Security Research Labs

t3rn Baseline Security Assurance

Threat model and hacking assessment report

V1.0, 15 Mar 2024

Aarnav Bos	aarnav@srlabs.de
Haroon Basheer	haroon@srlabs.de
Kevin Valerio	kevin@srlabs.de
Mostafa Sattari	mostafa@srlabs.de

Abstract. This work describes the result of a thorough and independent security assurance audit of the t3rn parachain platform performed by Security Research Labs. Security Research Labs is a consulting firm that has been providing specialized audit services in the Polkadot ecosystem since 2019, including for the Substrate and Polkadot projects. During this study, t3rn provided access to relevant repositories for the research team. The code of t3rn was verified to assure that the business logic of the product is resilient to hacking and abuse.

The research team has identified a variety of security issues ranging from high to low severity, in areas such as authorization, benchmarking, and runtime configuration. Considering these findings, it is recommended that the t3rn team prioritize the mitigations of these issues in their upcoming updates, beginning with the highest severity issues. Addressing these vulnerabilities systematically will minimize the risks to their network's functionality, business logic, and exposure to potential attacks significantly.

Furthermore, t3rn may adhere to Substrate's best practices and security guidelines during the remediation and future development process to prevent the recurrence of such issues. Additionally, a shift towards a high code quality and security-first mindset is recommended for the t3rn developers in future developments. This includes the adoption of secure coding practices, better repository management, and improvements in documentation. These measures will enhance long-term evolution, maintainability, and the overall security posture of the codebase.

Additionally, this report also emphasizes the importance of conducting regular security audits to ensure continuous improvement and adherence to security best practices.

Content

1 Disclaimer

This report describes the findings and core conclusions derived from the audit conducted 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

Blockchains evolve in a trustless and decentralized environment, which by its own nature could lead to security issues. Ensuring availability and integrity is a priority for t3rn. 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 t3rn team can then integrate into their own threat modeling and development process to enhance the security of the product.

The t3rn blockchain is built on top of Substrate. Like other Substrate-based blockchain networks, the t3rn code is written in Rust, a memory safe programming language. Substrate-based chains utilize three technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, and a block production engine.

The t3rn runtime consists of multiple modules compiled into a WASM Binary Large Object (blob) that is stored on-chain. Nodes execute the runtime code either natively or will execute the on-chain WASM blob.

The core business logic of t3rn is to provide a platform and protocol to facilitate reversible, interoperable execution across blockchains. This includes transaction validation, consensus mechanisms specific to t3rn, and the execution of smart contracts in a way that they can operate across different blockchain platforms withing the defined security constraints.

Security Research Labs collaborated with the t3rn team to create an overview containing the runtime modules in scope and their audit priority. The in-scope components and their assigned priorities are reflected in Table 1. During the audit, Security Research Labs used a threat model to guide efforts on exploring potential security flaws and realistic attack scenarios.

Repository	Priority	Component(s)	Reference
t3rn	High	circuit	[1]
	Medium	runtime configuration pallet-attesters pallet-rewards pallet-account-manager	
	Low	pallet-portal pallet-xdns pallet-clock	
eth2-light-client	High	eth2-light-client	[2]

Table 1. In-scope t3rn components with audit priority

3 Methodology

This report details the baseline security assurance results for the t3rn parachain with the aim of creating transparency in four steps, namely, threat 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 t3rn parachain 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 t3rn. Our threat modeling process is outlined in Chapter 3.

Security design coverage check. Next, the t3rn 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 t3rn implementation was evaluated for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the t3rn codebase, we derived our code review strategy based on the threat model that we established as the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.

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 pallets and the runtime configuration.

2. Identified viable strategies for the code review. Manual code audits, fuzz testing, and manual tests were performed where appropriate.

3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks, 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.

We conducted a hybrid strategy utilizing a combination of code review and dynamic tests (e.g., fuzz testing) to assess the security of the t3rn codebase.

While fuzz testing and dynamic tests establish a baseline assurance, the focus of this audit was a manual code review of the t3rn codebase to identify logic bugs, design flaws, and best practice deviations. Security Research Labs reviewed the t3rn repository up to the commit *d8654548* and eth2-light-client up to the commit *1da666d7.* 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 t3rn 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 t3rn's case is extrinsics in the runtime. (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 last step is supporting the t3rn development team with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation solution is implemented, the auditors verify the fix to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, the *eth2-light-client* private GitHub repository [2] was used for sharing the finding. We also used a Slack channel for asynchronous communication, weekly status updates on the fuzzing and code review progress along with identified issues.

4 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in t3rn'sblockchain 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 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 t3rn network
- Medium: Risk scenarios pose a considerable threat to t3rn's functionality as a network.
- High: Risk scenarios pose an existential threat to t3rn'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 t3rn project, 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 to steal native tokens from nodes participating in the t3rn ecosystem and claiming the assets (represented in the token) for themselves.

Integrity:

Integrity threat scenarios threaten to disrupt the functionality of the entire network by undermining or bypassing the rules that ensure that t3rn transactions/operations are fair and equal for each participant. Undermining t3rn'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 t3rn'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 t3rn 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 t3rn 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. This list can serve as a starting point to the t3rn developers 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 t3rn, the auditors attributed the most hacking value to the integrity class of threats. Since the efforts required to exploit this kind of issue is considered lower, we identified threat scenarios to the integrity of t3rn as of the highest risk category. Undermining the integrity of the t3rn chain means making unauthorized modifications to the system. Some of the scenarios can have a direct effect on the financial model of the system. This can include taking over permissioned actors such as gateways, disrupting executor auctions, spamming circuit or crashing remote node involved in cross chain transactions that causes financial damages to the end user and t3rn network.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas

Integrity	High	- An attacker may hijack the gateway to disrupt its functionality - An attacker may bypass the fees (e.g., by exploiting bugs/misconfigurations) - An attacker may impersonate a permission actor such as *Relayers* and *Executors*	Medium	- Hijack a gateway by calling the *set_owner* - Exploit bugs (logic, arithmetic) in the transaction implementation to bypass fees - Exploit collisions in the creation of bogus contracts or by taking over escrow accounts
Availability	Medium	- An attacker may compromise the node availability by crashing nodes - An attacker may abuse the block time difference between various blockchains - An attacker may spam the network with bogus messages	Low	- Exploit different codecs scheme across chains that do not deserialize into the same values causing execution issues - Abuse the transaction batches to nest calls and create an infinite loop

Table 4. Risk overview. The threats for t3rn'sblockchain were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

5 Findings summary

We identified 10 issues - summarized in Table 5- during our analysis of the runtime modules in scope in the t3rn codebase that enable some of the attacks outlined above. In summary 5 high severity, 3 medium severity, 1 low severity and 1 info severity issues were found.

Please note that in our methodology, critical severity issues refer to high severity issues that could be exploited immediately by an attacker on already deployed infrastructure, including a parachain or a non-incentivized testnet.

Issue	Severity	References	Status
Implementation error allows attackers to impersonate any *CircuitRole*	High	[3]	Open
Extrinsic allows takeover of gateway's escrow account	High	[4]	Open
Underweight extrinsics may cause block production timeout	High	[5]	Open
General benchmarking mismatch across multiple extrinsics	High	[6]	Open

Extrinsics that have default weights should be properly benchmarked	High	[7]	Open
Insecure randomness algorithm for Attester's shuffling	Medium	[8]	Open
The values of *IsTeleporter* and *IsReserve* should not have overlapping assets	Medium	[9]	Open
IdentityFee implementation used to configure *WeightToFee*	Medium	[10]	Open
Incorrect benchmarks for dependency pallets	Low	[11]	Open
Lack of authorization in extrinsic exposes potential for spamming	Info	[12]	Open

Table 5 Issue summary

6 Detailed findings

6.1 Implementation error allows attackers to impersonate any *CircuitRole*

Attack scenario	Implementation error allows attackers to impersonate any *CircuitRole*
Location	Circuit pallet
Tracking	[3]
Attack impact	Any user can impersonate infrastructure-critical users and perform permissioned transactions
Severity	High
Status	Open

In Circuit pallet different *CircuitRoles* are defined. The Requester role initiates cross-chain requests, such as data retrieval or smart contract executions. The Relayer role facilitates the transfer of information between different blockchains, ensuring interoperability. Lastly, the Executor carries out the actions specified in the requests, such as executing contracts or processing transactions. Therefore, some extrinsics in this pallet can only be called by users who have a valid *CircuitRole* based on their Authorization. However, a coding error allows any user to impersonate these roles and call any of the affected extrinsics.

As an example in the *bid_sfx* extrinsic, the *authorize* function is called to check whether the origin is an Executor and thus can execute the extrinsic:

```
pub fn bid_sfx(
            origin: OriginFor<T>,
            sfx_id: SideEffectId<T>,
            bid_amount: BalanceOf<T>,
        ) -> DispatchResultWithPostInfo {
            // Authorize: Retrieve sender of the transaction.
```

```
            let bidder = Self::authorize(origin,
CircuitRole::Executor)?;
```

However, the *authorize* function simply matches the *CircuitRole* and then performs a universal signed origin check:

```
fn authorize(
        origin: OriginFor<T>,
        role: CircuitRole,
    ) -> Result<T::AccountId, sp_runtime::traits::BadOrigin> {
        match role {
            CircuitRole::Requester | CircuitRole::ContractAuthor =>
ensure_signed(origin),
            CircuitRole::Relayer => ensure_signed(origin),
            CircuitRole::Executor => ensure_signed(origin),
            _ => return Err(sp_runtime::traits::BadOrigin.into()),
        }
    }
```

Since there are no checks performed in either of these functions whether the origin actually has the expected *CircuitRole*, and the *CircuitRole* is hardcoded in the extrinsic logic, it means that any origin can call the affected extrinsics, not only the ones with a *CircuitRole*. Affected extrinsics are:

- *cancel_xtx*
- *on_remote_origin_trigger*
- *on_extrinsic_trigger*
- *escrow*
- *bid_sfx*
- *confirm_side_effect*

To mitigate this issue, we suggest retrieving and cross-checking the origin's role from storage when calling the *authorize* function.

6.2 Extrinsic allows takeover of gateway's escrow account

Attack scenario	Extrinsic allows takeover of gateway's escrow account.
Location	Xdns pallet
Tracking	[4]
Attack impact	By taking over an escrow account, an attacker can cause fraudulent transactions, resulting in complete loss of escrowed funds to the requester.
Severity	High
Status	Open

In the xdns pallet, the gateway is associated with an escrow account which is holding the funds for that gateway.

The *reboot_self_gateway* extrinsic checks [13] whether the caller is signed. If the caller is unsigned, it uses the *AccountId* configured as the pallet's Treasury's *AccountId*. This *AccountId* is then used to set the Gateway's escrow account. Since any signed user can call this extrinsic, they can set their own account as the gateway's escrow account, allowing them to steal any funds used in escrow.

By taking over an escrow account, an attacker can cause fraudulent transactions, resulting in complete loss of escrow funds to the requester.

To mitigate this issue, we suggest making this extrinsic only callable by root origins.

6.3 Underweight extrinsics may cause block production timeout.

Attack scenario	Underweight extrinsics may cause block production timeout
Location	Circuit pallet
Tracking	[5]
Attack impact	Under weighted extrinsics enables attacker to create overweight blocks that could cause block production timeouts.
Severity	High
Status	Open

In Substrate-based blockchains, the weight of an extrinsic is used to determine the computational complexity of an extrinsic. This will further be used to calculate the fees of the extrinsic which is a factor of the weight. Furthermore, validators use the weight of the extrinsics to calculate the number of extrinsics that fit in a block which should execute in one slot (6 seconds).

In the circuit pallet, multiple extrinsics do not account for the worst-case scenario when benchmarking for weights. Underweighted extrinsics enable attackers to create overweight blocks that could cause block production timeouts. This can slow down transaction processing and potentially stall the chain if all collators miss their block production slots.

We suggest taking into consideration the worst-case scenario for different situations when benchmarking. For example, when an extrinsic's computation includes looping over a list, consider the maximum possible length for that item in the benchmarking implementations.

6.4 General benchmarking mismatch across multiple extrinsics

Attack scenario	General benchmarking mismatch across multiple extrinsics
Location	Circuit and xdns pallets
Tracking	[6]
Attack impact	Underweighted extrinsics enable attackers to create overweight blocks that could cause block production timeouts.

Severity	High
Status	Open

Extrinsics must have a weight function which factors in storage, database access and computation. The reuse of the same weight across multiple extrinsics may result in a mismatch of computational requirements and the cost of execution.

In circuit and xdns pallets the extrinsics are reusing the weight functions that are generated for other extrinsics. For example, *verify_event_inclusion* uses the weight function of the *verify* extrinsic. This can lead to underweighted extrinsics and overweight blocks that could disrupt block production similar to 6.1.

We suggest creating unique benchmarks and weight functions for each extrinsic, factoring in their computational complexity.

6.5 Extrinsics that have default weights should be properly benchmarked

Attack scenario	Extrinsics that have default weights should be properly benchmarked
Location	Circuit and attesters pallets
Tracking	[7]
Attack impact	Slow down the transaction processing and potentially stall the chain if all collators miss their block production slots
Severity	High
Status	Open

In Substrate-based blockchains the weight of the extrinsics is produced by the benchmarking code. This will yield a unique weight for each extrinsic (e.g., *2_000_000*) which shows the computational complexity of the extrinsic.

A number of extrinsic throughout the t3rn codebase are assigned default weights (e.g., *100_000*). For example, the claim extrinsic in circuit pallet does not perform any authentication checks. Therefore, if claiming is not halted, any user could call this extrinsic while not paying properly for its execution time and thus spamming the blockchain and bloating its block size.

We suggest making sure that all the extrinsics are assigned benchmarked weights, in accordance with their computational complexity and database access.

6.6 Insecure randomness algorithm for Attester's shuffling

Attack scenario	Insecure randomness algorithm for Attester's shuffling
Location	All runtime configurations
Tracking	[8]
Attack impact	A malicious collator, also participating as an Attester could influence the randomness outcome in their favor

Severity	Medium
Status	Open

Polkadot-SDK provides the *RandomnessCollectiveFlip* as a basic randomness implementation which is inherently insecure and should not be used directly in production environments. The reason is that the output from the collective flip relies on the last 81 blocks, making it highly predictable.

In t3rn the Attesters selection is randomized, where the randomness is derived from *RandomnessCollectiveFlip.* If a malicious collator is also an active participant in the Attesters, they could potentially manipulate the outcome of the randomness algorithm. This will allow them to ensure their consistent selection in subsequent Attesters committees and gain an unfair financial advantage by controlling the committee shuffling process.

Furthermore, the Contracts pallet also uses the *RandomnessCollectiveFlip* in order to generate randomness which may have adverse consequences.

We suggest using a secure source of randomness for the Contracts and Attesters pallets by utilizing a secure randomness implementation like a verifiable random function (VRF).

6.7 Overlapping values for *IsTeleporter* and *IsReserve*

Attack scenario	The values of *IsTeleporter* and *IsReserve* should not have overlapping assets
Location	All runtime configurations
Tracking	[9]
Attack impact	An attacker may empty the *CheckedAccount* and cause a DoS
Severity	Medium
Status	Open

In XCM, the *IsTeleporter* and *IsReserve* settings are used to configure which origins are allowed to teleport and reserve transfer assets via the XCM executor.

In t3rn the *IsTeleporter* and *IsReserve* settings in multiple runtime configurations overlap and this means that the parachain trusts the relay chain for both teleport and do reserve transfers of *NativeAssets*.

An attacker could do a reserve transfer of some *NativeAsset* and then teleport it back to the initial account. Therefore, by only paying a small transaction fee and repeating the process with the same amount of *NativeAsset*, an attacker can empty the *CheckedAccount*.

Emptying the *CheckedAccount* can cause a DoS by making it impossible for other users to teleport their assets back from the parachain to the relay chain. Since the same funds can be used repeatedly for the attack, effective DoS can even be reached by an attacker with limited funding.

We recommend making a clear separation between teleporting and reserve transfers. A chain should either use teleporting or a reserve account (with a single, well-defined reserve location) for a given token, but not both at the same time.

6.8 *IdentityFee* used to configure *WeightToFee*

Attack scenario	*IdentityFee* implementation used to configure *WeightToFee*
Location	All runtime configurations
Tracking	[10]
Attack impact	The simplistic fee calculation can lead to weight fee underestimation and can be used by malicious network participants to spam the chain.
Severity	Medium
Status	Open

When configuring TransactionPayment pallet, if the *WeightToFee* is set to *IdentityFee*, it might lead to underestimation of fees. This is because *IdentityFee* does not apply any conversion or scaling to the weight, potentially causing the calculated fee to be lower than necessary for the transaction's actual resource consumption. An attacker can use this underestimation to spam the chain cheaply with bogus transactions.

In all runtime configurations, the transaction-payment pallet has the *WeightToFee* set to *IdentityFee*. The *IdentityFee* considers the exact weight as the fee which does not take into account the current network's economic conditions or congestion times.

The *WeightToFee* trait should be implemented in such a way that it dynamically adjusts the fee to reflect changes in the network's requirements or economic conditions.

An example could be implementing *WeightToFeePolynomial* for *WeightToFee* as done for the runtime configuration for Rococo. [14]

6.9 Incorrect benchmarks for dependency pallets

Attack scenario	Incorrect benchmarks for dependency pallets
Location	All runtime configurations
Tracking	[11]
Attack impact	Non-accessible extrinsic calls due to incorrectly benchmarked weights
Severity	Low
Status	Open

The t3rn project relies on FRAME pallets from Polkadot-SDK for constructing the runtime logics, with each runtime having custom pallets enabled in its configuration. Consequently, benchmarking necessitates adherence to the custom-built runtime

specifications to ensure accurate evaluation and optimization of performance metrics.

However, in t3rn this was incorrectly done using template substrate-runtime which will not accurately reflect the t3rn runtime performance.

As benchmarks can be dependent on the actual runtime configuration, this can lead to:

- Overweight extrinsics
- Underweight extrinsics

For t3rn, this could lead to non-accessible extrinsic calls due to incorrectly benchmarked weights.

All pallet extrinsics, even the Substrate ones, should be benchmarked with the actual runtime configuration by including them in the runtime's *define_benchmarks!* block. A best practice example can be found in the Kusama runtime implementation [15].

6.10 Lack of authorization in extrinsic exposes potential for spamming

Attack scenario	Lack of authorization in extrinsic exposes potential for spamming
Location	Circuit/Vacuum pallet
Tracking	[12]
Attack impact	Attackers can waste chain resources without paying for them by repeatedly calling extrinsics that lack any origin checks before execution
Severity	Info
Status	Open

Every extrinsic must be associated with an origin to maintain security and control. This ensures that only authorized entities can execute transactions, preventing unauthorized activities. Attackers can also waste chain resources without paying for them by repeatedly calling extrinsics that lack origin checks.

In t3rn, the absence of origin checks within some extrinsics in CircuitVacuum pallet will expose the blockchain to spamming attacks. This means that anyone may call the extrinsic without paying any fees for it.

For instance, the extrinsics *read_order_status* and *read_all_pending_order_status* in CircuitVacuum pallet do not perform any origin checks.

Introduce an origin check (by using *ensure_signed* or similar functionality) to enforce signed callers.

7 Evolution suggestions

The core findings from the security audit revealed that the t3rn repository and logic needs improvement in coding style, maturity, and repository organization. As a first maturity step, it is recommended to remove all the unused logics in the pallets,

improve the code readability through Rust documentation and inline commenting to create a transparent business logic for developers and auditors. Creating this transparency helps with dealing the complexity of implementation logic and reasoning about data flow. Given the complexity of t3rn's business logic, the coding style could also be improved to make the function and variable naming succinct, consistent, and easy to follow along. All these shortcomings currently make it increasingly challenging to improve the repository's security against both known and unforeseen threats.

7.1 Address currently open security and broader design issues

We recommend addressing already known security issues from Chapter 6 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 cause financial harm to user and reputation damage to t3rn. In addition to security concerns, we have also identified recurring patterns of broader issues throughout the repository.

Address inadequate benchmarking for extrinsics. The absence of adequate benchmarking, particularly without utilizing the t3rn runtime, significantly undermines the operation and production of the blockchain. It is imperative that t3rn developers thoroughly consult the Polkadot-SDK's knowledge base to ensure proper benchmarking for extrinsics. Failure to do so risks exposing the chain to low-effort attacks such as spamming and bloating, which can severely disrupt t3rn's functionality and availability. Benchmarking and validating extrinsic weights are fundamental security measures for parachain developers and should not be overlooked.

Implement proper safeguards for critical operations, such as authorization. Security considerations designed to prevent impersonation through roles design and filtering are currently insufficient, especially for key roles such as gateways that facilitate cross-chain transactions. It is crucial that all participants involved in cross-chain transactions undergo thorough validation and filtering with appropriate guard conditions. Additionally, documenting security policies and restrictions for each role is essential to enhance visibility and prevent the recurrence of such errors.

Fix all runtime misconfiguration issues. To mitigate runtime panics and security vulnerabilities in t3rn due to misconfiguration and insecure API usage, it is imperative to closely adhere to the recommendations and security advisories provided by the upstream Polkadot-SDK's documentation. This involves meticulously reviewing runtime configurations, ensuring alignment with best practices, and promptly addressing any deviations or vulnerabilities. By avoiding insecure API usage patterns, staying updated with security patches, and conducting thorough testing and training, developers can safeguard their codebase against potential risks, promoting both stability and security throughout the product's lifecycle.

Consider improving security posture and creating an internal roadmap for long-term evolution. t3rn's repository's lack of extensive documentation and forking practices underline a deeper issue of insufficient consideration for security maturity and long-term evolution goals. Without documentation on business logic, current and new developers may inadvertently introduce vulnerabilities. This insufficiency leads to fragmented codebases with varying levels of security maturity across pallets, runtime, and roles implementation. To address this, improve documentation while

emphasizing security considerations among developers such that the t3rn codebase can adapt and evolve securely. Creating and documenting a roadmap of reaching maturity in core features with well-defined milestones can further ensure t3rn's resilience against potential threats and attacks.

7.2 Further recommended best practices

Improve code documentation. The lack of clear documentation makes it challenging for auditors and internal reviewers to understand the intent and functionality of the code, leading to increased time and resources spent in attempting to decipher the programming constructs. Components like the *CirquitVacuum* module would benefit greatly from a more comprehensive set of documentation and detailed inline commentary. Enhancing these aspects could significantly streamline the review process, facilitate a better understanding of the code's purpose and design, and contribute to a more efficient and effective security evaluation.

Clean up the code base. The production code base contains numerous instances of uncompilable code blocks that have been commented out or functionalities that are partially implemented. It is advisable to remove all such unused and uncompilable code fragments (instead of commenting them out) to maintain a clean and efficient code base. This will help in preventing the inadvertent introduction of bugs during refactoring or forking processes, thereby contributing to the overall integrity and clarity of the software.

Improve repository organization. Implementing test logic and business logic in a single file is considered a best practice deviation. It will clutter the code making it difficult to navigate and maintain the code. We recommend separating the test logic from the pallet implementation in a test file or module. This will improve the long-term maintainability and may prevent the introduction of bugs to the pallets as the pallet implementation continues to evolve.

Regular code review and continuous fuzz testing. 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, t3rn should continuously fuzz their code on each commit made to the codebase. We recommend using the *substrate-runtime-fuzzer* [16] as a good starting point for getting started with runtime fuzzing.

Regular updates. New releases of Substrate may contain fixes for critical security issues. Since t3rn 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.

Avoid chain forking of pallets. Using forked repositories should be avoided in most cases: for instance the Merkle-Patricia Trie implementation in eth2-light-client is forked from *carver/eth-trie.rs* [17], which in turn is forked from *cita_trie* [18]. 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/t3rn/t3rn.

[2] [Online]. Available: https://github.com/t3rn/eth2-light-client.

[3] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/47.

[4] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/46.

[5] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/43.

[6] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/42.

[7] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/44.

[8] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/49.

[9] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/48.

[10] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/50.

[11] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/41.

[12] [Online]. Available: https://github.com/t3rn/eth2-light-client/issues/45.

[13] [Online].Available:https://github.com/t3rn/t3rn/blob/development/pallets/xdns/src/lib.rs#L1029.

[14] [Online].Available:https://github.com/paritytech/polkadot-sdk/blob/master/polkadot/runtime/rococo/constants/src/lib.rs#L87-L102.

[15] [Online].Available:https://github.com/paritytech/polkadot/blob/01fd49a7fafa01f133e2dec538a2ef7c697a26aa/runtime/kusama/src/lib.rs#L1578-L1587.

[16] [Online]. Available: https://github.com/srlabs/substrate-runtime-fuzzer.

[17] [Online]. Available: https://github.com/carver/eth-trie.rs.

[18] [Online]. Available: https://github.com/citahub/cita_trie.