CERTIK

Security Assessment

PLASMA

CertiK Assessed on Oct 11th, 2023

CertiK Assessed on Oct 11th, 2023

PLASMA

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Exchange	PLASMA	Manual Review, Static Analysis
LANGUAGE	**TIMELINE**	**KEY COMPONENTS**
Golang	Delivered on 10/11/2023	N/A

CODEBASE
ece01172905fed0e7b5f5f6247e757ed6da1273e
View All in Codebase Page

COMMITS
ece01172905fed0e7b5f5f6247e757ed6da1273e
View All in Codebase Page

Vulnerability Summary

23	16	0	0	6	1
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

1	Critical	1 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
0	Major		Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
6	Medium	6 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
8	Minor	5 Resolved, 3 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
8	Informational	4 Resolved, 3 Acknowledged, 1 Declined	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | PLASMA

CODEBASE

Repository

ece01172905fed0e7b5f5f6247e757ed6da1273e

Commit

ece01172905fed0e7b5f5f6247e757ed6da1273e

AUDIT SCOPE

31 files audited ● 2 files with Declined findings ● 11 files with Acknowledged findings ● 14 files with Resolved findings ● 4 files without findings

ID	Repo	Commit	File	SHA256 Checksum
DKR	lib	ece0117	tss/key/dkg/dkg_round.go	45575b8cefcf4e8d11183e74e572930f35e790297743c84fa5ddc0caad36ec84
UPD	/threshold-	ece0117	tss/key/reshare/update_round.go	02c1f08a77ae5811ac43dd39b0cad8d05d5b972975a7234463eabf4d0c76bd14
COI	/threshold-	ece0117	crypto/commitment/commitment.go	247e80f7e369af99fe7139482aaa177b5ee7582fcb591181a2fdbb2d706d6945
PAL	/threshold-	ece0117	crypto/paillier/paillier.go	662484576f01881166bf292f5b6161cfa08ab511c36464cbd4e8b0278a361859
SCN	/threshold-	ece0117	crypto/schnorr/schnorr_proof.go	2e0ce7f3cfc575a4411f5448a44bda3edc4c403dbc028dbae3f8306fdd8f4c82
UTI	/threshold-	ece0117	crypto/utils.go	9798aa7d9f168eaf7d3119d64df52320fe3597a3a6234ce6c6ca4ea66b5829dc
PAR	/threshold-	ece0117	tss/ecdsa/sign/party1.go	9f744042e35a945e195acfb3b731a6d59316e765ed564dd62460f9c6ce5c305e
PAT	/threshold-	ece0117	tss/ecdsa/sign/party2.go	12283901009072debf6bbf92dc1cc5273af3133db2053a12e5fd02fdb2e6ee49
UTL	/threshold-	ece0117	tss/ed25519/sign/utils.go	36b6b9e5f2db2378609e6df0638e32a30c5e89bc07423a14bd04ecf9c48bca75
TSK	/threshold-	ece0117	tss/key/bip32/tsskey.go	d576b1fead7ee5319ce24ee62fe74e4ffde0a42bb01c1c4d881657414d5fbb97
UPA	/threshold-	ece0117	tss/key/reshare/update_round1.go	bdd119e200526ac4d9a456202e20046e64fd318bb49eab07bff8d9669e8bc027
UPT	/threshold-	ece0117	tss/key/reshare/update_round2.go	10b9c2ab734a13b63207094a2dcba185b85b92e1f8f0be9a28290bbb3c2979d9

ID	Repo	Commit	File	SHA256 Checksum
GOE	/threshold-	ece0117	go.mod	ded98cbba29a1a212b63cc80568ae7bcf5adde0be70a1622837d1ccc384451eb
CUV	/threshold-	ece0117	crypto/curves/curve.go	3a954a6794621bed807970aadb6f9d1e1e415d3f408b398164d8589fdcc3412a
COR	/threshold-	ece0117	crypto/paillier/correct_key_ni.go	9de5ffaca7702d6a09ecdcdea29eee7ecee862b8831c538c6f813c22a8f72c6d
FEL	/threshold-	ece0117	crypto/vss/feldman.go	198301ac8bdc01b2e94d4982cac130b5cfdc07bb880bf23ae35689275ce2f627
POL	/threshold-	ece0117	crypto/vss/polynomial.go	fe8c9d4631ff6ae87864ce8fe8fd281f809dc1bea06b4e00e5090a8a6d167f02
DLN	/threshold-	ece0117	crypto/zkp/dlnproof.go	503e15158382fdbf21cb02404a89c4fc06228e7f9790a65125d339e5c4679514
PDL	/threshold-	ece0117	crypto/zkp/pdl_w_slack_proof.go	54a0142afbf9792c6623bddbbb22a7e35711e761e472db1b4b92914c3e5db599
ALI	/threshold-	ece0117	tss/ecdsa/keygen/alice.go	8fe2f2f70697d80785187096c280b43a9585a46ee27243eb52bf6dd4fc0454a6
BOB	/threshold-	ece0117	tss/ecdsa/keygen/bob.go	735807d6de52318aacc8b02cd3fdcc467a42c4408da187c5e123ef416f24b5ea
ED5	/threshold-	ece0117	tss/ed25519/sign/ed25519.go	01d7ef94bb1e6f1a1084568acb52e77bfd9b57f14ca3139fd887ebc50e49ed79
ROD	/threshold-	ece0117	tss/ed25519/sign/round3.go	65ba0756a4b5ce149c3d46180ec9a19ffd71b2dca739063a27aea7805543ab92
DKO	/threshold-	ece0117	tss/key/dkg/dkg_round1.go	4ec820b3b4fc96c64b633827a895abe277e2a53bf364c1a8ee47076f23159c71
DKN	/threshold-	ece0117	tss/key/dkg/dkg_round3.go	310335d520cfe72f7921812eebc9962d058ae7fb92467d36307c4db03b63af78
UPE	/threshold-	ece0117	tss/key/reshare/update_round3.go	4d8a8c748315225589f600fb38413b02e4d4eddfb375f90a7a801944dc045658
COO	/threshold-	ece0117	tss/common.go	56bda0dc78dc34f7ffd715c26f354b0d9075cf76964e80f2078494a06e09bf79

ID	Repo	Commit	File	SHA256 Checksum
ECP	/threshold-	ece0117	crypto/curves/ecpoint.go	396d9e02f9c7328093a103534926628a2fa7482658026f6e361af6d61e233100
ROU	/threshold-	ece0117	tss/ed25519/sign/round1.go	c5a22a9fc595023757e14d3a8ec036315dc88ff350868cac710dacd16e69a040
RON	/threshold-	ece0117	tss/ed25519/sign/round2.go	0525b1addfc413a4d6cb03ccdcd08df11ab67c17998450c39a9af18e5cac4337
DKU	/threshold-	ece0117	tss/key/dkg/dkg_round2.go	0f5ef8ef61ee5769dd224b003b42db4861132b4b0aa7dc062519ae9f8c30c19e

APPROACH & METHODS

This report has been prepared for to discover issues and vulnerabilities in the source code of the (Threshold-lib) project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis techniques.

Audit Approach

The audit process was comprehensive, focusing on both the security and functional aspects of the threshold signature crypto library. The library was evaluated for adherence to industry standards and practices, and for alignment with the specifications and intentions of the client. The audit was conducted in the following stages:

- Specification and Literature Review: A thorough review of the library specifications and any related literature was conducted. This provided a clear understanding of the intended behavior of the library and set a benchmark for the subsequent review.
- Functional Review: A detailed review of the functional matching between the code and the specified intended behavior was carried out. This involved a line-by-line manual review of the entire codebase by industry experts to ensure that the library functioned as intended.
- Cryptographic Primitives Assessment: The cryptographic primitives used in the library were assessed. This involved checking the security of the cryptographic algorithms and implementations in respect to their reference specification papers.
- Software Security Code Review: A software security code review was conducted to identify any potential vulnerabilities. This involved testing the library against both common and uncommon attack vectors.
- Codebase Best Practices Assessment: The codebase was assessed for compliance with current best practices and industry standards. This involved cross-referencing the library structure and implementation against similar libraries produced by industry leaders.
- Documentation Review: The comments and documentation provided in the codebase were also reviewed for readability and understanding.

Audit Methodology

The audit was conducted in a static manner, focusing on the source code of the library. No dynamic analysis was performed on the codebase. Here's a detailed breakdown of the methodology:

- Manual Code Review: Industry experts conducted a thorough line-by-line manual review of the entire codebase. This helped identify any potential issues or vulnerabilities in the code.
- Automated Static Analysis: Automated static analysis tools were used to identify common coding errors and vulnerabilities. This supplemented the manual code review.
- Unit Testing: The existing unit tests were reviewed and additional tests were suggested to cover possible use cases. This helped ensure the robustness of the library.

- Documentation Review: The comments and documentation provided in the codebase were reviewed. Improvements were suggested to enhance readability and understanding, especially for contracts that were verified in public.
- Comparison with Industry Standards: The library structure and implementation were cross-referenced against similar libraries produced by industry leaders. This ensured that the library adhered to industry standards.
- Security Assessment: A security assessment of the library was conducted, identifying findings that ranged from critical to informational. Recommendations were provided to address these findings and enhance the security of the library.

The primary goal of this audit was to evaluate the overall robustness of the threshold signature crypto library against a range of potential real-world attacks targeting the library's controls and functions. By identifying any weaknesses, we aimed to provide recommendations to address these vulnerabilities and enhance the library's overall security posture. The audit was particularly focused on the library's use of cryptographic primitives, its handling of potentially untrusted inputs, and its adherence to the protocol specifications. The findings of the audit offer valuable insights that can guide the ongoing development and refinement of the library, ensuring it remains a reliable and secure tool for web3 use cases.

Three members of our audit team were involved in this engagement, which spanned over the course of 15 days in June 2023 and resulted in 25 security-relevant findings. The most significant findings, while of medium severity, highlight areas for improvement in the following categories:

- Loop Control and Termination: There were issues with the termination of loops in the Public Share Map Calculation, which could potentially lead to inefficiencies or errors in the execution of the program.
- Resource Management: Inappropriate closure of channels was observed, which could lead to resource leakage or unexpected behavior in concurrent operations.
- Error Handling: There were instances where nonexistent curves could potentially cause the program to panic, indicating a need for better error handling and validation of inputs.
- Cryptographic Protocol Adherence: The library was found to be missing a proof of correct Paillier encryption, which is crucial for ensuring the integrity and security of the cryptographic operations. The DLNProof Algorithm was found to have reduced iterations, which could potentially impact the security and effectiveness of the cryptographic processes.

These findings highlight areas where the threshold signature crypto library could be improved to enhance its overall security posture. While none of these issues are critical, they could potentially impact the library's robustness and reliability if not addressed.

Other weaknesses were also identified and are detailed in the Findings section of the report. We recommend addressing these findings to ensure a high level of security standards and industry practices, and to enhance the overall security posture of the threshold signature crypto library. Our team is confident that by addressing these issues, the library will continue to serve as a reliable and secure tool for web3 use cases.

REVIEW NOTES

The threshold signature crypto library, developed by and open-sourced on GitHub, implements algorithms for Multi-Party Computation aimed at 2 out of n key-pair management trying to balance signing efficiency while meeting the business requirements of web3 use cases.

The library uses Feldman's Verifiable Secret Sharing scheme (Feldman's VSS) for the distributed key generation problem and relies on the Lindell '17 protocol to effectively compute ECDSA signatures using the previously generate private local shares. EdDSA is supported exploiting the additive properties of the signature scheme and new key-pairs can be derived through the support to the BIP-32 non-hardened key derivation. Finally a re-share algorithm is provided to allow local key share refresh in the case new participants join the signing group.

Significantly, the library has incorporated implementations from Binance and ZenGo's threshold crypto libraries. This adoption of tested and proven code from reputable sources in the industry adds to the reliability and robustness of the library. However, it is crucial to ensure that these implementations are correctly integrated and that they align with the overall design and functionality of the library.

Audit Phases

The audit was structured into three distinct phases:

- **Go Safety Programming Review**: The code was scrutinized for potential software defects, with a particular emphasis on how it handles untrusted inputs. The review specifically focused on Go safety programming, including the handling of nil pointers, error handling, data races, and memory leaks. Attention was also given to the use of third-party packages, as they can introduce vulnerabilities if not properly vetted. The review also examined the code for susceptibility to known vulnerabilities, unsafe behavior, leakage of secrets or sensitive data, susceptibility to misuse and system errors, and safety against malformed or malicious input from other network participants.
- **Cryptography Analysis**: The cryptographic primitives and protocols employed were thoroughly analyzed. This included a detailed examination of randomness and hash generation, signatures, key management, zero-knowledge proofs, and encryption. The review ensured that the cryptographic primitives were appropriately matched to the required cryptographic functionality, and that they maintained a high security level.
- **Protocol Specification Matching**: The audit team analyzed the original paper and cross-checked the code to ensure it aligns with the given specification. This involved checking the correct implementation of protocol phases, error handling, zero-knowledge proofs, and adherence to the protocol's logical description.

Audit Comments

The audit also resulted in the following comments:

1. The library operates under an optimistic assumption that all interacting parties will behave correctly. This means that the library's algorithms are designed with the expectation that all parties involved in a transaction or operation will

follow the prescribed rules and protocols. This approach has the advantage of speeding up the execution of algorithms, as it reduces the number of messages that need to be exchanged between parties. However, this optimistic assumption also has potential drawbacks. If a party does not behave correctly - for example, if they provide incorrect or malicious input - the protocol may proceed to its conclusion before the error is detected. This could result in an unusable outcome, such as a failed transaction or an incorrect computation, after computational resources have already been expended. Therefore, while the optimistic assumption can improve efficiency, it also underscores the importance of robust error detection and handling mechanisms within the library.

2. The library's re-share protocol allows for generating a new set of shares for a new set of participants in relation to the same global key-pair. This feature can be useful when new actors need to be added to the participants. However, it's important to note that once a re-share is completed, the old set of shares remains valid. This means that the original participants still have access to the shared key, which could pose a security risk if those original participants are no longer trusted or if they have had their shares compromised. Therefore, the re-share protocol should be used with caution. If the goal is to remove an entity from the set of participants, simply resharing the key would not be sufficient to prevent the removed entity from accessing the shared key. In such cases, a brand new key-pair must be generated with the new set of trusted participants. This ensures that removed or compromised participants no longer have access to the shared key.

3. The ECDSA signing protocol based on Lindell 17 uses a range zero-knowledge proof complemented with a Discrete Logarithm proof regarding the parameters of the range proof. However, no reference was provided about this second proof, so the audit team could not cross-check in the cryptography literature the theory and security assumptions behind this implementation decision.

4. The audit of the threshold signature crypto library revealed that the current test suite primarily focuses on verifying the successful execution of the different cryptographic primitives and protocols. While this is crucial for validating the library's functionality under ideal conditions, it does not fully account for real-world scenarios where the library might be used incorrectly or misused. Misuse can occur due to user error, misunderstanding of the library's functions, or even malicious attempts to exploit potential vulnerabilities. The absence of tests simulating misuse or incorrect usage means the library might not be fully equipped to handle these scenarios, potentially leading to unexpected behavior or exploitable vulnerabilities. It is recommended to expand the test suite to cover potential misuse or incorrect usage scenarios. This would help ensure the library's robustness and security across a wider range of scenarios and use cases.

Scopes and Limitations

The audit of the threshold signature crypto library was conducted with a focus on the library's implementation of a secure t/n ECDSA/EdDSA signature scheme. It's important to note that while the library provides the tools for secure multi-party computation, the overall security of a solution built using the library heavily relies on proper usage practices by the library users.

In a 2/3 scheme, for instance, the full private key can be reconstructed if two parties have their key shares leaked. Therefore, library users must ensure the safe storage and handling of their key shares to prevent such leaks. When using the re-share protocol, library users must not only generate a new set of shares but also properly invalidate and erase the old shares. If the old shares remain valid and fall into the wrong hands, the security of the system could be compromised.

Additionally, the threshold library does not serve as an access policy component. If one party's key share is leaked and the other party continues to perform the signature process without question, the security benefits of having multiple shares are negated. Therefore, library users should implement additional checks and balances to ensure that all parties are behaving correctly and that key shares have not been compromised.

Furthermore, the audit does not guarantee that the library will be free of issues if misused. Misuse can occur in various ways, such as not using the correct parameters or curves, not following the correct order of API calls, or not implementing the correct usage scenario and access policy. These factors can significantly impact the security of the system and are beyond the scope of the library itself.

In essence, while the library provides a set of tools for secure multi-party computation, library users must also follow best practices in key management and access control to maintain the security of the system. The audit's scope was limited to the library itself, and it is the responsibility of the library users to ensure the library is used correctly and securely in their specific applications.

Conclusion

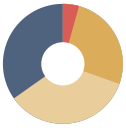
While the audit revealed the high-risk Lindell17 Abort Vulnerability (CVE-2023-33242), the client promptly addressed it. The client introduced a ban list design in commit: de1431b2c9b6d601e0bf7e3566537c3d22e9eb8b, where failed ECDSA requests result in a permanent ban, efficiently mitigating the risk albeit with a potential Denial of Service on the user side. Further improvement was made by adopting a zero knowledge proof approach in pull request: c191c162d2f7cd52f9ce6eb5b945d3cd5f21be68. Leveraging the Paillier affine operation with group commitment in range zk proof, Paillier Blum modulus zk proof and No Small Factor zk proof from CGGMP21, the last message is now proofed and verified. Any failed zero knowledge proof results in early termination to prevent signature result leak. This approach, used in conjunction with the ban list, introduces a more robust defense mechanism.

The auditors also would like to take a note that the library heavily relies on existing cryptographic primitives. This approach has both advantages and potential drawbacks. On the positive side, using established cryptographic primitives reduces the risk of introducing new vulnerabilities. However, it also suggests a need for a deeper understanding of the underlying cryptographic principles to ensure that these primitives are being used correctly and optimally, which is partially reflected in the CVE-2023-33242 scenario.

The audit has identified areas for improvement, particularly in loop control and termination, resource management, error handling, and adherence to cryptographic protocols. Addressing these areas could enhance the library's robustness and reliability in all its implemented functionalities.

Finally, although the adoption of implementations from Binance and ZenGo's threshold crypto libraries builds on top of robust packages, it is recommended to provide more references and documentation. In particular, the Discrete Logarithm proof used in the ECDSA signing protocol should be complemented with reference documentation and security proofs, while in general, a deeper understanding of the cryptographic principles underlying the used primitives could further enhance the library's security and efficiency. It's also important to note that the security landscape is constantly evolving, and regular audits are recommended to ensure that the library continues to meet the necessary security standards and industry practices.

FINDINGS



23	1	0	6	8	8
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for (Threshold-lib). Through this audit, we have uncovered 23 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
GLOBAL-02	CVE-2023-33242 Lindell17 Abort Vulnerability	Private Key Leakage	Critical	Resolved
ALI-01	Missing Proof Of Correct Paillier Encryption	Inconsistency	Medium	Resolved
CUV-01	Potential Panic Caused By Nonexistent Curve	Coding Issue	Medium	Resolved
DLN-01	Reduced Iterations In The DLNProof Algorithm	Volatile Code	Medium	Resolved
ECE-01	Inappropriate Channel Closure	Coding Issue	Medium	Resolved
ECE-02	Possible Go Routine Leakage	Coding Issue	Medium	Resolved
KET-01	Incorrect Loop Termination On Public Share Map Calculation	Logical Issue	Medium	Resolved
DLN-02	Missing Preliminary Validation In DLNProof Algorithm Verification Function	Inconsistency	Minor	Resolved
PAR-01	Missing Validation On Message Encoding	Incorrect Calculation	Minor	Resolved
POL-01	Missing Error Check And Boundary Check In Function `InitPolynomial`	Coding Issue	Minor	Resolved

ID	Title	Category	Severity	Status
SCN-01	Discrepancy Between Implementation And Specification In Schnorr Proof Algorithm	Coding Issue	Minor	Acknowledged
SIG-01	Missing Round Enforcement In ECDSA Contexts	Volatile Code	Minor	Acknowledged
TSK-01	Mismatch On Chaincode Usage In BIP-32 Key Derivation	Inconsistency	Minor	Acknowledged
TSK-02	Missing Validation On Child Key Pair Calculation	Inconsistency	Minor	Resolved
TSK-03	Missing Hardened Key Derivation Implementation	Inconsistency	Minor	Resolved
COI-01	Non Timing-Constant Int Value Comparison	Language Design Issue	Informational	Acknowledged
COR-01	Outdated Reference Paper For Pailler Correctness Proof	Inconsistency	Informational	Resolved
CRY-03	Inconsistent Random Number Error Handling	Coding Style	Informational	Resolved
ECE-03	Dependency Import Order Format	Coding Style	Informational	Resolved
GOE-01	Potential Vulnerable Runtime Version	Language Version	Informational	Acknowledged
KET-02	Panic Used Instead Of Error Messages	Coding Style	Informational	Declined
PDL-01	Invalid Reference Paper URL	Invalid Reference	Informational	Resolved
RES-01	Unnecessary Computation Of Random Polynomial In Non-Devotees	Coding Issue	Informational	Acknowledged

GLOBAL-02 | CVE-2023-33242 LINDELL17 ABORT VULNERABILITY

Category	Severity	Location	Status
Private Key Leakage	Critical		Resolved

Description

The audited version of the library was found to be vulnerable to the Lindell17 Abort Vulnerability (CVE-2023-33242), where an attacker could potentially extract the full private key from a wallet implementing the Lindell17 2PC protocol. This vulnerability arises from deviations in the Lindell17 implementations from the specification of the academic paper, particularly in handling aborts during failed signature attempts. It was discovered that an attacker, with privileged access, could exploit this vulnerability to exfiltrate the key after approximately 200 malicious signature requests, thereby posing a severe security risk to the affected systems and their users.

The root cause of the Lindell17 Abort Vulnerability stems from some implementations of the Lindell17 protocol mishandling or ignoring aborts in cases of failed signatures. This oversight allows an attacker, assuming privileged access, to exploit this flaw and extract a full private key by initiating malicious signature requests, posing a significant security threat to the affected systems.

Vulnerability detail: https://www.fireblocks.com/blog/lindell17-abort-vulnerability-technical-report

Proof of Concept

Following the description provided in https://www.fireblocks.com/blog/lindell17-abort-vulnerability-technical-report. The auditors performed a quick proof of concept verification on the audited code base. As shown in the screenshot, the attacker can deduce the last bit of the party two's private key by checking if the signing process is successful. This can be extended to extract all bits of the other party's key share.

```
threshold-lib (ece0117) [!?] via  v1.21.0 took 11s
❯ go run trail.go
CVE-2023-33242 trails.
=========2/2 keygen==========
p1 private key: 145043300967419279221646883009716431125841613829320376042984763863466579757696
p1 private key is even: true
=========2/2 sign==========
1047627487292000646713667857159717902203643869858502944224122709573271575300 72 480924101933161369471148119461480066720011583541715537665379673290933260853 86 <nil>

threshold-lib (ece0117) [!?] via  v1.21.0 took 17s
❯ go run trail.go
CVE-2023-33242 trails.
=========2/2 keygen==========
p1 private key: 202119219376594240945236510817167248636861355784153010719660068520419212434353
p1 private key is even: false
=========2/2 sign==========
<nil> <nil> ecdsa sign verify fail

threshold-lib (ece0117) [!?] via  v1.21.0 took 7s
❯
```

Recommendation

As discussed in the vulnerability detail, there are several ways to address this issue. The developers can either introduce blacklist mechanism to permanently ban the malicious user and corresponding key share. Another potential solution is to

introduce zero knowledge proof during the last message interaction stage.

Alleviation

The client has introduced the ban list design in commit: de1431b2c9b6d601e0bf7e3566537c3d22e9eb8b. The failed ecdsa request will result in permanent ban. This approach is efficient yet might cause Denial of Service on user side. To further improve the solution, a zero knowledge proof approach is also introduced in pull request: c191c162d2f7cd52f9ce6eb5b945d3cd5f21be68. More specifically, leveraging the Paillier affine operation with group commitment in range zk proof, Paillier Blum modulus zk proof and No Small Factor zk proof from CGGMP21, the last message is now proofed and verified. Any failed zero knowledge proof will result in early termination to prevent signature result leak. This approach will also be used in conjunction with the ban list approach to introduce more robust defense mechanism.

ALI-01 | MISSING PROOF OF CORRECT PAILLIER ENCRYPTION

Category	Severity	Location	Status
Inconsistency	Medium	tss/ecdsa/keygen/alice.go: 83	Resolved

Description

The key generation sub-protocol in Lindell17 ends up with the two parties having the global public key, their local private key share. Moreover, the entity playing party 2 also obtains a Paillier encryption of party 1 private share that will be necessary in the signing sub-protocol.

In order to convince party 2 that the shared encryption actually is the chipertext corresponding to the local private share behind the shared partial public key, party 1 also sends to party 2 a proof demonstrating such assertion. A method to compute and verify such proof is reported in section 6 of Lindell17.

In the scenario implemented in the codebase in scope, parties generate key shares using a combination of multiple (one per party) VSS sessions. Then, in order to run the Lindell protocols, the Paillier encryption of the local share is shared between the signing actors (the c_{key} value). However, no proof that such encryption is correct is shared by party 1, so party 2 can only trust that the shared information is correct. In the case in which such encryption is not correct, party 1 can induce party 2 either in computing useless data or in participating in the calculation of a signature verifiable by a random key-pair that only party 1 is aware of.

Recommendation

The auditors recommend implementing the proof generation and verification reported in section 6 of Lindell17 to verify that the shared cipher-text is a valid encryption of the local private secret.

Alleviation

File `pdl_w_slack_proof.go` is used to introduce the proof action to validate the ciphertext/private share relationship. Note that also this proof carries the range proof by design but the range proof verification step is skipped to avoid redundant range proof action.

CUV-01 | POTENTIAL PANIC CAUSED BY NONEXISTENT CURVE

Category	Severity	Location	Status
Coding Issue	Medium	crypto/curves/curve.go: 26~29	Resolved

Description

In the following code, the developers did not check if curve is existed in curve map. In Go programming language, the value will be `nil` or `0` if key is not found in the map.

```
func GetCurveByName(curveName string) elliptic.Curve {
    return curveMap[curveName]
}
```

This function is further used in the following code, which takes unsanitized JSON input and retrieve the curve by JSON `curve` field.

```
func (p *ECPoint) UnmarshalJSON(payload []byte) error {
    aux := &struct {
        Curve string
        X     *big.Int
        Y     *big.Int
    }{}
    if err := json.Unmarshal(payload, &aux); err != nil {
        return err
    }
    p.X = aux.X
    p.Y = aux.Y
    p.Curve = GetCurveByName(aux.Curve)

    if !p.IsOnCurve() {
        return fmt.Errorf("UnmarshalJSON error, point not on the curves ")
    }
    return nil
}
```

However, the code will panic (caused by null pointer dereference) if curve's name is not precisely `secp256k1` or `ed25519`.

Proof of Concept

```
var test elliptic.Curve
test = GetCurveByName("test")
_ = test.IsOnCurve(big.NewInt(10), big.NewInt(10))
```

caused panic:

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x28 pc=0x4d8e5d]

goroutine 1 [running]:
main.main()
    /tmp/sandbox1107371530/prog.go:37 +0xbd
```

Recommendation

Please use error code scheme when curve name is not found in pre-defined curve map.

Alleviation

Curve name is now checked against existence and error message is returned if curve not found.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/curves/curve.go#L26

DLN-01 | REDUCED ITERATIONS IN THE DLNPROOF ALGORITHM

Category	Severity	Location	Status
Volatile Code	Medium	crypto/zkp/dlnproof.go: 19~20	Resolved

Description

The current `DlnProof` implementation is largely borrowed from Binance open source threshold signature library. The corresponding code is located at https://github.com/bnb-chain/tss-lib/blob/master/crypto/dlnproof/proof.go.

However, the adapted code reduced the number of components of the proof from the original `128` to `12`. This reduction in iterations weakens the security of the proof. In fact, the number of iterations in an interactive zk-proof (even though made non-interactive with the Fiat-Shamir heuristic) directly impacts the probability that the proof correctly demonstrates its assertion. A reduced number of iterations may make possible for an attacker to issue a proof which verifies as `true` while the underlying fact to demonstrate is `false`.

While `12` iterations may speed up the computation, they do not represent a big enough value to trust the outcome of the proof verification, thereby potentially compromising the security of the system.

Recommendation

The auditors recommend restoring the number of iterations to its original value of 128 to ensure the robustness of the DLN.

Alleviation

Client has increased the number of iterations into 30 and reaches the confidence > 99.9999999%.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/zkp/dln_proof.go#L14

ECE-01 | INAPPROPRIATE CHANNEL CLOSURE

Category	Severity	Location	Status
Coding Issue	Medium	crypto/paillier/paillier.go: 41~50; tss/ecdsa/keygen/alice.go: 30~40	Resolved

Description

In the mentioned code snippet, the developers use the go routine function `crypto.GenerateSafePrime` to generate two safe prime numbers. The `crypto.GenerateSafePrime` function takes `quit` channel parameter as a control signal to terminate the prime generation process.

Using code from `paillier.go` as an example:

```
var values = make(chan *big.Int)
var quit = make(chan int)
var p, q *big.Int
for p == q {
    for i := 0; i < currency; i++ {
        go crypto.GenerateSafePrime(PrimeBits/2, values, quit)
    }
    p, q = <-values, <-values
    close(quit)
}
```

The above code creates a channel `quit` before launching go routines. The for loop checks if generated two prime numbers are the same. The loop continues if two same prime numbers are acquired. However, the channel `quit` is closed in every iteration of the for loop, meaning if the loop continues then the repeated `close` action on the already closed `quit` channel will cause unexpected `panic`.

Recommendation

Please consider remove unused control channel `quit` from `crypto.GenerateSafePrime` function. Alternatively, the developers can use `sync.Once` to ensure a channel is only closed once.

Alleviation

The `quit` channel now has been moved into the `for` loop which addressed the raised issue.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/paillier/paillier.go#L44
https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/ecdsa/keygen/party1.go#L36

ECE-02 | POSSIBLE GO ROUTINE LEAKAGE

Category	Severity	Location	Status
Coding Issue	Medium	crypto/paillier/paillier.go: 41~50; tss/ecdsa/keygen/alice.go: 31~40	Resolved

Description

Using code from `paillier.go` as example:

```
var values = make(chan *big.Int)
var quit = make(chan int)
var p, q *big.Int
for p == q {
    for i := 0; i < currency; i++ {
        go crypto.GenerateSafePrime(PrimeBits/2, values, quit)
    }
    p, q = <-values, <-values
    close(quit)
}
```

This code launches currency number of go routines in each iteration of the loop, each running the GenerateSafePrime function. However, only two values (i.e., `p` and `q`) are read from the values channel in each iteration. This means that if currency is greater than 2, there will always be some go routines whose results are not read in each iteration. These go routines will be blocked waiting for their results to be read, leading to a go routine leak. This leak can cause unnecessary resource consumption and potential performance degradation in the program.

Recommendation

1. Limit the number of go routines: Since the developers are only interested in two values p and q per iteration, it would be more efficient to only spawn two go routines per iteration. This way, it ensures that all go routines are able to send their results to the values channel and no go routines are left hanging.

2. Use buffered channels: If the developers still want to run more than two go routines per iteration, consider using a buffered channel. This allows a go routine to send its result to the channel and terminate, even if its result isn't immediately read from the channel. For example, developers could create the channel like this: `values := make(chan *big.Int, currency)`. This creates a channel with a buffer size equal to currency, so up to currency results can be sent to the channel without blocking.

Alleviation

Client now uses buffered channel to receive generated safe prime number.

See:

1. https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/paillier/paillier.go#L41
2. https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/ecdsa/keygen/party1.go#L33

KET-01 INCORRECT LOOP TERMINATION ON PUBLIC SHARE MAP CALCULATION

Category	Severity	Location	Status
Logical Issue	Medium	tss/key/dkg/dkg_round3.go: 92; tss/key/reshare/update_round3.go: 92	Resolved

Description

The `for` constructs at the pointed locations initializes the `sharePubKeyMap` map with the public keys obtained from the secret shares commitments sent by the parties of the protocol. Such data are then used to verify the correctness of the owned share.

The `map` initialization is stopped after $t + 1$ iterations, where t is the threshold to overcome to reconstruct the global shared secret. However, n can be greater than $t + 1$ as the number of parties participating in the protocol is not related to t. In this way, when $n > t + 1$, the parties with ID higher than $t + 1$ can not verify the correctness of their share and the DKG protocol is interrupted.

The pointed locations refer to the correspondent steps the key generation and reshare, which both present the same incorrect behavior.

Proof of Concept

The following test runs the DKG algorithm with a threshold of 2 among 4 parties, so $n > t + 1$. It panics in line 42 since the final result for the 4th peer is not computed.

```go
func TestKeyGen2_4(t *testing.T) {
    curve := secp256k1.S256() // edwards.Edwards()
    setUp1 := NewSetUp(1, 4, curve)
    setUp2 := NewSetUp(2, 4, curve)
    setUp3 := NewSetUp(3, 4, curve)
    setUp4 := NewSetUp(4, 4, curve)

    msgs1_1, _ := setUp1.DKGStep1()
    msgs2_1, _ := setUp2.DKGStep1()
    msgs3_1, _ := setUp3.DKGStep1()
    msgs4_1, _ := setUp4.DKGStep1()

    msgs1_2_in := []*tss.Message{msgs2_1[1], msgs3_1[1], msgs4_1[1]}
    msgs2_2_in := []*tss.Message{msgs1_1[2], msgs3_1[2], msgs4_1[2]}
    msgs3_2_in := []*tss.Message{msgs1_1[3], msgs2_1[3], msgs4_1[3]}
    msgs4_2_in := []*tss.Message{msgs1_1[4], msgs2_1[4], msgs3_1[4]}

    msgs1_2, _ := setUp1.DKGStep2(msgs1_2_in)
    msgs2_2, _ := setUp2.DKGStep2(msgs2_2_in)
    msgs3_2, _ := setUp3.DKGStep2(msgs3_2_in)
    msgs4_2, _ := setUp4.DKGStep2(msgs4_2_in)

    msgs1_3_in := []*tss.Message{msgs2_2[1], msgs3_2[1], msgs4_2[1]}
    msgs2_3_in := []*tss.Message{msgs1_2[2], msgs3_2[2], msgs4_2[2]}
    msgs3_3_in := []*tss.Message{msgs1_2[3], msgs2_2[3], msgs4_2[3]}
    msgs4_3_in := []*tss.Message{msgs1_2[4], msgs2_2[4], msgs3_2[4]}

    p1SaveData, err := setUp1.DKGStep3(msgs1_3_in)
    if err != nil {
        panic(fmt.Sprintf("Error on step 3 party 1: %s", err))
    }
    p2SaveData, err := setUp2.DKGStep3(msgs2_3_in)
    if err != nil {
        panic(fmt.Sprintf("Error on step 3 party 2: %s", err))
    }
    p3SaveData, err := setUp3.DKGStep3(msgs3_3_in)
    if err != nil {
        panic(fmt.Sprintf("Error on step 3 party 3: %s", err))
    }
    p4SaveData, err := setUp4.DKGStep3(msgs4_3_in)
    if err != nil {
        panic(fmt.Sprintf("Error on step 3 party 4: %s", err))
    }

    fmt.Println("setUp1", p1SaveData, p1SaveData.PublicKey)
    fmt.Println("setUp2", p2SaveData, p2SaveData.PublicKey)
    fmt.Println("setUp3", p3SaveData, p3SaveData.PublicKey)
    fmt.Println("setUp4", p4SaveData, p4SaveData.PublicKey)
}
```

Recommendation

The auditors recommend letting the pointed `for` cycles to terminate after `info.Total` iterations instead of `info.Threshold+1`, so that the DKG algorithm is completed on all parties with `ID > t + 1` when `n` > `t+1`.

Alleviation

Client fixed the issue and the pointed cycle is terminated after `n` iterations to cover all the involved parties.

See:

1. https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/key/reshare/update_round3.go#L93
2. https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/key/dkg/dkg_round3.go#L93

DLN-02 MISSING PRELIMINARY VALIDATION IN DLNPROOF ALGORITHM VERIFICATION FUNCTION

Category	Severity	Location	Status
Inconsistency	Minor	crypto/zkp/dlnproof.go: 50~73	Resolved

Description

The current `DlnProof` implementation is largely borrowed from Binance open source threshold signature library. The corresponding code is located at https://github.com/bnb-chain/tss-lib/blob/master/crypto/dlnproof/proof.go. However, the adapted code removed the preliminary validation code from original function, as shown in the following:

```
if p == nil {
    return false
}
if N.Sign() != 1 {
    return false
}
modN := common.ModInt(N)
h1_ := new(big.Int).Mod(h1, N)
if h1_.Cmp(one) != 1 || h1_.Cmp(N) != -1 {
    return false
}
h2_ := new(big.Int).Mod(h2, N)
if h2_.Cmp(one) != 1 || h2_.Cmp(N) != -1 {
    return false
}
if h1_.Cmp(h2_) == 0 {
    return false
}
for i := range p.T {
    a := new(big.Int).Mod(p.T[i], N)
    if a.Cmp(one) != 1 || a.Cmp(N) != -1 {
        return false
    }
}
for i := range p.Alpha {
    a := new(big.Int).Mod(p.Alpha[i], N)
    if a.Cmp(one) != 1 || a.Cmp(N) != -1 {
        return false
    }
}
```

The `Verify` function of DLNProof Algorithm takes input from other party and it is necessary to perform preliminary checking to ensure the legitimacy of passed parameters before performing actual verification step.

Recommendation

It is recommended for the developers to add back the parameters validation code in `Verify` function of DLNProof Algorithm.

Alleviation

The preliminary validation has been improved and aligned with original implementation.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/zkp/dln_proof.go#L46

PAR-01 | MISSING VALIDATION ON MESSAGE ENCODING

Category	Severity	Location	Status
Incorrect Calculation	Minor	tss/ecdsa/sign/party1.go: 26, 32	Resolved

Description

The code to initiate a signature session for the ECDSA algorithm takes as an initialization parameter the `message` to sign. Such parameter will not be taken into consideration before the second message by party 2, since some coefficient generation is necessary before proceeding with the calculation of the final signature components.

The `message` parameter is treated as an hexadecimal string and converted into its byte representation. However, no check is enforced on `message` in the initialization phase. If the passed string is not a valid hexadecimal string, the implementation would abort the signing procedure with an error and the effort put in the first part of the protocol may be wasted.

Recommendation

The auditors recommend including a check that the `message` string contains hexadecimal characters only so that its decoding operation to a byte string can not go in error for such reason in the final steps of the signing protocol.

Alleviation

The team included the suggested checks on the correctness of the hexadecimal message.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/ecdsa/sign/party1.go#L45
https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/ecdsa/sign/party2.go#L42

POL-01 MISSING ERROR CHECK AND BOUNDARY CHECK IN FUNCTION `InitPolynomial`

Category	Severity	Location	Status
Coding Issue	Minor	crypto/vss/polynomial.go: 19~34	Resolved

Description

Inadequate Error Handling: The function `rand.Prime(rand.Reader, q.BitLen())` used in the `InitPolynomial` function is designed to return a randomly generated prime number and an error. The current implementation ignores this error, leading to a potential issue where the function could fail silently. If the random number generation fails for any reason, `r` may be `nil` or invalid, and this could lead to unexpected behavior or runtime errors when `r` is used later in the code.

Lack of Boundary Checks: The `InitPolynomial` function accepts a parameter degree which is used to create an array and also controls the flow of a loop. Currently, there is no check to ensure that degree is a non-negative value. If a negative value is passed as degree, it could lead to unexpected behavior such as an error in creating the array or an infinite loop.

Recommendation

First, the auditors suggest to implement error handling for the `rand.Prime` function. At the very least, check if the error is not `nil` and if so, return an error from `InitPolynomial`. This will ensure that any issues with the generation of the random prime number are caught and handled appropriately.

Second, please consider add a boundary check at the beginning of the function to verify that `degree` is a non-negative value. If `degree` is negative, the function should return an error or handle the situation in a way that is appropriate for the context in which it is used.

Alleviation

The `degree` parameter is validated and the random number generator error is returned at well.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/vss/polynomial.go#L22

SCN-01 DISCREPANCY BETWEEN IMPLEMENTATION AND SPECIFICATION IN SCHNORR PROOF ALGORITHM

Category	Severity	Location	Status
Coding Issue	Minor	crypto/schnorr/schnorr_proof.go: 24~26	Acknowledged

Description

In the Schnorr proof implementation, the computation of the challenge value `h` doesn't match with the specification.

According to the specification, the challenge c is computed as `H(G || V || A)`, where `H` is a hash function, `G` is the base point of the elliptic curve, `V = rG` is the random commitment, and `A` is Alice's public key.

However, in the implementation, the challenge `h` is computed as `crypto.SHA256Int(X.X, X.Y, R.X, R.Y)`, which is a hash of the `X` and `Y` coordinates of the public key `X` and the random commitment `R`. The base point `G` is not included in the computation.

This discrepancy between the implementation and the specification can lead to potential vulnerabilities or incorrect results when using the Schnorr proof. The implementation needs to be corrected to follow the specification.

Recommendation

To align the implementation with the Schnorr proof specification, the challenge `h` should be calculated according to the specification as `H(G || V || A)`.

The developers should consider modifying the code to include the base point `G` in the hash computation. The base point `G` could be represented as its coordinates `(G.X, G.Y)`.

The line in the Prove function should be updated from:

```
h := crypto.SHA256Int(X.X, X.Y, R.X, R.Y)
```

To:

```
Gx, Gy := X.Curve.Params().Gx, X.Curve.Params().Gy
h := crypto.SHA256Int(Gx, Gy, R.X, R.Y, X.X, X.Y)
```

Alleviation

Client stated that G is a constant point, and therefore does not impact the validity of proof.

SIG-01 | MISSING ROUND ENFORCEMENT IN ECDSA CONTEXTS

Category	Severity	Location	Status
Volatile Code	Minor	tss/ecdsa/sign/party1.go: 21~29; tss/ecdsa/sign/party2.go: 15~23	Acknowledged

Description

The cryptography algorithms implemented in the codebase in scope strictly follow their reference paper by wrapping each step or round of a party participating in the protocols in dedicated methods of a single struct. The context of each session of an algorithm is kept by storing information in the assigned method struct, while messages coming from the interacting counterparties are passed as methods parameters.

By following this pattern it is important that struct methods are executed according to the prescribed order so that the struct state is consistent with the algorithm progress in a certain session. Any wrong method call would invalidate the state, cause an information loss and force the algorithm session abort. In order to avoid such situation, the structs wrapping the context for Distributed Key Generation with Verified Secret Sharing, Key Reshare and Ed25519 signature include a `RoundNumber` field which explicitly track the algorithm progress. Then, all the methods implementing algorithm steps check if the current `RoundNumber` is the one supposed to be in the invoked round before executing the assigned logic and update the `RoundNumber`, as well.

However, such verification mechanism of the current round is absent in the ECDSA signature implementation based on Lidell17. In this way, the correctness of methods call order is completely delegated to the library caller, which may not be aware of such constraints or may unwillingly implement a faulty logic.

Recommendation

The auditors recommend including the verification of the current algorithm round in the ECDSA implementation, too, so that the library denies calls to the algorithm round logic in the wrong order.

Alleviation

Client stated the verification of algorithm rounds was implemented in upper level services.

TSK-01 MISMATCH ON CHAINCODE USAGE IN BIP-32 KEY DERIVATION

Category	Severity	Location	Status
Inconsistency	Minor	tss/key/bip32/tsskey.go: 86, 88	Acknowledged

Description

The BIP-32 specification document adopts HMAC-SHA512 for generating derived child data from parent information. Such algorithm provides data authentication through an hash function and a key. It takes two parameters, an authentication key and the data to authenticate.

The BIP-32 specification uses what it calls chaincode as authentication key, and the parent public key concatenated with the derivation index as data to authenticate.

However the derivation function implemented at the pointed location uses a constant hard-coded string as authentication key and prepends the chaincode to the data to authenticate. Even though such changes do not introduce security problems thanks to the hash function properties, the BIP-32 definitions are not respected, so the library can not claim to be compliant and interoperable with correct implementations of such standard.

Recommendation

The auditors recommend following the BIP-32 specification by using the chaincode as HMAC-SHA512 authentication key and removing the constant label from the child key calculation.

Alleviation

The client team decided to remain unchanged as the BIP-32 specification is not considered a strict requirement for them.

TSK-02 | MISSING VALIDATION ON CHILD KEY PAIR CALCULATION

Category	Severity	Location	Status
Inconsistency	Minor	tss/key/bip32/tsskey.go: 41, 47, 49	Resolved

Description

The BIP-32 specification provides a method to generate child key-pairs from a parent one through an hash function and exploiting the linearity of operations in the elliptic curve group.

When, given a chaincode, an index, and a parent key-pair, a new key is calculated, there are two outcomes that need to be validated before the procedure can be successfully concluded.

First, the integer resulting from the 32 least significant bytes of the HMAC-SHA512 outcome must be strictly less than the order of the elliptic curve (the `offset` variable). Second, the obtained public key (`ecPoint` variable) must not be the point-at-infinite of the curve. In either cases the procedure did not generate a valid child key pair. In particular, the BIP-32 document specifies that a new generation attempts should be made with the next value of the index.

Even though there is vary low probability that the describes conditions arise, the algorithm should manage them in order to cover all the possible cases.

Recommendation

The auditors recommend including the described checks in the pointed function to both correctly generate a usable key-pair and be compliant with the BIP-32 specification.

Alleviation

The client team added the suggested validation on the generated child key.

See https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/key/bip32/tsskey.go#L56

TSK-03 | MISSING HARDENED KEY DERIVATION IMPLEMENTATION

Category	Severity	Location	Status
Inconsistency	Minor	tss/key/bip32/tsskey.go: 41	Resolved

Description

The BIP-32 specification makes a distinction between the generation of non-hardened and hardened key-pairs.

The former are child key-pairs meant to work themselves as parent key-pairs for new derivations, while the latter do not allow for the derivation of a new public key from the parent public key.

Such distinction is made explicit by partitioning the index space into two intervals, respectively $[0; 2^{31} - 1]$ and $[2^{31}; 2^{32} - 1]$.

However, the derivation algorithm implemented in the `NewChildKey` does not account for the described distinction and all indexes are used for the non-hardened derivation.

Recommendation

The auditors recommend following the distinction between hardened and non-hardened derivation through indexes and, if the hardened method is not planned to be supported, we suggest including a check which rejects the generation of non-hardened key-pair using indexes assigned to hardened key-pairs.

Alleviation

The client team solved the issue validating that the requested child key index is not an hardened generation, which is not supported.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/tss/key/bip32/tsskey.go#L46

COI-01 | NON TIMING-CONSTANT INT VALUE COMPARISON

Category	Severity	Location	Status
Language Design Issue	Informational	crypto/commitment/commitment.go: 46~49	Acknowledged

Description

The `hash.Cmp(C) == 0` code used in commitment verification is not a constant comparison procedure which could potentially lead to timing side channel attack. Although the auditors did not notice it is exploitable at this moment but it is suggested to perform constant timing comparison.

Recommendation

Constant time arithmetic is currently not supported in Go int package, see proposal: math/big: support for constant-time arithmetic. We suggest the developers to keep an eye on the development of Go constant time comparison implementation for future improvement.

Alleviation

Client stated that this is a Go limitation and thus cannot be remediated at this moment.

COR-01 OUTDATED REFERENCE PAPER FOR PAILLER CORRECTNESS PROOF

Category	Severity	Location	Status
Inconsistency	Informational	crypto/paillier/correct_key_ni.go: 13	Resolved

Description

The reference paper reporting the parameters for the proof of correctness in the Pailler key-pair generation is outdated. In fact, it currently reports an update of the Lindell algorithm, made by Lindell et al., which gets rid of the Pailler homomorphic encryption scheme.

Recommendation

We recommend updating the reference paper to the correct version so that the codebase in scope has a clear pointer to the rationale behind the implementation.

Alleviation

Reference link updated.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/paillier/correct_key_ni.go#L13

CRY-03 | INCONSISTENT RANDOM NUMBER ERROR HANDLING

Category	Severity	Location	Status
Coding Style	Informational	crypto/paillier/paillier.go: 173~187; crypto/utils.go: 36~49	Resolved

Description

Random number generation plays a critical role in the threshold library. There are two main random number generation functions used in this library.

1. `getRandom` in `paillier.go` for generating a random number `r` which is `r < n` and `gcd(r,n) = 1`.

```
func getRandom(n *big.Int) (*big.Int, error) {
    gcd := new(big.Int)
    r := new(big.Int)
    var err error
    for gcd.Cmp(one) != 0 {
        r, err = rand.Int(rand.Reader, n)
        if err != nil {
            return nil, err
        }
        gcd = new(big.Int).GCD(nil, nil, r, n)
    }
    return r, nil
}
```

2. `RandomNum` in `utils.go` which generate a random number `r` which is `1 < r < n`.

```
func RandomNum(n *big.Int) *big.Int {
    if n == nil {
        panic(fmt.Errorf("RandomNum error, n is nil"))
    }
    for {
        r, err := rand.Int(rand.Reader, n)
        if err != nil {
            panic(fmt.Errorf("RandomNum error"))
        }
        if r.Cmp(one) == 1 {
            return r
        }
    }
}
```

There are several inconsistent coding issues presented in the above code.

1. `getRandom` does not check against `n == 0` where `RandomNum` does.
2. `getRandom` returns error message where `RandomNum` simply panics if are any random number generation issue.
3. Both function do not check if `n` is negative value.

Recommendation

The auditors suggest developers to unify the error handling scheme in function `RandomNum` and `getRandom`. Especially the `RandomNum` function should carefully handle and return error message to caller as a library, instead of panic.

Alleviation

The parameter validations are updated according to suggestions.

ECE-03 | DEPENDENCY IMPORT ORDER FORMAT

Category	Severity	Location	Status
Coding Style	Informational	crypto/commitment/commitment.go: 6; crypto/paillier/paillier.go: 6; crypto/schnorr/schnorr_proof.go: 4~7; crypto/vss/feldman.go: 4~7; crypto/zkp/dlnproof.go: 15~16; crypto/zkp/pdl_w_slack_proof.go: 18~21; tss/common.go: 8; tss/ecdsa/keygen/alice.go: 7; tss/ecdsa/keygen/bob.go: 12; tss/ecdsa/sign/party2.go: 12; tss/ed25519/sign/ed25519.go: 7; tss/ed25519/sign/round3.go: 14; tss/key/dkg/dkg_round.go: 9; tss/key/dkg/dkg_round1.go: 10; tss/key/dkg/dkg_round3.go: 13; tss/key/reshare/update_round.go: 10; tss/key/reshare/update_round1.go: 10; tss/key/reshare/update_round3.go: 12	Resolved

Description

The `math/big` library is always included at the end on the import list, while according to the `gofmt` formatting, it should be placed with the Go standard library before external dependencies. Also, external dependencies should go at the end of the import list.

Recommendation

We recommend complying with the `gofmt` formatting guidelines, including library imports in the correct order and running the `gofmt` tool on all the source files.

Alleviation

The client team changed import order of dependencies according to the `gofmt` style.

Commit `8f5867bb383539ef2ad8f32b991d61af7cee7a61` contains such changes.

GOE-01 | POTENTIAL VULNERABLE RUNTIME VERSION

Category	Severity	Location	Status
Language Version	Informational	go.mod: 1~4	Acknowledged

Description

The audited Go project is designated to run with version 1.17. However, Go language (before version 1.19.5) suffers an issue (GO-2023-1621) where the ScalarMult and ScalarBaseMult methods of the P256 Curve may return an incorrect result if called with some specific unreduced scalars. Although this does not directly impact the audited project for now, this may impact future version of the audited library.

Similarly, Go language runtime (version 1.17) also suffers an issue (GO-2023-1840) on Unix platform which could lead to information leakage or privilege escalation. Although the audited code is used as library instead of full binary, the users of this library may keep the same Go version thus auditor still consider this as potential impact.

Recommendation

Please consider use latest version of Go language if possible. Auditor did not find any hard dependencies on old version Go language features.

Alleviation

The client stated that they will upgrade the GO version in the future according to the internal policy.

KET-02 | PANIC USED INSTEAD OF ERROR MESSAGES

Category	Severity	Location	Status
Coding Style	Informational	tss/key/dkg/dkg_round.go: 29~34; tss/key/reshare/update_round.go: 31~37	Declined

Description

In function `NewRefresh` and `NewSetUp` of this library, the developers check parameter `total` to unsure the number of participated parties is no less than 2. However, the code simply panics if the constrain does not meet. As a library which can be used as part of code in other developers' project, the auditors suggest that use error message instead of panic to show that the passed parameters are inappropriate, which is consistent with the rest of the library coding style.

Recommendation

It is recommended to use error message instead of panic.

For instance, change the code style to:

```
func NewRefresh(deviceNumber, total int, devoteList [2]int, ShareI *big.Int,
PublicKey *curves.ECPoint) (*RefreshInfo, error) {
    if total < 2 || deviceNumber > total || deviceNumber <= 0 {
        return nil, fmt.Errorf("NewRefresh params error")
    }

    // Rest of your code...

    return refreshInfo, nil
}
```

Alleviation

The client stated that if this library is misused in the described manner, it is a manifestation of a deep misunderstanding and the code should panic.

PDL-01 | INVALID REFERENCE PAPER URL

Category	Severity	Location	Status
Invalid Reference	Informational	crypto/zkp/pdl_w_slack_proof.go: 10~11	Resolved

Description

The address of cited paper regarding `NewPDLwSlackProof` design is invalid. Accessing to `https://www.cs.unc.edu/~reiter/papers/2004/IJIS.pdf` is no longer valid.

Recommendation

Please consider update the link to referred paper to `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.2603`

Alleviation

Reference paper link has been updated.

See: https://github.com/okx/threshold-lib/blob/8f5867bb383539ef2ad8f32b991d61af7cee7a61/crypto/zkp/pdl_w_slack_proof.go#L15

RES-01 UNNECESSARY COMPUTATION OF RANDOM POLYNOMIAL IN NON-DEVOTEES

Category	Severity	Location	Status
Coding Issue	Informational	tss/key/reshare/update_round.go: 53~55; tss/key/reshare/update_round1.go: 18~37, 45~55; tss/key/reshare/update_round2.go: 33~37, 45~59	Acknowledged

Description

The aim of the key reshare process is to make a change in the set of shareholders while keeping the same global key-pair. This is achieved through the participation of two shareholders from the old set which, relying on their shares, act as dealers of new shares for the new set of shareholders. Such dealers are called `devotees` in the codebase while non-devotees simply verify the incoming information in order to assess that the process was concluded successfully.

In the current implementation, non-devotees generate a local secret, a random polynomial to share such secret, a Schnorr proof to demonstrate the knowledge of the secret and the verifiers of the polynomial coefficient. Such data are also forwarded in the messages of step 1 and 2. However, all these computations are not necessary since the only source of information is represented by the devotees, and only their messages will be actually verified and taken into account in step 3 when finalizing the reshare algorithm.

Recommendation

The auditors recommend saving the generation of the useless data reported in the description by restricting the generation of new polynomials along with their proof and verifiers only to the devotees, in charge to distribute the refreshed shares.

Alleviation

The client team decided to remain unchanged the current implementation as it does not affect the security of the algorithm.

OPTIMIZATIONS

ID	Title	Category	Severity	Status
CRY-02	Hard-Coded Source Of Randomness	Coding Style	Optimization	Acknowledged
UTL-01	Unnecessary Memory Allocation	Coding Issue	Optimization	Acknowledged

CRY-02 | HARD-CODED SOURCE OF RANDOMNESS

Category	Severity	Location	Status
Coding Style	Optimization	crypto/paillier/paillier.go: 179; crypto/utils.go: 42, 61	Acknowledged

Description

The generation of random numbers in a crypto safe way plays a paramount role in ensuring the security of the implemented cryptographic algorithms. Such task can be delegated to the underlying operating system which offer such service using a mixture of hardware and software mechanism to generate unpredictable and unique random values. In particular, the codebase in scope always relies on the standard `rand.Reader` utility which, calling the operating system, generates random strings of bytes.

However, there are some cases in which users of crypto wallets require strong guarantees of randomness for some reasons: the runtime environment may not be fully trusted, several source of randomness may be required, an external dedicated hardware for random number generation may be available, and so on.

Moreover, given the sensibility of the implemented algorithms, a common testing practice is to compare the implementation with the same algorithm executed in a different programming language. Since many algorithms rely on random numbers and results are not deterministic, in order to perform such tests, deterministic random number generators are put in place so the final results and intermediate outcomes can be compared.

Since the codebase in scope implements cryptographic algorithm built **on top** of a secure random number generator, the randomness security is not a responsibility of the library and hard-coding it to the default random byte generator of Go may be limiting for the library usage.

Recommendation

We recommend replacing the hard-coded `rand.Reader` usage with a generic `io.Reader` interface so that the source of randomness can be passed to the library as a parameter and the use case reported in the description can be realized.

Alleviation

The client team decided to leave unchanged the current implementation.

UTL-01 | UNNECESSARY MEMORY ALLOCATION

Category	Severity	Location	Status
Coding Issue	Optimization	tss/ed25519/sign/utils.go: 8	Acknowledged

Description

The `s` variable in the `bigIntToEncodedBytes` function is initialized as a reference to a 32 byte array through the `new` operator which also allocates memory for the specified type.

However, the memory allocated at variable initialization is practically unused since the a new `byte` array is allocated by the `copyBytes` function and the reference is replaced.

Recommendation

It is recommend initializing the `s` variable with the `copyBytes` outcome and allocating a new empty array only in the case of `a == nil`.

Alleviation

The client team decided to leave unchanged the current implementation.

APPENDIX

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Language Version	Language Version findings indicate that the code uses certain compiler versions or language features with known security issues.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.