Security Research Labs

InvArch Baseline Security Assurance

Threat model and hacking assessment report

V1,0, 12 April 2024

Tobias Müller tobias@srlabs.de

Florian Wilkens florian@srlabs.de

Abstract. This work describes the result of the thorough and independent security assurance audit of the InvArch 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, InvArch provided access to relevant documentation and supported the research team. The code of InvArch 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 info to high severity. In cooperation with the auditors, InvArch already remediated a subset of the identified high severity issues.

In addition to mitigating the remaining open issues, Security Research Labs recommends documenting the intended behaviour to ease further comprehension and deploying runtime fuzzers for continuous assessment of the code.

Content

1 Motivation and scope

InvArch positions itself as a transformative platform in realm of blockchain, emphasizing the creation of Multichain Accounts that empower users to interact seamlessly across diverse blockchains with a single non-custodial account. By leveraging Cross-Consensus Messaging (XCM), it ensures cross-chain account control.

The network introduces a governance model through a multisig protocol, offering customizable roles, multisig members, dynamic voting, and self-executing governance. InvArch utilizes Substrate, Polkadot, ORML and Cumulus Frame as well as custom pallets to implement its core business logic.

In this engagement, the audit team focused on InvArch's runtime configuration code and its custom pallets.

Security Research Labs collaborated with the InvArch team to create an overview of the threats in scope and the priority of the audit. During the audit, Security Research Labs created a threat model to guide the efforts on exploring potential security flaws and realistic attack scenarios.

During the assessment of the codebase, security critical parts were identified and security issues in these components were communicated to the InvArch development team in the form of GitHub issues in a private repository.

Repository	Priority	Component(s)
Repo	High	INV4 Pallet Rings Pallet Checked Inflation Pallet Runtime Configuration (tinkernet)
	Medium	OCIF Pallet

Table 1. In-scope InvArch components with audit priority

2 Methodology

This report details the baseline security assurance results for the InvArch parachain 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 based on *hacking incentives*, i.e., the motivations to achieve the goals of breaching the confidentiality, integrity, or availability of InvArch 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 InvArch. Our threat modeling process is outlined in Chapter 3.

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

To effectively review the InvArch 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 3.

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

While fuzz testing and dynamic tests establish baseline assurance, the focus of this audit was a manual code review of the InvArch codebase to identify logic bugs, design flaws, and best practice deviations. We reviewed the InvArch repository up to the commit `ea35f6fbe2f364897cf7358128ccec51550e6e5a` for the runtime and commit `ce1c1421550019472c622b7896e3fbd7f03d2ec5` for the pallets. 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, or abuse these components or trigger unexpected behavior on the blockchain due to logic bugs or missing checks. Since the InvArch 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 InvArch'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 final step is supporting InvArch 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.

During the audit, findings were shared via a private GitHub repository. We also used a private Telegram group chat for asynchronous communication and weekly status updates.

3 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in InvArch'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 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 InvArch network
- Medium: Risk scenarios pose a considerable threat to InvArch's functionality as a network.
- High: Risk scenarios pose an existential threat to InvArch'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 InvArch 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 to steal native tokens from nodes participating in the InvArch 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 InvArch transactions/operations are fair and equal for each participant. Undermining InvArch'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 InvArch'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 InvArch 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 InvArch 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 `SRL-InvArch-Threat-Model.xlsx` that SRL has previously shared with InvArch. This list can serve as a starting point to the InvArch 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 InvArch, 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 InvArch as of the highest risk category. Undermining the integrity of the InvArch chain means making unauthorized modifications to the system. Some of the scenarios can have a direct effect on the financials of the system. This can include market manipulation, gaining tokens for free or as a vault stealing collateral without repercussions.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confidentiality	High	Steal token from node (scenario also applies to single node)	High	Attack to calculate private keys of network participants
Integrity	High	Circumvent approval mechanism of DAOs to execute calls without approval majority	Medium	Multisig approval circumvention
Availability	Medium	Delay the new block production by slowing it	Low	Transaction spamming

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

4 Findings summary

We identified 8 issues – summarized in Table 5 – during our analysis of the runtime modules in scope in the InvArch codebase that enable some of the attacks outlined above. In summary, 0 critical severity, 5 high severity, 2 medium severity, 0 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
All XCM fee payments are waived due to setting `FeeManager` to the unit type	High	Issue #8	Open
No XCM delivery fees configured for sibling parachain messages	High	Issue #7	Open
Underestimated worst-case weight for `OcifStaking::unregister_core`	High	Issue #6	Open
Incorrect runtime weights for XCM and a set of pallets	High	Issue #5	Open
Missing decode depth limit in INV4 pallet allows stack exhaustion	High	Issue #1	Fixed
Incorrect benchmarks for dependency Substrate-native pallets	Medium	Issue #4	Open
Malicious users can bloat storage at little cost via `operate_multisig`	Medium	Issue #3	Open
Incorrect weight returned by `pallet_checked_inflation::on_initialize`	Low	Issue #9	Open
Unconditional call decoding in `vote_multisig` is inefficient and potentially inflates weight	Info	Issue #2	Fixed

Table 5 Issue summary

5 Detailed findings

5.1 XCM fee payments waived due to setting FeeManager to Unit type

Location	Tinkernet Runtime
Tracking	Issue #8
Attack impact	Attackers can cause congestion, possibly leading to long delivery delays, storage exhaustion and/or dropping of messages.
Severity	High
Status	Open

The current `XcmConfig` **for both *tinkernet* and *invarch*** configures XCM fees through `type FeeManager = ();` **effectively waiving all fee payments rendering fee-based congestion control ineffective as fees are not actually charged.**

The risk is that attackers can cause congestion, potentially leading to long delivery delays, storage exhaustion and/or message drops**.**

A suitable mitigation is to not waive fees by configuring an appropriate `FeeHandler`, as performed in the Kusama ecosystem.

5.2 No XCM delivery fees configured for sibling Parachain messages

Location	Tinkernet Runtime
Tracking	Issue #7
Attack impact	Attackers can cause congestion, possibly leading to long delivery delays, storage exhaustion and/or dropping of messages.
Severity	High
Status	Open

There are no fees charged for delivering XCM messages across parachains. In the `tinkernet` runtime configuration, this is configured through `PriceForSiblingDelivery` by `type PriceForSiblingDelivery = ();`.

The risk is that attackers may send spam messages across chains without paying an fee. Excessive messages could lead to XCM queue size exhaustion by excessive storage usage until messages are delivered. This could also lead to delays in message delivery for other users.

To mitigate this issue, charge adequate message delivery fees in the runtime configuration template. To prevent excessive delivery times and storage exhaustion, an exponential fee mechanism should be used as configured in Kusama.

5.3 Underestimated worst-case weight for OcifStaking::unregister_core

Location	OCIF Pallet
Tracking	Issue #6

Attack impact	Under-weighted extrinsics enable attacker to create overweight blocks that could cause block production timeouts.
Severity	High
Status	Open

The unregister_core extrinsic in OcifStaking is underestimating the worst-case weight by a factor 100.

In the worst-case, the unregister_core function underestimates the weight by factor 100. A single core can hold up to MaxStakersPerCore unique stakers that can unstake at the same time by calling unregister_core extrinsic.

```
#[pallet::call_index(1)]
#[pallet::weight(
    <T as Config>::WeightInfo::unregister_core() +
    <T as Config>::MaxStakersPerCore::get().div(100) * <T as
Config>::WeightInfo::unstake()
)]
pub fn unregister_core(origin: OriginFor<T>) ->
DispatchResultWithPostInfo {
...
```

A risk exists, because extrinsics must have a weight that is calculated based on the worst-case computational complexity and database access of the extrinsic. Under-weighted extrinsics enable attackers to create overweight blocks that could subsequently cause block production timeouts. This can slow down transaction processing and potentially stall the chain if all collators miss their block production slots.

To mitigate the risk, ensure that unregister_core is using the worst-case weight of:

```
<T as Config>::WeightInfo::unregister_core() +
<T as Config>::MaxStakersPerCore::get() * <T as
Config>::WeightInfo::unstake()
```

5.4 Incorrect runtime weights for XCM and a set of pallets

Location	Tinkernet Runtime
Tracking	Issue #5
Attack impact	Under weighted extrinsics enables attacker to create overweight blocks that could cause block production timeouts.
Severity	High
Status	Open

The runtime weights for pallet XCM is configured using TestWeightInfo.

In tinkernet, runtime weights are configured to Zero via type WeightInfo = (); as for the following pallets:

- orml_tokens
- orml_currencies

- `orml_vesting`
- `pallet_scheduler`
- `pallet_preimage`
- `pallet_multisig`
- `pallet_uniques`
- `orml_tokens2`

The risk stems from these pallet extrinsic weights not depending on the actual runtime configuration. This could lead to underweight extrinsic. Setting the weights to `()` effectively make it a zero-cost execution for extrinsic which can lead to an attacker spamming and bloating network storage freely.

All pallet extrinsics, even the Substrate ones, should be benchmarked with the actual runtime configuration by including them in the `define_benchmarks!` block.

A best practice example can be found in the Kusama runtime implementation.

5.5 Missing decode depth limit in INV4 pallet allows stack exhaustion

Location	INV4 Pallet
Tracking	Issue #1
Attack impact	Stack exhausting could lead to a crash of the runtime which in turn impacts the availability of the nodes
Severity	High
Status	Fixed

The INV4 pallet allows executing an encoded call via the extrinsic `vote_multisig`. The call has to be proposed for voting first by `operate_multisig` and gets decoded without any depth limit once a vote was processed via `vote_multisig`.

The risk is that attackers can cause stack exhaustion, which will lead to a crash of the wasm runtime. Having a user-reachable panic in an extrinsic is a bad situation but it can be recovered by creating blocks without including the failing calls from gossip (possibly with a small number of patched collators/validators). That way a chain can continue producing blocks so that an emergency code upgrade can be applied via on-chain governance.

This risk can be mitigated by using a depth limit when decoding calls by way of `decode_with_depth_limit`.

5.6 Incorrect benchmarks for dependency Substrate-native pallets

Location	Tinkernet runtime
Tracking	Issue #4
Attack impact	Under weighted extrinsics enables attacker to create overweight blocks that could cause block production timeouts.
Severity	Medium
Status	Open

InvArch depends on a subset of FRAME pallets. The benchmarks for these pallets are done using the **substrate-node** template, instead of the correct **InvArch** runtime (i.e., tinkernet).

InvArch relies on weights for their FRAME pallet dependencies that are benchmarked with the substrate-node template runtime instead of the actual runtime.

Below you can find an example of an incorrect benchmark for `pallet_identity`:

```
impl pallet_identity::Config for Runtime {
...
type WeightInfo = pallet_identity::weights::SubstrateWeight<Runtime>;
}
```

So far, this issue has been spotted for most pallets in the runtime, also for some that are already part of `define_benchmark!`

As pallet extrinsic benchmarks can be dependent on the actual runtime configuration, this can lead to either overweighted or underweighted extrinsics for all extrinsics that are using the substrate-node template runtime weights (`SubstrateWeight`).

All pallet extrinsics, even the Substrate ones, should be benchmarked with the actual runtime configuration by including them in the `define_benchmarks!` block.

A best practice example can be found in the Kusama runtime implementation.

5.7 Malicious users can bloat storage at little cost via operate_multisig

Location	INV4 Pallet
Tracking	Issue #3
Attack impact	Storage clutter
Severity	Medium
Status	Open

The `operate_multisig` extrinsic (and more specifically the `inner_operate_multisig` function) accepts parameter like metadata and call. These will ultimately be inserted into the Multisig storage if `owner_balance` is below `minimum_support`. As both the boxed call and the metadata arguments have a significant size (up to 60kb combined), a malicious user could insert a high number of Multisig objects into the storage (utilizing accounts that do not have approval majority). This action can be executed the cost of the weight for `operate_multisig` only.

This is the relevant snippet from `inner_operate_multisig`:

```
// Wrap the call making sure it fits the size boundary
let bounded_call: BoundedCallBytes<T> = (*call)
    .encode()
    .try_into()
    .map_err(|_| Error::<T>::MaxCallLengthExceeded)?;
    // SRL: the above is bounded to Config::MaxCallLength which is set
to 50k in tinkernet
```

```
// Insert proposal in storage, it's now in the voting stage
Multisig::<T>::insert(
    core_id,
    call_hash,
    MultisigOperation {
        tally: Tally::from_parts(
            owner_balance,
            Zero::zero(),
            BoundedBTreeMap::try_from(BTreeMap::from([(
                owner.clone(),
                Vote::Aye(owner_balance),
            )]))
            .map_err(|_| Error::<T>::MaxCallersExceeded)?,
        ),
        original_caller: owner.clone(),
        actual_call: bounded_call,
        metadata,
        // SRL: metadata is bounded to Config::MaxMetadata which is set
to 10k in tinkernet
        fee_asset,
    },
);
```

An attacker could call the extrinsic multiple times to clutter the storage.

To mitigate this issue, we suggest implementing deposits for all extrinsic that save data to the storage to prevent storage bloating issues. The deposit can optionally be refunded once the storage is freed up, i.e., the call is executed. This can be achieved via the `FeeCharger` that is already used to collect the deposit for creating a *Core*.

5.8 Incorrect weight returned by pallet_checked_inflation::on_initialize

Location	Checked Inflation Pallet
Tracking	Issue #9
Attack impact	Underestimated weight may lead to block deadlines not being met in the worst case
Severity	Low
Status	Open

The `on_initialize` hook of all included pallets are executed as part of every produced block. Their return values of type `Weight` are used by the runtime to calculate the remaining time available for extrinsic calls in the block to still meet the block deadline.

The `Weight` returned by `pallet_checked_inflation::on_initialize` does not correctly reflect the storage accesses made in the respective control flow paths and thus underestimates the actual runtime required to execute the function. The following return statements are affected:

In line 181:
```
// should be: T::DbWeight::get().reads_writes(7, 3)
T::DbWeight::get().reads_writes(3, 4)
```

In line 247:
```
// should be: T::DbWeight::get().reads_writes(8, 2)
T::DbWeight::get().reads_writes(5, 2)
```

In line 249:

```
// should be: T::DbWeight::get().reads_writes(6, 2)
T::DbWeight::get().reads_writes(4, 2)
```

In line 252:

```
// should be: T::DbWeight::get().reads(6)
T::DbWeight::get().reads(4)
```

While the issue is not directly exploitable, as attackers cannot actively call an `on_initialize` hook compared to an extrinsic, a wrongly estimated `Weight` affects the remaining runtime of any block as the hook is included in all of them. In the worst-case this can lead to block deadlines not being met as the runtime tries to fit in a costly extrinsic that would have fit given the wrongly estimated runtime budget.

To mitigate this, adjust the affected return statements and continuously update and review the values in case of changes in business logic that alter the number of storage accesses performed.

5.9 Unconditional call decoding in vote_multisig is inefficient and potentially inflates weight

Location	INV4 Pallet
Tracking	Issue #2
Attack impact	Unnecessary decoding of calls may lead to higher resource consumption
Severity	Info
Status	Fixed

The `vote_multisig` extrinsic processes a given vote on a previously proposed call and executes it once support and approval thresholds are met. However, the decoding of the proposed call executes before the thresholds are tested although the decode result is only needed in case the conditions are met.

While this issue is not directly security related, the unconditional call to decode is inefficient and poses computational overhead in the case of unmet thresholds. This problem is increasingly relevant for large multisigs where proposed calls are likely to stay in an extended voting phase where each call to `vote_multisig` unnecessarily decodes the call before just updating the new vote tally.

To mitigate this, simply move the decoding of the proposed call into the condition after thresholds are checked.

6 Evolution suggestions

The overall impression of the auditors was that InvArch as a product is designed and written with security in mind. To ensure that InvArch is secure against unknown or yet undiscovered threats, we recommend considering the evolution suggestions and best practices described in this section.

6.1 Address currently open security issues

We recommend addressing already known 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 InvArch.

The issues identified are mainly concerned with underweighted extrinsics which allows an attacker to perform attacks relatively cheaply. We recommend two main strategies for short term mitigation of the issues identified: 1) Assign correct weights and 2) benchmark all pallets with the actual runtime configuration. To sustainably address weight issues, we suggest a period re-evaluation of the weights to determine whether the weights assigned still reflect the effort required by attackers.

6.2 Further recommended best practices

Documentation: We recommend producing an explanatory document describing the higher-level goals and the mechanics of the implemented functionality. A description of the intended behaviour will help users, developers, and auditors alike to comprehend the code and to assess whether the implementation matches the description.

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, InvArch should continuously fuzz their code on each commit made to the codebase. The Polkadot codebase provides a good example of multiple fuzzing harnesses based on *honggfuzz* [3].

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

7 Bibliography

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