respective distinguished names match the ones described in the IAS documentation.

Note that IETF RFC 5280 section 4.1.2.4 also notes the following about the `Issuer` field:

> Certificate users MUST be prepared to process the issuer distinguished name and subject distinguished name (Section 4.1.2.6) fields to perform name chaining for certification path validation (Section 6). Name chaining is performed by matching the issuer distinguished name in one certificate with the subject name in a CA certificate.

Impact By omitting checks for the `Subject` and `Issuer`, it will be easier for an attacker to provide a fraudulent signing certificate in the event of a breach. For example, if the smart contracts interfacing with the RAVe smart contracts are compromised, and an attacker is able to supply an attacker-controlled parent certificate (e.g., such as a self-signed certificate), then the attacker can supply any signing certificate, thereby allowing the attacker to arbitrarily forge signatures with RAVe.

In contrast, with such checks in place, the attacker will be forced to provide both a parent and a signing certificate whose respective distinguished names must match the ones specified in the IAS documentation. This improves the security of the system by making such an attack slightly more difficult. Furthermore, even without a breach, having fixed strings in the preimage of the signature will make it more difficult for an attacker to forge a valid leaf certificate (which can then be used to forge a signature for the report). Additionally, such checks can also help guard against configuration errors.

Recommendation The `verifySignedX509()` function should check that the attributes in the `Issuer` and `Subject` match the ones in the IAS documentation. This also helps enforce specific structures on the certificate that could help prevent attacks that exploit V-RAV-VUL-008.

As an additional layer of security, the developers could also implement some basic "name chaining" and validate the `Issuer` and `Subject` of the parent certificate as well.

4.1.5 V-RAV-VUL-005: readNodeLength does not handle length 0 data values correctly

Severity	Medium	**Commit**	664d724
Type	Logic Error	**Status**	Open
File(s)	`ASN1Decode.sol`		
Location(s)	readNodeLength()		

The function `readNodeLength()` parses an ASN.1 DER data value (see ITU-T Rec. X.690) at a given index `ix` in the raw DER data in order to build a packed "pointer" data structure to the data value starting at `ix`. To build this pointer, `readNodeLength()` must parse the length octets representing the number of bytes of the contents octets.

The length has one of two forms. The short form consists of a single byte with the highest bit set to 1 (corresponding to the then-branch), with the other bits storing the length; and the long form consists of an initial byte that indicates the number of subsequent bytes will store the actual length. In either case, the length will then be used to calculate the starting and ending indices of the contents octets (`ixFirstContentByte` and `ixLastContentByte`). Lastly, the packed pointer will be constructed by a call to `NodePtr.getPtr()`. However, it appears that all

```
1  function readNodeLength(bytes memory der, uint256 ix) private pure returns (uint256)
       {
2      uint256 length;
3      uint80 ixFirstContentByte;
4      uint80 ixLastContentByte;
5      if ((der[ix + 1] & 0x80) == 0) {
6          length = uint8(der[ix + 1]);
7          ixFirstContentByte = uint80(ix + 2);
8          ixLastContentByte = uint80(ixFirstContentByte + length - 1);
9      } else {
10         uint8 lengthbytesLength = uint8(der[ix + 1] & 0x7F);
11         if (lengthbytesLength == 1) {
12             length = der.readUint8(ix + 2);
13         } else if (lengthbytesLength == 2) {
14             length = der.readUint16(ix + 2);
15         } else {
16             length = uint256(der.readBytesN(ix + 2, lengthbytesLength) >> (32 -
       lengthbytesLength) * 8);
17         }
18         ixFirstContentByte = uint80(ix + 2 + lengthbytesLength);
19         ixLastContentByte = uint80(ixFirstContentByte + length - 1);
20     }
21     return NodePtr.getPtr(ix, ixFirstContentByte, ixLastContentByte);
22 }
```

Snippet 4.5: Definition of `readNodeLength`

functions related to `NodePtr` assumes that the range starting at `ixFirstContentByte` and ending at `ixLastContentByte` is inclusive on both ends. This implies that the length is always greater than zero. There is no validation in `readNodeLength()` that the length is nonzero; and in fact, a length of zero is allowed in actual DER, as stated by section 8.1.3.4 of ITU-T Rec. X.690:

> In the short form, the length octets shall consist of a single octet in which bit 8 is zero and bits 7 to 1 encode the number of octets in the contents octets (which may be zero)

Impact If there is a data value with a length of zero, then `ixLastContentByte` will become `uint80(ixFirstContentByte - 1)`, i.e. the computed end of the contents octets will be before the start. This will lead to invalid pointers being generated by `NodePtr.getPtr()`, which can cause `X509Verifier` to traverse certificates incorrectly in `verifySignedX509()`. In particular, because the `NodePtr.nextSiblingOf()` uses the last index to calculate the pointer to the next ASN.1 data value in a sequence, code such as `verifySignedX509()` that traverses the ASN.1 data will read the data incorrectly.

Recommendation The developers should change `readNodeLength()` to construct the `ixLastContentByte` as an exclusive index and then make all methods that reference the `ixf()` function to be consistent with the change.

4.1.6 V-RAV-VUL-006: Unchecked assumption that leaf signature algorithm is PKCS#1 v1.5 with RSA-SHA256

Severity	Low	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	`X509Verifier.sol`		
Location(s)	verifySignedX509(), verifyChildCert()		

The `verifySignedX509()` function will call `verifyChildCert()` to perform signature validation on the leaf certificate using PKCS#1 v1.5 with RSA-SHA256. However, there is no check in `verifySignedX509()` that the `algorithm` field of the `signature` of the leaf certificate matches `sha256WithRSAEncryption`, or that it is even a valid object identifier value.

Impact If the Intel Attestation Service issues new certificates that are signed with an algorithm other than PKCS#1 v1.5 with RSA-SHA256, then `verifyChildCert()` will still attempt to use the algorithm. This creates the (very low probability) risk of the certificate being accepted when it should not be.

Furthermore, note that the `signatureAlgorithm` can be modified by attacker, who might try to exploit ASN.1 parsing vulnerabilities such as V-RAV-VUL-008 in order to work around the certificate verification logic.

Recommendation `verifySignedX509()` should be modified to check that `signatureAlgorithm` is indeed an object identifier and that the contents match the expected algorithm. For `sha256WithRSAEncryption`: the length is exactly 9 bytes, and the contents octets of the `signatureAlgorithm` should explicitly be compared with `2A 86 48 86 F7 0D 01 01 0B` (as defined in IETF RFC 4055 section 6).

4.1.7 V-RAV-VUL-007: Missing check for X.509 KeyUsage extension

Severity	Low	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	`X509Verifier.sol`		
Location(s)	verifySignedX509()		

The `verifySignedX509()` function is used to check whether an X.509 certificate is valid. However, these checks do not include a check for the `KeyUsage` extension described by IETF RFC 5280:

> The key usage extension defines the purpose (e.g., encipherment, signature, certificate signing) of the key contained in the certificate. The usage restriction might be employed when a key that could be used for more than one operation is to be restricted. [. . .] Conforming CAs MUST include this extension in certificates that contain public keys that are used to validate digital signatures on other public key certificates or CRLs.

Furthermore, the `KeyUsage` extension imposes the following constraint:

> The digitalSignature bit is asserted when the subject public key is used for verifying digital signatures, other than signatures on certificates (bit 5) and CRLs (bit 6), such as those used in an entity authentication service, a data origin authentication service, and/or an integrity service.

Arguably, the Intel Attestation Service (IAS) is a "data origin authentication service", so the signing certificate should assert the `digitalSignature` bit (indeed, a sample IAS signing certificate provided by the developers does assert this bit). However, `verifySignedX509()` does not check that this bit is asserted.

Impact If an attacker breaches IAS and steals a leaf certificate signed by the IAS CA certificate that is not used to sign reports, then the attacker can forge valid signatures which may be accepted by the RAVe smart contracts, even though such a certificate should not be allowed to sign messages.

Recommendation To reduce the attack surface, `verifySignedX509()` should be extended to ensure that (1) a `KeyUsage` extension is present in the signing certificate; and (2) the `digitalSignature` bit of the `KeyUsage` extension bit is asserted.

4.1.8 V-RAV-VUL-008: getNodeLength does not handle out-of-bounds integers

Severity	Low	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	ASN1Decode.sol		
Location(s)	getPtr(), readNodeLength()		

The `NodePtr.getPtr()` helper function is used to pack three `uint80`s into a single `uint256`, representing a "pointer" into DER-encoded data (see section 8). These "pointers" consist of the index (in the raw DER data) of the start of a data value, the index of the start of the contents octets, and the index of the end of the contents octets.

However, the arguments to `getPtr()` are actually of type `uint256`, not `uint80`. Furthermore, if the caller passes in numbers that are larger than the max `uint80` value, the high bits will not be cleared first, so in effect there will be a "buffer overflow" into the integer components stored in the higher bit positions. For example, if `_ixs` is set to `0x010000000000000000` and `_ixf` and `_ixl` are set to 0, then the packed value of `_ixs` will be zero but the packed value of `_ixf` will be 1. The only caller of `NodePtr.getPtr()` is `ASN1Decode.readNodeLength()`, where `_ixs` is set to `_ix`

```
function getPtr(uint256 _ixs, uint256 _ixf, uint256 _ixl) internal pure returns (
    uint256) {
    _ixs |= _ixf << 80;
    _ixs |= _ixl << 160;
    return _ixs;
}
```

Snippet 4.6: Implementation of `getPtr()`

(an unbounded `uint256`). The `_ixf` and `_ixl` parameters will be set to `ixFirstContentByte` and `ixLastContentByte`, respectively, which can be truncated if `ix` or `length` are sufficiently large. In conclusion, the overall effect is:

- If `ix` is larger than a `uint80`, this will result in "buffer overflow" in `getPtr()`.
- If `ixFirstContentByte` or `ixLastContentByte` is actually truncated, then an invalid pointer will be constructed by `getPtr()`.

Impact If an attacker provides an invalid or corrupt X.509 certificate, they may be able to cause `getPtr()` to produce an invalid pointer; consequently, the DER data may be parsed incorrectly. Assuming that such a malicious certificate can still be verified by its parent certificate, this could potentially be exploited to work around the subsequent X.509 validation mechanisms in `verifySignedX509()` or make it easier to forge signatures. For example, an attacker may choose to overflow the indices so that they may insert extra "garbage" data in the certificate, making it easier to forge a signature.

Recommendation The `ix` should be validated to fit within a `uint80`, and the `ixFirstContentByte` and `ixLastContentByte` should be validated to fit within a `uint80` before they are truncated. if such validation fails, then the DER data is practically invalid anyways, as this would require the raw DER data to be at least of size 2^{80} bytes (an unrealistic amount of memory).

```
function readNodeLength(bytes memory der, uint256 ix) private pure returns (uint256)
    {
    uint256 length;
    uint80 ixFirstContentByte;
    uint80 ixLastContentByte;
    if ((der[ix + 1] & 0x80) == 0) {
        length = uint8(der[ix + 1]);
        ixFirstContentByte = uint80(ix + 2);
        ixLastContentByte = uint80(ixFirstContentByte + length - 1);
    } else {
        uint8 lengthbytesLength = uint8(der[ix + 1] & 0x7F);
        if (lengthbytesLength == 1) {
            length = der.readUint8(ix + 2);
        } else if (lengthbytesLength == 2) {
            length = der.readUint16(ix + 2);
        } else {
            length = uint256(der.readBytesN(ix + 2, lengthbytesLength) >> (32 -
    lengthbytesLength) * 8);
        }
        ixFirstContentByte = uint80(ix + 2 + lengthbytesLength);
        ixLastContentByte = uint80(ixFirstContentByte + length - 1);
    }
    return NodePtr.getPtr(ix, ixFirstContentByte, ixLastContentByte);
}
```

Snippet 4.7: Definition of `readNodeLength()`

4.1.9 V-RAV-VUL-009: Unchecked assumption that X.509 leaf public key algorithm is RSA

Severity	Low	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	`X509Verifier.sol`		
Location(s)	verifySignedX509()		

The `verifySignedX509()` function will perform validation on the X.509 certificate and then extract the RSA modulus and exponent of the `subjectPublicKey`. However, it does not check that the `algorithm` field of the `subjectPublicKeyInfo` matches the RSA algorithm. Based on a sample signing certificate provided by the developers, the `algorithm` is likely supposed to be `rsaEncryption`.

Impact If the Intel Attestation Service issues new certificates that use an algorithm other than RSA, then the RSA modulus and exponent extraction steps in `verifySignedX509()` will return bogus results. Consequently, it is likely that signature verification of the report will fail and cause a `BadReportSignature` error to be thrown.

Recommendation The developers should insert checks that the `algorithm` field of the `subjectPublicKeyInfo` in the leaf certificate is the constant equal to `rsaEncryption` (see definition in IETF RFC 4055 section 6). Specifically, the content octets of `algorithm` should be equal to `2A 86 48 86 F7 0D 01 01 01`. The contract should throw an error if the `algorithm` does not match the expected value.

4.1.10 V-RAV-VUL-010: ASN.1 data values with more than one identifier octet not correctly handled

Severity	Low	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)	`ASN1Decode.sol`		
Location(s)	readNodeLength()		

The function `readNodeLength()` parses an ASN.1 DER data value at a given index `ix` in the raw DER data. The identifier octet refers to the first octet(s) of an ASN.1 encoded node, which specifies the class, form, and tag number of the encoded object. Although it is possible for an ASN.1 data value to have more than one identifier octet when the tag number is 31 or greater (see section 8.1.2 of ITU-T Rec. X.690), `readNodeLength()` seems to assume that there can only be one identifier octet, as evidenced by how the length is read from index `ix + 1`.

```
1  function readNodeLength(bytes memory der, uint256 ix) private pure returns (uint256)
       {
2      uint256 length;
3      uint80 ixFirstContentByte;
4      uint80 ixLastContentByte;
5      if ((der[ix + 1] & 0x80) == 0) {
6          length = uint8(der[ix + 1]);
7          ixFirstContentByte = uint80(ix + 2);
8          ixLastContentByte = uint80(ixFirstContentByte + length - 1);
9      } else {
10         uint8 lengthbytesLength = uint8(der[ix + 1] & 0x7F);
11         if (lengthbytesLength == 1) {
12             length = der.readUint8(ix + 2);
13         } else if (lengthbytesLength == 2) {
14             length = der.readUint16(ix + 2);
15         } else {
16             length = uint256(der.readBytesN(ix + 2, lengthbytesLength) >> (32 -
       lengthbytesLength) * 8);
17         }
18         ixFirstContentByte = uint80(ix + 2 + lengthbytesLength);
19         ixLastContentByte = uint80(ixFirstContentByte + length - 1);
20     }
21     return NodePtr.getPtr(ix, ixFirstContentByte, ixLastContentByte);
22 }
```

Snippet 4.8: Definition of `readNodeLength()`

Impact If there is an ASN.1 data value that uses an identifier consisting of more than one octet, then the length (and therefore first and last contents octets indices) will be read incorrectly. This can result in potential misinterpretation of the DER-encoded data, which can be exploited by attackers.

Recommendation Based on the structure of an X.509 certificate, it seems unlikely for a certificate to include an ASN.1 data value that has a tag number equal to 31 or larger (and

therefore multiple identifier octets). Thus, we recommend rejecting the ASN.1 data if `(der[ix] & 0x1F) == 0x1F` (e.g., by reverting or throwing an error). This will ensure that such tag numbers cannot be considered, as there can only be multiple identifier octets if the lowest five bits of the first identifier octet are all set to 1's.

4.1.11 V-RAV-VUL-011: BadReportSignature error thrown if reconstructed JSON is inconsistent, but signature is valid

Severity	Warning	**Commit**	664d724
Type	Usability Issue	**Status**	Open
File(s)	`RAVE.sol`		
Location(s)	verifyRemoteAttestation()		

If the reconstructed JSON contains extra keys or is missing keys that should otherwise be in the report, then `verifyRemoteAttestation()` will throw a `BadReportSignature` error even though the cause is likely due to the JSON reconstruction or status, not due to the signature verification failing.

```
1  // Decode the encoded report JSON values to a Values struct and reconstruct the
       original JSON string
2  (Values memory reportValues, bytes memory reportBytes) = _buildReportBytes(report);
3
4  // Verify the report was signed by the SigningPK
5  if (!verifyReportSignature(reportBytes, sig, signingMod, signingExp)) {
6      revert BadReportSignature();
7  }
8
9  // Verify the report's contents match the expected
10 payload = _verifyReportContents(reportValues, mrenclave, mrsigner);
```

Snippet 4.9: Relevant lines in `verifyRemoteAttestation()`

Impact If there is an issue in reconstructing the JSON, the error message will be a "false positive" in that it identifies the issue incorrectly. This can cause developers or users to waste time investigating a supposed signature error when in fact the problem lies with key handling in the report.

Recommendation To clarify the error, the call to `_verifyReportContents()` can be moved above the call to `verifyReportSignature()`. Since the former checks the `isvEnclaveQuoteStatus` (which then will make some fields optional or mandatory), this will aid debugging by making the error reporting more accurate.

4.1.12 V-RAV-VUL-012: No logic to check certificate revocation

Severity	Warning	**Commit**	664d724
Type	Data Validation	**Status**	Open
File(s)		N/A	
Location(s)		N/A	

There does not appear to be any logic in the RAVe smart contracts that checks the signing certificate chains against certificate revocation lists (CRLs). The developers noted that they currently do not have any such logic in the callers of RAVe, either.

Impact Section 3.3 of IETF RFC 5280 states that the purpose of the CRL is:

> When a certificate is issued, it is expected to be in use for its entire validity period. However, various circumstances may cause a certificate to become invalid prior to the expiration of the validity period. Such circumstances include change of name, change of association between subject and CA (e.g., an employee terminates employment with an organization), and compromise or suspected compromise of the corresponding private key. Under such circumstances, the CA needs to revoke the certificate.

However, without logic to check the CRL, the RAVe smart contracts will still accept certificates that have been revoked.

Recommendation If feasible, we recommend that the developers implement checks against the CRLs indicated in the X.509 certificate signing chain, either in RAVe or the code that uses RAVe.

4.1.13 V-RAV-VUL-013: bitstringAt() reverts on bitstrings that have length not multiple of 8

Severity	Warning	**Commit**	664d724
Type	Logic Error	**Status**	Open
File(s)	`ASN1Decode.sol`		
Location(s)	bitstringAt()		

The `bitstringAt()` function is used to parse a DER-encoded bitstring from raw DER data. In DER, the contents octet consists of an "initial octet" and the "subsequent octets", where subsequent octets contain the bitstring data and the initial octet indicates the number of unused bits in the last subsequent octet (see section 8.6 of ITU-T Rec. X.690). However, `bitstringAt()` requires that the initial octet is equal to 0, so that there are no unused bits in the last subsequent octet. This constrains the length of the bitstring to be a multiple of 8. If there are unused bits, then the function will revert.

```
1 function bitstringAt(bytes memory der, uint256 ptr) internal pure returns (bytes
      memory) {
2     require(der[ptr.ixs()] == 0x03, "Not type BIT STRING");
3     // Only 00 padded bitstr can be converted to bytestr!
4     require(der[ptr.ixf()] == 0x00);
5     uint256 valueLength = ptr.ixl() + 1 - ptr.ixf();
6     return der.substring(ptr.ixf() + 1, valueLength - 1);
7 }
```

Snippet 4.10: Definition of `bitstringAt()`

Impact Currently, the only use of `bitstringAt()` is to extract the DER-encoded RSA public key of the leaf certificate. This currently has no impact as DER data is always made up of octets (not individual bits). However, if in the future `bitstringAt()` is used to read bitstrings that are not always guaranteed to have lengths that are multiples of 8, then the function could revert and cause the RAVe smart contracts to reject valid leaf certificates.

Recommendation The developers may want to change `bitstringAt()` to correctly handle the last subsequent octet if they intend for RAVe to read other bitstring fields in the future.