Security Research Labs

zkVerify Baseline Security Assurance

Threat model and hacking assessment report

v1.0, 3 September 2025

- 17 pages -

Content

Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the timeframe and scope detailed in Chapter 3.

Please note that this report does not guarantee that all existing security vulnerabilities were discovered in the codebase exhaustively and that following all suggestions may not ensure future code to be bug free.

Version:	v1.0
Client:	Horizen Labs
Date:	3 September 2025
Audit Team:	Lara Kaiser Louis Merlin Daniel Schmidt

Timeline

Security Research Labs performed the zkVerify source code security assessment. The analysis took 1 month, starting from the 1st of July 2025.

Date	Event
1 July 2025	Initial engagement
31 July 2025	Draft report on the baseline security check delivered

Table 1: Audit timeline

1 Executive summary

1.1 Engagement overview

Security Research Labs consultants have been providing specialized audit services for substrate-based projects since 2019.

This report documents the results of a baseline security assurance audit of zkVerify that Security Research Labs performed in July 2025.

During this study, the zkVerify team provided access to relevant documentation and effectively supported the research team. We verified the protocol design, concept documentation, and relevant available source code of zkVerify.

This audit focused on assessing zkVerify's codebase for resilience against hacking and abuse scenarios. Key areas of scrutiny included runtime configuration, pallet logic and XCM configurations. The testing approach combined static and dynamic analysis techniques, leveraging both automated tools and manual inspection. We prioritized reviewing critical functionalities and conducting thorough security tests to ensure the robustness of zkVerify's platform. We collaborated closely with the zkVerify developers, utilizing full access to source code and documentation to perform a rigorous assessment.

1.2 Observations and risk

The research team identified several issues ranging from critical to informational level severity, many of which concerned low-level code in cryptography dependencies.

The zkVerify team acknowledged these issues and is actively working on remediation in cooperation with us. By now, the client has successfully remediated most impact of both of the identified critical issues.

1.3 Recommendations

In addition to mitigating the remaining open issues, Security Research Labs recommends open-sourcing the rest of zkVerify's infrastructure, sanitizing user input, avoiding forking dependencies as well as running thorough dynamic tests.

2 Evolution suggestions

To ensure that zkVerify is secure against further unknown or yet undiscovered threats, we recommend considering the following evolution suggestions and best practices described in this section.

2.1 Secure development improvement suggestions

We recommend further strengthening the security of the blockchain by implementing the following best practices:

Open-source core infrastructure. Some parts of zkVerify's core infrastructure, namely the proof-submission bot, are currently closed-source and were not submitted to external security review. To increase transparency and gather goodwill from the community, they should be thoroughly viewed and eventually open-sourced. This might not be an issue at mainnet launch, as the team aims to migrate to other submission systems.

Perform dynamic analysis. Developing fuzzing harnesses for consensus mechanisms, cryptographic operations, domain-specific areas, and other critical components is essential for identifying security vulnerabilities and business logic issues. By employing invariants, these fuzzing tests can effectively uncover subtle flaws that might otherwise go unnoticed. The Polkadot codebase exemplifies this approach by utilizing multiple fuzzing harnesses based on the substrate-runtime-fuzzer. This demonstrates how comprehensive and targeted fuzz testing can significantly enhance the security and reliability of complex systems. For details, see the *substrate-runtime-fuzzer* [1]. This technique should be employed on a regular basis, either with every new version, or through automated continuous integration.

Launch a bounty program. Establishing a bounty program to encourage external discovery and reporting of security vulnerabilities is crucial for enhancing code security. By offering incentives, such programs motivate individuals to responsibly report vulnerabilities to zkVerify rather than exploit them. This approach not only broadens the pool of people actively searching for security flaws but also fosters a collaborative security environment, helping to identify and address issues more quickly and effectively.

2.2 Address currently open security issues

We recommend addressing already known security issues before going live. 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 zkVerify.

2.3 Further recommended best practices

Regular updates. New releases of Polkadot SDK may contain fixes for critical security issues. Since zkVerify is a product that heavily relies on the Polkadot SDK, update to the latest version as soon as possible whenever a new release is available.

Avoid forking dependencies. While it may not always be possible to contribute upstream to popular dependencies, such as Plonky2 and Plonky3, avoid forking in most cases. Having a fork of a known crate makes getting upstream fixes a manual process and is harder to maintain. Moreover, the adapted fix for the forked project may not mitigate the underlying security issue, or it may introduce new flaws.

Avoid deprecated dependencies. This is a recent development, so we do not consider it a high-priority recommendation, but the Plonky2 library was recently deprecated [2] in favor of Plonky3. We recommend the zkVerify team migrate to this new library, as Plonky2 will stop receiving upstream security updates.

3 Motivation and scope

zkVerify is a blockchain offering zero-knowledge proof verification for other decentralized projects. It consists of a central chain, zkVerify, which acts as a relay-chain (main source of consensus for all chains on the network) and is the main source of zero-knowledge-proof verification, and of an EVM parachain, VFlow, which acts as a gateway from the EVM world to zkVerify.

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

zkVerify's runtime consists of multiple modules compiled into a WASM Binary Large Object (blob) that is stored on-chain. Nodes execute this WASM blob which contains the runtime.

Security Research Labs collaborated with the zkVerify 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 2. During the audit, zkVerify's online documentation provided us with a good runtime module design and implementation overview.

Repository	Priority	Component(s)	Reference
zkVerify	High	Runtime XCM Configuration	[3]
	Moderate	ZK Verification Logic	[4]
VFlow	High	Runtime EVM Integration	[5]

Table 2: In-scope zkVerify components with audit priority

4 Methodology

We applied the following four-step methodology when performing feature reviews for the zkVerify network: (1) threat modeling, (2) security design coverage checks, (3) implementation baseline check, and finally (4) remediation support.

4.1 Threat modeling and attacks

The goal of the threat model framework is to determine specific areas of risk in zkVerify network. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios.

The risk level is categorized into low, medium, and high and considers both the hacking value and the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

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

The *Hacking Value* is similarly categorized into low, medium, and high and considers the incentive of an attacker, as well as the effort required by an adversary to successfully execute the attack. The hacking value is calculated as follows:

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

While incentives describe what an adversary 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

Easiness:

- High: 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
- Low: 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 Easiness are divided according to Table 3.

Hacking Value/Likelihood	Low Incentive	Medium Incentive	High Incentive
Low Easiness	Low	Medium	Medium
Medium Easiness	Medium	Medium	High
High Easiness	Medium	High	High

Table 3: Hacking value measurement scale

Hacking scenarios are classified by the risk they pose to the system. Conversely, the *Damage* describes the negative impact that a given attack, if 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 zkVerify network
- Medium: Risk scenarios pose a considerable threat to zkVerify's functionality as a network
- High: Risk scenarios pose an existential threat to zkVerify network functionality

Damage and Hacking Value are divided according to Table 4.

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

Table 4: Risk measurement scale

Table 5 highlights some relevant threats that the team identified, as well as attack examples.

Threat	Attack Example	Risk
Hacker wants to cause direct financial damage to zkVerify	Attacker abuses a bug to mint new tokens	High
	Attacker exploits a flaw to burn treasury funds	High
Hacker wants to disrupt the verification process	Attacker executes a DoS due to a vulnerable extrinsic, halting the chain's operations	High
Hackers wants to abuse the XCM bridge for financial gains	Attacker exploits a bug to divert funds to their address	Medium
Hacker wants to execute arbitrary code on the EVM parachain	Attacker crafts an EVM call that is able to bypass filters and create a smart contract	Low

Table 5: In-scope zkVerify components with audit priority

4.2 Security design coverage check.

Next, we reviewed the zkVerify design for coverage against relevant hacking scenarios. For each scenario, we have investigated the following two aspects:

a. **Coverage**. Is each potential security vulnerability sufficiently covered by our audit?
b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

4.3 Implementation check

As a third step, we tested the current zkVerify implementation for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the zkVerify 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 tested.

Prioritizing potential risk for the network, the code was assessed for present protection against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, we:

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. We performed manual code audits, fuzz testing, and manual tests where appropriate
3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks. Otherwise, we 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 carried out a hybrid strategy utilizing a combination of code review, static tests, and dynamic tests to assess the security of the zkVerify codebase.

While static and dynamic testing establishes baseline assurance, the focus of this audit was on manual code review of the zkVerify codebase to identify logic bugs, design flaws, and best practice deviations. We reviewed the zkVerify repository which contains zkVerify implementation up to commit *6907163* from June 27th 2025, and the VFlow implementation up to commit *646ef6f* from June 27th 2025. We aimed 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 zkVerify codebase is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

4.4 Remediation support

The final step is supporting zkVerify 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 us to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via a private Slack channel. In addition, calls were to provide detailed updates and address open questions.

5 Findings summary

We identified 5 issues during our analysis of the runtime modules in scope in the zkVerify codebase that enabled some of the attacks outlined above. In summary, we found 2 critical-severity, 2 low-severity and 1 information-level issues. An overview of all findings can be found in Table 6.

5.1 Issue summary

ID	Issue	Severity	Status
1	Unchecked allocation in key deserialization	Critical	Partially mitigated [6] (Severity now Low)
2	Memory Allocation panic in sp1 verification	Critical	Partially mitigated [6] (Severity now Low)
3	Silent failure of ensure_signed in submit_proof	Low	Mitigated [7]
4	Shift left overflow when registering an invalid VK	Low	Risk Accepted
5	Locked inflation rate increases risk of network takeover	Info	Risk Accepted

Table 6: Findings overview

6 Detailed findings

6.1 Unchecked allocation in key deserialization

Attack scenario	**An attacker wants to disrupt the zkVerify relay-chain**
Component	zkVerify/verifiers/plonky2/ and zkVerify/runtime/src/xcm_config.rs
Attack impact	Attackers may stop the relay-chain from producing blocks until there is manual intervention from the block producers
Severity	Critical
Status	Partially Mitigated (Severity downgraded to Low)

Background

Even though Rust is a memory-safe language, it can still be subject to panics in undefined scenarios. These can sometimes be abused in substrate-based chains in case extrinsics are subject to forced execution (e.g. `on_initialize` via XCM or governance).

Issue Details

A call to `SettlementPlonky2Pallet::submit_proof` with a specifically crafted vk will make the WASM runtime panic and terminate. A portion of this byte-array directly controls the size of an allocated vector during deserialization in `read_usize_vec` [8] inside of the plonky2 library. This leads to a *capacity overflow*.

Risk

While the risks associated with a runtime panic alone are low, when associated with other mechanisms that force execution of extrinsics (e.g. governance, xcm) they can lead to total chain halt, until a patched node can be built and distributed.

This can happen here due to permissive XCM parameters, allowing `Everything` [9] to go through to the zkVerify relay-chain.

Mitigation

To mitigate most of the effects of this issue, we recommend hardening the XCM configuration of the relay-chain. The zkVerify team has taken this action [6] immediately upon receiving this issue, effectively **lowering the severity of this issue to low**.

The panic itself should also be patched, either in the pallet or in the dependency, via bounds checking the problematic length value.

To avoid creating compatibility issues with the underlying cryptographic libraries, the zkVerify team has decided not to take additional action.

6.2 Memory Allocation panic in sp1 verification

Attack scenario	**An attacker wants to disrupt the zkVerify relay-chain**
Component	zkVerify/verifiers/sp1/ and zkVerify/runtime/src/xcm_config.rs
Attack impact	Attackers may stop the relay-chain from producing blocks until there is manual intervention from the block producers
Severity	Critical
Status	Partially Mitigated (Severity downgraded to Low)

Background

Even though Rust is a memory-safe language, it can still be subject to panics in undefined scenarios. These can sometimes be abused in substrate-based chains in case extrinsics are subject to forced execution (e.g. `on_initialize` via XCM or governance).

Issue Details

A call to `SettlementSp1Pallet::submit_proof` with a specifically crafted proof will make the WASM runtime panic and terminate. The crash is due to an unbounded memory allocation in plonky3's poseidon2 library [10]. The `WIDTH` constant is meant to be bounded, but the bound is [not enforced in release mode [11].

An older version of the code (before zkVerify#326 [12]) crashed with the same input but at a different place in the code: the crash was due to a custom macro-generated borrow implementation [13] inside of the sp-1 library.

Risk

While the risks associated with a runtime panic alone are low, when associated with other mechanisms that force execution of extrinsics (e.g. governance, xcm) they can lead to total chain halt, until a patched node can be built and distributed.

This can happen due to permissive XCM parameters, allowing `Everything` [9] to go through to the zkVerify relay-chain.

Mitigation

To mitigate most of the effects of this issue, we recommend hardening the XCM configuration of the relay-chain. The zkVerify team has taken this action [6] immediately upon receiving this issue, effectively **lowering the severity of this issue to low**.

The panic itself should also be patched, either in the pallet or in the dependency, through better input validation.

The affected libraries were extracted from the Plonky3 project, which warns that panics in the verification code are a known issue [14]. Additionally, the vulnerable code does not exist anymore upstream [15], which means that this specific issue would be resolved by using the latest version of the code (but it would not guarantee the absence of another issue).

To avoid creating compatibility issues with the underlying cryptographic libraries, the zkVerify team has decided not to take additional action.

6.3 Silent failure of ensure_signed in submit_proof

Attack scenario	**An attacker wants to cheaply spam the chain**
Component	zkVerify/pallets/verifiers/
Attack impact	A malicious block producer can fill blocks with useless free extrinsics
Severity	Low
Status	Mitigated

Background

Unsigned extrinsics are rare in Substrate. Some are mandatory, such as the `Timestamp::set` extrinsic, and the only origin that can add them to a block is usually the block's author. Because nobody is the origin of an unsigned extrinsic, nobody pays for their fees.

Issue Details

In `submit_proof` [16], the origin is validated using `ensure_signed` [17]. However, the result of `ensure_signed` is postfixed with `.ok()`, which converts `Result<T, E>` into `Option<T>`. If the transaction is not signed, the origin is not `RawOrigin::Signed`, `ensure_signed` would return `Err(BadOrigin)`, which after applying `.ok()` results in `None`.

This `None` is then stored in `account`, which is passed to `on_proof_verified` [18]. Since `account` is `None`, `on_proof_verified` returns early [19], and the extrinsic continues without requiring a signed origin.

As a result, an attacker can call `submit_proof` and execute verification logic, bypassing fee payment all together.

`submit_proof` becomes an unsigned extrinsic and can thus only be injected into the block by a block producer.

Risk

A malicious block producer can fill-up a block with these extrinsics, paying no fees for them and thereby wasting block space.

However, since this behaviour only affects the blocks they produce, the overall impact on the network is minimal.

Mitigation

Remove the `.ok()` from the `ensure_none` check to verify that this extrinsic has been signed and the user pays for the execution costs.

6.4 Left shift overflow when registering an invalid VK

Attack scenario	**An attacker wants to abuse cryptographic verification**
Component	zkVerify/verifiers/plonky2/
Attack impact	An attacker could submit a malicious verification key, leading to incorrect verification
Severity	Low
Status	Risk Accepted

Background

Cryptographic primitives often rely on Rust mathematical code that was not written with substrate or WASM in mind, or that assume that the input was sufficiently sanitized beforehand.

Issue Details

When performing a `register_vk` for a Plonky2 verification key with a specifically crafted input, the runtime panics in debug mode during the deserialization of the VK due to a left shift overflow [20].

Risk

When the nodes are compiled in release mode, the left shift overflow will not panic. This will lead to the `cap_length` overflowing which could potentially have logic implications affecting the chain's security.

Mitigation

Safer alternative arithmetic operations should be used, such as `checked_shl`, which returns an error on overflow instead of overflowing.

The zkVerify team has accepted the potential risks associated with the issue. They are providing their users with Plonky2 verification, and the overflow is expected behavior in the context of Plonky2 (even if it might be a bug).

6.5 Locked inflation rate increases risk of network takeover

Attack scenario	**An attacker wants to takeover block producing**
Component	zkVerify/runtime/src/payout.rs
Attack impact	The risk of a 51% attack might be increased
Severity	Info
Status	Risk Accepted

Background

Inflation is often at the core of a chain's economic model, and subject to a great amount of scrutiny.

Issue Details

In zkVerify's runtime configuration, the inflation multiplier [21] value is set to **0**. This means that the inflation will stay locked at **2.5% annual** forever.

The entire system of staking target and sensitivity is bypassed [22] by this configuration.

Risk

Fixed inflation without self-correction will most likely result in low participation in the staking system.

This could increase the risk of a 51% attack substantially.

If, let's say, only 5% of all tokens are locked for participation in staking, it would be manageable for a sufficiently motivated attacker to acquire 6% of the total supply and take-over the block production.

Mitigation

Use a multiplier greater than **0**.

This issue was initially shared with Moderate severity, and after some discussion with the zkVerify team we lowered the severity to Info. They rightly pointed-out that Polkadot's inflation rate was modified in October 2024, going from a self-correcting mechanism to a fixed annual rate, and that this had not had a significant impact on staking.

7. Bibliography

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

[2] [Online]. Available: https://github.com/0xPolygonZero/plonky2?tab=readme-ov-file#%EF%B8%8F-plonky2-deprecation-notice.

[3] [Online]. Available: https://github.com/zkVerify/zkVerify/tree/4424351aa2bc7abefd6b67b6a6eb66c31e96578b/runtime/src.

[4] [Online]. Available: https://github.com/zkVerify/zkVerify/tree/4424351aa2bc7abefd6b67b6a6eb66c31e96578b/runtime/src.

[5] [Online]. Available: https://github.com/zkVerify/zkVerify-EVM-Parachain/tree/ff768ca5ea875dc0ddf4594b9bcdec250f397faf/runtime/src.

[6] [Online]. Available: https://github.com/zkVerify/zkVerify/pull/328.

[7] [Online]. Available: https://github.com/zkVerify/zkVerify/pull/325.

[8] [Online]. Available: https://github.com/zkVerify/plonky2/blob/4e0834d71150af6001a82411bd1934e0648fae2f/plonky2/src/util/serialization/mod.rs#L144.

[9] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/0a98af83f2cc29c2197247fd4359264df30f183b/runtime/src/xcm_config.rs#L229.

[10] [Online]. Available: https://github.com/zkVerify/sp1/blob/v5.0.5-no_std/crates/p3/p3-poseidon2-0.2.3-succinct/src/lib.rs#L40.

[11] [Online]. Available: https://github.com/zkVerify/sp1/blob/v5.0.5-no_std/crates/p3/p3-baby-bear-0.2.3-succinct/src/baby_bear.rs#L141.

[12] [Online]. Available: https://github.com/zkVerify/zkVerify/pull/326.

[13] [Online]. Available: https://github.com/zkVerify/sp1/blob/6544380d8d71640929c862a65174613c6274baf8/crates/derive/src/lib.rs#L75.

[14] [Online]. Available: https://github.com/Plonky3/Plonky3?tab=readme-ov-file#known-issues.

[15] [Online]. Available: https://github.com/Plonky3/Plonky3/commit/9b267c411cc8965c709b27d7b0ef81e225603c59#diff-2f9d60be40eb2f425d7ac2b8df89508e9c352b2fd408e5f877802d8a1cd891abL32-L33.

[16] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/bfd83e7db8a5f64953157018b66b9e6cfe1f91eb/pallets/verifiers/src/lib.rs#L349.

[17] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/bfd83e7db8a5f64953157018b66b9e6cfe1f91eb/pallets/verifiers/src/lib.rs#L373.

[18] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/bfd83e7db8a5f64953157018b66b9e6cfe1f91eb/pallets/aggregate/src/lib.rs#L180-L180.

[19] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/bfd83e7db8a5f64953157018b66b9e6cfe1f91eb/pallets/aggregate/src/lib.rs#L196-L196.

[20] [Online]. Available: https://github.com/zkVerify/plonky2/blob/4e0834d71150af6001a82411bd1934e0648fae2f/plonky2/src/util/serialization/mod.rs#L297.

[21] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/f024a83646c7eefe2ec53df2cb1d022815c7cee2/runtime/src/payout.rs#L88-L89.

[22] [Online]. Available: https://github.com/zkVerify/zkVerify/blob/f024a83646c7eefe2ec53df2cb1d022815c7cee2/runtime/src/payout.rs#L115-L123.