Tezos Kernel Scope 1, 2 and 3 Security Review

Auditors

Lukasz Glen, Lead Security Researcher

Wei Tang, Lead Security Researcher

Defsec, Security Researcher

Report prepared by: Lucas Goiriz

August 15, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Etherlink is an EVM-compatible, non-custodial Layer 2 blockchain powered by Tezos Smart Rollup technology. It enables seamless integration with existing Ethereum tools, including wallets and indexers, and facilitates asset transfers to and from other EVM-compatible chains.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of tezos according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 30 days in total, Etherlink engaged with Spearbit to review the tezos protocol. In this period of time a total of **35** issues were found.

Summary

Project Name	Etherlink
Repository	tezos
Commit	4f4457...78fd
Type of Project	Infrastructure, Kernel
Audit Timeline	Jun 1st to Jun 28th
Two week fix period	Jul 1 - Jul 5

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	2	2	0
High Risk	1	1	0
Medium Risk	4	3	1
Low Risk	11	3	8
Gas Optimizations	0	0	0
Informational	17	3	14
Total	**35**	**12**	**23**

After reviewing the fixes, the team concluded that the components in scope at the provided commit hash can be considered safe as major issues are fixed.

5 Findings

5.1 Critical Risk

5.1.1 `CALLCODE` to `withdrawal` precompile may lead to fund loss

Severity: Critical Risk

Context: withdrawal.rs#L23

Description: The current `withdrawal` precompile implementation blindly accepts a `Transfer` without checking its `source` and `target`.

When a `CALLCODE` instruction executes, it creates a `Transfer` object with the following values:

- `source`: the context address.
- `target`: same as `source`.

The transfer would then be complied by sputnikvm. It may sound like a no-op, but it's an important implementation detail due to EVM's special account existence rules.

The fund will not be transferred to the precompile, but the precompile routine is invoked, passed with the above `Transfer` object. The precompile then blindly creates a `Withdrawal` message, but the source money is not deducted at all.

This is a critical severity issue due to its attack simplicity and impact of fund lose of the L2.

Recommendation: To be discussed. Two options are available:

1. Properly check the `Transfer` object.
2. Disable all call variants except `CALL`.

(2) is preferred by the auditor.

Etherlink: Fixed by MR 13997.

Spearbit: Verified.

5.1.2 Small amount of native token transfer dropped from mempool

Severity: Critical Risk

Context: withdrawal.rs#L23

Description: When attempting to send a small amount of native tokens (e.g., 0.000000000000000001 ether) using the `cast send` command to invoke the `withdraw_base58` function of a precompile contract, the transaction is being dropped from the mempool. This issue occurs even when providing a high gas limit and gas price.

The specific command used is:

```
cast send 0xff0000000000000000000000000000000000000001 "withdraw_base58(string)"
↪  "tz1PSeFiWEu5Ry9h6PvNnd8EJG51zEB2yA8Y" --value 0.000000000000000001ether --rpc-url
↪  https://node.ghostnet.etherlink.com/ --private-key xxxxxx --legacy --gas-limit 20000000 --gas-price
↪  100000000000
```

The error message received indicates that the transaction was not found and might have been dropped from the mempool:

```
Error: tx not found, might have been dropped from mempool:
↪  0x4023fd45ae7642c231a8cd89acba948cd55bf79edcdb8cc196095db3a8d8059c
```

In the logs when try to execute a withdrawal with `value = 1wei` I get:

```
------------------ Kernel Invocation ------------------
[Info] Going to run an Ethereum transaction
  - from address: 0x1074...f3eb
  - to address: Some(0xff0000000000000000000000000000000000001)
[Info] Withdrawal to "tz1fp5ncDmqYwYC568fREYz9iwQTgGQuKZqX"
[Error] Block 5 failed with 'Invalid ticket'. Reverting.
```

The team introduced a failsafe mechanism in the kernel:

- When the kernel starts to process a block, everything is copied to a temporary directory.
- If the block was succesfuly applied, we promote the temp directory.
- If the block fails, we revert the blocks and drop the transactions.

The block fails here, because of a **kernel bug**, therefore we can attack the kernel by spamming these withdrawals. Fortunately as we have a sequencer as frontend it'll probably just drop the transaction because it cannot produce a block with it. But we can always use the delayed inbox to spam it.

Recommendation: Examine the logic inside the `withdrawal_precompile` function to ensure it can handle very small amounts of native token transfers correctly.

Etherlink: Fixed was included in MR 13620.

Spearbit: Fix is verified.

5.2 High Risk

5.2.1 Non monotonic L2 block timestamps

Severity: High Risk

Context: stage_one.rs#L62

Description: The L2 blocks use two different sources for the block timestamps:

- The blocks produced from the sequencer's blueprints use the blueprint timestamp, which are set arbitrary by the sequencer.
- The blocks produced from the delayed inbox base on the L1 block timestamp.

It is almost certain that the blocks produced from the delayed inbox break monotonicity of block timestamps.

Recommendation: A blueprint from the delayed inbox is triggered upon a sequencer's blueprint. The easiest way would be to take the sequencer's blueprint timestamp for this. But the Ethereum spec is clear: block timestamps must be strictly increasing. So it could be the sequencer's blueprint `timestamp - 1` but additional checks would be required.

An additional check would be recommended: to check the sequencer's blueprints actually hold timestamps monotonicity. But there is a catch: a malicious sequencer can set the blueprint timestamp as max(i64) and it would hold any subsequent blocks.

Etherlink: The issue has been fixed by the following merge requests:

1. Accept only increasing timestamps from sequencer: MR 13807.
2. The sequencer can send blueprints that at most 5 minutes in the future with respecto to L1 timestamp: MR 13827.
3. Forced blueprint checks always use a valid timestamp: MR 13832.

Spearbit: Verified.

5.3 Medium Risk

5.3.1 Inconsistency between hardcoded gas limit and actual gas left in smart contract execution

Severity: Medium Risk

Context: tick_model.rs#L21

Description: During the execution of a smart contract function `a()` in contract `A`, an inconsistency was observed between the hardcoded maximum gas limit defined in the code and the actual remaining gas reported by the `gasleft()` function.

In the code snippet provided, the `MAXIMUM_GAS_LIMIT` constant is set to `30_000_000`, which represents the maximum amount of gas allowed for a transaction. This value is derived from the block gas limit defined in EIP-1559 as 2 times the gas target.

However, when executing the `a()` function using the `cast call` command with a specified gas limit of `999999999`, the `gasleft()` function returned a value of `999978817`, which is significantly higher than the hardcoded `MAXIMUM_GAS_LIMIT`.

This inconsistency raises concerns about the accuracy and reliability of the gas limit enforcement in the smart contract execution environment. If the actual gas left exceeds the defined maximum gas limit, it suggests that the gas limit is not being properly enforced or that there may be issues with the gas accounting mechanism.

Proof of concept:

```
contract A {
    function a() public view returns(uint) {
        return gasleft();
    }
}
```

- Deploy contract:

```
forge create --rpc-url https://node.ghostnet.etherlink.com/ --private-key xxx src/A.sol:A
↪  --legacy
```

- Call public function:

```
cast call 0x131fC1CAC96D7c279C6D46810f37595bC0F2fc7E "a()(uint256)" --rpc-url
↪  https://node.ghostnet.etherlink.com/ --gas-limit 999999999
999978817 [9.999e8]
```

Recommendation: Consider fixing the discrepancy between the hardcoded `MAXIMUM_GAS_LIMIT` and the actual gas left reported by `gasleft()`.

Etherlink: Fixed by MR 13729.

Spearbit: Verified.

5.3.2 Unbounded gas limit in the simulation & precompile

Severity: Medium Risk

Context: simulation.rs#L275, modexp.rs#L57

Description: In the simulations, there is an inconsistency between the defined maximum gas limit for Ethereum transactions and the usage of `u64::MAX` as the default gas limit in the `run` function of the `Evaluation` struct.

In the `constants` module, the `MAXIMUM_GAS_LIMIT` constant is defined as follows:

- tick_model.rs#L21:

```
pub const MAXIMUM_GAS_LIMIT: u64 = 30_000_000;
```

This constant sets the maximum gas limit for Ethereum transactions to 30 million gas units. However, in the `run` function of the `Evaluation` struct, when calling the `run_transaction` function, the gas limit is set to `u64::MAX` if no gas limit is provided in the `Evaluation` instance:

```
self.gas.or(Some(u64::MAX))
```

Recommendation: Use the `MAXIMUM_GAS_LIMIT` constant as the default gas limit.

Etherlink: Fixed by MR 13729.

Spearbit: Fix is verified.

5.3.3 Stack-based call stack may overflow the stack

Severity: Medium Risk

Context: handler.rs#L1134

Description: EVM execution requires a call stack to store all information of sub-calls as they happen. Etherlink uses a custom implementation of its call/create routines. Within it, the sub-call execution directly happens on the (native) stack.

This will likely lead to (native) stack overflow, because the depth limit of EVM sub-calls is 1024.

Recommendation: Switch to sputnikvm's provided heap-based call stack.

Etherlink: Fixed by MRs 13850 and 14018.

Spearbit: Confirmed for short-term fix. Long-term fix of switching to heap-based call stack is still recommended.

5.3.4 Fallback mechanism risks data loss due to lack of backup file validation

Severity: Medium Risk

Context: fallback_upgrade.rs#L48

Description: The `fallback_backup_kernel` function assumes that the backup files (`BACKUP_KERNEL_ROOT_HASH` and `BACKUP_KERNEL_BOOT_PATH`) exist and contain valid data. However, it does not perform any checks to verify the existence and validity of the backup files before copying them back to the original locations (`KERNEL_ROOT_HASH` and `KERNEL_BOOT_PATH`).

If the backup files are missing or corrupt, copying them back to the original locations could lead to data loss or unexpected behavior. The function blindly copies the backup files without ensuring their integrity, which can result in a faulty fallback mechanism.

```
pub fn fallback_backup_kernel(host: &mut impl Runtime) -> Result<(), RuntimeError> {
    log!(
        host,
        Error,
        "Something went wrong, fallback mechanism is triggered."
    );

    host.store_copy(&BACKUP_KERNEL_ROOT_HASH, &KERNEL_ROOT_HASH)?;
    host.store_copy(&BACKUP_KERNEL_BOOT_PATH, &KERNEL_BOOT_PATH)
}
```

Recommendation: To address this issue, it is recommended to add appropriate checks and error handling in the `fallback_backup_kernel` function before copying the backup files.

Etherlink: Acknowledged. This issue has not been fixed. As long as the code that is upgraded correctly backup these information, these information should exist and be valid. There is:

1. The kernel that creates the backups.
2. The kernel that uses the backups.

What you can review is only (1), we don't know in advance what is (2). What we can make sure is that (1) is valid in the version we're auditing.

Spearbit: Acknowledged.

5.4 Low Risk

5.4.1 Old rustls version vulnerable to RUSTSEC-2024-0336

Severity: Low Risk

Context: `Cargo.lock`

Description: The dependency of `rustls` in Etherlink is `0.21.10`. This is vulnerable to RUSTSEC-2024-0336.

The vulnerability has a CVSS score of **high**. However, as it's only related to networking and we did not find a direct usage of `rustls::ConnectionCommon::complete_io`, we set our severity of this advisory to *low*.

Recommendation: Upgrade `rustls`.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.4.2 Wrong expression precedence in withdrawal rounding check

Severity: Low Risk

Context: MR#13928

Description: The expression is wrong:

```
!amount < U256::from(u64::max_value())
```

Rust's expression precedence is that unary (`!`) is applied before comparison (<).

The current expression does not do what it is supposed. Practically, it returns `true` only for values greater than `0xfff...(x192)...000...`, but we want to check for values greater than or equal to `0x000...(x191)...1000...`.

We rate this as a low severity issue because practically the amount will not be greater than `U256::from(u64::max_value())` unless another bug is triggered.

Recommendation: Should be `!(amount < U256::from(u64::max_value()))`.

Etherlink: Fixed by MR 13964.

Spearbit: Verified.

5.4.3 Risk of blueprint drop on a sequencer upgrade

Severity: Low Risk

Context: parsing.rs#L305, upgrade.rs#L241

Description: The check if the current sequencer submitted a blueprint (a signature verification) takes place before a potential sequencer upgrade. And if there is a sequencer upgrade, then all stored blueprints are cleared.

This has two implications. If the upgrade does not actually change a sequencer address, a valid blueprint may be dropped. On the other hand, if a new sequencer is eager and submits a blueprint in the same time the upgrade takes place, its blueprint is dropped because the signature is checked before the upgrade.

Quite unlikely to happen, but worth to consider a fix.

Recommendation: It seems that it would be more reasonable to check blueprints signatures when bip is produced.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.4.4 Sequencer upgrades overriding allows the old sequencer to operate for a while

Severity: Low Risk

Context: upgrade.rs#L41

Description: There is a single place to store a sequencer upgrade. If the sequencer upgrade is not yet activated and another one is read, then the previous setting is overridden. There is one caveat. Assume we have an old sequencer, the first upgrade request with a sequencer s_A and an activation timestamp t_A, the second overriding upgrade request with a sequencer s_B and an activation timestamp t_B. Then between timestamps t_A and t_B, the old sequencer is eligible to create blueprints instead of the sequencer s_A.

Recommendation: Sequencer upgrade requests should not be overridden but rather processed in order they are read.

Etherlink: Acknowledged. We are aware of this, and keep as is for the moment.

Spearbit: Acknowledged.

5.4.5 Kernel Upgrade Override

Severity: Low Risk

Context: upgrade.rs#L39

Description: The kernel upgrade is stored in a single place. If there is a kernel upgrade stored and another is received, then the new kernel upgrade overrides the previous one.

This is still ok despite the activation time shift. But if the first kernel upgrade provides a store migration, it cannot be omitted. The kernel does not need to and should not rely on governance aware decisions.

Recommendation: Kernel upgrades approved by governance should not be dropped. A queue of kernel upgrades instead of a single slot would do the work, although it is best to keep it simple if possible since it is very rare case.

Etherlink: Acknowledged. We have decided to keep as is. It's on the responsibility of the governance to play with this carefully.

Spearbit: Acknowledged.

5.4.6 Excessive logging through smart contract fallback function can cause DoS

Severity: Low Risk

Context: Global scope

Description: The potential problem has been identified with the Counter smart contract, specifically in its fallback function. The fallback function contains code that can lead to excessive logging, potentially causing a DoS Condition.

- Contract:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

contract Counter {
    fallback () external {
        uint m = 5000000;
        assembly {
            revert(0, m)
        }
    }
}
```

In this code, the fallback function is defined with an external visibility. When the fallback function is triggered, it assigns a large value of 5000000 to the variable `m`. Then, using inline assembly, it calls the revert function with the arguments `0` and `m`.

The problem lies in the excessive size of the `m` value passed to the revert function. When a node processes a transaction that triggers this fallback function, it will attempt to generate an error message with a size of 5000000 bytes. Generating and logging such a large error message can consume significant computational resources on the node.

If an attacker repeatedly triggers this fallback function with high frequency, it can lead to a DoS condition on the node.

```
curl -X POST https://node.ghostnet.etherlink.com/ -H "Content-Type: application/json" -d '{"jsonrpc":
↪  "2.0", "id": 1, "method": "eth_call", "params": [{"to":
↪  "0x466a7e59905991c5Bdf9f21CE26e097db6CBeC31", "gas": "0x2faf080"}]}'
```

Recommendation: It is recommended to implement rate-limiting at the middleware and disable excessive logging.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.4.7 Incorrect use of handler's `ExitError`

Severity: Low Risk

Context: evm_execution/src/handler.rs

Description: `SputnikVM v0.x` exposes a `handler` trait, where library users customize the environmental behaviors (for example, log and storage). Many of the methods in the `handler` trait can return `ExitError`, indicating that things have gone wrong within the handler.

The use of `ExitError` in `handler` should be limited to "flat errors". For example, if the environment has additional permission logic and certain get/set storage operation does not have the necessary access permission, it can return `ExitError`.

Here, it's important to note the control flow after `ExitError` is returned. In this case, the current EVM call stack will immediately stop, with all state changes in the call stack reverted. The parent caller will then know that the child call stack has error, and *may continue execution*.

Most errors returned in Etherlink's `handler` are not "flat errors" as we discussed above, but environmental/native errors. In those cases, the control flow after `ExitError` is inappropriate. The parent caller has no way to distinguish whether the child exit is related to a normal error (like out-of-gas) or an environmental error, and may erroneously continue execution.

At this moment we mark it as low severity because we found that all environmental errors in the handler are only returned in case of logical errors. If other parts of the codebase is done correctly, then those errors will never happen. However, we want to note that in case the errors are found to actually happen, this may quickly turn into a high severity or critical severity issue.

Recommendation: Change the error return type to `ExitFatal` instead. The control flow after `ExitFatal` is that it will recursively fail all call stacks, which is the suitable behaviour of environmental/native errors.

This will require changes in `SputnikVM`'s handler trait. But it should be relatively straightforward.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.4.8 Revoke admin access for kernel upgrade

Severity: Low Risk

Context: parsing.rs#L497

Description: The current implementation of the Tezos smart rollup contract grants privileged access to the admin contract address, allowing it to initiate kernel upgrades by sending specific messages to the rollup contract. This centralized approach introduces a single point of failure and goes against the principles of decentralization.

While having a centralized authority for critical upgrades may provide a degree of control and coordination, it also poses risks such as censorship, collusion, or a single entity compromising the system. A decentralized approach would distribute the power across multiple parties, reducing the risk of a single point of failure and increasing the overall security and resilience of the system.

```
fn parse_internal_transfer<Host: Runtime>(
    host: &mut Host,
    transfer: Transfer<RollupType>,
    smart_rollup_address: &[u8],
    tezos_contracts: &TezosContracts,
    context: &mut Mode::Context,
) -> Self {
    if transfer.destination.hash().0 != smart_rollup_address {
        log!(
            host,
            Info,
            "Deposit ignored because of different smart rollup address"
        );
        return InputResult::Unparsable;
    }

    let source = transfer.sender;

    match transfer.payload {
        MichelsonOr::Left(left) => match left {
            MichelsonOr::Left(MichelsonPair(receiver, ticket)) => {
                Self::parse_deposit(
                    host,
                    ticket,
                    receiver,
                    &tezos_contracts.ticketer,
                    context,
                )
            }
            MichelsonOr::Right(MichelsonBytes(bytes)) => {
                Mode::parse_internal_bytes(source, &bytes, context)
            }
        },
        MichelsonOr::Right(MichelsonBytes(bytes)) => {
            if tezos_contracts.is_admin(&source)
                || tezos_contracts.is_kernel_governance(&source)
                || tezos_contracts.is_kernel_security_governance(&source)
            {
                Self::parse_kernel_upgrade(&bytes)
            } else if tezos_contracts.is_sequencer_governance(&source) {
                Self::parse_sequencer_update(&bytes)
            } else {
                Self::Unparsable
            }
        }
    }
}
```

Recommendation: Revoke the admin access for initiating kernel upgrades by removing the following condition from the `InputResult::parse_internal_transfer`.

Etherlink: The admin contract is a special contract that is part of the kernel config. It is either sets at the genesis of the chain or when doing an upgrade. On mainnet such admin contract is not set and upgrades are voted on by the community.

This contract is set on our etherlink testnet (on our l1 testnet ghostnet), which I think is ok.

Spearbit: Acknowledged.

5.4.9 Potential integer overflow in `add_ticks` function

Severity: Low Risk

Context: block_in_progress.rs#L237

Description: The `add_ticks` function in the `BlockInProgress` struct is responsible for incrementing the `estimated_ticks_in_run` and `estimated_ticks_in_block` fields by the provided `ticks` value. However, the function does not perform any checked addition or overflow checks. If the sum of the current values and the `ticks` parameter exceeds the maximum value representable by `u64`, an integer overflow will occur, leading to unexpected behavior and potentially corrupting the state of the `BlockInProgress`.

```
fn add_ticks(&mut self, ticks: u64) {
    self.estimated_ticks_in_run += ticks;
    self.estimated_ticks_in_block += ticks;
}
```

Recommendation: To mitigate the risk of integer overflow, it is recommended to use checked arithmetic operations when adding the `ticks` value to the `estimated_ticks_in_run` and `estimated_ticks_in_block` fields.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.4.10 Inconsistent nonce increment logic

Severity: Low Risk

Context: storage.rs#L860

Description: The codebase contains two functions that increment nonces: `increment_nonce` and `get_and_increment_deposit_nonce`. However, there is an inconsistency in how these functions handle the nonce increment operation.

In the `increment_nonce` function, the code uses the `checked_add` method to safely increment the nonce value. If the addition overflows, it returns an error (`AccountStorageError::NonceOverflow`). This approach ensures that the nonce increment is performed safely and any overflow condition is properly handled.

On the other hand, the `get_and_increment_deposit_nonce` function does not use `checked_add` when incrementing the nonce. Instead, it directly adds 1 to the current nonce value using the `+` operator (`let new_nonce = nonce + 1;`). This approach does not include any overflow checks or error handling.

```
pub fn get_and_increment_deposit_nonce<Host: Runtime>(
    host: &mut Host,
) -> Result<u32, Error> {
    let current_nonce = || -> Option<u32> {
        let bytes = host.store_read_all(&DEPOSIT_NONCE).ok()?;
        let slice_of_bytes: [u8; 4] = bytes[..]
            .try_into()
            .map_err(|_| Error::InvalidConversion)
            .ok()?;
        Some(u32::from_le_bytes(slice_of_bytes))
    };

    let nonce = current_nonce().unwrap_or(0u32);
    let new_nonce = nonce + 1;
    host.store_write_all(&DEPOSIT_NONCE, &new_nonce.to_le_bytes())?;
    Ok(nonce)
}
```

Recommendation: Update the `get_and_increment_deposit_nonce` function to use `checked_add` when incrementing the nonce value. This can be done by modifying the line `let new_nonce = nonce + 1;` to `let new_nonce = nonce.checked_add(1).ok_or(Error::NonceOverflow)?;`.

Etherlink: Fixed by MR 13712.

Spearbit: Fix is verified.

5.4.11 Transaction failure encountered during spam withdrawals

Severity: Low Risk

Context: withdrawal.rs#L23

Description: A smart contract named `SpamWithdrawals` was created to test the behavior of the withdrawal precompile function when making a large number of small native token transfers.

```
contract SpamWithdrawals {

    function doWithdrawals() external payable  {
        for (uint256 i = 0; i < 10000; i++) {
            address to = 0xff0000000000000000000000000000000000000001;
            to.call{value: 0.0000001 ether}(abi.encodeWithSignature("withdraw_base58(string)",
↪   "tz1WrbkDrzKVqcGXkjw4Qk4fXkjXpAJuNP1j"));
        }
    }
}
```

The contract has one function:

1. `doWithdrawals()`: Performs 10,000 iterations of calling the `withdraw_base58` function of the precompile contract (`0xff0000000000000000000000000000000000000001`) with a small amount of 0.0000001 ether.

The `doWithdrawals()` function was invoked using the `cast send` command with a value of 0.01 ether:

```
cast send 0xfa5ac9f11eaB8782B7D8C2CDBa85e03cf6407c6e "doWithdrawals()" --value 0.01ether --rpc-url
↪  https://node.ghostnet.etherlink.com/ --private-key xxxx --legacy --gas-limit 2000000 --gas-price
↪  10000000000
```

However, executing this command resulted in a failure, see the example transaction 0xc749b822ce461d6619b78c10e7dbd687e5

Recommendation: Consider implementing rate limiting mechanisms in the withdrawal precompile function to prevent excessive withdrawals within a single transaction.

Etherlink: Fixed by MR 13620.

Spearbit: Verified.

5.5 Informational

5.5.1 `Info` logging in user invoked code may lead to spamming

Severity: Informational

Context: withdrawal.rs#L172

Description: The `withdrawal` precompile contains an `Info` log in case the function selector is invalid. `CALL` itself into the precompile, even taking account of `WITHDRAWAL_COST` is really cheap.

An attacker may attempt to repeatedly call into the `withdrawal` precompile with invalid function selector in a single transaction invocation to spam the node log.

Recommendation: Change the logging level to `Debug`.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.2 `StartOfLevel` and `EndOfLevel` should not be `Unparsable`

Severity: Informational

Context: parsing.rs#L519

Description: Along with `InfoPerLevel`, the kernel gets `StartOfLevel` and `EndOfLevel` messages. And they are considered `Unparsable`.

Recommendation: In order to distinct unexpected `Unparsables` and expected messages, `StartOfLevel` and `EndOfLevel` should be just logged and dropped but not return as `Unparsable`.

Etherlink: Acknowledged, we don't think it's necessary.

Spearbit: Acknowledged.

5.5.3 Total issuance of Etherlink not accounting for the `withdrawal` precompile

Severity: Informational

Context: withdrawal.rs

Description: When a user send funds to the `withdrawal` precompile, the fund is transferred to the precompile contract itself and is effectively "burned". However, the balance then stays at the precompile.

This may cause external applications (for example, dapps) designed for Ethereum to read the total issuance of Etherlink wrongly, if they do not consider the particular situation of this precompile.

Recommendation: Document the behavior of `withdrawal` precompile so that it's clear balances on the `withdrawal` precompile should be excluded from total issuance. Or `reset_balance` on `withdrawal` precompile every time it is called.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.4 Unexpected output on `eth_call` with custom state override

Severity: Informational

Context: Global scope

Description: When making an `eth_call` request to the Ethereum JSON-RPC API with a custom state override, an unexpected output or error is encountered. The issue arises when including an additional parameter in the `eth_call` request to specify a custom state override.

The expected behavior is for the `eth_call` request to execute the specified smart contract function with the provided input parameters and return the result, taking into account the custom state override. However, the API returns an error or an unexpected response format.

For example, when making the following `eth_call` request:

```
curl https://node.ghostnet.etherlink.com/ \
  -X POST \
  -H "Content-Type: application/json" \
  --data '{"method":"eth_call","params":[{"from": null,"to":"0x6b175474e89094c44da98b954eedeac495271d0f
  ↪  ","data":"0x70a08231000000000000000000000006E0d01A76C3Cf4288372a29124A26D4353EE51BE"}, "latest",
  ↪  {"0x1111111111111111111111111111111111111111": {"balance":
  ↪  "0xFFFFFFFFFFFFFFFFFFFF"}}],"id":1,"jsonrpc":"2.0"}'
```

The response received is an error:

```
{"jsonrpc":"2.0","error":{"code":-32600,"message":"Json_encoding.Cannot_destruct at /: No case
↪  matched:\n                                        Unexpected array of size 3 instead of 2\n
↪                              Unexpected array of size 3 instead of 1"},"id":1}
```

Similarly, when making the same request to other Ethereum&BSC API endpoints, such as `https://bsc-testnet.public.blastapi.io` or `https://sepolia.era.zksync.dev`, the response is `0x`.

```
{"jsonrpc":"2.0","result":"0x","id":1}
```

Recommendation: Consider implementing error handling and validation mechanisms in the Ethereum API implementations to provide more informative and consistent error messages when encountering issues with custom state overrides in `eth_call` requests.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.5 Centralization and single point of failure in sequencer

Severity: Informational

Context: Global scope

Description: The Etherlink sequencer currently operates as a centralized entity, posing significant risks to the security, reliability, and decentralization of the rollup system. The following issues need to be addressed:

1. Single Point of Failure: The sequencer acts as a single point of failure. If the sequencer experiences downtime, malfunction, or becomes compromised, it can severely impact the entire rollup, leading to disruptions in transaction processing and user experience.
2. Lack of Decentralization: The sequencer has complete control over the selection and ordering of transactions. This centralized control raises concerns about potential censorship, manipulation, or extraction of MEV (Miner Extractable Value) through transaction ordering.

Recommendation: Implement a decentralized transaction ordering mechanism, such as a consensus algorithm or a rotating sequencer model. This ensures that no single entity has complete control over the transaction selection and ordering process. Decentralizing the transaction ordering mitigates the risks of censorship, manipulation, and MEV extraction.

Etherlink: Here's how we are going to improve these aspects:

- The single point of failure: It's indeed an issue, but on the other hand it gives you extremely fast latency because it doesn't need a consensus among a comittee. It's not excluded to introduce a small consensus like round robin to mitigate this issue but the experience might get degraded (in normal condition). It will be up to users to decide what they prefer.
- Lack of Decentralization: The sequencer has complete control over the selection and ordering of transactions, yes. On the other hand, the community has complete control over the selection of the sequencer. If the sequencer starts to be malicious or extract value, we can simply vote him out by exposing his plays.

On the specific aspect of MEV and manipulation, we are working towards a "*treshold encryption*" design where the user would submit an encrypted transaction to a committee, which be partially decrypted by all members of the committee. Therefore, it should give the most powerful resistance to MEV, manipulation and censorship as the sequencer will blindly include transactions in its blueprints.

Spearbit: Acknowledged.

5.5.6 Discrepancy in gas usage estimation for transaction execution

Severity: Informational

Context: Global scope

Description: In the provided example, the output from the `cast run` command shows a "Gas used" value of 184,134, while the on-chain transaction details indicate a gas usage of 3,450,132 out of a limit of 3,521,481 (97.97% of the gas limit):

```
cast run 0x7b13f38fd5090a9683d6b0bf30c204a98e5654310b6444354d44df9e0795530e --rpc-url
↪  https://node.ghostnet.etherlink.com/
```

Output:

```
Executing previous transactions from the block.
Traces:
  [121296] → new <unknown>@0x12dF1071Abdc7903E89C3E2e585a5C7662701F9F
    ← [Return] 494 bytes of code


Transaction successfully executed.
Gas used: 184134
```

See the transaction in the explorer: 0x7b13f38fd5090a9683d6b0bf30c204a98e5654310b6444354d44df9e0795530e.

Output:

> Value 0 XTZ Transaction fee 0.003450132 XTZ Gas price 0.000000001 XTZ (1 Gwei) Gas usage & limit by txn 3,450,132 | 3,521,481 97.97% Gas fees (Gwei) Base: 1

Recommendation: To address this issue, it is recommended to thoroughly review the gas usage estimation logic in the cast call.

Etherlink: The gas usage seems to be correct but yet badly documented. The gas of the transaction includes DA fees, whereas tracing the transaction gives the gas used for execution only.

Spearbit: Acknowledged.

5.5.7 Logging should be disabled in production stage 0 and stage 1

Severity: Informational

Context: kernel/src/lib.rs#100

Description: Logging in Etherlink is implemented in `tezos-evm-logging` crate, where it calls `alloc::format!` and then pass the returned string to `tezos_smart_rollup_debug::debug_msg`. We want to note that Rust formatter is not panic safe. It's also not simple to eliminate panics, as each `Display/Debug` trait implementation itself may return error or panic.

Because it's critical that no panic happens in stage 0 and stage 1, logging should be disabled in production.

Recommendation: Disable `Info` logging in production stage 0 and stage 1.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.8 Optimization through wasm-opt

Severity: Informational

Context: Global scope

Description: The current deployment process does not include optimization steps for the generated wasm bytecode (see `https://github.com/WebAssembly/binaryen`).

Recommendation: Incorporate the wasm-opt tool into the deployment process to optimize the generated Wasm bytecode.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.9 Unchecked arithmetic in `set_balance` function

Severity: Informational

Context: lib.rs#L270

Description: The `set_balance` function in the provided code is susceptible to an underflow issue. In the `set_balance` function, there is a condition that checks if the `current_balance` is greater than the balance parameter. If this condition is true, it attempts to subtract balance from `current_balance` and update the account's balance using `balance_remove`. However, if balance is smaller than `current_balance`, an underflow will occur:

```
fn set_balance(
    host: &mut MockHost,
    evm_account_storage: &mut EthereumAccountStorage,
    address: &H160,
    balance: U256,
) {
    let mut account = evm_account_storage
        .get_or_create(host, &account_path(address).unwrap())
        .unwrap();
    let current_balance = account.balance(host).unwrap();
    if current_balance > balance {
        account
            .balance_remove(host, current_balance - balance)
            .unwrap();
    } else {
        account
            .balance_add(host, balance - current_balance)
            .unwrap();
    }
}
```

Recommendation: To mitigate the underflow vulnerability, it is recommended to use Rust's built-in checked arithmetic operations or the `checked_sub` method provided by the `U256` type.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.10 Incompatibility with updated `selfdestruct` behavior

Severity: Informational

Context: Global scope

Description: The smart contract deployed at address `0x3853C7834d74ed8Afa05cEA06a2f2a3FB355EC61` is not compatible with the recent changes introduced to the behavior of the `SELFDESTRUCT` opcode.

Under the new rules, the `SELFDESTRUCT` opcode will only destroy a contract when it's called in the same transaction that created the contract. In all other cases, `SELFDESTRUCT` will only transfer the contract's Ether balance to the specified target address without destroying the contract's bytecode.

In the provided example, the `Exploit` contract is deployed and then subsequently destroyed using the `destroy()` function in a separate transaction.

```
pragma solidity 0.7.6;

contract Exploit {
    constructor() payable {}

    function destroy() public {
        selfdestruct(payable(address(this)));
    }

    function take() public {
        msg.sender.transfer(address(this).balance);
    }
}
```

Command

```
cast send 0x3853C7834d74ed8Afa05cEA06a2f2a3FB355EC61 "destroy()" --rpc-url
↪  https://node.ghostnet.etherlink.com/ --private-key --legacy
```

Code is set to 0x.

```
curl https://node.ghostnet.etherlink.com/ \
  -X POST \
  -H "Content-Type: application/json" \
  --data '{"method":"eth_getCode","params":["0x3853C7834d74ed8Afa05cEA06a2f2a3FB355EC61","latest"],"id"
  ↪  :1,"jsonrpc":"2.0"}'
```

```
{"jsonrpc":"2.0","result":"0x","id":1}
```

For more context, see EIP-6780.

Recommendation: Consider applying EIP requirements on the kernel evm.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.11 Excessive logging verbosity

Severity: Informational

Context: lib.rs#L168, parsing.rs#L459

Description: The code snippets provided for the `run_transaction` and `parse_deposit` functions contain multiple logging statements at the `Info` level, which can lead to excessive verbosity and potential performance issues.

1. `run_transaction` function; the logging statement in question is:

   ```
   log!(host, Info, "Going to run an Ethereum transaction\n  - from address: {}\n  - to address:
   ↪  {:?}", caller, address);
   ```

   This statement logs detailed information about every Ethereum transaction processed by the `run_transaction` function, including the caller address and the destination address (if available). While this information can be useful for debugging and tracing purposes, logging it at the `Info` level for every transaction can introduce performance overhead and clutter the log output.

2. `parse_deposit` function; the `parse_deposit` function contains multiple logging statements at the `Info` level:

   ```
   log!(host, Info, "Deposit of {} to {}.", amount, receiver);
   ```

   These logging statements provide information about deposit transactions, including cases where deposits are ignored due to different ticketers or invalid receiver addresses, as well as successful deposit details. However, logging this information at the `Info` level for every deposit transaction can lead to excessive verbosity and impact performance, especially if the `parse_deposit` function is called frequently.

Recommendation: Modify the logging statements to use the `Debug` level instead of `Info`.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.12 Inconsistent error message structure for `eth_getLogs` method

Severity: Informational

Context: Global scope

Description: The `eth_getLogs` method on the Etherlink Ghostnet node returns an error message that does not follow the standard JSON-RPC error format. The error message is returned as a string within the "message" field, instead of being a separate "error" field with a "code" and "message" subfields.

When making a request to the `eth_getLogs` method with an invalid block hash (e.g., `0x0000000000000000000000000000000000000000000000000000000000000000`), the Etherlink Ghostnet node returns the following error response:

```
curl --location 'https://node.ghostnet.etherlink.com' \
--header 'Content-Type: application/json' \
--data '{
    "jsonrpc":"2.0",
    "id":1,
    "method":"eth_getLogs",
    "params": [{"blockHash":"0x0000000000000000000000000000000000000000000000000000000000000000"}]
  }'
```

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32600,
    "message": "Evm_node_lib_dev.Durable_storage.Make(Reader).Invalid_block_structure(\"Couldn't decode
↪  bytes\")"
  },
  "id": 1
}
```

In comparison, when making the same request to a different node, such as the Moonbase Alpha TestNet node (`https://rpc.api.moonbase.moonbeam.network`), the error response follows the standard JSON-RPC error format:

```
{
  "jsonrpc": "2.0",
  "error": {
    "code": -32000,
    "message": "unknown block"
  },
  "id": 1
}
```

Recommendation: To ensure compatibility and consistency with the JSON-RPC specification and other Ethereum nodes, it is recommended to update the error handling mechanism of the Etherlink Ghostnet node to return error responses in the standard JSON-RPC format.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.13 Unsupported JSON-RPC methods

Severity: Informational

Context: Global scope

Description: The Etherlink node on the Ghostnet network does not support certain JSON-RPC methods, specifically `eth_feeHistory` and `eth_maxPriorityFeePerGas`. When attempting to call these methods using curl, the node responds with an error indicating that the methods are not supported.

Proof of concept:

1. Send a POST request to `https://node.ghostnet.etherlink.com/` with the following JSON payload:

```
{
  "method": "eth_feeHistory",
  "params": [4, "latest", [25, 75]],
  "id": 1,
  "jsonrpc": "2.0"
}
```

Command:

```
curl https://node.ghostnet.etherlink.com/ \
  -X POST \
  -H "Content-Type: application/json" \
  --data '{"method":"eth_feeHistory","params":[4, "latest", [25, 75]],"id":1,"jsonrpc":"2.0"}'
```

2. Send a POST request to `https://node.ghostnet.etherlink.com/` with the following JSON payload:

```
{
  "method": "eth_maxPriorityFeePerGas",
  "params": [],
  "id": 1,
  "jsonrpc": "2.0"
}
```

Command:

```
curl https://node.ghostnet.etherlink.com/ \
  -X POST \
  -H "Content-Type: application/json" \
  --data '{"method":"eth_maxPriorityFeePerGas","params":[],"id":1,"jsonrpc":"2.0"}'
```

Command

```
 curl https://node.ghostnet.etherlink.com/ \
-X POST \
-H "Content-Type: application/json" \
--data '{"method":"debug_traceBlockByNumber","params":["0xccde12", {"tracer":
↪  "callTracer"}],"id":1,"jsonrpc":"2.0"}'
```

The lack of support for these methods on the Etherlink node can impact developers and applications that rely on fetching historical fee information (`eth_feeHistory`) or retrieving the maximum priority fee per gas (`eth_maxPriorityFeePerGas`). It may limit the ability to analyze past transaction fees or optimize gas pricing strategies.

Recommendation: Implement support for the `eth_feeHistory` and `eth_maxPriorityFeePerGas` methods on the Etherlink node to align with the Ethereum JSON-RPC specification.

Etherlink: Fee history has been implemented in MR 13259. Max priority fee per gas has been implemented in MR 13161.

Spearbit: Verified.

5.5.14 Blocks can be filled of transactions with zero value

Severity: Informational

Context: Global scope

Description: A potential issue has been identified with the current configuration of the blockchain. As it stands, users are able to send transactions with zero value, which in itself is not a problem. However, there is currently no mechanism to prevent the blockchain blocks from being entirely filled with such zero-value transactions. This could potentially be exploited by malicious actors to spam the network, fill blocks, and slow down transaction processing times for legitimate, non-zero value transactions.

If blocks are filled with zero-value transactions, the overall processing efficiency and performance of the blockchain could be affected. This could also increase transaction waiting times and lead to a suboptimal user experience. In worst-case scenarios, it might disrupt the normal operation of the blockchain, leading to a potential loss of trust in the system's reliability.

Proof of concept:

```
const { ethers } = require('ethers');

// Set the RPC URL and private key
const rpcUrl = 'https://node.ghostnet.etherlink.com/';
const privateKey = '-';

// Create a provider instance
const provider = new ethers.providers.JsonRpcProvider(rpcUrl);

// Create a wallet instance using the private key
const wallet = new ethers.Wallet(privateKey, provider);

// Set the recipient address
const recipientAddress = '0x1234567890123456789012345678901234567890';

// Set the minimum gas price (adjust as needed)
const minGasPrice = ethers.utils.parseUnits('1', 'gwei');

// Set the number of transactions to send
const numTransactions = 100;

// Send multiple transactions with zero value
async function sendZeroValueTransactions() {
  try {
    for (let i = 0; i < numTransactions; i++) {
      const tx = {
        to: recipientAddress,
        value: 0,
        gasLimit: 2000000,
        gasPrice: minGasPrice,
      };

      const transaction = await wallet.sendTransaction(tx);
      console.log(`Transaction ${i + 1} sent:`, transaction.hash);
    }

    console.log('All transactions sent.');
  } catch (error) {
    console.error('Error sending transactions:', error);
  }
}

// Call the function to send the transactions
```

```
sendZeroValueTransactions();
```

Recommendation: In order to mitigate this issue, we recommend instituting a mechanism to increase the minimum gas price for transactions. This change would economically disincentivize spamming zero-value transactions, as each transaction would incur a minimum cost. While care should be taken to ensure that this minimum price does not unduly burden legitimate users, this measure would ensure the integrity and performance of the blockchain, enhancing system security, efficiency, and overall user experience.

As an another solution, It's recommended to introduce some form of throttling or restrictions for zero-value transactions. Another approach could be to implement a mechanism that prioritizes non-zero value transactions over zero-value ones during the block creation process. A thorough review of transaction policies should be considered to address this issue.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.15 Withdrawal precompile reverts directly instead of returning `false`

Severity: Informational

Context: withdrawal.rs#L85

Description: The withdrawal precompile implementation in the provided Rust code follows a design pattern where it reverts directly using `ExitReason::Revert` or `ExitReason::Error` instead of returning a boolean value to indicate the success or failure of the operation. This approach deviates from the expected behavior when interacting with the precompile from Solidity contracts.

In the Solidity code, when calling the precompile using `call`, `delegatecall`, or `staticcall`, the return value is expected to be a tuple `(bool success, bytes memory result)`. The `success` variable is then checked using a `require` statement to determine if the call was successful. If `success` is `false`, the transaction is meant to revert with an error message.

However, the current implementation of the withdrawal precompile does not adhere to this convention. Instead of returning a boolean value to indicate the success or failure, it reverts directly using `ExitReason::Revert` or `ExitReason::Error` in various scenarios, such as when there is an error recording the cost, when there is no transfer or the transfer value is zero, or when the input data or target address is invalid.

This mismatch between the expected behavior in Solidity and the actual behavior of the precompile can lead to confusion and unexpected transaction reverts. Developers interacting with the precompile from Solidity may assume that the precompile will return a boolean value, but instead, the transaction reverts directly without returning the expected `false` value.

- **Example Transaction:** 0x6f8681ae9b8f6165fd40453ec26da02d8f935b13da591e84eb43acb5e832cd6a
- **Example Contract:**

```
contract CallPrecompile {

    function testCall() public payable {
        address to = 0xff0000000000000000000000000000000000000001;
        (bool success, bytes memory result) = to.call{value: 0.0000001
↪  ether}(abi.encodeWithSignature("withdraw_base58(string)",
↪  "tz1WrbkDrzKVqcGXkjw4Qk4fXkjXpAJuNP1j"));
        require(success, "call failed");
    }

    function testDelegatecall() public {
        address to = 0xff0000000000000000000000000000000000000001;
        (bool success, bytes memory result) =
↪  to.delegatecall(abi.encodeWithSignature("withdraw_base58(string)",
↪  "tz1WrbkDrzKVqcGXkjw4Qk4fXkjXpAJuNP1j"));
        require(success, "call failed");
    }

    function testStaticcall() public {
        address to = 0xff0000000000000000000000000000000000000001;
        (bool success, bytes memory result) =
↪  to.staticcall(abi.encodeWithSignature("withdraw_base58(string)",
↪  "tz1WrbkDrzKVqcGXkjw4Qk4fXkjXpAJuNP1j"));
        require(success, "call failed");
    }
}
```

Recommendation: Update the error handling in the withdrawal precompile to return appropriate error codes or error messages as output instead of reverting directly.

Etherlink: Acknowledged.

Spearbit: Acknowledged.

5.5.16 `execute_transfer` does not use the `source` and `target` value of the `Transfer` object

Severity: Informational

Context: handler.rs#L1088

Description: The EVM execution code of Etherlink contains a customized implementation with its own `execute_call` and `execute_transfer`. The `execute_transfer` within a `call` is implemented as follows:

```
self.execute_transfer(
    transaction_context.context.caller,
    transaction_context.context.address,
    transfer.value,
)
```

However, `Transfer` object contains explicit `source` and `target` field.

Only two invocations will invoke this code path: on invocation of `CALL`, and on invocation of `CALLCODE`. The transfer source and target happens to be correct in both cases, but is confusing.

Recommendation: Use `Transfer` object's `source` and `target` field instead.

```
self.execute_transfer(
    transfer.source,
    transfer.target,
    transfer.value,
)
```

Etherlink: Fixed by MR 13997.

Spearbit: Verified.

5.5.17 `withdrawal` precompile should disable invocation by `CALLCODE`, `DELEGATECALL`, `STATICCALL`

Severity: Informational

Context: withdrawal.rs#L23

Description: `withdrawal` is a custom precompile for Etherlink. Upon invocation, the EVM engine passes on a parameter `is_static`, to determine if the current execution is within a `STATICCALL` context. Its `Context` also determines if the invocation is a `CALL`, `DELEGATECALL`, or `CALLCODE`.

The `withdrawal` precompile should respect the `is_static` parameter, and it should disable all other call variants other than `CALL`. Those additional invocation paths increases the attack surfaces and provides no benefits to the functionality.

Recommendation: Disable `CALLCODE`, `DELEGATECALL`, `STATICCALL` by checking the `Context` object and `is_static` boolean.

Etherlink: Fixed by MR 13997.

Spearbit: Verified.