BLOCKSEC

Security Audit
Report for Puffer - Protocol v2

Date: September 9, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Puffer
Target	Puffer-Protocol v2

Version History

Version	Date	Description
1.0	September 9, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-09-09 09:17:06+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repositories [1] [2] of Puffer-Protocol v2 of Puffer.

Puffer-Protocol v2 is a cross-chain restaking and attestation integration that connects Puffer validator operations with Automata-verified TEE sessions. The audit scope covers two main layers of code. The first is the Puffer contract layer, which handles validator registration and provisioning, Guardian enclave key rotation, withdrawal batch processing, Validator Ticket accounting, reward distribution, and L1/L2 reward bridging . The second layer is the Automata TEE workload measurement layer, which verifies TDX/SNP reports, TPM quotes, AK and session-key delegation, base image and workload policies, PCR constraints, and session lifecycle operations consumed by Puffer's Guardian trust path.

Specifically, for code in `puffer-contracts-updates-private`, the audit focuses on the codes located in the following directories/files:

- l2-contracts/src
- mainnet-contracts/src

For code in `automata-tee-workload-measurement`, the audit focuses on the codes located in the following directories/files:

- src/SessionRegistry.sol
- src/BaseImageRegistry.sol
- src/WorkloadRegistry.sol
- src/TeeVerifier.sol
- src/SignatureVerifier.sol
- src/KeyResolver.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. For `Version 2` of `automata-tee-workload-measurement`, newly introduced features and refactoring changes are outside the audit scope. Code prior to and including the

[1] https://github.com/PufferFinance/puffer-contracts-updates-private

[2] https://github.com/automata-network/automata-tee-workload-measurement

baseline version (Version 0), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
puffer-contracts-updates-private	Version 0	380600060cd231fd8616ba167e674d4140486dbb
	Version 1	08ecce346f092f4ba5f0f71399b08d5d065a48bb
	Version 2	b3a53d9ce026d1c6a18fd02e53b0c6c354e6b12b
automata-tee-workload-measurement	Version 1	101d828ded943dc75865e65907e057de1532da9f
	Version 2	0e0991b656fd7568ebfff9a9cf3554899695073a

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management

- Whitelist and blacklist mechanisms
- Initialization consistency
- Improper use of the proxy system
- Reentrancy
- Denial of Service (DoS)
- Untrusted external call and control flow
- Exception handling
- Data handling and flow
- Events operation
- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

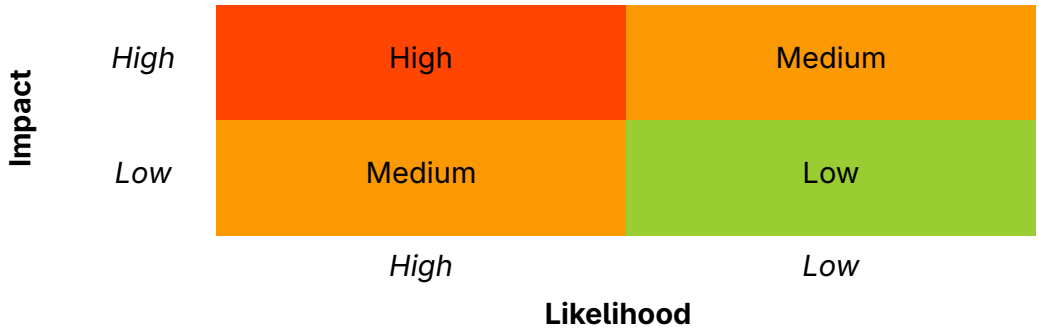
Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

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

Table 1.1: Vulnerability Severity Classification



- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **nine** potential security issues. Besides, we have **eight** recommendations and **fourteen** notes.

- Medium Risk: 4
- Low Risk: 5
- Recommendation: 8
- Note: 14

ID	Severity	Description	Category	Status
1	Medium	Incorrect accounting when reverting partially claimed intervals	Security Issue	Fixed
2	Medium	Incorrect reversal due to insufficient L1 interval state	Security Issue	Fixed
3	Medium	Replay of stale Guardian rotation proofs	Security Issue	Fixed
4	Medium	Lack of workload status check	Security Issue	Fixed
5	Low	Lack of check on duplicate enclave addresses	Security Issue	Fixed
6	Low	Circumvention of static PCR policy	Security Issue	Fixed
7	Low	Lack of freshness validation for DCAP ZK proofs	Security Issue	Fixed
8	Low	Lack of check on debug requirements	Security Issue	Fixed
9	Low	Front-running of base image and workload registration	Security Issue	Fixed
10	-	Align the documentation with the current implementation	Recommendation	Fixed
11	-	Add a maximum Workload Session TTL	Recommendation	Fixed
12	-	Prevent key reuse during session rotation	Recommendation	Fixed
13	-	Bind successor keys to owner authorization	Recommendation	Confirmed
14	-	Add RSA exponent validation	Recommendation	Fixed
15	-	Flush removed Guardian reward tokens	Recommendation	Fixed
16	-	Add zero-address checks for immutable dependencies	Recommendation	Confirmed
17	-	Add explicit Guardian action domains	Recommendation	Confirmed
18	-	Verify the workload before allowlist operations	Note	-
19	-	Atomic upgrade for `PufferDepositorV2` initialization	Note	-
20	-	Session lifecycle does not preserve workload continuity	Note	-

21	-	DCAP verifier integration assumes a zero-fee backend	Note	-
22	-	Workload base-image lists are identifier policies	Note	-
23	-	Guardian enclave key is reused for encryption and signing	Note	-
24	-	Potential centralization risks	Note	-
25	-	Trust assumptions of enclave keys	Note	-
26	-	Session and enclave keys must not be exportable	Note	-
27	-	Monitor VT balances to prevent blocked withdrawals	Note	-
28	-	Check the base image status separately	Note	-
29	-	Migration requirements in the upgrade script	Note	-
30	-	Assumption of vTPM Migration	Note	-
31	-	Trust assumption of GCP TEE-to-vTPM binding	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect accounting when reverting partially claimed intervals

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `L2RewardManager`, the function `_isClaimingLocked()` derives an interval's lock status using the current global `claimingDelay` rather than an interval-specific unlock time. Once the `claimingDelay` is increased, the interval can be frozen and reverted.

When function`_revertInterval()` is invoked, it returns the interval's original reward amount to L1 and deletes the interval status. If any rewards from that interval have already been claimed, those prior claims are not accounted for when calculating the amount returned to L1. As rewards across intervals are held in a shared balance rather than in interval-specific reserves, the rollback can consume assets backing later intervals, while deletion of the interval status prevents users from claiming its remaining rewards.

```
85          if (_isClaimingLocked(claimOrders[i].intervalId)) {
86              // If timeBridged is 0, the interval has been reverted (permanently locked)
87              uint256 lockedUntil = epochRecord.timeBridged == 0 ? 0 : epochRecord.timeBridged +
                    $.claimingDelay;
88
```

```
89              revert ClaimingLocked({
90                  intervalId: claimOrders[i].intervalId,
91                  account: claimOrders[i].account,
92                  lockedUntil: lockedUntil
93              });
94          }
95
96          // Alice may run many Puffer validators in the same interval `totalETHEarned = sum(
                aliceValidators)`
97          // The leaf is: keccak256(abi.encode(AliceAddress, isL1Contract, totalETHEarned))
98          bytes32 leaf = keccak256(
99              bytes.concat(
100                 keccak256(abi.encode(claimOrders[i].account, claimOrders[i].isL1Contract,
                        claimOrders[i].amount))
101             )
102         );
103         if (!MerkleProof.verifyCalldata(claimOrders[i].merkleProof, epochRecord.rewardRoot,
                leaf)) {
104             revert InvalidProof();
105         }
106
107         // Mark it claimed and transfer the tokens
108         $.claimedRewards[claimOrders[i].intervalId][claimOrders[i].account] = true;
109
110         uint256 amountToTransfer = (claimOrders[i].amount * epochRecord.ethToPufETHRate) / 1
                ether;
111
112         recipient = recipient == address(0) ? claimOrders[i].account : recipient;
113
114         // if the custom claimer is set, then transfer the tokens to the set claimer
115         // First we transfer any remaining xPufETH in this contract to the recipient
116         uint256 xPufETHBalance = IERC20(xPufETH).balanceOf(address(this));
117         if (xPufETHBalance > 0) {
118             if (xPufETHBalance >= amountToTransfer) {
119                 IERC20(xPufETH).safeTransfer(recipient, amountToTransfer);
120             } else {
121                 IERC20(xPufETH).safeTransfer(recipient, xPufETHBalance);
122                 IERC20($.pufETHOFT).safeTransfer(recipient, amountToTransfer - xPufETHBalance);
123             }
124         } else {
125             IERC20($.pufETHOFT).safeTransfer(recipient, amountToTransfer);
```

Listing 2.1: puffer-contracts-updates-private/l2-contracts/src/L2RewardManager.sol

```
356     if (newDelay > 72 hours) {
357         revert InvalidDelayPeriod();
358     }
359     RewardManagerStorage storage $ = _getRewardManagerStorage();
360
361     emit ClaimingDelayChanged({ oldDelay: $.claimingDelay, newDelay: newDelay });
362     $.claimingDelay = newDelay;
363 }
364
```

```
365   function _isClaimingLocked(bytes32 intervalId) internal view returns (bool) {
366       RewardManagerStorage storage $ = _getRewardManagerStorage();
367
368       uint256 timeBridged = $.epochRecords[intervalId].timeBridged;
369
370       // If the timeBridged is 0, the interval is either reverted, or has not been bridged yet
371       // we consider that the claiming is locked in both cases
372       if (timeBridged == 0) {
373           return true;
374       }
375
376       return block.timestamp < timeBridged + $.claimingDelay;
377   }
378
379   function _freezeClaimingForInterval(uint256 startEpoch, uint256 endEpoch) internal {
380       RewardManagerStorage storage $ = _getRewardManagerStorage();
381
382       bytes32 intervalId = getIntervalId(startEpoch, endEpoch);
383
384       // Revert if the claiming is not locked
385       if (!_isClaimingLocked(intervalId)) {
386           revert UnableToFreezeInterval();
387       }
388
389       // revert for non-existing interval
390       if ($.epochRecords[intervalId].rewardRoot == bytes32(0)) {
391           revert UnableToFreezeInterval();
392       }
393
394       // To freeze the claiming, we set the timeBridged to 0
395       $.epochRecords[intervalId].timeBridged = 0;
```

Listing 2.2: puffer-contracts-updates-private/l2-contracts/src/L2RewardManager.sol

```
403   function _revertInterval(uint256 startEpoch, uint256 endEpoch) internal {
404       RewardManagerStorage storage $ = _getRewardManagerStorage();
405
406       bytes32 intervalId = getIntervalId(startEpoch, endEpoch);
407
408       EpochRecord memory epochRecord = $.epochRecords[intervalId];
409
410       // We only want to revert the frozen intervals, if the interval is not frozen, we revert
411       if (epochRecord.timeBridged != 0 && epochRecord.rewardRoot != bytes32(0)) {
412           revert UnableToRevertInterval();
413       }
414
415       if (epochRecord.rewardRoot == bytes32(0)) {
416           revert UnableToRevertInterval();
417       }
418
419       // Delete the epoch record before to minimize re-entrancy
420       delete $.epochRecords[intervalId];
421
```

```
422       // We bridge the pufETH back to the L1
423       // We don't need to approve since the oft is itself the pufETH token
424       IPufETH($.pufETHOFT).send{ value: msg.value }(
425           IOFT.SendParam({
426               dstEid: $.destinationEID,
427               to: bytes32(uint256(uint160(L1_REWARD_MANAGER))),
428               amountLD: epochRecord.pufETHAmount,
429               minAmountLD: 0,
```

Listing 2.3: puffer-contracts-updates-private/l2-contracts/src/L2RewardManager.sol

Impact Remaining rewards may become unclaimable, and later reward intervals may be blocked due to insufficient funds.

Suggestion Store an interval-specific unlock time and remaining liability, and prevent full rollback after any claim.

Feedback from the project A new `claimableAt` field is introduced to fix the issue. However, intervals created before the upgrade may experience the same issue during the maximum 72-hour delay window. To mitigate this risk, the project will schedule the upgrade outside the reward distribution window, ensuring that no pre-upgrade intervals remain subject to changes in the global delay.

2.1.2 Incorrect reversal due to insufficient L1 interval state

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `L1RewardManager`, rewards can be minted every 20 hours while the L2 claim delay can be configured up to 72 hours, allowing several intervals to remain locked and reversible simultaneously. However, contract `L1RewardManager` tracks only `lastIntervalEndEpoch` and `currentIntervalEndEpoch`. When the function `lzCompose()` receives a reversal for an interval, it sets `currentIntervalEndEpoch` back to `lastIntervalEndEpoch` without checking whether the reverted interval is the latest pending interval.

As a result, reverting an older interval can inadvertently revert the latest interval (i.e., `lastIntervalEndEpoch`) on L1, even though the latest interval remains active on L2, which is incorrect.

```
352   function _setClaimingDelay(uint256 newDelay) internal {
353       if (newDelay < 6 hours) {
354           revert InvalidDelayPeriod();
355       }
356       if (newDelay > 72 hours) {
357           revert InvalidDelayPeriod();
358       }
359       RewardManagerStorage storage $ = _getRewardManagerStorage();
360
361       emit ClaimingDelayChanged({ oldDelay: $.claimingDelay, newDelay: newDelay });
362       $.claimingDelay = newDelay;
363   }
```

Listing 2.4: puffer-contracts-updates-private/l2-contracts/src/L2RewardManager.sol

```
113         if (($.lastRewardMintTimestamp + $.allowedRewardMintFrequency) > block.timestamp) {
114             revert NotAllowedMintFrequency();
115         }
116
117         // Update lastIntervalEndEpoch to the previous currentIntervalEndEpoch
118         $.lastIntervalEndEpoch = $.currentIntervalEndEpoch;
119
120         // Check that startEpoch is greater than the last processed end epoch
121         if (params.startEpoch <= $.lastIntervalEndEpoch) {
122             revert InvalidStartEpoch();
123         }
124
125         // Update current interval end epoch
126         $.currentIntervalEndEpoch = params.endEpoch;
127
128         // Update the last mint timestamp
129         $.lastRewardMintTimestamp = uint48(block.timestamp);
```

Listing 2.5: puffer-contracts-updates-private/mainnet-contracts/src/L1RewardManager.sol

```
205         // We decode the actual message to get the amount of shares(pufETH) and the ETH amount.
206         L2RewardManagerStorage.EpochRecord memory epochRecord =
207             abi.decode(actualMessage, (L2RewardManagerStorage.EpochRecord));
208
209         // This contract has already received the pufETH from pufETHAdapter after bridging back to
                L1
210         // The PufferVault will burn the pufETH from this contract and subtract the ETH amount from
                the ethRewardsAmount
211         PUFFER_VAULT.revertMintRewards({ pufETHAmount: epochRecord.pufETHAmount, ethAmount:
                epochRecord.ethAmount });
212
213         // When reverted, set current end epoch back to last end epoch
214         RewardManagerStorage storage $ = _getRewardManagerStorage();
215         $.currentIntervalEndEpoch = $.lastIntervalEndEpoch;
```

Listing 2.6: puffer-contracts-updates-private/mainnet-contracts/src/L1RewardManager.sol

```
317         bytes32 intervalId = getIntervalId(params.startEpoch, params.endEpoch);
318
319         $.epochRecords[intervalId] = EpochRecord({
320             ethToPufETHRate: params.ethToPufETHRate,
321             startEpoch: uint104(params.startEpoch),
322             endEpoch: uint104(params.endEpoch),
323             timeBridged: uint48(block.timestamp),
324             rewardRoot: params.rewardsRoot,
325             pufETHAmount: uint128(amount),
326             ethAmount: uint128(params.rewardsAmount)
327         });
```

Listing 2.7: puffer-contracts-updates-private/l2-contracts/src/L2RewardManager.sol

Impact The reward accounting and interval records can become incorrect.

Suggestion Track pending intervals on L1, or allow only the latest interval to be reverted.

2.1.3 Replay of stale Guardian rotation proofs

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `GuardianModule`, the function `rotateGuardianKey()` accepts a valid Session proof from any relayer and only checks whether the committed `blockNumber` is within `FRESHNESS_BLOCKS`. However, the function does not use a nonce or record consumed proofs. A valid proof can therefore be replayed during the freshness window to restore an old guardian enclave key. If the restored key is unavailable or compromised, the rollback may reduce quorum availability or restore a compromised signer to the active signer set.

```
367   function rotateGuardianKey(uint256 blockNumber, bytes calldata pubKey, GuardianSessionProof
          calldata proof)
368       external
369   {
370       // The ownerKey is provided by the operator during TEE workload deployment (via atakit
              deploy --owner-private-key).
371       // The CVM agent binds the ownerKey to the session, so a valid session signature attests
              that the message originated from the operator.
372       require(proof.ownerKey.typeId == ALGO_ID_ES256K, InvalidECDSAPubKey());
373       require(proof.ownerKey.key.length == _ECDSA_KEY_LENGTH, InvalidECDSAPubKey());
374
375       address guardian = address(uint160(uint256(keccak256(proof.ownerKey.key[1:]))));
376
377       if (!_guardians.contains(guardian)) {
378           revert Unauthorized();
379       }
380
381       if (pubKey.length != _ECDSA_KEY_LENGTH) {
382           revert InvalidECDSAPubKey();
383       }
384
385       if ((block.number - blockNumber) > FRESHNESS_BLOCKS) {
386           revert StaleEvidence();
387       }
388
389       // Since rotateGuardianKey is called infrequently, blockNumber-based freshness is used for
              replay protection instead of a nonce.
390       bytes32 signedMessageHash =
391           keccak256(abi.encode("ROTATE_GUARDIAN_KEY", address(this), block.chainid, blockNumber,
                  pubKey));
392       bool isValid = SESSION_REGISTRY.verifySessionSignature(
393           proof.sessionId, proof.sessionKey, signedMessageHash, proof.signature
394       );
395       if (!isValid) {
396           revert InvalidSignature();
```

```
397         }
398
399         bytes32 ownerFingerprint = SESSION_REGISTRY.getSessionOwner(proof.sessionId);
400         require(ownerFingerprint == LibKey.computeKeyFingerprint(proof.ownerKey),
                InvalidECDSAPubKey());
401
402         CVMSession memory session = SESSION_REGISTRY.getSession(proof.sessionId);
403         require(_allowedWorkloads[session.workloadId], WorkloadNotAllowed());
404
405         // pubKey[1:] means we need to strip the first byte '0x' if we want to get the correct
                address
406         address computedAddress = address(uint160(uint256(keccak256(pubKey[1:]))));
407
408         _guardianEnclaves[guardian].enclaveAddress = computedAddress;
409         _guardianEnclaves[guardian].enclavePubKey = pubKey;
410
411         emit RotatedGuardianKey(guardian, computedAddress, pubKey);
```

Listing 2.8: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

Impact Guardian signer state can be rolled back, potentially reducing quorum availability or restoring a compromised signer within the proof's validity window.

Suggestion Bind a nonce and the old key to each proof, and consume the nonce on success.

Feedback from the project The project fixes this issue by tracking the last block number a guardian successfully rotated at, and requiring the new session proof to commit to a strictly higher block number.

2.1.4 Lack of workload status check

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `GuardianModule`, function `validateGuardiansEnclaveSignatures()` validates signatures against the stored enclave addresses. However, the function does not check whether their associated workloads remain allowed in the mapping `_allowedWorkloads`. The mapping is checked only when the function `rotateGuardianKey()` registers an enclave key. Updating this mapping does not invalidate existing enclave addresses. As a result, enclave keys from disallowed workloads remain valid.

```
283   function setAllowedWorkload(bytes32 workloadId, bool allowed) external restricted {
284       require(workloadId != bytes32(0), WorkloadNotAllowed());
285       _allowedWorkloads[workloadId] = allowed;
286       emit WorkloadAllowanceChanged(workloadId, allowed);
287   }
```

Listing 2.9: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

```
263   function validateGuardiansEnclaveSignatures(bytes[] calldata enclaveSignatures, bytes32
          signedMessageHash)
```

```
264         public
265         view
266         returns (bool)
267     {
268         return _validateSignatures(getGuardiansEnclaveAddresses(), enclaveSignatures,
                signedMessageHash);
269     }
```

Listing 2.10: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

Impact Enclaves from revoked workloads can continue participating in Guardian authorization.

Suggestion Associate each guardian enclave key with its workload or session and verify its current status during signature validation.

Feedback from the project The project confirms that the workload allowlist is intentionally enforced only during enclave key registration to avoid immediate quorum loss when retiring a workload. The project adds the DAO-only function `revokeGuardianEnclave()` to invalidate a guardian's affected enclave key without changing guardian membership. The guardian's enclave signatures will no longer count toward the threshold until it rotates onto an allowed workload.

2.1.5 Lack of check on duplicate enclave addresses

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `GuardianModule`, the function `rotateGuardianKey()` associates a guardian with an enclave signing address authorized by a valid session proof. However, the function does not prevent multiple guardians from registering the same enclave address. Consequently, a single enclave key may represent multiple guardians and produce signatures that count toward multiple positions in the guardian set, reducing the number of distinct enclave keys required to satisfy the threshold.

```
367     function rotateGuardianKey(uint256 blockNumber, bytes calldata pubKey, GuardianSessionProof
            calldata proof)
368         external
369     {
370         // The ownerKey is provided by the operator during TEE workload deployment (via atakit
                deploy --owner-private-key).
371         // The CVM agent binds the ownerKey to the session, so a valid session signature attests
                that the message originated from the operator.
372         require(proof.ownerKey.typeId == ALGO_ID_ES256K, InvalidECDSAPubKey());
373         require(proof.ownerKey.key.length == _ECDSA_KEY_LENGTH, InvalidECDSAPubKey());
374
375         address guardian = address(uint160(uint256(keccak256(proof.ownerKey.key[1:]))));
376
377         if (!_guardians.contains(guardian)) {
378             revert Unauthorized();
```

```
379         }
380
381         if (pubKey.length != _ECDSA_KEY_LENGTH) {
382             revert InvalidECDSAPubKey();
383         }
384
385         if ((block.number - blockNumber) > FRESHNESS_BLOCKS) {
386             revert StaleEvidence();
387         }
388
389         // Since rotateGuardianKey is called infrequently, blockNumber-based freshness is used for
                replay protection instead of a nonce.
390         bytes32 signedMessageHash =
391             keccak256(abi.encode("ROTATE_GUARDIAN_KEY", address(this), block.chainid, blockNumber,
                    pubKey));
392         bool isValid = SESSION_REGISTRY.verifySessionSignature(
393             proof.sessionId, proof.sessionKey, signedMessageHash, proof.signature
394         );
395         if (!isValid) {
396             revert InvalidSignature();
397         }
398
399         bytes32 ownerFingerprint = SESSION_REGISTRY.getSessionOwner(proof.sessionId);
400         require(ownerFingerprint == LibKey.computeKeyFingerprint(proof.ownerKey),
                InvalidECDSAPubKey());
401
402         CVMSession memory session = SESSION_REGISTRY.getSession(proof.sessionId);
403         require(_allowedWorkloads[session.workloadId], WorkloadNotAllowed());
404
405         // pubKey[1:] means we need to strip the first byte '0x' if we want to get the correct
                address
406         address computedAddress = address(uint160(uint256(keccak256(pubKey[1:]))));
407
408         _guardianEnclaves[guardian].enclaveAddress = computedAddress;
409         _guardianEnclaves[guardian].enclavePubKey = pubKey;
410
411         emit RotatedGuardianKey(guardian, computedAddress, pubKey);
412     }
```

Listing 2.11: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

```
263     function validateGuardiansEnclaveSignatures(bytes[] calldata enclaveSignatures, bytes32
            signedMessageHash)
264         public
265         view
266         returns (bool)
267     {
268         return _validateSignatures(getGuardiansEnclaveAddresses(), enclaveSignatures,
                signedMessageHash);
269     }
```

Listing 2.12: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

```
431   function getGuardiansEnclaveAddresses() public view returns (address[] memory) {
432       uint256 guardiansLength = _guardians.length();
433       address[] memory enclaveAddresses = new address[](guardiansLength);
434
435       for (uint256 i; i < guardiansLength; ++i) {
436           // If the guardian doesn't have an enclave address, we use `0xdead` address
437           // The reason for this is that we use .tryRecover in signature verification, and a
                  valid signature can be crafted to recover to address(0)
438           address enclaveAddress = _guardianEnclaves[_guardians.at(i)].enclaveAddress == address
                  (0)
439               ? address(0x000000000000000000000000000000000000dEaD)
440               : _guardianEnclaves[_guardians.at(i)].enclaveAddress;
441           enclaveAddresses[i] = enclaveAddress;
442       }
443
444       return enclaveAddresses;
445   }
```

Listing 2.13: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

```
526   function _validateSignatures(address[] memory signers, bytes[] calldata signatures, bytes32
          signedMessageHash)
527       internal
528       view
529       returns (bool)
530   {
531       uint256 validSignatures;
532
533       // We only count signature as valid if it's from the correct signer
534       for (uint256 i; i < signers.length; ++i) {
535           (address currentSigner, ECDSA.RecoverError recoverError,) =
536               ECDSA.tryRecover(signedMessageHash, signatures[i]);
537           if (recoverError == ECDSA.RecoverError.NoError) {
538               if (currentSigner == signers[i]) {
539                   ++validSignatures;
540               }
541           }
542       }
543
544       return validSignatures < _threshold ? false : true;
545   }
546}
```

Listing 2.14: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

Impact Guardian enclave threshold can be met with fewer unique enclave keys than intended.

Suggestion Reject duplicate enclave addresses.

2.1.6 Circumvention of static PCR policy

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The TPM attestation library permits a zero `PcrValue.value` when event log hashes are present. In this case, the library reconstructs the PCR from the event log and uses the reconstructed value to validate the quote digest. However, contract `TpmVerifier` returns the original caller-supplied PCR array without replacing the zero sentinel with the reconstructed PCR. Function `SessionRegistry._evaluateSinglePcr()` then evaluates a `STATIC` policy against this returned PCR. Consequently, a non-zero PCR derived from the event log can therefore satisfy a `STATIC` policy whose expected value is zero.

```
126       // Check PCR measurements (library emits events, reverts on failure)
127       (bool pcrSuccess,) = tpmAttestation.checkPcrMeasurements(quoteReport.tpm2bAttest,
              quoteReport.pcrValues);
128
129       if (!pcrSuccess) {
130           revert TpmQuotePcrCheckFailed(quoteReport.pcrValues.length);
131       }
132
133       return TpmQuoteVerificationResult({valid: true, pcrValues: quoteReport.pcrValues});
```

Listing 2.15: automata-tee-workload-measurement/src/bases/TpmVerifier.sol

```
1438   function _evaluateSinglePcr(PcrSpec memory spec, PcrValue memory measured) internal pure {
1439       if (spec.verifyType == PcrVerifyType.STATIC) {
1440           uint256 matchDataLength = spec.matchData.length;
1441           if (matchDataLength != 1) {
1442               revert InvalidStaticMatchDataLength(spec.pcrIndex, matchDataLength);
1443           }
1444           // STATIC: exact value match
1445           if (measured.value != spec.matchData[0]) {
1446               revert PCRStaticMismatch(spec.pcrIndex, measured.value, spec.matchData[0]);
1447           }
```

Listing 2.16: automata-tee-workload-measurement/src/SessionRegistry.sol

```
1033       TpmQuoteVerificationResult memory quoteResult =
1034           verifyTpmQuote(tpmQuoteReport, akPub, abi.encodePacked(expectedExtraData));
```

Listing 2.17: automata-tee-workload-measurement/src/SessionRegistry.sol

Impact An environment with a non-zero measured PCR may be accepted despite violating the configured `STATIC` policy.

Suggestion Return the normalized PCR values used for Quote verification, or require each submitted PCR value to equal the value recomputed from its event log.

2.1.7 Lack of freshness validation for DCAP ZK proofs

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The DCAP ZK journal binds a proof timestamp together with the verified attestation output. However, `IDcapAttestation.verifyAndAttestWithZKProof()` returns only the verified output, and `TeeVerifier._verifyIntelTdx()` does not receive or validate the proof timestamp. Consequently, contract `SessionRegistry` cannot reject future or stale ZK proofs before their output is used to create a Session. If a quote can be proven against a historical time at which now-expired or outdated collateral was valid, evidence that fails current collateral or TCB checks may still be admitted.

```
45  /// @notice Verifies a DCAP attestation using a pre-generated ZK proof
46  /// @param output The verification output/public inputs from the ZK circuit execution,
47  ///       containing the attestation claims to be verified
48  /// @param zkCoprocessor The type of ZK coprocessor that generated the proof,
49  ///       determining which verifier contract to use
50  /// @param proofBytes The serialized ZK proof bytes in the format expected by
51  ///       the corresponding verifier (RiscZero or Succinct)
52  /// @return success True if the ZK proof is valid and the attestation passes, false otherwise
53  /// @return verifiedOutput The verified attestation output after proof validation,
54  ///        containing the same claims as the input output if verification succeeds
55  function verifyAndAttestWithZKProof(
56      bytes calldata output,
57      ZkCoProcessorType zkCoprocessor,
58      bytes calldata proofBytes
59  ) external payable returns (bool success, bytes memory verifiedOutput);
```

Listing 2.18: automata-tee-workload-measurement/src/interfaces/external/IDcapAttestation.sol

```
502  function _verifyIntelTdx(TeeReport memory teeReport) private returns (TeeVerificationResult
         memory result) {
503      bool success;
504      bytes memory output;
505
506      if (teeReport.verificationBackendType == VerificationBackendType.Solidity) {
507          // Direct on-chain verification
508          (success, output) = dcapAttestation.verifyAndAttestOnChain(teeReport.data);
509      } else {
510          // ZK proof verification
511          ZkProof memory zkProof = abi.decode(teeReport.data, (ZkProof));
512
513          // Cast backend type to ZK coprocessor type (ordinals align after reordering)
514          IDcapAttestation.ZkCoProcessorType zkType =
515              IDcapAttestation.ZkCoProcessorType(uint8(teeReport.verificationBackendType));
516
517          (success, output) = dcapAttestation.verifyAndAttestWithZKProof(zkProof.output, zkType,
                 zkProof.proofBytes);
518      }
519
520      // Surface DCAP failure with the verifier's raw output so off-chain decoders
521      // can pick out the specific reason (was silently swallowed before).
522      if (!success) {
523          revert DcapVerificationFailed(output);
524      }
```

```
525
526         if (output.length <= DCAP_TCB_STATUS_OFFSET) {
527             revert TeeReportTooShort(output.length, DCAP_TCB_STATUS_OFFSET + 1);
528         }
529         uint8 tcbStatus = uint8(output[DCAP_TCB_STATUS_OFFSET]);
530         if (
531             tcbStatus > TCB_STATUS_RELAUNCH_ADVISED_CONFIGURATION_NEEDED
532                 || (tcbStatus > TCB_STATUS_OUT_OF_DATE_CONFIGURATION_NEEDED && tcbStatus <
                        TCB_STATUS_RELAUNCH_ADVISED)
533         ) {
534             revert DcapTcbStatusNotAccepted(tcbStatus);
535         }
536         uint256 tcbStatusBit = uint256(1) << tcbStatus;
537
538         bytes memory quoteBody = extractDcapQuoteBody(output);
539         bool debugEnabled = _validateTdxReport(quoteBody);
540         uint256 enabledTeeAttributes = debugEnabled ? TEE_ATTRIBUTE_INTEL_TDX_DEBUG_BIT : 0;
541
542         return TeeVerificationResult({
543             valid: true,
544             reportData: quoteBody,
545             teeType: TEEType.IntelTDX,
```

Listing 2.19: automata-tee-workload-measurement/src/TeeVerifier.sol

```
605         PolicyContext memory policyCtx = _lookupPolicy(workloadId, baseImageId, platformProfileId,
                measurementVariantId);
606         result.attestation = _verifyAttestation(
607             evidence,
608             ownerFingerprint,
609             policyCtx.platformProfile.attributes,
610             policyCtx.variant.attributes,
611             policyCtx.workloadSpec.requirements
612         );
613         result.sessionId =
614             _computeSessionId(result.attestation.tpmSignatureHash, teeVerifier.getTeeReportHash(
                    evidence.teeReport));
615         if (_sessions[result.sessionId].exists) revert SessionAlreadyExists();
```

Listing 2.20: automata-tee-workload-measurement/src/SessionRegistry.sol

Impact TDX evidence that fails current collateral or TCB checks may create a new active Session.

Suggestion Return the proof-bound timestamp to `TeeVerifier`, reject future timestamps, and enforce a maximum proof age consistent with the Solidity backend.

Feedback from the project The project clarifies that the underlying DCAP verifier validates collateral freshness for the Solidity backend. For the ZK backend, it exposes the proof-committed timestamp on-chain and adds explicit freshness checks.

2.1.8 Lack of check on debug requirements

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `GuardianModule`, function `setAllowedWorkload()` only checks that the `workloadId` is not zero before adding it to the local allowlist. However, the function does not verify the workload's debug requirement. In the contract `WorkloadRegistry`, function `_validateRequirements()` permits the debug requirement `[false, true]`. If an allowed base image profile or variant declares debug as `true`, function `_verifyBooleanAttribute()` of contract `AmdSnpSecurityPolicyRegistry` accepts a debug-enabled TEE report. Consequently, contract `GuardianModule` may authorize a debug-enabled session because it does not independently require debug mode to be disabled.

```
280     * @inheritdoc IGuardianModule
281     * @dev Restricted to the DAO
282     */
283    function setAllowedWorkload(bytes32 workloadId, bool allowed) external restricted {
284        require(workloadId != bytes32(0), WorkloadNotAllowed());
285        _allowedWorkloads[workloadId] = allowed;
286        emit WorkloadAllowanceChanged(workloadId, allowed);
287    }
```

Listing 2.21: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

```
350                 key == TEE_ATTRIBUTE_INTEL_TDX_DEBUG || key == TEE_ATTRIBUTE_AMD_SEV_SNP_DEBUG
351                     || key == TEE_ATTRIBUTE_AMD_SEV_SNP_MIGRATE_MA
352             ) {
353                 bytes32[] calldata allowedValues = requirements[i].allowedValues;
354                 uint256 allowedLen = allowedValues.length;
355                 if (allowedLen != 1 && allowedLen != 2) {
356                     revert InvalidTeeAttributeRequirementLength(key, allowedLen);
357                 }
358                 if (allowedValues[0] != bytes32(0)) {
359                     revert InvalidTeeAttributeRequirementValue(key, allowedValues[0]);
360                 }
361                 if (allowedLen == 2 && allowedValues[1] != TEE_ATTRIBUTE_TRUE) {
362                     revert InvalidTeeAttributeRequirementValue(key, allowedValues[1]);
363                 }
```

Listing 2.22: automata-tee-workload-measurement/src/WorkloadRegistry.sol

Impact Debug-enabled sessions may be authorized as guardians, weakening TEE security.

Suggestion Only allow workloads that require debug to be `false`, or reject debug-enabled sessions in the guardian authorization path.

2.1.9 Front-running of base image and workload registration

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `BaseImageRegistry`, function `registerBaseImage()` derives the identifier `baseImageId` only from the corresponding name and version. When registration is public, an attacker can front-run the intended base image registration using the same name and version. The same issue exists for function `registerWorkload()` in contract `WorkloadRegistry`.

Additionally, function `registerWorkload()` stores each supplied `baseImageId` without verification, potentially allowing an attacker-controlled base image policy.

```
156         // Populate base image set
157         for (uint256 i = 0; i < spec.baseImageIds.length; i++) {
158             _baseImageSet[workloadId][spec.baseImageIds[i]] = true;
159         }
```

Listing 2.23: automata-tee-workload-measurement/src/WorkloadRegistry.sol

```
129         workloadId = keccak256(abi.encode(WORKLOAD_DOMAIN, spec.name, spec.version));
```

Listing 2.24: automata-tee-workload-measurement/src/WorkloadRegistry.sol

```
167         baseImageId = keccak256(abi.encode(BASEIMAGE_DOMAIN, spec.name, spec.version));
```

Listing 2.25: automata-tee-workload-measurement/src/BaseImageRegistry.sol

Impact The base image and workload can be registered publicly when contracts are unpaused, allowing an unexpected base image to be bound to the workload.

Suggestion Verify that each `baseImageId` exists before adding it to a `workload`, and compute the fingerprint of the publisher of the base image or workload.

2.2 Recommendation

2.2.1 Align the documentation with the current implementation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current documents contain several inconsistencies with the implementation.

1. Interface `IGuardianModule` describes `setAllowedWorkload()` as setting a Workload allowance. However, the implementation controls eligibility for later `rotateGuardianKey()` calls and does not revoke the cached `enclaveAddress`.
2. Interface `IPufferDepositorV2` describes `rescueAnything()` as an admin operation frozen with the vault target. However, the implementation uses `AccessManager` authorization on the depositor target, and the access setup does not assign the `rescueAnything()` selector.
3. Contract `WorkloadRegistry` describes an empty `WHITELIST` as a valid deny-all configuration. However, `registerWorkload()` reverts with `EmptyBaseImageWhitelist`.
4. In the contract `ProofSubmitter`, the contract-level `NatSpec` title is expected to identify the `EigenPod` proof submission wrapper. However, the title still says `PUFFER Token`, although the contract is an `EigenPod` proof submission wrapper for the `operations paymaster`. This does not affect runtime behavior but makes documentation and indexing less clear.

It is recommended to align the affected interfaces, documentation, and AccessManager deployment configuration with the implemented authority, lifecycle, and input rules.

```
130   /**
131    * @notice Set allowed workloads for guardians
132    * @dev Workload policies are managed in WorkloadRegistry
133    * @param workloadId Id of the workload
134    * @param allowed Bool indicating whether the workload is allowed or not
135    */
136   function setAllowedWorkload(bytes32 workloadId, bool allowed) external;
```

Listing 2.26: puffer-contracts-updates-private/mainnet-contracts/src/interface/IGuardianModule.sol

```
82    function _getPublicSelectorsForDepositor(address pufferDepositorProxy) internal pure returns (
          bytes memory) {
83        // PufferDepositor public selectors
84        bytes4[] memory publicSelectorsDepositor = new bytes4[](2);
85        publicSelectorsDepositor[0] = PufferDepositorV2.depositStETH.selector;
86        publicSelectorsDepositor[1] = PufferDepositorV2.depositWstETH.selector;
87
88        return abi.encodeWithSelector(
89            AccessManager.setTargetFunctionRole.selector, pufferDepositorProxy,
                  publicSelectorsDepositor, PUBLIC_ROLE
90        );
```

Listing 2.27: puffer-contracts-updates-private/mainnet-contracts/script/GenerateAccessManagerCallData.sol

```
120       // An empty whitelist denies every base image, so isBaseImageAllowed always returns false
121       // and no session can ever reference this workload. Registration is one-shot: the spec is
122       // immutable and the name/version pair remains claimed after deactivation, so the mistake
123       // is unrecoverable under that identifier. Reject it rather than record it.
124       if (spec.baseImageMode == AccessMode.WHITELIST && spec.baseImageIds.length == 0) {
125           revert EmptyBaseImageWhitelist();
```

Listing 2.28: automata-tee-workload-measurement/src/WorkloadRegistry.sol

```
 8 * @title PUFFER Token
 9 * @author Puffer Finance
10 * @dev This contract limits the functions from IEigenPod that can be called by the operations
       paymaster. It is intended to be used as a "proof submitter" for EigenPods,
11 *      allowing the operations paymaster to submit proofs on behalf of the pod owner.
12 *      This prevents the proof submitter to call other more critical functions like `
       requestWithdrawal()`.
13 * @custom:security-contact security@puffer.fi
14 */
15contract ProofSubmitter is AccessManaged {
```

Listing 2.29: puffer-contracts-updates-private/mainnet-contracts/src/ProofSubmitter.sol

Suggestion Update the affected interfaces and documentation, and configure deployment permissions consistently with the implementation.

2.2.2 Add a maximum Workload Session TTL

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `registerWorkload()` accepts any `sessionTtl` and stores the Workload specification as immutable registration data. The function `SessionRegistry._prepareFullSession()` adds this configured TTL to `uint64(block.timestamp)` when preparing a full Session. A TTL greater than the remaining `uint64` timestamp range causes Session registration, renewal, and recovery to revert, while the Workload name and version remain claimed and the specification cannot be corrected. It is recommended to enforce a product-defined maximum TTL during Workload registration.

```
128       // Compute workload ID
129       workloadId = keccak256(abi.encode(WORKLOAD_DOMAIN, spec.name, spec.version));
130
131       // Check for duplicate
132       if (_workloads[workloadId].exists) {
133           revert WorkloadAlreadyExists(workloadId);
134       }
135
136       // Compute owner fingerprint after duplicate check
137       bytes32 ownerFingerprint = LibKey.computeKeyFingerprint(ownerIdentity);
138
139       // Check whitelist if paused
140       _checkRegistrationAllowed(ownerFingerprint);
141
142       // Build signed message (operation-specific domain, no msg.sender, raw params)
143       bytes32 message = sha256(abi.encode(WORKLOAD_REGISTER_MSG, block.chainid, address(this),
              opExpiresAt, spec));
144
145       // Verify signature
146       if (!signatureVerifier.verify(ownerIdentity, message, ownerSignature)) {
147           revert InvalidSignature(message, ownerFingerprint);
148       }
149
150       // Store workload
151       _workloads[workloadId].exists = true;
152       _workloads[workloadId].isRevoked = false;
153       _workloads[workloadId].owner = ownerFingerprint;
154       _workloads[workloadId].workloadSpec = spec;
```

Listing 2.30: automata-tee-workload-measurement/src/WorkloadRegistry.sol

```
632       result.sessionExpiresAt = policyCtx.workloadSpec.sessionTtl == 0
633           ? uint64(block.timestamp) + DEFAULT_CVM_TTL
634           : uint64(block.timestamp) + policyCtx.workloadSpec.sessionTtl;
```

Listing 2.31: automata-tee-workload-measurement/src/SessionRegistry.sol

Suggestion Reject `sessionTtl` values above the supported maximum with a dedicated error and document the allowed range.

2.2.3 Prevent key reuse during session rotation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `SessionRegistry`, the function `rotateKey()` allows a successor session to reuse the predecessor's TPM signing key and session key. The function `_createSession()` revokes the predecessor and clears its session-key fingerprint before updating the reverse mapping, so the same keys can be registered again. It is recommended to require successor keys to differ from the predecessor's keys and to retain historical fingerprints if key reuse must be prohibited.

```
807   function _finalizeRotation(
808       bytes32 oldSessionId,
809       bytes32 newSessionId,
810       bytes32 newTpmSigningKeyFingerprint,
811       bytes32 sessionKeyFingerprint,
812       PublicIdentity memory newTpmSigningKey,
813       RotationContext memory ctx,
814       uint64 opExpiresAt,
815       PublicIdentity calldata ownerIdentity,
816       bytes calldata ownerSignature,
817       PublicIdentity calldata akPub,
818       PublicIdentity calldata sessionKey
819   ) private {
820       _requireNotExpired(opExpiresAt);
821
822       bytes32 message = sha256(
823           abi.encode(SESSION_ROTATE_KEY_MSG, block.chainid, address(this), opExpiresAt,
                  oldSessionId, newSessionId)
824       );
825       _requireSignature(ownerIdentity, message, ownerSignature);
826
827       _sessions[oldSessionId].isRevoked = true;
828       _clearSessionFingerprint(oldSessionId, _sessions[oldSessionId].session.
              sessionKeyFingerprint);
829       emit SessionRevoked(oldSessionId, ctx.ownerFingerprint);
830
831       _createSession(
832           SessionParams({
833               sessionId: newSessionId,
834               ownerFingerprint: ctx.ownerFingerprint,
835               akPubKeyFingerprint: _sessions[oldSessionId].session.akPubKeyFingerprint,
836               tpmSigningKeyFingerprint: newTpmSigningKeyFingerprint,
837               sessionKeyFingerprint: sessionKeyFingerprint,
838               baseImageId: ctx.baseImageId,
839               workloadId: ctx.workloadId,
840               platformProfileId: ctx.platformProfileId,
841               measurementVariantId: ctx.measurementVariantId,
842               sessionExpiresAt: ctx.sessionExpiresAt
843           })
844       );
```

Listing 2.32: automata-tee-workload-measurement/src/SessionRegistry.sol

Suggestion Reject rotations that reuse the predecessor's TPM signing key or session key. Preserve tombstones for retired fingerprints if the protocol requires one-time key usage.

2.2.4 Bind successor keys to owner authorization

Status Confirmed

Introduced by `Version 1`

Description In contract `SessionRegistry`, function `_finalizeRotation()` verifies the owner's signature over only `oldSessionId` and `newSessionId`. Function `_verifyRotationAuthorization()` separately checks the successor TPM signing-key and session-key fingerprints using the old TPM signing key. However, function `_computeSessionId()` derives `newSessionId` only from the TPM quote signature hash and `teeReportBytesHash`. A party controlling the old TPM credentials can therefore keep the signed quote and TEE report hash unchanged while replacing the successor keys without invalidating the owner's signature. It is recommended to bind both successor key fingerprints to the owner-signed rotation message.

```
822       bytes32 message = sha256(
823           abi.encode(SESSION_ROTATE_KEY_MSG, block.chainid, address(this), opExpiresAt,
                  oldSessionId, newSessionId)
824       );
825       _requireSignature(ownerIdentity, message, ownerSignature);
```

Listing 2.33: automata-tee-workload-measurement/src/SessionRegistry.sol

Suggestion Include the successor TPM signing-key fingerprint and session-key fingerprint in the owner-signed rotation message.

2.2.5 Add RSA exponent validation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `SignatureVerifier`, function `verify()` dispatches `ALGO_ID_RS256` identities to function `_verifyRsa()`. Function `_verifyRsa()` passes the DER-encoded exponent `e` directly to function `RSA.pkcs1Sha256()` without validating its value. The OpenZeppelin implementation checks only the modulus and signature lengths and whether `s < n` before performing modular exponentiation. When `e = 1`, modular exponentiation becomes an identity operation, allowing an attacker to construct a valid PKCS#1 encoded signature without a private key. It is recommended to validate the RSA public exponent before signature verification.

```
57    function _verifyRsa(bytes calldata key, bytes32 message, bytes calldata signature)
58        internal
59        view
60        returns (bool valid)
61    {
```

```
62      // Copy key from calldata to memory (Asn1Decode requires memory)
63      bytes memory keyMem = key;
64
65      // Parse DER structure: SEQUENCE { n, e }
66      uint256 rootPtr = keyMem.root();
67      uint256 nPtr = keyMem.firstChildOf(rootPtr);
68      uint256 ePtr = keyMem.nextSiblingOf(nPtr);
69
70      // Extract modulus (n) and exponent (e) as bytes
71      bytes memory n = keyMem.uintBytesAt(nPtr);
72      bytes memory e = keyMem.uintBytesAt(ePtr);
73
74      // Copy signature from calldata to memory (OZ RSA requires memory)
75      bytes memory sigMem = signature;
76
77      // Verify using OpenZeppelin RSA library
78      return RSA.pkcs1Sha256(message, sigMem, e, n);
79  }
```

Listing 2.34: automata-tee-workload-measurement/src/SignatureVerifier.sol

```
34  function verify(PublicIdentity calldata identity, bytes32 message, bytes calldata signature)
35      external
36      view
37      returns (bool valid)
38  {
39      uint8 typeId = identity.typeId;
40
41      if (typeId == ALGO_ID_RS256) {
42          return _verifyRsa(identity.key, message, signature);
43      } else if (typeId == ALGO_ID_ES256) {
44          return _verifyP256(identity.key, message, signature);
45      } else if (typeId == ALGO_ID_ES256K) {
46          return _verifySecp256k1(identity.key, message, signature);
47      } else {
48          revert UnsupportedAlgorithm(typeId);
49      }
50  }
```

Listing 2.35: automata-tee-workload-measurement/src/SignatureVerifier.sol

Suggestion Require function `_verifyRsa()` to accept only the standard exponent `65537`. Apply the same validation when registering or allowlisting RSA identities.

2.2.6 Flush removed Guardian reward tokens

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `GuardianModule`, function `removeAllowedToken()` removes a token without distributing the existing balance to guardians via function `splitGuardianFunds()`. A removed token balance can remain in the contract until governance re-adds the token or uses another recovery path.

```
144         // ERC20 tokens
145         uint256 allowedTokensLength = _allowedTokens.length();
146         for (uint256 i = 0; i < allowedTokensLength; ++i) {
147             address token = _allowedTokens.at(i);
148             uint256 tokenBalance = IERC20(token).balanceOf(address(this));
149             uint256 tokenAmountPerGuardian = tokenBalance / numGuardians;
150             for (uint256 j = 0; j < numGuardians; ++j) {
151                 // slither-disable-start reentrancy-unlimited-gas
152                 // slither-disable-next-line calls-loop
153                 IERC20(token).safeTransfer(guardians[j], tokenAmountPerGuardian);
154                 // slither-disable-end reentrancy-unlimited-gas
155             }
156         }
157     }
158
159     /**
160      * @inheritdoc IGuardianModule
161      */
162     function validateSkipProvisioning(bytes32 moduleName, uint256 skippedIndex, bytes[] calldata
          eoaSignatures)
163         external
164         view
165     {
166         bytes32 signedMessageHash =
167             LibGuardianMessages._getSkipProvisioningMessage(address(this), moduleName, skippedIndex
                  );
168
169         // Check the signatures
170         bool validSignatures =
171             validateGuardiansEOASignatures({ eoaSignatures: eoaSignatures, signedMessageHash:
                  signedMessageHash });
172
173         if (!validSignatures) {
174             revert Unauthorized();
175         }
```

Listing 2.36: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

```
330     function removeAllowedToken(address token) external restricted {
331         if (token == address(0)) {
332             revert InvalidAddress();
333         }
334         bool success = _allowedTokens.remove(token);
335         if (!success) {
336             revert InvalidAddress();
337         }
338
339         emit AllowedTokenRemoved(token);
340     }
```

Listing 2.37: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

Suggestion Flush the token balance before removal.

2.2.7 Add zero-address checks for immutable dependencies

Status Confirmed

Introduced by Version 1

Description In the contracts `SessionRegistry`, `WorkloadRegistry`, `BaseImageRegistry`, and `TeeVerifier`, the constructors are expected to set immutable dependencies used for `session` registration, `attestation` verification, `policy` lookup, and signature verification. Several immutable dependencies are accepted at deployment and then used without a zero-address check. A zero or incorrect dependency address would make the affected registry path unusable.

```
252   constructor(
253       ITeeVerifier teeVerifier_,
254       ITpmAttestation tpmAttestation_,
255       ISignatureVerifier signatureVerifier_,
256       IAkCollateralVerifier akCollateralVerifier_,
257       IBaseImageRegistry baseImageRegistry_,
258       IWorkloadRegistry workloadRegistry_,
259       IAmdSnpSecurityPolicyRegistry amdSnpSecurityPolicyRegistry_
260   ) TpmBase(tpmAttestation_) {
261       teeVerifier = teeVerifier_;
262       signatureVerifier = signatureVerifier_;
263       akCollateralVerifier = akCollateralVerifier_;
264       baseImageRegistry = baseImageRegistry_;
265       workloadRegistry = workloadRegistry_;
266       amdSnpSecurityPolicyRegistry = amdSnpSecurityPolicyRegistry_;
267       _disableInitializers();
268   }
```

Listing 2.38: automata-tee-workload-measurement/src/SessionRegistry.sol

```
 88   constructor(ISignatureVerifier _signatureVerifier) {
 89       signatureVerifier = _signatureVerifier;
 90       _disableInitializers();
 91   }
```

Listing 2.39: automata-tee-workload-measurement/src/WorkloadRegistry.sol

```
116   constructor(ISignatureVerifier _signatureVerifier) {
117       signatureVerifier = _signatureVerifier;
118       _disableInitializers();
119   }
```

Listing 2.40: automata-tee-workload-measurement/src/BaseImageRegistry.sol

```
218   constructor(IDcapAttestation _dcapAttestation, ISnpAttestation _snpAttestation) {
219       dcapAttestation = _dcapAttestation;
220       snpAttestation = _snpAttestation;
221   }
```

Listing 2.41: automata-tee-workload-measurement/src/TeeVerifier.sol

Suggestion Validate immutable dependency addresses in constructors.

2.2.8 Add explicit Guardian action domains

Status Confirmed

Introduced by Version 1

Description In the library `LibGuardianMessages`, the functions `_getBeaconDepositMessageToBeSigned()`, `_getSkipProvisioningMessage()`, `_getHandleBatchWithdrawalMessage()`, and `_getSetNumberOfValidatorsMessage()` are expected to define replay-resistant `Guardian` action messages. The current messages do not include an explicit action-specific domain for deposit approval, skip provisioning, batch withdrawal, or validator count updates. There is no practical collision under the current naming convention. However, a changed naming convention in future could make the parameter domains overlap and allow guardian approvals intended for one operation to be replayed as approvals for the other.

```
26  function _getBeaconDepositMessageToBeSigned(
27      address verifyingContract,
28      uint256 pufferModuleIndex,
29      bytes memory pubKey,
30      bytes memory signature,
31      bytes memory withdrawalCredentials,
32      bytes32 depositDataRoot
33  ) internal view returns (bytes32) {
34      return keccak256(
35          abi.encode(
36              verifyingContract,
37              block.chainid,
38              pufferModuleIndex,
39              pubKey,
40              withdrawalCredentials,
41              signature,
42              depositDataRoot
43          )
44      ).toEthSignedMessageHash();
45  }
46
47  /**
48   * @notice Returns the message to be signed for skip provisioning
49   * @param verifyingContract is the address of the contract that will verify the signature
50   * @param moduleName is the name of the module
51   * @param index is the index of the skipped validator
52   * @return the message to be signed
53   */
54  function _getSkipProvisioningMessage(address verifyingContract, bytes32 moduleName, uint256
        index)
55      internal
56      view
57      returns (bytes32)
58  {
59      // All guardians use the same nonce
60      return keccak256(abi.encode(verifyingContract, block.chainid, moduleName, index)).
            toEthSignedMessageHash();
61  }
```

```
62
63    /**
64     * @notice Returns the message to be signed for handling the batch withdrawal
65     * @param verifyingContract is the address of the contract that will verify the signature
66     * @param validatorInfos is an array of validator information
67     * @return the message to be signed
68     */
69    function _getHandleBatchWithdrawalMessage(address verifyingContract, StoppedValidatorInfo[]
          memory validatorInfos)
70        internal
71        view
72        returns (bytes32)
73    {
74        return keccak256(abi.encode(verifyingContract, block.chainid, validatorInfos)).
              toEthSignedMessageHash();
75    }
76
77    /**
78     * @notice Returns the message to be signed updating the number of validators
79     * @param verifyingContract is the address of the contract that will verify the signature
80     * @param numberOfValidators is the new number of validators
81     * @param epochNumber is the epoch number
82     * @return the message to be signed
83     */
84    function _getSetNumberOfValidatorsMessage(
85        address verifyingContract,
86        uint256 numberOfValidators,
87        uint256 epochNumber
88    ) internal view returns (bytes32) {
89        return keccak256(abi.encode(verifyingContract, block.chainid, numberOfValidators,
              epochNumber))
90            .toEthSignedMessageHash();
91    }
92}
```

Listing 2.42: puffer-contracts-updates-private/mainnet-contracts/src/LibGuardianMessages.sol

Suggestion Add a fixed domain constant as the first encoded field for each Guardian action message.

2.3 Note

2.3.1 Verify the workload before allowlist operations

Introduced by Version 1

Description When the workload registry opens registration to the public, any party can register a workload. Contract `GuardianModule` determines whether a session originates from an allowed workload solely by checking the `workloadId`. It does not revalidate the workload owner or its TEE policy. The design is secure only if the DAO adds a `workloadId` to the allowlist after

the corresponding workload has been registered and its owner, debug policy, and other security configurations have been reviewed. The DAO should not pre-approve a `workloadId` for a workload that has not yet been registered or reviewed. The review should also account for variant attribute overrides. In the contract `BaseImageRegistry`, the function `addPlatformVariants()` allows new measurement variants to share keys with existing profile attributes. Meanwhile, the function `_effectiveAttribute()` determines the effective TEE attribute by checking the variant attributes, the profile attributes, and the default value in that order. A variant can therefore widen policy within what the base image owner publishes and what each workload explicitly permits. The DAO should account for this property when reviewing workloads.

2.3.2 Atomic upgrade for `PufferDepositorV2` initialization

Introduced by `Version 1`

Description In contract `PufferDepositorV2`, the implementation constructor invokes function `_disableInitializers()`. The proxy exposes the function `initialize()` as a public entrance without caller authorization. Function `initialize()` transfers `0.201 pufETH` from the `PufferDepositorV2` contract to the hardcoded `_PUFFER` address. The deployment is safe only when the proxy upgrade and function `initialize()` call execute atomically through function `upgradeToAndCall()` with `abi.encodeCall(PufferDepositorV2.initialize, ())`. Otherwise, any account can consume version 2 initialization before the intended migration.

```
40        PUFFER_VAULT = pufferVault;
41        _ST_ETH = stETH;
42        _disableInitializers();
43    }
44
45    /**
46     * @notice Returns the pufETH sent to this contract by mistake
47     */
48    // nosemgrep tin-unprotected-initialize
49    function initialize() public reinitializer(2) {
50        // https://etherscan.io/token/0xd9a442856c234a39a81a089c06451ebaa4306a72?a=0
              x4aa799c5dfc01ee7d790e3bf1a7c2257ce1dceff
51        // slither-disable-next-line unchecked-transfer
52        PUFFER_VAULT.transfer(_PUFFER, 0.201 ether);
53    }
```

Listing 2.43: puffer-contracts-updates-private/mainnet-contracts/src/PufferDepositorV2.sol

Feedback from the project The project will invoke the initialization in the same transaction as the upgrade.

2.3.3 Session lifecycle does not preserve workload continuity

Introduced by `Version 1`

Description In the contract `SessionRegistry`, the function `renewSession()` and the function `recoverSession()` are expected to create successor sessions from a predecessor. The successor evidence is evaluated with caller-supplied `workload`, `base image`, `platform profile`,

and `measurement variant` identifiers. A successor session should therefore be authorized by its current session fields rather than by assuming continuity from the predecessor.

```
359   function rotateKey(
360       bytes32 oldSessionId,
361       bytes32 teeReportBytesHash,
362       SessionKeyRotationEvidence calldata rotationEvidence,
363       uint64 opExpiresAt,
364       PublicIdentity calldata ownerIdentity,
365       bytes calldata ownerSignature
366   ) external returns (bytes32 newSessionId) {
367       return _rotateKey(
368           oldSessionId, teeReportBytesHash, rotationEvidence, opExpiresAt, ownerIdentity,
                ownerSignature
369       );
370   }
371
372   /// @inheritdoc ISessionRegistry
373   function renewSession(
374       bytes32 oldSessionId,
375       AttestationEvidence calldata newEvidence,
376       bytes32 workloadId,
377       bytes32 baseImageId,
378       bytes32 platformProfileId,
379       bytes32 measurementVariantId,
380       SessionRenewalAuthorization calldata renewalAuthorization,
381       uint64 opExpiresAt,
382       PublicIdentity calldata ownerIdentity,
383       bytes calldata ownerSignature
384   ) external returns (bytes32 newSessionId) {
385       _requireNotExpired(opExpiresAt);
386       CVMSessionStorage storage predecessor = _sessions[oldSessionId];
387       if (!predecessor.exists) revert SessionNotFound();
388       if (!_isSessionActive(predecessor)) revert SessionNotActive();
389
390       bytes32 ownerFingerprint = _requireFingerprint(ownerIdentity, predecessor.owner);
391       _requireFingerprint(renewalAuthorization.oldTpmSigningKey, predecessor.session.
              tpmSigningKeyFingerprint);
392
393       FullSessionResult memory result = _prepareFullSession(
394           newEvidence, workloadId, baseImageId, platformProfileId, measurementVariantId,
                ownerFingerprint
395       );
396       newSessionId = result.sessionId;
397
398       bytes32 evidenceHash = keccak256(abi.encode(newEvidence));
399       bytes32 renewalMessage = keccak256(
400           abi.encode(SESSION_RENEW_DOMAIN, block.chainid, address(this), oldSessionId,
                newSessionId, evidenceHash)
401       );
402       _requireSignature(renewalAuthorization.oldTpmSigningKey, renewalMessage,
              renewalAuthorization.signature);
403       _requireLifecycleOwnerSignature(
```

```
404             SESSION_RENEW_MSG,
405             oldSessionId,
406             newSessionId,
407             workloadId,
408             baseImageId,
409             platformProfileId,
410             measurementVariantId,
411             result.sessionKeyFingerprint,
412             opExpiresAt,
413             ownerIdentity,
414             ownerSignature
415         );
416
417         predecessor.isRevoked = true;
418         _clearSessionFingerprint(oldSessionId, predecessor.session.sessionKeyFingerprint);
419         emit SessionRevoked(oldSessionId, ownerFingerprint);
420         _finalizeFullSession(
421             result, workloadId, baseImageId, platformProfileId, measurementVariantId, newEvidence.
                    sessionKey
422         );
423         emit SessionRenewed(oldSessionId, newSessionId, ownerFingerprint);
424     }
425
426     /// @inheritdoc ISessionRegistry
427     function recoverSession(
428         bytes32 oldSessionId,
429         AttestationEvidence calldata newEvidence,
430         bytes32 workloadId,
431         bytes32 baseImageId,
432         bytes32 platformProfileId,
433         bytes32 measurementVariantId,
434         uint64 opExpiresAt,
435         PublicIdentity calldata ownerIdentity,
436         bytes calldata ownerSignature
437     ) external returns (bytes32 newSessionId) {
438         _requireNotExpired(opExpiresAt);
439         CVMSessionStorage storage predecessor = _sessions[oldSessionId];
440         if (!predecessor.exists) revert SessionNotFound();
441         // A recovery consumes its predecessor by revoking it, so an already-revoked predecessor
442         // cannot authorize a second one. This makes recovery single-use per predecessor without
443         // a dedicated flag. It costs the caller nothing: recovery is registerSession plus
444         // revokeSession, so an owner whose predecessor is already revoked has nothing left to
445         // revoke and simply calls registerSession. Checked before _prepareFullSession so a
446         // repeat attempt fails cheaply.
447         if (predecessor.isRevoked) revert SessionAlreadyRevoked();
448         bytes32 ownerFingerprint = _requireFingerprint(ownerIdentity, predecessor.owner);
449
450         FullSessionResult memory result = _prepareFullSession(
451             newEvidence, workloadId, baseImageId, platformProfileId, measurementVariantId,
                    ownerFingerprint
452         );
453         newSessionId = result.sessionId;
454         _requireLifecycleOwnerSignature(
```

```
455             SESSION_RECOVER_MSG,
456             oldSessionId,
457             newSessionId,
458             workloadId,
459             baseImageId,
460             platformProfileId,
461             measurementVariantId,
462             result.sessionKeyFingerprint,
463             opExpiresAt,
464             ownerIdentity,
465             ownerSignature
466         );
467
468         predecessor.isRevoked = true;
469         _clearSessionFingerprint(oldSessionId, predecessor.session.sessionKeyFingerprint);
470         emit SessionRevoked(oldSessionId, ownerFingerprint);
471         _finalizeFullSession(
472             result, workloadId, baseImageId, platformProfileId, measurementVariantId, newEvidence.
                    sessionKey
473         );
474         emit SessionRecovered(oldSessionId, newSessionId, ownerFingerprint);
475     }
```

Listing 2.44: automata-tee-workload-measurement/src/SessionRegistry.sol

2.3.4 DCAP verifier integration assumes a zero-fee backend

Introduced by `Version 1`

Description In the contract `TeeVerifier`, the function `_verifyIntelTdx()` is expected to invoke the configured `DCAP` verifier for `Intel TDX` reports. The implementation invokes functions `verifyAndAttestOnChain()` and `verifyAndAttestWithZKProof()` without forwarding `msg.value`. Deployments therefore assume these verifier functions can be used without caller-funded fees, and a paid verifier backend would require a contract change or a fee-handling wrapper.

```
502     function _verifyIntelTdx(TeeReport memory teeReport) private returns (TeeVerificationResult
            memory result) {
503         bool success;
504         bytes memory output;
505
506         if (teeReport.verificationBackendType == VerificationBackendType.Solidity) {
507             // Direct on-chain verification
508             (success, output) = dcapAttestation.verifyAndAttestOnChain(teeReport.data);
```

Listing 2.45: automata-tee-workload-measurement/src/TeeVerifier.sol

```
19      /**
20       * @notice full on-chain verification for an attestation
21       * @dev must further specify the structure of inputs/outputs, to be serialized and passed to
               this method
22       * @param input - serialized raw input as defined by the project
```

```
23    * @return success - whether the quote has been successfully verified or not
24    * @return output - the output upon completion of verification. The output data may require
          post-processing by the
25    * consumer.
26    * For verification failures, the output is simply a UTF-8 encoded string, describing the
          reason for failure.
27    * @dev can directly type cast the failed output as a string
28    */
29   function verifyAndAttestOnChain(bytes calldata input) external payable returns (bool success,
         bytes memory output);
```

Listing 2.46: automata-tee-workload-measurement/src/interfaces/external/IDcapAttestation.sol

Feedback from the project The project confirms that the DCAP verifier backend will remain zero-fee in future deployments, so not forwarding `msg.value` is acceptable.

2.3.5 Workload base-image lists are identifier policies

Introduced by `Version 1`

Description In the contract `WorkloadRegistry`, the function `registerWorkload()` is expected to store workload-level `baseImageIds` policy. The function stores `baseImageIds` without checking whether the referenced base images already exist in the contract `BaseImageRegistry`. This supports pre-registration against deterministic IDs, but an incorrect ID can make later session policy lookup reject the intended base image until a matching base image exists.

```
106   function registerWorkload(
107       WorkloadSpec calldata spec,
108       uint64 opExpiresAt,
109       PublicIdentity calldata ownerIdentity,
110       bytes calldata ownerSignature
111   ) external returns (bytes32 workloadId) {
112       // Check signature expiration
113       if (block.timestamp > opExpiresAt) {
114           revert SignatureExpired(opExpiresAt, uint64(block.timestamp));
115       }
116
117       _validatePcrSpecsSorted(spec.pcrs);
118       _validateRequirements(spec.requirements);
119
120       // An empty whitelist denies every base image, so isBaseImageAllowed always returns false
121       // and no session can ever reference this workload. Registration is one-shot: the spec is
122       // immutable and the name/version pair remains claimed after deactivation, so the mistake
123       // is unrecoverable under that identifier. Reject it rather than record it.
124       if (spec.baseImageMode == AccessMode.WHITELIST && spec.baseImageIds.length == 0) {
125           revert EmptyBaseImageWhitelist();
126       }
127
128       // Compute workload ID
129       workloadId = keccak256(abi.encode(WORKLOAD_DOMAIN, spec.name, spec.version));
130
131       // Check for duplicate
```

```
132         if (_workloads[workloadId].exists) {
133             revert WorkloadAlreadyExists(workloadId);
134         }
135
136         // Compute owner fingerprint after duplicate check
137         bytes32 ownerFingerprint = LibKey.computeKeyFingerprint(ownerIdentity);
138
139         // Check whitelist if paused
140         _checkRegistrationAllowed(ownerFingerprint);
141
142         // Build signed message (operation-specific domain, no msg.sender, raw params)
143         bytes32 message = sha256(abi.encode(WORKLOAD_REGISTER_MSG, block.chainid, address(this),
                opExpiresAt, spec));
144
145         // Verify signature
146         if (!signatureVerifier.verify(ownerIdentity, message, ownerSignature)) {
147             revert InvalidSignature(message, ownerFingerprint);
148         }
149
150         // Store workload
151         _workloads[workloadId].exists = true;
152         _workloads[workloadId].isRevoked = false;
153         _workloads[workloadId].owner = ownerFingerprint;
154         _workloads[workloadId].workloadSpec = spec;
155
156         // Populate base image set
157         for (uint256 i = 0; i < spec.baseImageIds.length; i++) {
158             _baseImageSet[workloadId][spec.baseImageIds[i]] = true;
159         }
160
161         emit WorkloadRegistered(workloadId, ownerFingerprint, spec.name, spec.version);
162     }
```

Listing 2.47: automata-tee-workload-measurement/src/WorkloadRegistry.sol

Feedback from the project The project will derive `baseImageIds` in a workload spec using off-chain tooling, so an incorrect ID will not be registered in practice.

2.3.6 Guardian enclave key is reused for encryption and signing

Introduced by `Version 1`

Description In the contract `GuardianModule`, the function `rotateGuardianKey()` is expected to establish the `Guardian enclave` key used by Puffer's custody flow. The same stored `secp256k1` public key is used as the enclave identity for Puffer message signing and as the recipient key for encrypted BLS share custody. This combines encryption and signing under one enclave key and should be treated as an explicit key-management assumption for the Guardian workload.

```
413
414     /**
415      * @inheritdoc IGuardianModule
416      */
417     function getEjectionThreshold() external view returns (uint256) {
```

```
418         return _ejectionThreshold;
419     }
420
421     /**
422      * @inheritdoc IGuardianModule
423      */
424     function getGuardiansEnclaveAddress(address guardian) external view returns (address) {
425         return _guardianEnclaves[guardian].enclaveAddress;
426     }
427
428     /**
429      * @inheritdoc IGuardianModule
430      */
431     function getGuardiansEnclaveAddresses() public view returns (address[] memory) {
432         uint256 guardiansLength = _guardians.length();
433         address[] memory enclaveAddresses = new address[](guardiansLength);
```

Listing 2.48: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

2.3.7 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., `ROLE_ID_OPERATIONS_PAYMASTER`, `ROLE_ID_OPERATIONS_MULTISIG`, and `ROLE_ID_DAO`) can conduct sensitive operations, which introduces potential centralization risks. For example, these roles can control validator registration and provisioning, skipped provisioning, batch withdrawal handling, EigenLayer withdrawal queueing, proof submitter configuration, and `EigenPod` proof or checkpoint submission through contract `ProofSubmitter`. The validator BLS share security model also relies on trusted NoOp key/share generation and Guardian enclave custody verification before Guardians sign provisioning approval. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project accepts this centralization trade-off and mitigates it through structural controls rather than reliance on any single key. Sensitive actions require multisig approval, while EOAs are limited to low-risk operations that would have minimal impact if compromised.

2.3.8 Trust assumptions of enclave keys

Introduced by Version 1

Description In contract `GuardianModule`, function `rotateGuardianKey()` verifies signatures by active session key to ensure that only an attested TEE instance can set or rotate the enclave key.

However, the internal lifecycle management of the enclave key, including generation, sealing, access control, and hardware protection, falls outside the scope of this audit. We rely on the standard assumption that the underlying TEE guarantees the confidentiality and integrity of the key material, and our review does not assess those implementation details.

```
389     // Since rotateGuardianKey is called infrequently, blockNumber-based freshness is used for
            replay protection instead of a nonce.
390     bytes32 signedMessageHash =
391         keccak256(abi.encode("ROTATE_GUARDIAN_KEY", address(this), block.chainid, blockNumber,
                pubKey));
392     bool isValid = SESSION_REGISTRY.verifySessionSignature(
393         proof.sessionId, proof.sessionKey, signedMessageHash, proof.signature
394     );
395     if (!isValid) {
396         revert InvalidSignature();
397     }
398
399     bytes32 ownerFingerprint = SESSION_REGISTRY.getSessionOwner(proof.sessionId);
400     require(ownerFingerprint == LibKey.computeKeyFingerprint(proof.ownerKey),
            InvalidECDSAPubKey());
401
402     CVMSession memory session = SESSION_REGISTRY.getSession(proof.sessionId);
403     require(_allowedWorkloads[session.workloadId], WorkloadNotAllowed());
404
405     // pubKey[1:] means we need to strip the first byte '0x' if we want to get the correct
            address
406     address computedAddress = address(uint160(uint256(keccak256(pubKey[1:]))));
407
408     _guardianEnclaves[guardian].enclaveAddress = computedAddress;
409     _guardianEnclaves[guardian].enclavePubKey = pubKey;
```

Listing 2.49: puffer-contracts-updates-private/mainnet-contracts/src/GuardianModule.sol

Feedback from the project The contracts enforce that an enclave key can only be registered from an attested, allowlisted workload, and treat the confidentiality of the key material inside that workload as a property of the TEE rather than something verifiable on-chain.

2.3.9 Session and enclave keys must not be exportable

Introduced by Version 1

Description In the contract `GuardianModule`, the function `rotateGuardianKey()` registers an enclave key based on a session proof. However, the function does not check whether the session remains active. Therefore, the session key and the enclave key must be generated within the trusted domain and must not be exportable.

Feedback from the project The session key is generated and held inside the TEE (supervisor daemon uid=1), while the workload has no access to it. Meanwhile, the enclave key cannot be exported outside the TEE after registration.

2.3.10 Monitor VT balances to prevent blocked withdrawals

Introduced by Version 1

Description In the contract `PufferProtocol`, the function `batchHandleWithdrawals()` is expected to process exited validators and burn the corresponding Node Operator's Validator

Tickets. The function `_getVTBurnAmount()` caps the early-exit minimum penalty branch to the operator's stored VT balance, while the time-based burn branch returns the accrued burn amount directly. Withdrawal batch processing therefore depends on the relevant node operator having sufficient stored VT balance. The protocol should monitor Node Operators' VT balances and eject their validators before those balances become insufficient to cover accrued burns.

```
323         uint256 vtBurnAmount = _getVTBurnAmount($, bondWithdrawals[i].node, validatorInfos[i]);
324
325         // Update the burnAmounts
326         burnAmounts.pufETH += burnAmount;
327         burnAmounts.vt += vtBurnAmount;
328
329         // Store the withdrawal amount for that node operator
330         // Underflow is not possible because of the checks in the `_getBondBurnAmount` function
                and the fact that we are capping the burn amount to the bond amount
331         // nosemgrep basic-arithmetic-underflow
332         bondWithdrawals[i].pufETHAmount = (bondAmount - burnAmount);
333
334         emit ValidatorExited({
335             pubKey: validator.pubKey,
336             pufferModuleIndex: validatorInfos[i].pufferModuleIndex,
337             moduleName: validatorInfos[i].moduleName,
338             pufETHBurnAmount: burnAmount,
339             vtBurnAmount: vtBurnAmount
340         });
341
342         // Decrease the number of registered validators for that module
343         _decreaseNumberOfRegisteredValidators($, validatorInfos[i].moduleName);
344         // Storage VT and the active validator count update for the Node Operator
345         // nosemgrep basic-arithmetic-underflow
346         $.nodeOperatorInfo[validator.node].vtBalance -= SafeCast.toUint96(vtBurnAmount);
347         --$.nodeOperatorInfo[validator.node].activeValidatorCount;
348
349         delete validator.node;
350         delete validator.bond;
351         delete validator.module;
352         delete validator.status;
353         delete validator.pubKey;
354     }
355
356     VALIDATOR_TICKET.burn(burnAmounts.vt);
```

Listing 2.50: puffer-contracts-updates-private/mainnet-contracts/src/PufferProtocol.sol

```
804 function _getVTBurnAmount(ProtocolStorage storage $, address node, StoppedValidatorInfo
        calldata validatorInfo)
805     internal
806     view
807     returns (uint256)
808 {
809     uint256 validatedEpochs = validatorInfo.endEpoch - validatorInfo.startEpoch;
```

```
810         // Epoch has 32 blocks, each block is 12 seconds, we upscale to 18 decimals to get the VT
                amount and divide by 1 day
811         // The formula is validatedEpochs * 32 * 12 * 1 ether / 1 days
                (4444444444444444.44444444...) we round it up
812         uint256 vtBurnAmount = validatedEpochs * 4444444444444445;
813
814         uint256 minimumVTAmount = $.minimumVtAmount;
815         uint256 nodeVTBalance = $.nodeOperatorInfo[node].vtBalance;
816
817         // If the VT burn amount is less than the minimum VT amount that means that the node
                operator exited early
818         // If we don't penalize it, the node operator can exit early and re-register with the same
                VTs.
819         // By doing that, they can lower the APY for the pufETH holders
820         if (minimumVTAmount > vtBurnAmount) {
821             // Case when the node operator registered the validator but afterwards the DAO
                    increases the minimum VT amount
822             if (nodeVTBalance < minimumVTAmount) {
823                 return nodeVTBalance;
824             }
825
826             return minimumVTAmount;
827         }
828
829         return vtBurnAmount;
830     }
```

Listing 2.51: puffer-contracts-updates-private/mainnet-contracts/src/PufferProtocol.sol

Feedback from the project The project has implemented monitoring of VT balances.

2.3.11 Check the base image status separately

Introduced by Version 1

Description In contract `BaseImageRegistry`, function `deactivateBaseImage()` can revoke a base image referenced by a workload's allowlist. However, the revocation does not remove the base image from the workload allowlist in contract `WorkloadRegistry`, and function `isBaseImageAllowed()` continues to return `true` for the revoked image. Therefore, integrators should query the contract `BaseImageRegistry` to verify that a base image remains active rather than relying solely on the workload allowlist.

```
255     function deactivateBaseImage(
256         bytes32 baseImageId,
257         uint64 opExpiresAt,
258         PublicIdentity calldata ownerIdentity,
259         bytes calldata ownerSignature
260     ) external {
261         // Check signature expiration
262         if (block.timestamp > opExpiresAt) {
263             revert SignatureExpired(opExpiresAt, uint64(block.timestamp));
264         }
265
```

```
266         // Check exists and active
267         if (!_baseImages[baseImageId].exists) {
268             revert BaseImageNotFound(baseImageId);
269         }
270         if (_baseImages[baseImageId].isRevoked) {
271             revert BaseImageNotActive(baseImageId);
272         }
```

Listing 2.52: automata-tee-workload-measurement/src/BaseImageRegistry.sol

```
227     function isBaseImageAllowed(bytes32 workloadId, bytes32 baseImageId) external view returns (
            bool) {
228         if (!_workloads[workloadId].exists) {
229             revert WorkloadNotFound(workloadId);
230         }
231
232         AccessMode mode = _workloads[workloadId].workloadSpec.baseImageMode;
233
234         if (mode == AccessMode.ANY) {
235             return true;
236         } else if (mode == AccessMode.WHITELIST) {
237             return _baseImageSet[workloadId][baseImageId];
238         } else {
239             // BLACKLIST
240             return !_baseImageSet[workloadId][baseImageId];
241         }
242     }
```

Listing 2.53: automata-tee-workload-measurement/src/WorkloadRegistry.sol

Feedback from the project The project confirms that session creation checks revocation separately, while function `isBaseImageAllowed()` reflects only the allowlist policy.

2.3.12 Migration requirements in the upgrade script

Introduced by `Version 1`

Description The production upgrade script should migrate the dependency on the contract `GuardianModule` and the selector-based configuration in the contract `AccessManager`. Specifically, the script should perform the following steps:

1. Deploy a new instance of the contract `GuardianModule` and pass pufETH to its constructor. Deploy new `ValidatorTicket` and `PufferProtocol` implementations with the new `GuardianModule` address, and verify their relevant immutable dependencies before executing each proxy upgrade.
2. In the governance batch that upgrades contract `PufferProtocol`, invoke the function `AccessManager.s` to assign `PUBLIC_ROLE` to the updated selector of the function `registerValidatorKey()` in the contract `PufferProtocol`. The upgrade script should also verify every selector changed by the ABI update.

```
272         // If we are over the burst threshold, send everything to the treasury
273         if (PUFFER_ORACLE.isOverBurstThreshold()) {
```

```
274             IERC20(PUFFER_VAULT).transfer(TREASURY, pufEthUsed);
275             emit DispersedPufETH({ treasury: pufEthUsed, guardians: 0, burned: 0 });
276             return pufEthUsed;
277         }
278
279         ValidatorTicket storage $ = _getValidatorTicketStorage();
280
281         uint256 treasuryAmount = _sendPufETH(TREASURY, pufEthUsed, $.protocolFeeRate);
282         uint256 guardiansAmount = _sendPufETH(GUARDIAN_MODULE, pufEthUsed, $.guardiansFeeRate);
283         uint256 burnAmount = pufEthUsed - (treasuryAmount + guardiansAmount);
284
285         PufferVaultV5(PUFFER_VAULT).burn(burnAmount);
286
287         emit DispersedPufETH({ treasury: treasuryAmount, guardians: guardiansAmount, burned:
                burnAmount });
```

Listing 2.54: puffer-contracts-updates-private/mainnet-contracts/src/ValidatorTicket.sol

```
22    function run() public {
23        GenerateValidatorTicketCalldata calldataGenerator = new GenerateValidatorTicketCalldata();
24
25        vm.startBroadcast();
26
27        ValidatorTicket validatorTicketImpl = new ValidatorTicket({
28            guardianModule: payable(address(_getGuardianModule())),
29            treasury: payable(_getTreasury()),
30            pufferVault: payable(_getPufferVault()),
31            pufferOracle: IPufferOracle(address(_getPufferOracle()))
32        });
33
34        validatorTicket = ValidatorTicket(payable(_getValidatorTicket()));
35
36        vm.label(address(validatorTicket), "ValidatorTicketProxy");
37        vm.label(address(validatorTicketImpl), "ValidatorTicketImplementation");
38
39        // Upgrade on mainnet
40        upgradeCallData = abi.encodeCall(UUPSUpgradeable.upgradeToAndCall, (address(
              validatorTicketImpl), ""));
```

Listing 2.55: puffer-contracts-updates-private/mainnet-contracts/script/UpgradeValidatorTicket.s.sol

Feedback from the project The project states that the migration will redeploy or upgrade affected contracts, verify the `GUARDIAN_MODULE` binding, check selector changes, and migrate the required `AccessManager` roles.

2.3.13 Assumption of vTPM Migration

Introduced by `Version 1`

Description In the contract `SessionRegistry`, the function `rotateKey()` rotates the TPM signing key and session key of an active session, without verifying a new TEE report or checking the GCP PCR15 binding. This design assumes that the AK and TPM signing key cannot be

migrated or cloned to a CVM with a different TEE state. GCP confidential VM live migration can preserve vTPM keys and PCR values across a migration event. If the selected machine types support live migration [1], a trusted session may be created without verifying the current TEE state.

Feedback from the project The project states that GCP and Azure vTPM keys are instance-bound, preventing their use on another machine under the original policy. This assumption will be revisited if future platforms support vTPM migration across measurements.

2.3.14 Trust assumption of GCP TEE-to-vTPM binding

Introduced by `Version 1`

Description In the contract `SessionRegistry`, the function `_verifyTeeAkBinding()` derives `expectedPcr15` from the TEE report and compares it with PCR15 from the TPM quote. For GCP TDX, function `_verifyGcpTdxBinding()` uses the UUID and checks `RTMR3`. For GCP SNP, the function `_verifyGcpSnpBinding()` uses `REPORT_ID`. This binding scheme is defined by the project, and PCR15 appears to be defined and extended by guest software. Therefore, the security of this binding depends on the correctness of the scheme and its implementation by the off-chain boot and attestation components.

Feedback from the project The project states that the purpose of this binding is to prove that the vTPM quote and the hardware-signed TEE report come from the same VM. For GCP TDX, the project's kernel module generates the UUID and places it in the TDX `REPORT_DATA`, `RTMR3`, and vTPM PCR15. The contract verifies the expected `RTMR3` and PCR15 values. For GCP SNP, the AMD PSP generates the `REPORT_ID` at guest launch. The boot component reads it from the SNP report and extends it into vTPM PCR15.

[1] https://docs.cloud.google.com/confidential-computing/confidential-vm/docs/troubleshoot-live-migration


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS