Security assessments for software that matters.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

# Audit-Report any-sync Protocol Implementation & Trust Assumptions 11.-12.2025

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi, MSc. H. Moesl-Canaval, MSc. A. Schloegl, BSc. D. Prodinger, BSc. S. Possegger, BSc. C. Mayr, Dipl.-Ing. D. Gstir, BSc. L. Kofler

## Index

- Introduction
- Scope
- Identified Vulnerabilities
  - ANY-01-001 WP2: Panic-based DoS through invalid public keys (Critical)
  - ANY-01-002 WP2: ACL metadata keys not rotated on ownership transfer (Medium)
  - ANY-01-003 WP2: Network configurations have no integrity checks (Low)
  - ANY-01-005 WP2: Indefinite authentication due to non-expiring JWTs (Medium)
  - ANY-01-010 WP1: Private key leakage due to faulty invite process (High)
  - ANY-01-011 WP2: Potential side-channel on sync-nodes (Medium)
  - ANY-01-013 WP1: Potentially ineffective invite revocation (Medium)
  - ANY-01-014 WP2: Pre-authentication DoS via receiveQueues (Low)
  - ANY-01-015 WP2: Missing authorization in AclReadKeyChange handling (Critical)
- Miscellaneous Issues
  - ANY-01-004 WP2: Outdated and vulnerable dependencies (Low)
  - ANY-01-006 WP2: Potentially dangerous panic calls in any-sync (Low)
  - ANY-01-007 WP2: Potential panic in calcMedian (Low)
  - ANY-01-008 WP2: Dangerous GetRandomString utility function (Info)
  - ANY-01-009 WP2: Disjointed verification in content encryption logic (Info)
  - ANY-01-012 WP2: Usage of deprecated CID constants (Info)
  - ANY-01-016 WP2: Large invite file can crash clients upon joining (Info)
  - ANY-01-017 WP2: Lack of ACL AcceptorSignature trust anchor (Low)
- Conclusions

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## Introduction

*“any-sync is an open-source protocol designed for the post-cloud era, enabling high-speed, peer-to-peer synchronization of encrypted communication channels (spaces). It provides a communication layer for building private, decentralized applications offering unparalleled control, privacy, and performance.”*

From https://github.com/anyproto/any-sync

This report describes the results of a security assessment of the any-sync complex, specifically the any-sync protocol and trust assumptions, as well as the protocol’s implementation. The project, which included a review of the any-sync architecture and an audit of the any-sync cryptography, was conducted by Cure53 in November and December 2025.

The examination, registered as *ANY-01*, was requested by Open Impact Foundation PCC in October 2025 and then scheduled to start the following month. The project is the first external examination of the any-sync complex commissioned to Cure53.

In terms of the exact timeline and specific resources allocated to *ANY-01*, the Cure53 team has completed their research over several weeks, starting it in CW48 and concluding in CW50. In order to achieve the expected coverage of the components in scope, a total of thirty-two days were invested. A team consisting of nine senior testers was formed and assigned to the preparation, execution, documentation, and delivery of this project.

For optimal structuring and tracking of tasks, the assessment was divided into two separate work packages (WPs):

- **WP1**: Architecture reviews against the any-sync protocol & trust assumptions
- **WP2**: Cryptography audits & reviews against the any-sync protocol implementation

As the discussion of the formulated WPs indicates, the white-box methodology was used. Cure53 was provided with documentation, as well as all further means of access required to complete the tests. All sources corresponding to the test targets were shared to ensure that the project could be executed in accordance with the agreed framework.

The project could be carried out without any major issues. To facilitate a smooth transition into the testing phase, all preparations were completed in CW47. Throughout the engagement, communications were conducted through a private, dedicated, and shared Slack channel. Stakeholders - including Cure53 testers and the internal staff responsible for the any-sync components in scope - were able to participate in discussions in this space.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 did not need to ask many questions, and the quality of all project-related interactions was consistently excellent. In addition to frequent status updates on the examination and emerging findings, live-reporting was offered and executed for all issues with elevated severity scores, i.e., *High* or *Critical* problems. The noted Slack channel was used for this purpose.

Continuous communication contributed positively to the overall results of this project. Significant roadblocks were avoided thanks to clear and careful preparation of the scope, as well as through subsequent support.

The Cure53 team achieved good coverage of the WP1-W2 objectives. On the long list of the seventeen security-related discoveries, nine were classified as security vulnerabilities and eight were judged as general weaknesses with lower exploitation potential. Notably, two issues were ranked as *Critical* and a further one was scored *High* in terms of severity of their implications.

The protocol architecture was observed to rely on correctly chosen cryptographic primitives and a generally thoughtful approach to permission management within the Access Control List (ACL) mechanisms. However, significant design shortcomings were identified within the *invitation* logic, where the confidentiality of key material was placed at risk due to trust assumptions inherent in the invite strategy.

Furthermore, while the core cryptographic operations were well-implemented, the system boundaries were found to lack sufficient validation for incoming data, resulting in susceptibility to Denial-of-Service conditions through invalid public keys. These observations suggest that while the development team understands fundamental security requirements, edge cases regarding input validation and cryptographic handling have not been fully accounted for.

Critical gaps were discovered in the enforcement of authorization, where message authentication frequently relied on embedded public keys without verifying the sender's actual permissions against the ACL. Issues regarding code stability and structure were also noted, as validation logic was found to be fragmented across multiple functions, and uncontrolled panic calls posed risks to the application’s robustness.

Last but not least, the handling of ownership transfers was determined to operate under an incomplete threat model, specifically regarding the rotation of keys. Ultimately, the codebase was perceived as large and occasionally riddled with undefined responsibilities. This complicated the evaluation of the trust model due to a lack of explicit separation of concerns.

The following sections first describe the scope and key test parameters, as well as how the work packages were structured and organized.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Next, all findings are presented as vulnerabilities and as miscellaneous flaws. The problems are then discussed chronologically within each category. In addition to technical descriptions, PoC and mitigation advice is provided where applicable.

The report ends with general conclusions relevant to this winter 2025 project. Based on the test team's observations and the evidence collected, Cure53 elaborates on the overall impressions and reiterates the verdict. The final section also includes tailored hardening recommendations for the any-sync protocol and trust assumptions, as well as the protocol’s implementation.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## Scope

- **Audits & reviews against the any-sync protocol implementation & trust assumptions**
  - **WP1:** Architecture reviews against the any-sync protocol & trust assumptions
    - **URL:**
      - https://sync.any.org/
    - **Shared documents:**
      - *any-sync. updates. 2025.06.pdf*
      - *audit q&a.pdf*
      - *any. tech stack overview. 2024.08.pdf*
  - **WP2:** Cryptography audits & reviews against the any-sync protocol implementation
    - **Source code repository URLs & commits:**
      - https://github.com/anyproto/any-sync
        - *26f17f11ba2e0a04d5788749eedc3dbe48c02e43*
      - https://github.com/anyproto/any-sync-node
        - *cf6d47a9f0e3c70711c034d6498e753b1df21f85*
      - https://github.com/anyproto/any-sync-consensusnode
        - *64849a63e7bbb186a30b7d079678569d8eacb8b8*
      - *https://github.com/anyproto/any-sync-coordinator*
        - *777db15a39098c94129a994374363c472a9a4c33*
      - *https://github.com/anyproto/any-block*
        - *c993a7efb7d766576f852c2863f73b6cfc0cf9a6*
      - *https://github.com/anyproto/anytype-heart*
        - *b05365d3846dd6d285a60f0ab8eed0597b488314*
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Every ticket has been given a unique identifier (e.g., *ANY-01-001*) to facilitate any follow-up correspondence.

### ANY-01-001 WP2: Panic-based DoS through invalid public keys *(**Critical**)*

***Note**: this issue was communicated to the any-sync development team during the audit, and fixed in https://github.com/anyproto/any-sync/pull/581/files.*

It was observed that the *Ed25519PublicKeyToCurve25519* function panics when provided with an invalid Ed25519 public key. This is due to insufficient validation at unmarshaling time. The function converts Ed25519 public keys to Curve25519 format by calling *edwards25519.Point.SetBytes()*, which returns an error when the input bytes do not represent a valid point on the Edwards curve.

Rather than handling this error gracefully, the function panics, causing the entire process to crash. This vulnerability stems from the *UnmarshalEd25519PublicKey* function, which only validates the length of the input (32 bytes) but fails to verify that the bytes represent a valid curve point. As a result, any attacker who can inject arbitrary 32-byte values as public keys through network protocols can trivially crash any node processing those keys.

This issue affects numerous critical code paths throughout the codebase, including shared key derivation for one-to-one spaces, message encryption in the inbox system, ACL record processing, and peer identity validation. The vulnerability is easily exploitable through multiple entry points that accept public keys from external sources, such as ACL writer entries, inbox recipient keys, credential validation, and space payload processing.

By leveraging this attack, an attacker can cause Denial-of-Service issues on any node in the network, doing so by sending crafted messages containing invalid 32-byte public keys. An attacker can:

- crash nodes processing ACL records by inserting invalid public keys in one-to-one space *writer* entries,
- disrupt the inbox messaging system by sending messages with malformed recipient keys,
- crash peers during credential validation or identity verification,
- prevent space synchronization by injecting invalid keys into space payloads.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The attack requires no authentication and can be executed remotely through standard protocol operations. Multiple attack vectors exist across different protocol layers, making this vulnerability highly exploitable in daily usage context.

**Affected files:**
*any-sync/util/crypto/x25519.go*
*any-sync/util/crypto/ed25519.go*

**Affected code:**

```
func Ed25519PublicKeyToCurve25519(pk ed25519.PublicKey) []byte {
    // Unmarshalling public key into edwards curve point
    epk, err := (&edwards25519.Point{}).SetBytes(pk)
    if err != nil {
        panic(err)
    }
    return epk.BytesMontgomery()
}

[...]

func UnmarshalEd25519PublicKey(data []byte) (PubKey, error) {
    if len(data) != 32 {
        return nil, errors.New("expect ed25519 public key data size to be
32")
    }
    return NewEd25519PubKey(data), nil
}
```

The following call sites trigger the panic:

- *util/crypto/x25519.go:128* - *GenerateSharedKey* function used for one-to-one space key derivation
- *util/crypto/ed25519.go:142* - *Ed25519PrivKey.Decrypt* method initialization
- *util/crypto/ed25519.go:179* - *Ed25519PubKey.Encrypt* method initialization

An attacker can exploit this vulnerability through the one-to-one space ACL mechanism by completing the steps enumerated below.

**Steps to reproduce:**

1. Create a one-to-one space ACL root with the attacker's valid public key and an invalid 32-byte sequence as the second *writer.*
2. When a victim-node processes this ACL root, the code path flows through *commonspace/object/acl/list/onetoone.go:111-117*.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

3. The invalid bytes pass *UnmarshalEd25519PublicKeyProto* validation (only length is checked).
4. When *GenerateSharedKey* is called, it invokes *Ed25519PublicKeyToCurve25519* with the invalid key.
5. The *SetBytes* call fails, triggering the panic and crashing the node.

Additional attack vectors exist through the inbox messaging system (*coordinator/inboxclient/client.go:123*), ACL record builders (*commonspace/object/acl/list/aclrecordbuilder.go*), and peer credential validation (*net/secureservice/credential.go:101*).

To mitigate this issue, Cure53 advises implementing point validation in the *UnmarshalEd25519PublicKey* function. The result should be a rejection of invalid curve points before they enter the system. This will validate the cryptographic integrity of public keys at the earliest possible point, preventing invalid keys from propagating through the codebase. The recommended fix is to add a validation check that attempts to decode the bytes as an Edwards curve point and returns an error if the operation fails.

### ANY-01-002 WP2: ACL metadata keys not rotated on ownership transfer *(Medium)*

Testing of the ACL metadata protection mechanisms revealed that ownership transfers do not trigger rotation of the metadata encryption keys. When an owner transfers ownership of a space to another member, only the permission attributes are updated. Conversely, the cryptographic keys used to encrypt member metadata remain unchanged. This allows former owners to retain the ability to decrypt all members-related metadata that was encrypted during their ownership tenure.

The metadata key pair consists of a public key used to encrypt user information (such as names and icons) and a private key encrypted with the read key of the space. While the system correctly rotates both the read key and metadata key when members are removed via *AclAccountRemove*, the *AclOwnershipChange* operation lacks equivalent key rotation logic.

**Affected file:**
*commonspace/object/acl/list/aclrecordbuilder.go*

**Affected code:**

```
func (a *aclRecordBuilder) BuildOwnershipChange(ownershipChange
OwnershipChangePayload) (rawRecord *consensusproto.RawRecord, err error) {
    newOwnerBytes, err := ownershipChange.NewOwner.Marshall()
    if err != nil {
        return nil, err
    }
    content := &aclrecordproto.AclContentValue{
        Value: &aclrecordproto.AclContentValue_OwnershipChange{
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            &aclrecordproto.AclOwnershipChange{
                NewOwnerIdentity:    newOwnerBytes,
                OldOwnerPermissions:
aclrecordproto.AclUserPermissions(ownershipChange.OldOwnerPermissions)},
        },
    }
    return a.buildRecord(content)
}
```

**Affected file:**
*commonspace/object/acl/list/aclstate.go*

**Affected code:**

```
func (st *AclState) applyOwnershipChange(ch
*aclrecordproto.AclOwnershipChange, record *AclRecord) (err error) {
    err = st.contentValidator.ValidateOwnershipChange(ch, record.Identity)
    if err != nil {
        return err
    }
    newOwnerIdentity, err :=
st.keyStore.PubKeyFromProto(ch.NewOwnerIdentity)
    if err != nil {
        return err
    }
    var (
        oldOwnerKey = mapKeyFromPubKey(record.Identity)
        newOwnerKey = mapKeyFromPubKey(newOwnerIdentity)
    )
    st.updatePermissions(oldOwnerKey, ch.OldOwnerPermissions, record)
    st.updatePermissions(newOwnerKey,
aclrecordproto.AclUserPermissions_Owner, record)
    return nil
}
```

In contrast, the *buildReadKeyChange* function in *aclrecordbuilder.go* demonstrates the expected behavior of generating new metadata keys and encrypting the metadata private key with a new read key. This function is called during member removal but not during ownership transfer.

The attack scenario involves a space owner who transfers ownership to another member while retaining membership themselves (e.g., demoted to *admin* or *writer*). The former owner could have cached the metadata private key during their ownership period. Even after the ownership transfer, they retain cryptographic access to decryption of all member metadata encrypted with that key. Their reach includes metadata of members who joined after the ownership transfer but before any subsequent key rotation event.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 recommends extending the *AclOwnershipChange* message to include an optional *AclReadKeyChange* field. When ownership transfers occur, both the read key and metadata key should be rotated, similar to the behavior implemented for member removal. This ensures forward secrecy of metadata when administrative control over spaces is changed.

### ANY-01-003 WP2: Network configurations have no integrity checks *(Low)*

***Note:** this issue was communicated to the any-sync development team during the audit, and was later determined to be a false positive: a sufficient cryptographic check does indeed appear to occur in the code.*

Network configurations are fetched from coordinators without cryptographic verification of the response. A compromised *coordinator* can provide malicious node configurations, enabling complete network eclipse. Attackers controlling the *coordinator* could redirect all traffic through malicious nodes.

**Affected file:**
*coordinator/nodeconfsource/nodeconfsource.go*

**Affected code:**

```
func (n *nodeConfSource) GetLast(ctx context.Context, currentId string) (c
nodeconf.Configuration, err error) {
	res, err := n.cl.NetworkConfiguration(ctx, currentId)
	if err != nil {
		return
	}
	if res.ConfigurationId == currentId {
		err = nodeconf.ErrConfigurationNotChanged
		return
	}
	nodes := make([]nodeconf.Node, len(res.Nodes))
	for i, node := range res.Nodes {
		types := make([]nodeconf.NodeType, 0, len(node.Types))
		for _, nt := range node.Types {
			switch nt {
			case coordinatorproto.NodeType_FileAPI:
				types = append(types, nodeconf.NodeTypeFile)
			case coordinatorproto.NodeType_CoordinatorAPI:
				types = append(types,
nodeconf.NodeTypeCoordinator)
			case coordinatorproto.NodeType_TreeAPI:
				types = append(types, nodeconf.NodeTypeTree)
			case coordinatorproto.NodeType_ConsensusAPI:
				types = append(types, nodeconf.NodeTypeConsensus)
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                        case coordinatorproto.NodeType_NamingNodeAPI:
                                types = append(types,
nodeconf.NodeTypeNamingNode)
                        case coordinatorproto.NodeType_PaymentProcessingAPI:
                                types = append(types,
nodeconf.NodeTypePaymentProcessingNode)
                        }
                }
                nodes[i] = nodeconf.Node{
                        PeerId:    node.PeerId,
                        Addresses: node.Addresses,
                        Types:     types,
                }
        }

        return nodeconf.Configuration{
                Id:           res.ConfigurationId,
                NetworkId:    res.NetworkId,
                Nodes:        nodes,
                CreationTime: time.Unix(int64(res.CreationTimeUnix), 0),
        }, nil
}
```

To mitigate this issue, Cure53 advises adding cryptographic signatures on *coordinator* responses. The *coordinator*'s public key should be pinned or distributed through a secure channel.

## ANY-01-005 WP2: Indefinite authentication due to non-expiring JWTs *(Medium)*

While auditing the middleware in the *anytype-heart* repository, a JWT generation function was noticed for the fact that it does not set an expiry in the generated tokens. Some research into the used library showed that if this does not happen, the generated JWTs will actually not have an expiry date set.

The expiry is technically optional, yet it is crucial for the secure use of JWTs. Not setting it means that the generated tokens are accepted until the corresponding private key changes, causing signature issues. This would also mean that an attacker could authenticate to the server indefinitely after successfully capturing a single JWT.

**Affected location:**
*anytype-heart/core/session/service.go*

**Affected code:**

```
func generateToken(privKey []byte) (string, error) {
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{
        // "expiresAt": time.Now().Add(10 * time.Minute).Unix(),
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        "seed": randStringRunes(8),
    })

    // Sign and get the complete encoded token as a string using the secret
    return token.SignedString(privKey)
}
```

To mitigate this issue, Cure53 recommends stringently setting and checking expiry in all JWTs.

## ANY-01-010 WP1: Private key leakage due to faulty *invite* process *(High)*

While auditing the any-sync protocol and especially its *invite* process, it was noticed that the *inviter* will generate a public/private keypair and send the private key to the i*nvitee.* This immediately leaks the private key to the *inviter.* It also requires the *inviter* and *invitee* to have a pre-established secure channel which can be used to transfer the private key securely. If the *invite* message is logged or leaked, so is the private key of the *invitee.*

Additionally, it was communicated by the any-sync team that *invite* messages are synchronized between sync nodes, which would additionally leak the private key to all other members of the space. This would also allow space members with permissions lower than the *invite* to escalate their permissions by impersonating the *invitee* (e.g, upgrading read to write permissions). This leakage of private keys also makes removing members from spaces potentially difficult, as an attacker could use any leaked private key to access space content after they have been removed.

**Affected location:**
*anytype-heart/core/publish/service.go*

**Affected code:**

```
func (s *service) applyInviteLink(ctx context.Context, spaceId string,
snapshot *PublishingUberSnapshot, includeInviteLink bool) error {
    if !includeInviteLink {
        return nil
    }
    inviteInfo, err := s.inviteService.GetCurrent(ctx, spaceId)
    if err != nil && errors.Is(err, inviteservice.ErrInviteNotExists) {
        return nil
    }
    if err != nil {
        return err
    }
    snapshot.Meta.InviteLink =
fmt.Sprintf(s.limitsConfig.InviteLinkUrlTemplate, inviteInfo.InviteFileCid,
inviteInfo.InviteFileKey)
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
    return nil }
```

To mitigate this issue, Cure53 recommends switching to a two-step *invitation* process. Instead of generating a keypair and transmitting the private key, the *inviter* should generate a random one-time code and transmit that. The *invitee's* any-type application can then generate a keypair and transmit the public key together with the one-time code. After verifying the code, the *invitee'*s public key can then be added to the ACL with the correct permissions.

In order to prevent replicating similar privilege escalation issues, the one-time code must be kept private from other space members. This means that exchanging the code must happen synchronously with a node that can perform the ACL change.

### ANY-01-011 WP2: Potential side-channel on sync-nodes *(Medium)*

Auditing the authorization controls on sync-nodes revealed a potential side-channel leak about the timing and size of messages for a specific object. Specifically, when a user is connected to a peer sync node, there are no authorization controls in place that prohibit downloading encrypted object data, even when the user has no read privileges.

While only users with read privileges can decrypt this data, the adopted approach still leaks information about the timing and size of the object tree changes (in AES-256 block increments) to all other users on the same network. In the context of a text message, an attacker might be able to infer when a message was sent and the size of said message in a specific chatroom, despite not having access to this location.

Although similar leakage could occur via a Man-in-the-Middle attack on any encrypted connection, the lack of authorization on sync nodes makes it possible for an attacker to obtain this information directly. In other words, it would happen without intercepting the victim's connection, offering the ability to target specific data objects individually.

Cure53 was not able to reproduce this issue as it required implementing the network-stack for communicating with and authenticating to the sync nodes. However, the any-sync team confirmed that users are able to download the encrypted data of changes even when they have no read privileges, pointing to the existence of this side-channel issue.

**Affected files:**
*any-sync/commonspace/sync/objectsync/synchandler.go*

Cure53 recommends implementing object level authorization controls on the sync nodes. Information about the timing and size of object changes should remain confidential. In order to achieve this, the identity public key could be used in order to verify whether a user can download a specific object tree.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## ANY-01-013 WP1: Potentially ineffective *invite* revocation *(Medium)*

***Note**: this issue is known to the any-sync team. It is still documented for posterity.*

While auditing the any-sync protocol, specifically its *invite* mechanics, security shortcomings due to the strategy of generating the *invitee*'s private key have been documented (see ANY-01-010). This issue highlights another weakness with the *AnyoneCanJoin* invite-types. The problem is due to the lack of backward security in the read-key. Once an *invite* has been leaked, it would cause an irreversible break in the confidentiality of the space.

**The issue arises as follows:**

- *Invite* is created, ACL entry is pushed.
- *Invite* is shared.
- *Invite* is revoked, creating a new read key. However, if the *invitee* ever receives the sync state of the space at that time, they can decrypt the corresponding read key.

At its core, this issue arises from the fact that the read key has no backward secrecy (by design), combined with the fact that a private key is generated for the *invitee*. The multi-step process suggested as remediation in ANY-01-010 would also solve this issue.

## ANY-01-014 WP2: Pre-authentication DoS via *receiveQueue*s *(Low)*

While auditing the sync nodes, it was noticed that before any validation can take place, all requests are stored in a *receiveQueue* with limited capacity. If the queue is full, the request cannot be added and will not be handled.

This requires considering an attacker with sufficient network resources who could send enough packets to sync nodes. These are potentially publicly known/available via any-sync intent. As such, the receiving nodes could not be able to handle any benign traffic anymore, creating a DoS situation.

**Affected location:**
*any-sync/commonspace/sync/sync.go*

**Affected code:**

```
func (s *syncService) HandleMessage(ctx context.Context, msg drpc.Message)
error {
    idMsg, ok := msg.(syncdeps.Message)
    if !ok {
        return ErrUnexpectedMessage
    }
    objectId := idMsg.ObjectId()
    err := s.receiveQueue.Add(ctx, objectId, msgCtx{
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
            ctx:      ctx,
            Sizeable: idMsg,
        })
        if errors.Is(err, mb.ErrOverflowed) {
            return nil
        }
        return err
    }
```

To increase robustness of any-sync against potential censorship, Cure53 recommends adding some form of authentication to the gRPC endpoints. This would increase the difficulty of abuse for bad actors attempting to silence the any-sync spaces.

### ANY-01-015 WP2: Missing authorization in *AclReadKeyChange* handling (*Critical*)

***Fix note:*** *This issue was reported to the any-sync team during the audit. It was mitigated by adding an authorization check in the ValidateReadKeyChange() function to ensure the sender has either admin or owner permissions in the space. Cure53 verified the fix, confirming that the problem no longer exists.*

Due to the lack of proper authorization in the *AclReadKeyChange* message handling, any sender, regardless of whether they have any permissions in the space, can update the read key of a space. Sending the *AclReadKeyChange* message adds a new *AclKeys* record which includes a new read key encrypted for each participant, along with the previous read key encrypted with the new read key (*encryptedPreviousReadKey*).

The *encryptedPreviousReadKey* is included in this context, so that members who possess the current read key have access to all previous read keys by iterating through the *AclKeys* records. An attacker with no permissions in the space can abuse this behavior and append a new *AclKeys* record with a garbage value as *encryptedPreviousReadKey*.

This effectively makes it impossible for space members iterating the *AclKeys* records to properly decrypt the *encryptedPreviousReadKey* in the previous record. As a result, all space data preceding the invalid *AclKeys* record becomes impossible to decrypt. Since the any-sync protocol does not provide functionality to update or remove *AclKeys* records, decryption is only possible again after the bad record is manually removed or repaired.

Furthermore, the missing authorization also allows space members with only viewer or editor permissions to update the read key, which should not be possible according to the permission model of *any-sync*. The following code snippet shows that the *authorIdentity* handed in by the caller is neither checked nor handed on to *c.validateReadKeyChange*.

**Affected file:**
*any-sync/commonspace/object/acl/list/validator.go*

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

**Affected code:**

```
func (c *contentValidator) ValidateReadKeyChange(ch
*aclrecordproto.AclReadKeyChange, authorIdentity crypto.PubKey) (err error)
{
   if !c.verifier.ShouldValidate() {
       return nil
   }
   return c.validateReadKeyChange(ch, nil)
}
```

To mitigate this issue, Cure53 recommends ensuring only authorized members of the space are given the capacity to update the read key.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

### ANY-01-004 WP2: Outdated and vulnerable dependencies *(Low)*

While auditing the various repositories in scope, a number of outdated components with known vulnerabilities were identified. The following software packages were identified as out-of-date and potentially insecure. Notably, the version information provided is based on data collected at the time of testing. Whether these vulnerabilities are exploitable entirely depends on how the relevant functionality is used in the targeted application at present.

**Affected file:**
*any-sync-consensusnode/go.mod*

| Dependency | Vulnerability | Installed version | Fixed version |
|---|---|---|---|
| github.com/btcsuite/btcd | CVE-2022-44797<br>CVE-2024-38365<br>CVE-2024-34478 | 0.22.1 | 0.24.2 |
| github.com/quic-go/quic-go | CVE-2025-64702 | 0.56.0 | 0.57.0 |
| golang.org/x/crypto | CVE-2025-47914<br>CVE-2025-58181 | 0.44.0 | 0.45.0 |

**Affected file:**
*any-sync-coordinator/go.mod*

| Dependency | Vulnerability | Installed version | Fixed version |
|---|---|---|---|
| github.com/disintegration/imaging | CVE-2023-36308 | 1.6.2 | None |
| github.com/btcsuite/btcd | CVE-2022-44797<br>CVE-2024-38365<br>CVE-2024-34478 | 0.22.1 | 0.24.2 |
| github.com/quic-go/quic-go | CVE-2025-64702 | 0.56.0 | 0.57.0 |

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

| | | | |
|---|---|---|---|
| golang.org/x/crypto | CVE-2025-47914<br>CVE-2025-58181 | 0.44.0 | 0.45.0 |

**Affected file:**
*any-sync-node/go.mod*

| Dependency | Vulnerability | Installed version | Fixed version |
|---|---|---|---|
| github.com/<br>disintegration/imaging | CVE-2023-36308 | 1.6.2 | None |
| github.com/btcsuite/<br>btcd | CVE-2022-44797<br>CVE-2024-38365<br>CVE-2024-34478 | 0.22.1 | 0.24.2 |
| github.com/quic-go/<br>quic-go | CVE-2025-64702 | 0.56.0 | 0.57.0 |
| golang.org/x/crypto | CVE-2025-47914<br>CVE-2025-58181 | 0.44.0 | 0.45.0 |

**Affected file:**
*any-sync-tools/go.mod*

| Dependency | Vulnerability | Installed version | Fixed version |
|---|---|---|---|
| github.com/<br>disintegration/imaging | CVE-2023-36308 | 1.6.2 | None |
| github.com/btcsuite/<br>btcd | CVE-2022-44797<br>CVE-2024-38365<br>CVE-2024-34478 | 0.22.1 | 0.24.2 |
| github.com/quic-go/<br>quic-go | CVE-2025-64702 | 0.56.0 | 0.57.0 |
| golang.org/x/crypto | CVE-2025-47914<br>CVE-2025-58181 | 0.44.0 | 0.45.0 |

**Affected file:**
*any-sync/go.mod*

| Dependency | Vulnerability | Installed version | Fixed version |
|---|---|---|---|
| github.com/ | CVE-2023-36308 | 1.6.2 | None |

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

| | | | |
|---|---|---|---|
| disintegration/imaging | | | |
| github.com/btcsuite/btcd | CVE-2022-44797<br>CVE-2024-38365<br>CVE-2024-34478 | 0.22.1 | 0.24.2 |
| github.com/quic-go/quic-go | CVE-2025-64702 | 0.56.0 | 0.57.0 |
| golang.org/x/crypto | CVE-2025-47914<br>CVE-2025-58181 | 0.44.0 | 0.45.0 |

**Affected file:**
*anytype-heart/go.mod*

| Dependency | Vulnerability | Installed version | Fixed version |
|---|---|---|---|
| github.com/disintegration/imaging | CVE-2023-36308 | 1.6.2 | None |
| github.com/go-chi/chi | GHSA-vrw8-fxc6-2r93 | 5.2.1 | 5.2.2 |
| github.com/golang-jwt/jwt | CVE-2025-30204 | 3.2.2+incompatible | 5.3.0 |

Notably, the testing team was unable to comprehensively prove any potential impact during the limited time frame granted for this review. As such, the wider implications remain unknown at this point and should be subjected to internal research at the earliest possible convenience for the in-house team.

Generally speaking, the provision of robust supply chain security can be considerably challenging to provide to an optimal standard. Oftentimes, an easy or comprehensive solution cannot be offered, while the results and efficacy of the selected protection framework can vary depending on the integrated version of the deployed libraries.

To mitigate the existing issues as effectively as possible, Cure53 recommends upgrading all affected libraries and establishing a policy to ensure libraries remain up-to-date moving forward.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## ANY-01-006 WP2: Potentially dangerous *panic* calls in *any-sync* *(Low)*

While auditing the *any-sync* library for other potentially problematic usages of *panic*, some usages were discovered that are not directly problematic at the moment, but could lead to issues in the future. Adding locations that could *panic* in a library is inherently dangerous, and such an abstraction is not helpful in the views of the testers.

**Affected locations:**

- *any-sync/util/strkey/strkey.go:50*
- *any-sync/util/strkey/strkey.go:87*
- *source/any-sync/commonspace/object/acl/list/list.go:357*
- *source/any-sync/commonspace/object/acl/list/list.go:364*
- *source/any-sync/commonspace/object/acl/list/list.go:357*

**Affected code (example):**

```
func MustDecode(expected VersionByte, src string) []byte {
    d, err := Decode(expected, src)
    if err != nil {
        panic(err)
    }
    return d
}
```

A similar dangerous *panic* was observed in the handling of DRPC connections. The *ReleaseDrpcConn* function will panic if the passed connection does not implement the *Unblocked* function. However, no such checks are present in the corresponding *AcquireDrpcConn* function. Thus, it is easy for users of the *any-sync* library to misuse the API in such a way that the application will panic.

**Affected location:**
*source/any-sync/net/peer/peer.go*

**Affected code:**

```
func (p *peer) ReleaseDrpcConn(ctx context.Context, conn drpc.Conn) {
    var closed bool
    select {
    [...]
    default:
        if connCasted, ok := conn.(encoding.ConnUnblocked); ok {
            [...]
        } else {
            panic("conn does not implement Unblocked()")
        }
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 recommends not providing quick abstractions that could *panic*, especially not in a library. Unexpected panics completely sidestep any sane error handling process, and creating abstractions for this should be viewed as inherently dangerous. If such utilities are meant only to speed up writing tests, they should be moved to a corresponding module. Alternatively, they should be made visible only internally.

In the context of this scenario, Cure53 also advises checking any required preconditions as early as possible, i.e., ensuring that the passed connection implements the *Unblock* function during acquisition. This prevents potential footguns for users of the library.

### ANY-01-007 WP2: Potential panic in *calcMedian* *(Low)*

During the audit of the *any-sync-node* repository, a *calcMedian* function used for statistical calculations on storage objects was identified. Closer analysis showed that passing an empty slice to this function would trigger a *panic* due to an out-of-bounds access.

However, the affected code path is only reachable via the */stat/{spaceId}* HTTP debug endpoint, which is disabled by default. As a result, this issue is not exploitable in a standard installation, which explains its filing as a miscellaneous issue.

**Affected file:**
*any-sync-node/nodestorage/stat.go*

**Affected code:**

```
func calcMedian(sortedLengths []int) (median float64) {
	mid := len(sortedLengths) / 2
	if len(sortedLengths)%2 == 0 {
		median = float64(sortedLengths[mid-1]+sortedLengths[mid]) / 
2.0
	} else {
		median = float64(sortedLengths[mid])
	}
	return
}
```

Cure53 recommends addressing the out-of-bounds read as a defensive hardening measure. Although the condition is not reachable in a default deployment due to the debug endpoint being disabled, the underlying behavior can still lead to a runtime panic if the code is invoked under non-standard configurations or future changes.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## ANY-01-008 WP2: Dangerous *GetRandomString* utility function *(Info)*

While auditing the *any-sync* repository, a few usages of the non-cryptographic *math/rand* random generator were noticed. As most are used in scenarios that are not security sensitive, there is no visible security impact of this practice at present.

There is one function, listed below, that would be extremely easy to use insecurely, as it simply returns a random string of the desired length. It is not communicated that this generated string is not cryptographically random and should not be used for identifiers or similar items. The questionable functions are not used, but are again a case of what seems like a leftover testing utility code.

**Affected files:**
*any-sync/util/slice/slice.go*
*anytype-heart/util/slice/slice.go*

**Affected code (identical code):**

```
import (
    "hash/fnv"
    "math/rand"
[...]

func GetRandomString(s []string, seed string) string {
    rand.Seed(int64(hash(seed)))
    return s[rand.Intn(len(s))]
}
```

Cure53 recommends not providing easy-to-misuse utility functions. While such functions are useful for testing, they should not be exported if possible. Similarly, they should be clearly highlighted. As the functions are not used, a simple removal is suggested.

## ANY-01-009 WP2: Disjointed verification in content encryption logic *(Info)*

While auditing the content encryption mechanism in the any-type code, it was found that code level assumptions in varying levels of abstraction might lead to issues after future changes. Specifically, the payload set by the *prepareBuilderContent* method includes the *ReadKey* based on the *ShouldBeEncrypted* parameter.

Subsequently, the *Build* method for the *changeBuilder* type encrypts the message only based on the existence of the *ReadKey* parameter. If said parameter is *nil*, no error is thrown, even though *ShouldBeEncrypted* might be set to *true*.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This finding is set as *Info*-only, because a *nil* check for the *readKey* in the *prepareBuilderContent* method prevents this issue from being exploited. However, should this code ever be refactored, it is not clear from the function context that this single check certainly prevents complete breakdown of confidentiality for any-sync clients.

**Affected file:**
*source/any-sync/commonspace/object/tree/objecttree/changebuilder.go*

**Affected code:**

```
func (c *changeBuilder) Build(payload BuilderContent) (ch *Change,
rawIdChange *treechangeproto.RawTreeChangeWithId, err error) {
[...]
if payload.ReadKey != nil {
        var encrypted []byte
        encrypted, err = payload.ReadKey.Encrypt(payload.Content)
        if err != nil {
            return
        }
        change.ChangesData = encrypted
    } else {
        change.ChangesData = payload.Content
    }
```

Implicit assumptions in the code - such as *ReadKey=nil → no encryption* - should be documented explicitly to prevent future changes from introducing unintended consequences. Additionally, the single-responsibility principle could be applied with respect to security concerns, so that critical encryption mechanisms can be easily audited and modified within the codebase. In this case, this would necessitate that the *ShouldBeEncrypted* parameter is checked when it is decided whether a specific content is to be encrypted or not, with a subsequent *nil* check of the *readKey* parameter inside of the *changeBuilder*.

## ANY-01-012 WP2: Usage of deprecated CID constants *(Info)*

Static analysis of the any-sync codebase revealed extensive use of Content Identifiers (CIDs) within internal components. CIDs are self-describing, content-addressed identifiers commonly used in distributed systems.

The analyzed implementation relies on the *go-cid* package for CID creation and validation. Closer review showed that the codebase uses string constants that are now deprecated in *go-cid* and flagged as part of a breaking change. Upstream guidance indicates that these constants should be replaced with definitions from the *go-multicodec* package instead, as documented in the referenced change[1].

[1] https://github.com/ipfs/go-cid/pull/137

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

**Affected file:**
*any-sync/util/cidutil/cid.go*

This finding does not currently introduce a direct security risk. However, continued reliance on deprecated constants increases the likelihood of subtle incompatibilities which are hard to detect. Migrating to *go-multicodec* in a timely manner would reduce technical debt, align the implementation with the current CID specification and lower the risk of regressions or rushed fixes when *go-cid* removes backward compatibility.

## ANY-01-016 WP2: Large *invite* file can crash clients upon joining *(Info)*

Reviewing the flow for joining a space via an *invite* link revealed that the *invite* is stored as a Protobuf-encoded file blob on an any-sync file-node. When joining, the client will fully load this file into memory for parsing. A malicious actor can upload a huge file blob for this *invite* file. This can cause the client to run out of memory and consequently crash when it tries to join the space.

The impact of this flaw is rather limited, as it will only affect the client when joining a space via an *invite* link. The following code snippet shows the problematic *io.ReadAll* call which will load the whole file into memory.

**Affected file:**
*anytype-heart/core/invitestore/invitestore.go*

**Affected code:**

```
func (s *service) GetInvite(ctx context.Context, id cid.Cid, key
crypto.SymKey) (*model.Invite, error) {
	rd, err := s.commonFile.GetFile(ctx, id)
	if err != nil {
		return nil, fmt.Errorf("get data from IPFS: %w", err)
	}
	defer rd.Close()

	data, err := io.ReadAll(rd)
	if err != nil {
		return nil, fmt.Errorf("read data: %w", err)
	}

	data, err = key.Decrypt(data)
	if err != nil {
		return nil, fmt.Errorf("decrypt data: %w", err)
	}
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 suggests hardening the client by limiting the file size an *invite* can have when it is downloaded. In general, the use of *io.ReadAll* from potentially untrusted sources should always be limited in size since it can easily be abused to fill up the caller's memory and crash the application.

### ANY-01-017 WP2: Lack of ACL *AcceptorSignature* trust anchor *(Low)*

Analysis of the ACL handling revealed that each ACL record carries two signatures: One from the identity which created the record and another from the *acceptor* which accepted this ACL record, namely the *consensus* node. When an ACL record is verified, both signatures are verified. In case of the *acceptor*'s signature, there is no direct trust established in the signing key. Therefore, when verifying an ACL record, it is not possible to determine if the signature was produced by a trusted identity or some malicious actor.

As shown below, the *UnmarshallWithId* method takes the raw record and verifies the *acceptor's* signature using *VerifyAcceptor*. The respective public key for this is taken from the ACL record itself. Therefore, this code will trust any public key, as long as it is a valid Ed25519 public key which successfully verifies the signature.

Note that the impact of this flaw is limited as the *consensus* node - which verifies this - will only accept RPC calls from the *coordinator* nodes. Thus, only a malicious *coordinator* node is in the position to inject rogue public key signatures in this context.

**Affected file:**
*any-sync/commonspace/object/acl/list/aclrecordbuilder.go*

**Affected code:**

```
func (a *aclRecordBuilder) UnmarshallWithId(rawIdRecord
*consensusproto.RawRecordWithId) (rec *AclRecord, err error) {
	[...]
		err = a.verifier.VerifyAcceptor(rawRec)
		if err != nil {
			return
		}
		aclRecord := &consensusproto.Record{}
		err = aclRecord.UnmarshalVT(rawRec.Payload)
		if err != nil {
			return
		}
		pubKey, err = a.keyStorage.PubKeyFromProto(aclRecord.Identity)
		if err != nil {
			return
		}
```

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

**Affected file:**
*any-sync/commonspace/object/acl/recordverifier/recordverifier.go*

**Affected code:**

```
func (r *recordVerifier) VerifyAcceptor(rec *consensusproto.RawRecord) (err
error) {
	identity, err := r.store.PubKeyFromProto(rec.AcceptorIdentity)
	if err != nil {
		identity, err =
crypto.UnmarshalEd25519PublicKey(rec.AcceptorIdentity)
		if err != nil {
			return fmt.Errorf("failed to get acceptor identity:
%w", err)
		}
	}
	verified, err := identity.Verify(rec.Payload, rec.AcceptorSignature)
	if !verified || err != nil {
		return fmt.Errorf("failed to verify acceptor: %w", err)
	}
	return nil
}
```

To harden the application, it is suggested to add a check which ensures that the *acceptor*'s public key matches the sender's (*consensus* node) public key. This will ensure that only ACL records signed by the sending *consensus* node are accepted.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

## Conclusions

This examination, which marks the first security audit of the any-sync protocol and infrastructure conducted by Cure53, concludes with a list of seventeen findings, two of them *Critical*. Based on the issues spotted during *ANY-01,* the testing team formulates a variety of recommendations for targeted hardening of any-sync components.

To reiterate, the work has been separated into two work packages, with WP1 covering the protocol and its trust assumptions, and WP2 concerning the implementation and codebase in general. The codebase mainly spans Go code, spread over a number of repositories. This audit mainly focused on the *any-sync* repository, but the testers were told to examine other supporting repositories as necessary. Overall, testers examined the following repositories for WP2:

- *https://github.com/anyproto/any-sync*
  - Commit: *26f17f11ba2e0a04d5788749eedc3dbe48c02e43*
- *https://github.com/anyproto/any-sync-node*
  - Commit: *cf6d47a9f0e3c70711c034d6498e753b1df21f85*
- *https://github.com/anyproto/any-sync-consensusnode*
  - Commit: *64849a63e7bbb186a30b7d079678569d8eacb8b8*
- *https://github.com/anyproto/any-sync-coordinator*
  - Commit: *777db15a39098c94129a994374363c472a9a4c33*
- *https://github.com/anyproto/anytype-heart*
  - Commit: *b05365d3846dd6d285a60f0ab8eed0597b488314*

Further discussing the available material for testing, the any-sync team also supplied the relevant documents, including pdf files titled *any-sync. updates. 2025.06.pdf, audit q&a .pdf,* and *any. tech stack overview. 2024.08.pdf.*

Moving on to final key aspects of this *ANY-01* project, it is important to shed some light on the summary of trust assumptions that the any-sync protocol shall follow and that, consequently, Cure53 was working with and integrated into the testing process.

In specific terms, any-sync anchors trust the PKI established for all members of a space. In this context, private keys are used to securely distribute the shared read key, as well as derive signing keys for verifying authenticity. Correspondingly, this clearly means a trust assumption to securely handle the shared key is placed on all members in the ACL. As the key is not backward-secret (new read-keys encrypt the old one), leakage of the read key would leak the entire state of the space up to this point.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It must be recognized that content of the space is encrypted with the read-key before synchronization, thus no trust is placed on the sync nodes. However, the fact that retrieving a space's state can be done without authentication can lead to potential security issues if a read-key is leaked. This has also been picked up in relation to ANY-01-010 and ANY-01-013).

The current invitation process places immense trust assumptions on the *invitee* and other space members. In simple terms, it is expected that they adhere to the protocol and do not misuse the *invitee*'s new private key (again, see ANY-01-010). The mitigation of the issue recommends a more robust *invite* strategy.

Moreover, ACL state-changes - on the protocol design - are sound if trust in the PKI is justified, i.e., no leaks of private keys have happened. Synchronization messages have their signatures verified and are integrity-checked using SHA 256 before head updates are applied. If all clients then use the same algorithm to order DAG head updates, the state will be consistent between all clients. Failure to adhere to the protocol for head updates will result in the update being discarded.

These assumptions have been considered in relation to the nine exploitable vulnerabilities confirmed during this November-December 2025 project. As already noted, one issue was ranked as *High* and two were judged as *Critical.* Besides these crucial problems, eight further miscellaneous issues were documented. Looking at the entire scope, the Cure53 must still argue for a verdict that leans positively. However, the size of the complex leaves ample room for bad implementations and misconfiguration. This is exhibited by the number of vulnerabilities identified in this report.

The cryptography audit performed by Cure53 against the any-sync protocol implementation focused on the core cryptographic operations, ACL mechanisms, P2P networking security, and *consensus* client behavior. Testing concentrated exclusively on the any-sync repository, examining how the protocol handles key material, validates cryptographic operations, and enforces access control across its distributed architecture.

Significant shortcomings were observed in the *invite* pattern. All current *invite* strategies send some form of key in the *invite*, which places immense trust assumptions on the confidentiality of the *invite*, which is vital in relation to the mentioned trust assumptions. The fact that the message in question might also be synced within the space would leak the private key of all new members. ANY-01-010 and ANY-01-013 detail these issues and suggest a remediation strategy.

Another important observation is that while the cryptographic primitives themselves are correctly chosen and generally well-implemented - AES-256-GCM for symmetric encryption, Ed25519 for signatures, X25519 for key agreement - the boundaries where cryptographic data enters the system lack sufficient validation.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This pattern is evidenced by ANY-01-001, where invalid Ed25519 public keys can crash nodes due to missing curve point validation at unmarshaling time. The fact that only length checks were implemented suggests that the development team understood the need for input validation but did not fully consider cryptographic edge cases.

The ACL implementation demonstrates a thoughtful approach to permission management and state atomicity. The use of deep copies during record application and the structured permission enforcement via the *contentValidator* interface indicate security-conscious design decisions from a cryptographic perspective.

At the same time, the ownership transfer mechanism reveals an incomplete threat model regarding key rotation, as documented in ANY-01-002. While the system correctly rotates metadata keys when members are removed, the same logic is not applied to ownership changes. The observed asymmetry suggests that the key rotation behavior was implemented reactively for the member removal case rather than as a comprehensive policy applied uniformly across all privilege-changing operations.

The P2P networking layer implements TLS-based secure channels correctly through the *secureservice* component. The authentication flow for inbound connections follows expected patterns. However, the *coordinator* integration lacks cryptographic integrity verification for network configurations, as noted in ANY-01-003. While this represents a lower-severity issue due to the trust already placed in *coordinators*, it is a deviation from defense-in-depth principles that could become problematic if the threat model evolves or if self-hosted deployments become more common.

The *consensus* client implementation handles peer selection and response validation adequately for the current trust assumptions. The quorum verification logic follows the expected patterns for the *consensus* protocol in use. No issues were identified in the Protobuf message handling for the areas examined - the protocol message structures for ACL records, *consensus* operations, and *coordinator* communications appear well-defined and seem to greatly benefit appropriate field validation.

However, during the review of the ACL message handling, it was observed that message authentication frequently relies on a public key embedded directly within the signed payload. While this mechanism effectively verifies data integrity and sender identity, the check alone fails to enforce authorization. Therefore, it is important to also implement authorization at a later stage of message processing. One issue was identified where such an authorization check was not present. This case made it possible for any sender, regardless of whether they had permissions to update the read key of the space (see ANY-01-015).

The mnemonic generation and key derivation code follows standard BIP39/BIP32 patterns correctly. Key material handling could benefit from more explicit memory management practices - while Go's garbage collection presents inherent challenges for cryptographic memory hygiene, the codebase makes no attempt to zero sensitive material after use.

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This is a common pattern in Go cryptography code, seen as an accepted tradeoff rather than a specific vulnerability.

Signature verification was also carefully reviewed. It was found that signatures are often verified with public keys that are part of the message to be verified. It is, therefore, often necessary to add additional checks which ensure that the public key is actually trusted. Most of these were found to be in place, however one instance where this is not the case was documented in ANY-01-017.

This area could be improved by not transmitting the public key in the same message, but instead requiring each verifier to build a list of trusted entities from the ACL records. This would then require the code to always use this list for looking up the respective public key and building the list would be the only place where trust in these public keys needs to be established.

During the assessment, the Go code across the reviewed repositories was also examined for unsafe or improper language constructs that could negatively impact stability and security. It was found that certain use-cases, such as uncontrolled panic calls, were present. These could lead to unexpected application termination or Denial-of-Service conditions if triggered under specific circumstances. These observations were documented in ANY-01-006, which indicates areas where defensive error handling could be improved to increase overall robustness.

In addition, the implementation was reviewed from a dependency management and supply-chain perspective. As part of this effort, the Go modules in use were analyzed for outdated or vulnerable third-party components that could introduce inherited security weaknesses. It was found that at least one dependency issue could expose the application to avoidable risk if left unaddressed, particularly in scenarios involving transitive dependencies. This issue was captured in finding ANY-01-004 and highlighted the need for more rigorous dependency lifecycle management.

The application logic was assessed with a focus on runtime behavior and error conditions. It was found that a specific code path could result in a crash when executed in debug environments, potentially impacting developer workflows and test stability. While the condition did not directly affect production deployments, it demonstrated insufficient defensive checks around debug-specific logic, as recorded in ANY-01-007.

The user data that is hosted on sync-nodes is encrypted on the object tree level. Only authorized users can read this data because only their private keys decrypt the data to plaintext. However, there are no other authorization controls implemented on the sync nodes beyond these. This means that any user is able to download the object tree data of other users, which leaks the time and size of the changes to the sync tree (see ANY-01-011).

**Dr.-Ing. Mario Heiderich, Cure53**
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Throughout the code audit, it was also found that validation logic is fragmented around multiple functions and files. For example, the signature verification of the tree changes is verified in the utility function for unmarshalling the data, while it is not checked in the actual function for verifying the tree.

If a change is introduced through other ways or the unmarshalling function is refactored, the tree builder might accept changes where the signature does not match. Another case in this realm is documented in ANY-01-009 where implicit code level assumptions might lead to problems at a later stage in the development lifecycle.

Overall, the codebase is large and responsibilities of individual functions are not always clearly defined. This lack of explicit separation of concerns made it more time-consuming to evaluate each function's role within the overall trust model. Nevertheless, this *ANY-01* audit reached comprehensive coverage of the any-sync protocol and infrastructure. It concludes that the tested project is reasonably well thought-out, but its implementation is lacking in certain areas. Planning ahead, Cure53 recommends recurring security assessments of the infrastructure and nodes, as well as regular audits of the client applications.

Cure53 would like to thank Sergey Cherepanov, Alex Danilin, Roman Khafizianov, Sergey Malenkov and Anatolii Smolianinov from the Open Impact Foundation PCC team for their excellent project coordination, support and assistance, both before and during this assignment.