Subspace Baseline Security Assurance

Threat model and hacking assessment report

V1.0, 28 June 2023

Bruno Produit	bruno@srlabs.de
Mostafa Sattari	mostafa@srlabs.de
Regina Bíró	regina@srlabs.de
Jakob Lell	jakob@srlabs.de

Abstract. This work describes the result of the thorough and independent security assurance audit of the Subspace platform performed by Security Research Labs. Security Research Labs is a consulting firm that has been providing specialized audit services for Substrate-based blockchains since 2019, including in the Polkadot ecosystem.

During this study, Subspace provided access to relevant documentation and supported the research team effectively. The code of Subspace was verified to assure that the business logic of the product is resilient to hacking and abuse.

The research team identified several issues ranging from critical severity to moderate, many of which concerned the execution layer.

In addition to mitigating the remaining open issues, Security Research Labs recommends conducting another assessment before the launch of the network, when all the security measures have been implemented.

Content

1 Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the agreed-on timeframe and scope as detailed in Chapter 2. Please note that this report does not guarantee that all existing security vulnerabilities were discovered in the codebase exhaustively and that following all evolution suggestions described in Chapter 7 may not ensure all future code to be bug free.

2 Motivation and scope

Subspace is a blockchain network built on top of Substrate, a framework for developing blockchains. The core business logic of Subspace consists of a proof-of archival storage blockchain, which incentivizes farmers to store a backup of other blockchains. Furthermore, it implements a namespaced execution layer which allows other distributed blockchains to implement custom business logic, similar to Polkadot's parachain ecosystem, on top of the Subspace domain execution layer.

Like other Substrate-based blockchain networks, the Subspace code is written in Rust, a memory safe programming language. Substrate-based chains utilize three main technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, and a block production engine.

Because blockchains evolve in a trustless and decentralized environment, security issues may arise from these properties. Therefore, ensuring availability and integrity is a priority for Subspace as it depends on its users to be incentivised to participate in the network. Furthermore, the Subspace network relies on a novel proof of archival storage consensus, which relies on availability and integrity of the stored data as its core functionality. As such, a security review of the project should not only highlight the security issues uncovered during the audit process, but also bring additional insights from an attacker's perspective, which the Subspace team can then integrate into their own threat modeling and development process to enhance the security of the product.

Two main workstreams were identified and split into two projects: first, the review of the new consensus protocol design, and second, the baseline assurance.

During the audit, findings were shared via the main Subspace repository, marked with the "Audit" tag [1]. Security Research Labs used a private slack channel for asynchronous communication and status updates – in addition, bi-weekly jour fixe meetings were held to provide detailed updates and address open questions.

2.1 Consensus whitepaper review

The first part of the audit entailed reviewing Subspace's new consensus protocol design. Starting with the whitepaper and documentation [2], Security Research Labs assessed the possibility of threats and attacks around the consensus protocol.

Subspace decided to move away from the consensus protocol v1 to remediate the incomplete ASIC resistance vulnerability identified during the 2022 audit in the encoding function that allowed a compression attack. Additionally, the four metrics taken by Subspace to make a good consensus protocol, namely initialization time, proof-generation time, proof size and verification time were not satisfied by the consensus protocol v1. As a result, the consensus protocol v2 reviewed in this audit does not have the same logic as the v1 that was reviewed by Security Research Labs in 2022.

As guidance for the v2 consensus protocol review, the consensus implementation code in Subspace [3] was used.

2.2 Baseline assurance

The first audit of Subspace was conducted by Security Research Labs in July 2022. The main focus of this assessment was the farmer logic and the proof-of-storage algorithm implemented by Subspace. Several issues were reported to the developers, most of which were already mitigated [4].

In this current engagement, the code assurance team focused on the namespaced execution layer and networking, as well as the consensus design as requested by the Subspace team.

The baseline assurance audit was split into two phases. In the first phase, the Security Research Labs team audited the existing namespaced execution and networking layers which were ready to be reviewed. After that, we reviewed the newly implemented components as part of pull requests and specific requests by the development team, as well components such as the consensus implementation.

2.2.1 First phase: Initial assessment

Security Research Labs collaborated with the Subspace team to create an overview containing the runtime modules in scope and their audit priority [5]. The in-scope components and their assigned priorities are reflected in Table 1. During the audit, Security Research Labs used a CIA threat model to guide efforts on exploring potential security flaws and realistic attack scenarios.

During the initial assessment of the ready-for-review parts of the code base, mainly the namespaced execution layer, security critical parts of the code were identified and potential security issues in these components were communicated to the Subspace development team in the form of public GitHub issues [6].

Repository	Priority	Component(s)	Reference
subspace	High	crates/pallet-domain crates/pallet-settlement domains/client/block-builder domains/client/domain-executor domains/pallets/domain-registry domains/pallets/executive domains/pallets/executor-registry	[3]
	Medium	crates/subspace-service domains/client/executor-gossip	

Table 1. In-scope Subspace components with audit priority

2.2.2 Second phase: Continuous review

In the second phase Security Research Labs focused on reviewing pull requests and specific parts of the code introducing new functionalities or refactoring existing features. The following table specifies the components that were reviewed during this phase.

Repository	Priority	Component(s)	Reference
subspace	High	crates/pallet-settlement domains/client/domain-executor	[3]

	crates/subspace-fraud-proof domains/pallets/messenger domains/pallets/transporter
Medium	domain/client/domain-client-relayer domain/client/cross-domain-message-gossip crates/subspace-verification crates/subspace-core-primitives

3 Methodology

This report details the baseline security assurance results for the Subspace network with the aim of creating transparency in four steps: treat modeling, security design coverage checks, implementation baseline check and finally remediation support:

Threat Modeling. The threat model is considered in terms of *hacking incentives*, i.e., the motivations to achieve the goals of breaching the integrity, confidentiality, or availability of Subspace network nodes. For each hacking incentive, hacking *scenarios* were postulated, by which these goals could be achieved. The threat model provides guidance for the design, implementation, and security testing of Subspace.

Security design coverage check. Next, the Subspace design was reviewed for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

a. **Coverage**. Is each potential security vulnerability sufficiently covered?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

Implementation baseline check. As a third step, the current Subspace implementation was tested for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the Subspace codebase, Security Research Labs derived the code review strategy based on the threat model that was established as the first step [7] . For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 04.

Prioritizing by risk, the codebase was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, the auditors:

1. Identified the relevant parts of the codebase, for example the relevant crates and the runtime configuration.

2. Identified viable strategies for the code review. Manual code review, fuzz testing, and tests via static analysis tools were performed where appropriate.

3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks, or otherwise ensured that sufficient protection measures against specific attacks were present.

4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations.

Security Research Labs carried out a hybrid strategy utilizing a combination of code review and dynamic tests (e.g., fuzz testing) to assess the security of the Subspace codebase.

While fuzz testing and dynamic tests establish a baseline assurance, the focus of this audit was a manual code review of the Subspace codebase to identify logic bugs, design flaws, and best practice deviations. The approach of the review was to trace the intended functionality of the runtime modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior on the blockchain due to logic bugs or missing checks. Since the Subspace codebase is entirely open source, it is realistic that a malicious actor would analyze the source code while preparing an attack.

Fuzz testing is a technique to identify issues in code, that handles untrusted input, which in Subspace's case is extrinsics in the main Subspace runtime as well as extrinsics in the domain runtimes. (Note that the network part is handled by Substrate, which was not in scope for this review, but is built with a strong emphasis on security and where fuzz testing is also used). Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process, fuzz testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. The fuzz testing methods written for this assessment utilized the test runtime genesis configuration as well as mocked externalities to execute the fuzz test effectively against the extrinsics in scope.

Remediation support. The final step is supporting Subspace with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation solution is implemented, the fix is verified by the auditors to ensure that it mitigates the issue and does not introduce other bugs.

4 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in Subspace’s blockchain system. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as the security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios. The *Hacking Value*, categorized into *low*, *medium*, and *high*, considers the incentive of an attacker, as well as the effort required by an attacker to successfully execute the attack. The hacking value is calculated as:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While *incentive* describes what an attacker might gain from performing an attack successfully, *effort* estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat.
- Medium: Attacks offer the hacker considerable gains from executing the threat.
- High: Attacks offer the hacker high gains by executing this threat.

Effort:

- Low: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources.
- Medium: Attacks are moderately difficult to execute. They might require bypassing countermeasures, the use of expensive resources or a considerable amount of technical knowledge.
- High: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors.

Incentive and Effort are divided according to Table 2.

Hacking Value	Low incentive	Medium Incentive	High Incentive
High effort	Low	Medium	Medium
Medium effort	Medium	Medium	High
Low effort	Medium	High	High

Table 2. Hacking value measurement scale.

Hacking scenarios are classified by the risk they pose to the system. The risk level, also categorized into low, medium, and high, considers the hacking value, as well as the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

$$Risk = Damage \times Hacking\ Value = \frac{Damage \times Incentive}{Effort}$$

Damage describes the negative impact that a given attack, performed successfully, would have on the victim. The degrees of damage are defined as follows:

Damage:

- Low: Risk scenarios would cause negligible damage to the Subspace network
- Medium: Risk scenarios pose a considerable threat to Subspace's functionality as a network.

- High: Risk scenarios pose an existential threat to Subspace's network functionality.

Damage and Hacking Value are divided according to Table 3.

Risk	Low hacking value	Medium hacking	High hacking
Low damage	Low	Medium	Medium
Medium damage	Medium	Medium	High
High damage	Medium	High	High

Table 3. Risk measurement scale

After applying the framework to the Subspace system, different threat scenarios according to the CIA triad were identified.

The CIA triad describes three security promises that can be violated by a hacking attack, namely confidentiality, integrity, availability.

Confidentiality:

Confidentiality threat scenarios concern sensitive information regarding the blockchain network and its users. Native tokens are units of value that exist on the blockchain - confidentiality threat scenarios include, for example, attackers abusing information leaks and gaining control over network participants' private keys. The private key can then be used to steal user funds or to impersonate them.

Integrity:

Integrity threat scenarios threaten to disrupt the functionality of the entire network by undermining or bypassing the rules that ensure that Subspace transactions/operations are fair and equal for each participant. Undermining Subspace's integrity often comes with a high monetary incentive, like for example, if an attacker can double spend or mint tokens for themselves. Other threat scenarios do not yield an immediate monetary reward, but rather, could threaten to damage Subspace's functionality and, in turn, its reputation. For example, invalidating already executed transactions would violate the core promise that transactions on the blockchain are irreversible.

Availability:

Availability threat scenarios refer to compromising the availability of data stored by the Subspace network as well as the availability of the network itself to process normal transactions. Important threat scenarios regarding availability for blockchain systems include Denial of Service (DoS) attacks on participating nodes, stalling the transaction queue, and spamming.

Table 4 provides a high-level overview of the hacking risks concerning Subspace with identified example threat scenarios and attacks, as well as their respective hacking value and effort. The complete list of threat scenarios identified along with attacks that enable them are described in the threat model deliverable [8]. This list can serve as a starting point to the Subspace developers in order to guide their security outlook for future feature implementations. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

For Subspace, the auditors attributed the most hacking value to the availability class of threats. Undermining the availability of the Subspace chain renders the main functionalities of the project unavailable. Hindering block production on the main Subspace chain will have an impact on every domain ecosystem tied to the main chain.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confiden-tiality	Medium	- Compromise a user's private key	High	- Targeted attacks to compromise a user's private key
Integrity	High	- Take over executor nodes - Fabricate false transactions - Perform privilege escalation between untrusted and trusted domains	Medium	- Slash honest executors by submitting false offence reports - Spoof XDM messages
Availa-bility	High	- Stall block production - Spam the blockchain with bogus transactions	Low	- Spam a target domain via fraudulent XDM messages - Fill up XDM channels' in/out box - Crash executor nodes

Table 4. Risk overview. The threats for Subspace’s blockchain were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

At the time of the audit, none of the assessed code was live, which lead the code assurance team to refrain from assigning severity to the publicly reported issues on GitHub. However, to effectively communicate the potential impact of these vulnerabilities and assist in prioritizing internal mitigation efforts, in this report Security Research Labs assigned severity levels assuming the criticality of the vulnerabilities as if they existed within a live system.

Security Research Labs defines the highest severity "Critical" as an issue that could be exploited immediately by an attacker on a live system.

5 Consensus whitepaper review

5.1 Findings summary

During the analysis of the consensus protocol, Security Research Labs identified three issues which are summarized in table 5. Two of these issues allow attackers to gain financial advantage over other farmers due to a bug in the implementation and were assigned critical severity. The remaining issue revolves around the usage of weak cryptographic primitives and was assigned a high severity.

Each issue described here was publicly shared on the GitHub mono-repository [3] as issues, which have been tagged "Audit".

Issue	Severity	References	Status
A fraction of plotted data is used to verify valid solutions	Critical	[9]	Open
XOR-based calculation of *evaluation_seed* may lead to seed reuse	Critical	[10]	Open
Global randomness update only depends on one single block	High	[11]	Open

Table 5 Consensus whitepaper issue summary

5.2 Detailed findings

5.2.1 A fraction of plotted data is used to verify valid solutions

Attack scenario	**Gain farming advantage over other nodes**
Location	subspace-verification
Tracking	[9]
Attack impact	Attackers can gain an illicit advantage in farming.
Severity	Critical
Status	Open

During the verification (called auditing) procedure of the stored data from a farmer, the function *calculate_solution_distance* will XOR the plotted data (*audit_chunk*) from disk with *sector_slot_challenge*. If the result is within the solution range, this is considered a valid solution.

If a farmer is only storing the 16 most significant bits of the 64-bit value *audit_chunk*, this will still allow ruling out most cases where the user does not have a valid solution, depending on the *global_challenge* value only one (or at most two) possible values for the most significant 16 bits can lead to a solution within the solution range. An attacker can therefore throw away 75% of the plotted data and still find out whether there may be a solution or not.

This issue can be fixed by using a hashing function to combine the *sector_slot_challenge* and the plotted data. We recommend using a HMAC function with the *sector_slot_challenge* as key and the plotted data as message.

5.2.2 XOR-based calculation of *evaluation_seed* may lead to seed reuse

Attack scenario	**Gain farming advantage over other nodes**
Location	subspace-core-primitives
Tracking	[10]
Attack impact	Attackers can reuse the (computationally expensive) PoS tables.
Severity	Critical
Status	Open

During the verification (called auditing) procedure, the *evaluation_seed* is used for the proof of space. This *evaluation_seed* depends on the *sector_index* (thus should be different for every audited sector), *history_size* and *piece_offset*. When XORing them together to compute the *evaluation_seed*, only the first bytes are used, which makes that attackers can reuse the (computationally expensive) PoS tables. We recommend hashing the parameters together instead of using XOR.

5.2.3 Global randomness update only depends on one single block

Attack scenario	**Gain farming advantage over other nodes**
Location	pallet-subspace
Tracking	[11]
Attack impact	An attacker may be able to manipulate the randomness in order to maximize the number of solution slots in the next randomness epoch.
Severity	High
Status	Open

The global randomness for the chain is updated once every 256 blocks. The next global randomness only depends on the last of the 256 blocks instead of mixing values from all of them. This has two possible consequences: prior knowledge of random challenges may make compression attacks more feasible. In addition to that, randomness could be manipulated, making a DoS attack against the network or other farmers to align the randomness update block with a slot in which the attacker has a good solution possible.

We recommend mitigating this issue by including other blocks to contribute to the randomness, for example hashing the current value to the new random value at each block, or a verifiable random function. After discussion with the Subspace team, this is an intentional design choice based on the paper “Proof-of-Stake Longest Chain Protocols: Security vs Predictability” [12].

6 Baseline assurance

6.1 Findings summary

6.1.1 First phase: Initial assessment

During the initial assessment of the in-scope runtime components, seven issues were identified. Security Research Labs found one critical issue [13] in the executor registry's implementation that allows any executor candidate to fill up the executor slots and gain control over the execution domain or prevent other executor candidates from joining the executors. Additionally, five high severity and one moderate severity issues were found as summarized in Table 6.

Each issue described here was publicly shared on the GitHub mono-repository [3] as issues, which have been tagged "Audit".

Issue	Severity	References	Status
Executor registry implementation allows DoS by executor candidates	Critical	[14]	Open
Missing bundle election proof validation allows spamming	High	[15]	Open
Missing bundle equivocation checks allow malicious behavior	High	[16]	Open
Missing fraud proof validation and slashing allow malicious behavior	High	[17]	Open
Missing exponential fee adjustments are enabling DoS	High	[18]	Open
Extrinsics with default weights can cause block production timeouts	High	[19]	Closed
Users could spam the blockchain with free (*Pays::No*) calls	Moderate	[20]	Open

Table 6 Initial assessment issue summary

6.1.2 Second phase: Continuous review

During the continuous review phase of the runtime components, another four issues as summarized in Table 7 were uncovered. One critical issue found in this phase allows any domain executor to execute root calls with no filters. The other identified vulnerabilities consist of two high severity and one moderate severity issues.

Issue	Severity	Reference	Status
System domain executor can do an unsigned root call	Critical	[21]	Open
The unsigned extrinsic *submit_fraud_proof* allows spamming	High	[22]	Open
Relayer registry implementation allows DoS by registry candidates	High	[13]	Open
Extrinsics with missing storage deposits could clutter the storage	Moderate	[23]	Open

Table 7 Continuous review issue summary

6.2 Detailed findings

6.2.1 Executor registry implementation allows DoS by executor candidates

Phase	Initial assessment
Attack scenario	An attacker can fill up executor slots due to low stake requirements
Location	executor-registry
Tracking	[14]
Attack impact	An attacker may censor certain transactions or perform a DoS attack on a target domain
Severity	Critical
Status	Open

The register extrinsic in executor-registry pallet allows any user to register as a domain executor given that they submit a deposit. This deposit is controlled via the *MinExecutorStake* in the system runtime configuration and is set to 10 SSC. Furthermore, the number of available executor slots which is controlled by the *MaxExecutors* configuration is set to 10. Therefore, anyone with 100 SSC may fill up the whole executor slots and gain control of the execution system of a domain. This allows an attacker to use their leverage and censor certain transactions or perform DoS on the target domain.

To mitigate this, we recommend implementing a staking election for domain executors. Additionally, by introducing “invulnerable” executors which are trusted by the organization/community the risk of such DoS attacks can be reduced.

6.2.2 Missing bundle election proof validation allows spamming

Phase	Initial assessment
Attack scenario	Spamming bundles due to missing validation
Location	domain-executor
Tracking	[15]
Attack impact	An attacker can spam the domain by submitting bogus domain bundles

Severity	High
Status	Open

On the secondary chain, the domain executors produce bundles (like block production on the primary chain). To be authorized to produce a bundle, executors must win the election by calculation a solution to the bundle production challenge. Once the bundle is gossiped to the rest of the network, the receiving node shall validate the bundle solution proof before processing the bundle. As this feature is currently not implemented [24], it means that domain executors may spam the chain by submitting bogus bundles.

We recommend implementing this feature to prevent spamming by bundle domain executors.

6.2.3 Missing bundle equivocation checks allow malicious behavior

Phase	Initial assessment
Attack scenario	Spamming the chain with equivocating transactions due to the missing validation logic
Location	domain-executor
Tracking	[16]
Attack impact	An attacker can undermine the integrity of the blockchain with equivocating transactions
Severity	High
Status	Open

To prevent equivocation, the executor nodes should validate the incoming bundles for equivocation upon bundle reception. This is currently not implemented for domain executors which allows a misbehaving validator to gossip equivocating bundles to the network and undermine the chain's integrity.

We recommend implementing this feature to prevent equivocating bundles.

6.2.4 Missing fraud proof validation and slashing allow malicious behavior

Phase	Initial assessment
Attack scenario	Missing fraud proof validation and slashing
Location	pallet-domains, pallet-receipts, pallet-registry
Tracking	[17]
Attack impact	The lack of misbehavior checks and punishment encourages misbehavior on the network.
Severity	High
Status	Open

In the Subspace codebase, slashing is not yet implemented for security critical actions of potentially untrusted participants in the protocol. For slashing to work, there needs to be mechanism to report any fraud committed by misbehaving parties to other participants; as well as implementing means to validate the fraud reports by means of proofs. Currently this is not implemented in Subspace which allows misbehaving parties to spam the chain by submitting false fraud reports.

To disincentivise misbehaviour the fraud proof validation should be implemented as well as measures of punishment for the misbehaving parties.

6.2.5 Missing exponential fee adjustments are enabling DoS

Phase	Initial assessment
Attack scenario	Targeted DoS attacks against certain transactions
Location	All runtime configurations
Tracking	[18]
Attack impact	An attacker can block/delay certain transactions
Severity	High
Status	Open

In all Subspace runtime configurations, the *FeeMultiplierUpdate* is not set to a *TargetedFeeAdjustment*. The *TargetedFeeAdjustment* is used to adjust the fees automatically when the chain is spammed with transactions. When this parameter is not configured properly, an attacker can cause DoS against all zero-tip transactions for an extended period by paying the regular transaction fees instead of the exponentially raising fees.

At the time of the audit, Subspace did not make use of any fee systems. As exponential fees are important to prevent spamming attacks, we suggest keeping this in mind when enabling fees in the future.

6.2.6 Extrinsics with default weights can cause block production timeouts

Phase	Initial assessment
Attack scenario	Underweighted extrinsics can cause block production timeouts
Location	domain-registry, executor-registry
Tracking	[19]
Attack impact	An attacker may slow down transaction processing and potentially stall the chain if all collators miss their block production slots
Severity	High
Status	Closed

In Substrate-based blockchains, the weight of an extrinsic is a way to define how much execution resources an extrinsic requires which is calculated via benchmarking. The weight of an extrinsic tells the block producer how many extrinsic they can fit in their blocks. In Subspace several extrinsics were not benchmarked and had a default weight, e.g., *10_000* which could cause a block production timeout.

Subspace addressed this issue by adding benchmarking code for the affected extrinsics [25].

6.2.7 Users could spam the blockchain with free (*Pays::No*) calls

Phase	Initial assessment
Attack scenario	Spamming attack via free extrinsics
Location	pallet-domains, pallet-feeds, pallet-object-store, pallet-subspace, domain-registry
Tracking	[20]

Attack impact	An attacker can spam the chain using this extrinsics
Severity	Moderate
Status	Open

Throughout the codebase, various modules expose free extrinsics (extrinsics with *Pays::No* weight annotation) which means that these extrinsics will be free for anyone who calls them. This allows an attacker to spam the chain by repeatedly calling free extrinsics, e.g., *join_relayer_set* and *exit_relayer_set*, and not paying any fees. While some these extrinsics are only callable by trusted entities (e.g., domain executors), it is still best practice to require a fee for most operations.

It is evident through code comments on some of these extrinsics that Subspace developers are aware of this risk imposed by free extrinsics and are planning to implement proper fees.

6.2.8 System domain executor can do an unsigned root call

Phase	Continuous review
Attack scenario	Root call execution via *sudo_unchecked_weight_unsigned* extrinsic
Location	pallet-executive
Tracking	[21]
Attack impact	A malicious executor gains control of the chain via a root call
Severity	Critical
Status	Open

The unsigned extrinsic *sudo_unchecked_weight_unsigned* in pallet-executive dispatches the given call parameter via the *call.dispatch_bypass_filter(frame_system::RawOrigin::Root.into())*. This will dispatch the call and bypass all the call filters configured in the runtime. A malicious domain executor may abuse this feature to call any extrinsic as root. Since it is using the normal execution procedure of the runtime, it is not possible for other executor nodes on the network to spot the malicious call and report it as fraud.

We suggest designing the executor pallet so that it adds origin validation and prevents misbehaving domain executors from making root calls.

6.2.9 The unsigned extrinsic *submit_fraud_proof* allows spamming

Phase	Continuous review
Attack scenario	Spamming attack via unsigned extrinsic *submit_fraud_proof*
Location	pallet-domains
Tracking	[22]
Attack impact	An attacker may spam the chain using this extrinsic
Severity	High
Status	Open

The extrinsic *submit_fraud_proof* in pallet-domain allows users to submit fraud reports to the network. However, these are unsigned extrinsics and therefore are not bound to any account IDs. This allows misbehaving entities to submit false fraud report to the network without being tracked and punished.

To mitigate this issue, we suggest implementing a form of cryptographic signature checking or implementing this extrinsic as a signed extrinsic.

6.2.10 Relayer registry implementation allows DoS by registry candidates

Phase	Continuous review
Attack scenario	Exhausting relayer slots cheaply due to low deposit requirements
Location	pallet-messenger
Tracking	[13]
Attack impact	An attacker can DoS a target domain at low cost
Severity	High
Status	Open

In Subspace, relayers relay the messages between domains. To become a relayer, users can use the *join_relayer_set* extrinsic in pallet-messenger which requires a deposit of 100 SSC from the calling origin and checks whether the number of relayers, currently set to 100, has not the upper limit. This allows an attacker to fill up the relayers slots using 10000 SSC. By controlling the relayers, the attackers may perform a DoS on the target domain or simply censor the messages.

We recommend using a better value for the *RelayerDeposit* and *MaximumRelayers* parameters based on some economic analysis.

6.2.11 Extrinsics with missing storage deposits could clutter the storage

Phase	Continuous review
Attack scenario	Repeatedly calling extrinsics that add large objects to the chain's storage without requiring proper deposits
Location	Numerous, see issue and comments
Tracking	[23]
Attack impact	An attacker can clutter the storage at low cost
Severity	Moderate
Status	Open

Storage deposit fees are missing from several extrinsics throughout the Subspace codebase. A malicious entity could call these extrinsics repeatedly and store non-relevant data into the blockchain database to clutter the underlying storage. If the data is never deleted from the blockchain storage, even without malicious interactions, normal operations could clutter the storage over time and increase the barrier of entry.

To mitigate this issue, we recommend charging storage deposits for objects that are stored on-chain which will be returned to the owner after the object is removed from the storage. Furthermore, on-chain storage items that are not relevant anymore should also be automatically cleaned from the storage. This issue being similar to 6.2.1, we recommend implementing a staking election for domain executors.

7 Evolution suggestions

To ensure that Subspace is secure against known and yet undiscovered threats alike, the auditors recommend considering the evolution suggestions and best practices described in this section.

7.1 Core improvement suggestions to improve security posture

Implement security features. At the time of the audit, some security features were not yet implemented, but commented as TODOs. Security Research Labs recommends undertaking the “shift left” approach and ironing out security as an integral part of the development process.

Address remaining security issues. Security Research Labs recommends addressing the already known and recently reported security issues in a timely manner to prevent attackers from exploiting them – even if an open issue has a limited impact, an attacker might use it as part of their exploitation chain, which may have a more severe impact on Subspace. Security Research Labs recommends fixing the issues identified during this audit before launching the incentivized network.

Iron out terminology and documentation. The existing documentation could benefit from improvements as it is not up to date.

Conduct a security-focused economic analysis of fees and deposits. Currently, some pallets contain configurable deposits and fees that are not following security best practices. For example, in the domain-registry pallet the *ReplayerDeposit* which is currently 100 SSC or *MinDomainDeposi* which is set to 10 SSC allow an attacker to perform a DoS attack against the blockchain (see 6.2.1). We suggest setting proper values that are obtained based on an economic analysis for such parameters.

7.2 Further recommended best practices

Audit the code with the first stable build. Once the code has reached maturity and is ready to be pushed to production, Security Research Labs advise to perform a holistic full-scope code audit before putting the network online. As the code will substantially change before the first stable build, some new vulnerabilities or risks may be introduced. Additionally, we recommend having the remediation code reviewed.

Regularly review the code and continuously fuzz test. Subspace code base has very few tests implemented, both unit test and integration tests. Testing the code thoroughly helps regressions in the code, as well as making sure security measures do work in practice. Security Research Labs recommends having more tests and better coverage of the code base. Additionally, regular code reviews are recommended to avoid introducing new logic or arithmetic bugs, while continuous fuzz testing can identify potential vulnerabilities early in the development process. Ideally, Subspace should continuously fuzz their code on each commit made to the codebase. The Polkadot codebase provides a good example of multiple fuzzing harnesses.

Regularly update. New releases of Substrate may contain fixes for critical security issues. Since Subspace is a product that heavily relies on Substrate, updating to the latest version as soon as possible whenever a new release is available is recommended. Security Research Labs recommend paying special attention about security fixes, specifically Substrate related ones. Security Research Labs recommends setting up a review process for every new main version of Substrate to be incorporated into the update process of Subspace.

Establish best practice review process. Finding vulnerabilities is only the start of the remediation process. From experience the auditors have seen that assigning issues to specific people is the single best way of fixing issues. A lot of issues go unfixed just by the fact that no person is assigned to fix them. To ensure that no issue goes unfixed, Security Research Labs recommends setting up a review process, establishing a set of guidelines and criteria for the review to ensure consistency and standardization.

Avoid forking pallets and libraries. While it may not always be possible to contribute upstream to popular pallets, such as *block-builder* or *executive* pallets, forking should be avoided in most cases. Having a fork of a known pallet makes getting upstream fixes a manual process and harder to maintain. Moreover, the adapted fix for the forked pallet may not mitigate the underlying security issue, or it may introduce new vulnerabilities.

8 Bibliography

[1] [Online]. Available: https://github.com/subspace/subspace/issues?q=is%3Aissue+label%3AAudit.

[2] [Online]. Available: https://github.com/subspace/consensus-v2-research-paper/tree/main.

[3] [Online]. Available: https://github.com/subspace/subspace/.

[4] [Online]. Available: https://securityresearchlabs.sharepoint.com/:b:/s/SubspaceNetwork/EUwNweQhTX1GnY0oCbpTs5MBfrmshDYNdBIC3ZZ5_dRQkg?e=ll6ypd.

[5] [Online]. Available: https://subspacelabs.notion.site/Subspace-SR-Labs-Audit-60ca83364bd441adbf03372d3622801b.

[6] [Online]. Available: https://github.com/subspace/subspace/issues?q=is%3Aissue+label%3AAudit+is%3Aopen.

[7] [Online]. Available: https://securityresearchlabs.sharepoint.com/:x:/s/SubspaceNetwork/Ed1iEa767FdJo8jazVAzWtoBIzLmnyoKeXeJej0LKXYmog?e=8KfeD5.

[8] [Online]. Available: https://securityresearchlabs.sharepoint.com/:x:/s/SubspaceNetwork/EX_nyxyCaf9AsBsj2AHC0GYBueSFEAaE6QUGerGOGAXb4Q?e=XXtTuN.

[9] [Online]. Available: https://github.com/subspace/subspace/issues/1478.

[10] [Online]. Available: https://github.com/subspace/subspace/issues/1468.

[11] [Online]. Available: https://github.com/subspace/subspace/issues/1482.

[12] Z. a. al., "Proof-of-Stake Longest Chain Protocols: Security vs Predictability," [Online]. Available: https://arxiv.org/abs/1910.02218.

[13] [Online]. Available: https://github.com/subspace/subspace/issues/1494.

[14] [Online]. Available: https://github.com/subspace/subspace/issues/1337.

[15] [Online]. Available: https://github.com/subspace/subspace/issues/1339.

[16] [Online]. Available: https://github.com/subspace/subspace/issues/1340.

[17] [Online]. Available: https://github.com/subspace/subspace/issues/1338.

[18] [Online]. Available: https://github.com/subspace/subspace/issues/1284.

[19] [Online]. Available: https://github.com/subspace/subspace/issues/1336.

[20] [Online]. Available: https://github.com/subspace/subspace/issues/1335.

[21] [Online]. Available: https://github.com/subspace/subspace/issues/1483.

[22] [Online]. Available: https://github.com/subspace/subspace/issues/1465.

[23] [Online]. Available: https://github.com/subspace/subspace/issues/1517.

[24] [Online]. Available: https://github.com/subspace/subspace/blob/799b0b24d9c5ac85c745d1356ac01f54634440d5/domains/client/domain-executor/src/core_gossip_message_validator.rs#L207.

[25] [Online]. Available: https://github.com/subspace/subspace/pull/1467.

[26] [Online]. Available: https://github.com/paritytech/polkadot/tree/master/xcm/xcm-simulator/fuzzer.

[27] [Online]. Available: https://securityresearchlabs.sharepoint.com/:b:/s/SubspaceNetwork/EUwNweQhTX1GnY0oCbpTs5MBZKQCnIvFZwGRg5JUMPx-SQ?e=ILzHNn.