Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Fluentbase

Veridise Inc.

September 22, 2025

► **Prepared For:**

Fluent
https://www.fluent.xyz

► **Prepared By:**

Benjamin Sepanski
Petr Susil

► **Contact Us:**

contact@veridise.com

► **Version History:**

Dec. 8, 2025	V2
Oct. 31, 2025	V1
Oct. 28, 2025	Initial Draft

Contents

1 Executive Summary

From Sep. 22, 2025 to Oct. 27, 2025, Fluent engaged Veridise to conduct a security assessment of Fluentbase. The security assessment covered a fork of revm* and Fluentbase, the core runtime manager for the Fluent blockchain. Veridise conducted the assessment over 10 person-weeks, with 2 security analysts reviewing the project over 5 weeks on commits `5e3b547` and `9ed9bd51` - `e88ea57`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. Fluent is an Ethereum L2 that implements "blended execution": a single state machine where EVM, Wasm, and (eventually) SVM smart contracts can interoperate via atomic, synchronous calls within one transaction. The security assessment covered portions of the Fluentbase repository which enable execution of Wasm binaries inside revm, and acts as the core runtime/state manager for all of the runtimes.

Fluentbase is built on top of revm-rwasm, a fork of revm. Fluent uses this robust, battle-tested codebase to track state and manage execution frames. Fluentbase works by defining a Fluent VM for use with revm-rwasm. This VM executes bytecode represented in rWasm (reduced WebAssembly)†, a Wasm-based VM optimized for Zero Knowledge proving developed by the Fluent team.

To support blended execution, Fluent exposes each external VM through a Wasm-based precompile. When a new frame is created (i.e. at the beginning of transaction execution, or when a `CALL` or `CREATE` has been invoked), the Fluentbase runtime checks which VM the contract is registered with. If the code is rWasm, the Fluent VM executes it directly. Otherwise, the Fluent VM loads the system bytecode stored in the external VM's precompile. The rWasm interpreter then executes this precompile, emulating the external VM on the contract bytecode and provided input.

The in-scope code implements this unified rWasm-centered execution stack, providing deterministic transaction processing with gas metering. The runtime loads and runs modules, exposing host syscalls for fuel management, I/O, state access, cryptographic routines, and other basic operations enabling wasm code to implement common smart contract functionality.

Interrupts & Syscalls. Understanding Fluentbase requires a careful understanding of project control flow. This section provides a brief overview of how syscalls and interruptions work inside of Fluentbase, giving context to some of the issues identified below. Non-technical readers may safely skip this section.

During execution of a smart contract, a node runs the Fluent VM described above. This interpreter in turn runs an rWasm host process, executing the rWasm guest code (the contract bytecode or external VM precompile). Several syscalls are provided. Many of these are based on the precompiles provided by SP1‡, while others relate to fuel management (see below). Each of these executes the associated precompile within the rWasm host process. There is one notable exception to this: the `Exec` syscall.

* `https://github.com/bluealloy/revm/tree/v82`
† `https://github.com/fluentlabs-xyz/rwasm/tree/devel`
‡ `https://github.com/succinctlabs/sp1/tree/db2a7b2f6d/crates/zkvm/lib/src`

The `Exec` syscall acts as the entrypoint for several other syscalls which must be executed within the Fluent VM. This includes accessing contract storage, calling/creating other smart contracts, and any other operations which require interfacing with blockchain state. `Exec` pauses execution of the rWasm host, returning control back to the Fluent VM. The Fluent VM then stores the paused host process, and executes the requested syscall. Once done, the host process is resumed. Fluent refers to invocations of the `Exec` call as an "interrupt", since it is implemented by throwing an interrupt trap code.

Gas Metering. One of the most important aspects of this system is accounting for computation, i.e. charging gas. Transaction formats and intrinsic gas computations are inherited directly from Ethereum. When executing an external VM, the Fluent VM relies on the trusted system bytecode to meter its own gas, enabling full compatibility with EVM semantics.

When deploying untrusted wasm code, the Fluent VM first compiles it to rWasm, which inserts instructions that charge gas directly into the bytecode.

Since executing rWasm is much cheaper than emulating EVM code, computation is denominated in "fuel", which currently corresponds to $\frac{1}{1000}$ of one gas. When executing rWasm bytecode, remaining gas is converted into units of fuel. After execution terminates, the remaining fuel is converted back into units of gas, and then charged inside the Fluent VM.

Code Assessment. The Fluentbase developers provided the source code of the Fluentbase for the code review. The codebase consisted of two repositories. The first is revm-rwasm[§], a fork of revm v82 adapted for rWasm integration. The second is Fluentbase[¶] a codebase developed almost entirely with original code from the Fluent team. The only small exception is the reuse of cryptographic syscall handlers adapted from SP1's hook design and implementations, with local changes to interfaces, memory layout, and error handling.

The project contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the Fluentbase developers provided links to documentation[‖] [**] and met regularly with the analysts.

The source code contained a test suite. This test suite leverages the testing done by revm to ensure backwards compatibility with Ethereum. Most of the unit testing is performed within the (out of scope) `sdk/` directory for custom contracts and `contracts/` directory for precompiles. The SDK testing validates things like serialization/ABI handling, storage correctness, and compilation of Rust crates to rWasm. The precompile testing focuses on correctness.

However, the Veridise security analysts also noted that several of the syscalls were untested at the beginning of the audit, though additional testing occurred in parallel to the audit effort. While integration testing was present, complex interactions between rWasm and EVM contracts does not seem to be tested. For example, EIP-7702 delegation to rWasm contracts or call patterns involving multiple rounds of back-and-forth between contracts from multiple VMs was not tested. Additionally, careful testing of gas metering in various scenarios (non-reverting execution, reverting execution which is handled, reverting execution which is unhandled, etc.) was untested at the beginning of the review.

[§] https://github.com/fluentlabs-xyz/revm-rwasm/tree/devel
[¶] https://github.com/fluentlabs-xyz/fluentbase/tree/devel
[‖] https://book.gblend.xyz/introduction/
[**] https://hackmd.io/@dmitry123/r18QhKZ7ll

During the security assessment, the Fluent developers made several functional changes to the code. This was to perform required updates to the `runtime` crate. Veridise analysts did not begin review of the `runtime` crate until these changes were finalized, but did review the `revm` crate under commit `9ed9bd51` while these changes were developed.

Summary of Issues Detected. The security assessment uncovered 37 issues, 19 of which are assessed to be of high or critical severity by the Veridise analysts.

The most common issue came from unhandled panics within host functions and the Fluent VM. This includes issues V-FNT-VUL-002, V-FNT-VUL-003, V-FNT-VUL-004, V-FNT-VUL-006, V-FNT-VUL-007, V-FNT-VUL-008, V-FNT-VUL-011, and V-FNT-VUL-012. Other serious issues include denial of service achieved by triggering unbounded memory allocation (V-FNT-VUL-001, V-FNT-VUL-013, V-FNT-VUL-015), infinite loops (V-FNT-VUL-009), or failing to charge/under-charging gas (V-FNT-VUL-016, V-FNT-VUL-019). Additionally, Veridise analysts identified invocation of the wrong function during a syscall (V-FNT-VUL-005), returning the incorrect value from a syscall (V-FNT-VUL-010, V-FNT-VUL-017, V-FNT-VUL-018), and dropping incrementally produced smart contract output (V-FNT-VUL-014). Finally, some wasm binaries trigger an underflow in the out of scope rWasm interpreter (V-FNT-APP-VUL-001).

Two of these issues were already known to the developers, and included in the report for full transparency and to ensure they reached a resolution (V-FNT-VUL-013, V-FNT-VUL-015).

The Veridise analysts also identified 6 medium-severity issues. These include a `CREATE` path that can be frontrun (V-FNT-VUL-020), a static-call bypass in metadata syscalls that lets state-changing operations slip through read-only entry points (V-FNT-VUL-022), a `BLOCKHASH` handler that returns zero on host error, risking logic divergence (V-FNT-VUL-023), and a failure mode that can strand execution or enable arbitrary code execution in the future (V-FNT-VUL-024).

Finally, the Veridise team identified 8 low-severity issues and 4 warnings.

All 37 issues have been acknowledged and fixed by the Fluent. The only partial exception to this is issue V-FNT-VUL-001, which is marked as partially fixed. During review, the Veridise analysts believe the provided fix removed this issue from the repository. However, it also introduces large architectural changes which require a deeper analysis beyond the scope of this fix review to validate absence of newly added denial-of-service vulnerabilities. Several other architectural changes were required to resolve the raised issues, such as V-FNT-VUL-016 and V-FNT-VUL-024, which led to a large number of changes to the system as a whole during the fix review process.

Following the review, the project was rebased onto revm v102[††]. Additionally, the Fluent team implemented a further recommendation discussed with the Veridise team: enabling upper-rounding for fuel denomination charging[‡‡].

Some implementation details initially identified as issues have been determined to be intended behavior or out of scope after discussions with Fluent. These can be found in Appendix A.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Fluentbase. During the fix review, many of these suggestions were implemented, including the gas accounting redesign and many notable testing improvements.

[††] https://github.com/fluentlabs-xyz/fluentbase/pull/248/files
[‡‡] https://github.com/fluentlabs-xyz/fluentbase/pull/207

Avoiding Panics. A large number of serious issues came from the usage of dangerous APIs like `unwrap()` and `expect()`, which trigger panics. These panics can be used to crash a node after a long computation, providing a denial of service vector to arbitrary users. Dangerous APIs like the above should be avoided whenever possible. Introduction of static analysis such as the Semgrep rules provided in Appendix B could prevent the introduction of similar issues in the future. Additionally, syscalls should be tested for both happy and unhappy cases to ensure there are no crashes.

Improve Testing. As mentioned above, the Veridise analysts assessed portions of the protocol to be insufficiently tested. Importantly, each syscall should be tested. Tests should include happy cases, in which inputs match their expected values. They should also include unhappy cases, checking what happens when each input's size and/or value is out of the expected range. These tests should check that the function does not panic, validate the returned value and exit code, and check the expected gas usage.

Additionally, the gas metering of wasm contracts should be more thoroughly tested. Tests which check gas metering on successful executions, reverting executions, successful executions which contain reverting sub-executions, and executions involving a large number of contracts from different VMs should be tested.

Contracts which stress resource usage (especially calls, storage, and memory) should be included in the test suite. These should compare system resources to charged gas/fuel to ensure pricing is appropriate.

Align Threat Model for Ported SP1 Code. Much of the panic/unwrap usage originates from SP1 code written for a trusted executor. When deployed in an adversarial runtime, all inputs must be treated as untrusted. Audit any imported/copied-and-modified modules for trust assumptions, replacing assertions with explicit error returns, and normalizing failures into deterministic traps.

Redesign Gas Accounting. The Veridise analysts believe the developers should redesign the gas accounting logic to make reasoning about the remaining gas left in the executing frame easier to justify and reason about. Currently, the system relies on three different gas counters: a gas counter used by the Fluent VM used in revm-rwasm functions, a gas counter tracked within the rWasm store, and a third gas counter stored in an ambient struct called the `ExecutionResult`. Synchronizing these three results and properly handling out of gas errors is of critical importance. However, it currently requires reasoning about four different cases: when fuel is manually accounted by a system contract, when fuel is charged automatically in an untrusted deployed smart contract, when work is done within a syscall called by a trusted contract, and when work is done within a syscall invoked by an untrusted contract. Each of these cases must synchronize all three counters and handle error cases appropriately.

The Veridise analysts believe this complexity led to severe issues such as V-FNT-VUL-012 and V-FNT-VUL-016. As part of the fix review, Veridise team is reviewing a large refactor which should address this point. However, large changes to such core accounting logic during a time-boxed fix review indicate a need for additional testing and a lack of stability in the codebase.

Simplifying Data Flow. Much of the interfacing done between different parts of the codebase is performed by setting values in the appropriate global variables or ambient contexts. Currently the control flow taken when executing a single `CALL` instruction touches a large swath of the repository. Important invariants require reasoning simultaneously about the frame creation performed in revm-rwasm, the execution runtime defined over revm-rwasm, the `RuntimeFactoryExecutor` instance acting as the rWasm host, and the various different VM implementations. Where

possible, replacing this with explicit APIs which pass data back and forth between modules would make the system easier to reason about, and clarify the expected flow of data.

Additional Testing / Reviews. Issues such as V-FNT-VUL-002, V-FNT-VUL-003, V-FNT-VUL-004, V-FNT-VUL-006, V-FNT-VUL-007, V-FNT-VUL-008, V-FNT-VUL-011, and V-FNT-VUL-012 relate to under-testing of the syscalls, as discussed above. Other severe issues such as V-FNT-VUL-010, V-FNT-VUL-017, V-FNT-VUL-018, and V-FNT-VUL-014 could likely have been caught by tests. As mentioned in the above recommendations, the Veridise analysts believe the developers should allocate dedicated time to thoroughly test all of the syscalls.

Tracking and identifying these issues required a large amount of time during the review, leaving less time for dedicated attacks against the runtime logic itself. Once the codebase and runtime have reached stability, additional testing and security review may further increase the security and resilience of this system.

Disclaimer. Given the complexity of the Fluentbase, the size of the code base, and the scope of the proposed changes, the Veridise team cannot make guarantees about the absence of high/critical issues in the protocol beyond the 19 identified. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Fluentbase	`9ed9bd51 - e88ea57`	Rust	Fluent
revm-rwasm	`5e3b547`	Rust	Fluent

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 22–Oct. 27, 2025	Manual & Tools	2	10 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	12	12	11
High-Severity Issues	7	7	7
Medium-Severity Issues	6	6	6
Low-Severity Issues	8	8	8
Warning-Severity Issues	4	4	4
Informational-Severity Issues	0	0	0
TOTAL	37	37	36

Table 2.4: Category Breakdown.

Name	Number
Logic Error	12
Denial of Service	10
Data Validation	7
Maintainability	3
Cryptographic Vulnerability	1
Frontrunning	1
Logic Error,Denial of Service	1
Authorization	1
Usability Issue	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Fluentbase's source code. During the assessment, the security analysts aimed to answer questions such as:

- Does runtime/bytecode identification resist spoofing so RWasm vs. legacy EVM cannot be misrouted?
- Does the Fluent VM dispatch to the correct runtime for `CALL`/`CREATE`, including during type flips, staging, and delegation edge cases?
- Does EIP–7702 delegation enforce correct runtime selection without unauthorized switching, cycles, or replay?
- Is `OwnableAccountBytecode` creation possible through unintended channels?
- Are panics unreachable from untrusted inputs, with traps/quotas/timeouts preventing resource-exhaustion DoS?
- Is gas↔ fuel accounting consistent, preventing free work or double charge?
- When `disable_fuel` is used, is work still bounded and is fuel accounted accurately across pauses and resumes?
- Can refunds/remaining fuel be manipulated across exec/resume/syscall paths, and are reconciliation caps/floors in place?
- Is `is_gas_free` constrained to prevent free invocation of expensive precompiles?
- Does fuel usage scale with payload/complexity in line with EVM expectations, with divergences bounded and documented?
- Do frame init/resume checks prevent state corruption or isolation breaks under interrupts?
- Do `EthFrame.process_next_action()` and `EthFrame.clear()` guarantee atomic, correctly ordered state transitions under interrupts, preventing double-commit, torn state, or use-after-free of frame data?
- Are critical invariants (stack depth/shape, PC monotonicity, call depth, reentrancy) enforced with deterministic traps on violation?
- Are `call_id` values unique, bounded, and non-reusable to prevent replay or confused-deputy issues?
- Do syscall validators enforce arity/types/side-effects so untrusted inputs cannot reach unsafe host paths?
- Are memory bounds checked and fuel precharged before unbounded allocations to prevent OOB and allocation DoS?
- Do cache integrity/eviction rules prevent stale-code execution, cross-tenant leakage, or unbounded growth?
- Are multi-region syscalls atomic with rollback on partial failure to prevent torn state?
- Do cryptographic syscalls enforce canonical encodings, range/subgroup checks to prevent malleability?
- Are error returns explicit and unambiguous?
- Do any intentional EVM divergences create consensus or safety risk, and are they gated and documented with mitigations?
- Are sentinel values and signed/unsigned conversions validated to prevent bounds-bypass or logic errors?

- Does applying EIP–7702 to Ownable accounts preserve isolation and prevent privilege escalation?
- Are exec/resume orderings constrained so adversaries cannot violate single-step progress or re-execute effects?
- Are race conditions across caches/state updates ruled out or made atomic under concurrency?
- What observability hooks exist (inspector scope, metrics, logs), and are they sufficient for security and performance diagnostics?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a thorough review of the source code by human experts. To identify potential common vulnerabilities, security analysts leveraged static analyzers and the open-source tool `semgrep`, see Appendix B.

Scope. The scope of this security assessment is limited to the following locations:

For the repository at `https://github.com/fluentlabs-xyz/revm-rwasm/tree/devel`, the scope encompassed all files within `crates` that have been modified since version `v82`.

For the repository at `https://github.com/fluentlabs-xyz/fluentbase/tree/devel`, the scope comprised the directories `crates/runtime` and `crates/revm`, with the following files explicitly excluded:

- `crates/revm/Cargo.toml`
- `crates/revm/README.md`
- `crates/runtime/Cargo.toml`
- `crates/runtime/README.md`
- `crates/runtime/src/executor/global_executor.rs`
- `crates/runtime/src/runtime/wasmtime_runtime.rs`
- `crates/runtime/src/utils/testing_store.rs`

Methodology. Veridise security analysts reviewed the reports of previous audits for Fluentbase, inspected the provided tests, and read the Fluentbase documentation*†.

They then began a review of the code assisted by both static analyzers and automated testing. During the security assessment, the Veridise security analysts regularly met with the Fluentbase developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

The likelihood of a vulnerability is evaluated according to the Table 3.2.

The impact of a vulnerability is evaluated according to the Table 3.3:

* `https://book.gblend.xyz/introduction/`
† `https://hackmd.io/@dmitry123/r18QhKZ7ll`

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Trust Model

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Fluentbase.

- **Interface exclusivity.** Guest code interacts with the host solely via the documented syscall/interrupt ABI. The `Exec` interrupt mediates access to chain state (e.g., `CALL`/`CREATE`, storage, logs) and execution resumes only through the corresponding resume path. No alternative channels exist to read or mutate host memory.
- **Gas/fuel metering.** For native rWasm, metering is compiler-inserted (`fuel`) and enforced at runtime; for external VM emulation, metering is provided by trusted system bytecode. Charges occur *before* any host-visible effects (storage, logs, return data). No control-flow path may invoke host functionality without passing through the metering prologue. The compilation performed during deployment statically validates for required instrumentation (e.g., presence and reachability of fuel prologues and absence of direct, unmetered syscall entry), and any module that omits, violates, or attempts to bypass metering is *rejected*.
- **Value-stack arity and post-growth consistency.** At every dynamic call site, the entry stack height must meet the callee's parameter arity. Whole-module validation restricts control-flow targets to typed boundaries so no path can reach a call without its operands. The runtime maintains a single source-of-truth stack index and, after any buffer growth, resynchronizes all derived cursors before executing the next instruction. If a call is encountered with insufficient operands, execution must raise a deterministic `StackUnderflow` trap.
- **Trusted roles.** It is assumed that the privileged roles behave only to upgrade system contracts according to system-wide forks.

4.2 Privileged Roles

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity. *Highly-privileged* roles may have a critical impact on the protocol if compromised, while *limited-authority* roles have a negative, but manageable impact if compromised. Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assumed that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, non-emergency roles:
  - `UPDATE_GENESIS_AUTH` is a privileged address allowing deployment of arbitrary code to arbitrary addresses, see V-FNT-VUL-031.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-FNT-VUL-001	DoS via memory allocation before gas charge	Critical	Partially Fixed
V-FNT-VUL-002	Expect in decompress causes node to panic	Critical	Fixed
V-FNT-VUL-003	Panic on out-of-bounds slicing	Critical	Fixed
V-FNT-VUL-004	Write FD syscall can crash node	Critical	Fixed
V-FNT-VUL-005	Secp256r1 decompress syscall uses wrong . . .	Critical	Fixed
V-FNT-VUL-006	Users can force panic in Weierstrass . . .	Critical	Fixed
V-FNT-VUL-007	Users can crash node via Weierstrass curve . . .	Critical	Fixed
V-FNT-VUL-008	Users can crash node via underflow in . . .	Critical	Fixed
V-FNT-VUL-009	Square root syscall can trigger infinite loop	Critical	Fixed
V-FNT-VUL-010	Weierstrass decompressor returns . . .	Critical	Fixed
V-FNT-VUL-011	Weierstrass decompression syscall always . . .	Critical	Fixed
V-FNT-VUL-012	Unsafe unwrap in resume	Critical	Fixed
V-FNT-VUL-013	Code copy ignores offset/length	High	Fixed
V-FNT-VUL-014	Contract output wiped during interrupt	High	Fixed
V-FNT-VUL-015	Memory leak in module caching allows DoS	High	Fixed
V-FNT-VUL-016	Syscall OOG not propagated	High	Fixed
V-FNT-VUL-017	Wrong index in syscall_write_fd_handler . . .	High	Fixed
V-FNT-VUL-018	Elliptic curve x-coordinate interpreted as . . .	High	Fixed
V-FNT-VUL-019	Fuel undercharge leading to stall	High	Fixed
V-FNT-VUL-020	CREATE syscall may be DoS'ed	Medium	Fixed
V-FNT-VUL-021	EIP-7702 delegation to ownable accounts . . .	Medium	Fixed
V-FNT-VUL-022	Static-call bypass in metadata syscalls	Medium	Fixed
V-FNT-VUL-023	BLOCKHASH may return zero on host . . .	Medium	Fixed
V-FNT-VUL-024	Failure before resume could lead to DoS . . .	Medium	Fixed
V-FNT-VUL-025	Write FD assumes elements are reduced	Medium	Fixed
V-FNT-VUL-026	Floor gas may be bypassed	Low	Fixed
V-FNT-VUL-027	Unsafe Offset/Length Handling	Low	Fixed
V-FNT-VUL-028	Unbounded metadata write via unchecked . . .	Low	Fixed
V-FNT-VUL-029	Bytecode hash unwrap prior to validation . . .	Low	Fixed
V-FNT-VUL-030	rWasm Deployment may get overwritten . . .	Low	Fixed
V-FNT-VUL-031	Trusted address can overwrite code	Low	Fixed
V-FNT-VUL-032	RSA Quotient incorrectly truncated	Low	Fixed
V-FNT-VUL-033	Return data unchanged when . . .	Low	Fixed
V-FNT-VUL-034	Interpreter state carryover when invoking . . .	Warning	Fixed
V-FNT-VUL-035	call_id may overflow	Warning	Fixed
V-FNT-VUL-036	SVM ELF Magic bytes form valid EVM . . .	Warning	Fixed
V-FNT-VUL-037	Maintainability	Warning	Fixed

5.1 Detailed Description of Issues

5.1.1 V-FNT-VUL-001: DoS via memory allocation before gas charge

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Partially Fixed
Location(s)	`crates/runtime/src/syscall_handler/` ▶ `hashing/` • `keccak256.rs:17` • `sha256.rs:18` ▶ `host/` • `debug_log.rs:18` • `exec.rs:72` • `resume.rs:25` • `write_fd.rs:37`		
Confirmed Fix At	`pull/219throughac9d9db`		

The exec syscall handler reads in syscall input from the application. It does this based on a caller-controlled `input_len`

```
let (hash32_ptr, input_ptr, input_len, fuel16_ptr, state) = (
       params[0].i32().unwrap() as usize,
       params[1].i32().unwrap() as usize,
       params[2].i32().unwrap() as usize,
       params[3].i32().unwrap() as usize,
       params[4].i32().unwrap() as u32,
   );
// ... ELIDED
let mut input = vec![0u8; input_len];
caller.memory_read(input_ptr, &mut input)?;
let input = Bytes::from(input);
```

Snippet 5.1: Snippet from `syscall_exec_handler()`

Gas for the syscall is not charged until later, in the `execute_rwasm_interruption()` function. In particular, no upper bound or pre-charging is done on `input_len` before allocating and copying that many bytes from guest memory. In particular, adversaries can force a node to allocate up to 4 GiB of memory before a syscall is invoked.

Impact By passing large input lengths, an attacker can cause a node to run out of memory and crash. This could lead to the shutdown of a large number of nodes, stopping operation of the system as a whole.

Similar issues exist in several other syscalls (flagged in the locations section of the issue).

Recommendation Do not read in memory until the size has been validated by a syscall. Provide a handle to the `execute_rwasm_interruption()` function which allows it to read input after the size has been validated to match its expected value.

In general, validate that any memory reads will not go out of bounds *before* allocating a buffer.

Developer Response The developers now check the size before reading from memory. To do this, they now moved the interruption system out of the runtime, requiring system contracts to not exit/trap, but instead return interruptions directly.

Updated Veridise Response The changes resolve the described issue. However, given the magnitude of the changes, it seems likely that other issues may have been introduced. While the analysts have not been able to identify any concrete issues at this time, it may be beneficial to introduce defense-in-depth approaches to minimize future risk.

One beneficial approach would be to replace newly added `assert!`s and `expect()`s with a special case that reverts the entire transaction without panicking. This would prevent logical errors about code reachability from becoming denial of service vectors. While the Fluent system has only one node implementation, this will not lead to consensus errors.

Down the road, as the code stabilizes, these revert special cases can be removed.

Updated Developer Response I don't think we can safely mine this transaction even with UnrechableCodeReached, because we need to make sure that transaction is fully rolled-back that can be an attack vector. Honestly, both solutions don't work, because we should guarantee that trap code or unreachable really never happen during execution. No matter what path we choose, an attacker can do the following: find a multi-send contract, add his address into distribution list (inside a pool or some distribution service) and during execution cause trap code. If a contract don't have infinite gas execution then such transaction won't be ever mined. So maybe assert or external fatal error it better here, because at least we can identify an issue and fix as soon as it possible to make sure it never happens.

PoC The below proof of concept shows an attack contract which can be used to force a node to allocate a large amount of memory. On the analyst's local machine, passing a `LARGE_LEN` of twice the one shown below causes the program to crash with out-of-memory. With the provided setting, the "large call" takes 30x longer to execute for the same amount of gas.

```
use std::time::Instant;

use crate::EvmTestingContextWithGenesis;
use fluentbase_sdk::{calc_create_address, Address, Bytes};
use fluentbase_testing::{EvmTestingContext, TxBuilder};
use wat::parse_str;

const EXEC_BALANCE_DOS_WAT: &str = r#"
    (module
        (import "fluentbase_v1preview" "_read" (func $_read (param i32 i32 i32)))
        (import "fluentbase_v1preview" "_exec" (func $_exec (param i32 i32 i32 i32
    i32) (result i32)))
        (import "fluentbase_v1preview" "_exit" (func $_exit (param i32)))
        (memory 1)
        (data (i32.const 64)
                  "\00\00\00\00\00\00\00\00"
                  "\00\00\00\00\00\00\00\00"
                  "\00\00\00\00\00\00\00\0b"
                )
        (func $main (local $len i32)
            ;; Read the caller-supplied length into memory[0..4].
```

```
            ;;
            ;; Note that the first 1024 bytes are encoded context from the system, so we need to load from byte 1024
            ;; https://github.com/fluentlabs-xyz/fluentbase/blob/e88ea5712c2eb568a6cd9c8946db48064de41ab0/crates/revm/src/executor.rs#L154-L158
            i32.const 0    ;; target ptr
            i32.const 1024 ;; offset
            i32.const 4    ;; length
            call $_read

            i32.const 0
            i32.load
            local.set $len

            ;; Provision a large memory buffer up-front so both test cases have identical growth costs.
            i32.const 512
            memory.grow
            drop

            ;; Invoke the BALANCE syscall via _exec with an attacker-controlled input length.
            i32.const 64      ;; pointer to SYSCALL_ID_BALANCE
            i32.const 128     ;; input pointer
            local.get $len    ;; attacker-chosen length
            i32.const 0       ;; no explicit fuel limit
            i32.const 0       ;; STATE_MAIN
            call $_exec
            drop

            ;; Exit without bubbling up the nested error.
            i32.const 0
            call $_exit
        )
        (export "main" (func $main))
        (export "memory" (memory 0))
    )
"#;

fn deploy_exec_balance_contract(ctx: &mut EvmTestingContext) -> Address {
    let wasm = parse_str(EXEC_BALANCE_DOS_WAT).expect("invalid wat");
    let deployer = Address::ZERO;
    let deploy_result = TxBuilder::create(ctx, deployer, wasm.into())
        .gas_price(0)
        .gas_limit(50_000_000)
        .exec();
    assert!(
        deploy_result.is_success(),
        "failed to deploy exec test contract: {deploy_result:?}"
    );
    calc_create_address(&deployer, 0)
}

fn call_with_len(
    ctx: &mut EvmTestingContext,
    contract: Address,
    len: u32,
```

```
) -> revm::context::result::ExecutionResult<fluentbase_revm::RwasmHaltReason> {
    let calldata = Bytes::from(len.to_le_bytes().to_vec());
    TxBuilder::call(ctx, Address::ZERO, contract, None)
        .gas_price(0)
        .gas_limit(21_576)
        .input(calldata)
        .exec()
}

#[test]
fn exec_balance_accepts_unbounded_inputs_without_gas_cost() {
    let mut ctx = EvmTestingContext::default().with_full_genesis();
    let contract = deploy_exec_balance_contract(&mut ctx);

    // Ensure gas cost is independent of input length.
    const SMALL_LEN: u32 = 32;
    // 1 GiB of unmetered input copying
    // Changing this to 2 GiB causes a panic due to OOM
    const LARGE_LEN: u32 = 2 * 512 * 1024 * 1024;

    let small = call_with_len(&mut ctx, contract, SMALL_LEN);
    assert!(
        small.is_success(),
        "baseline call unexpectedly failed: {small:?}"
    );

    let large = call_with_len(&mut ctx, contract, LARGE_LEN);

    assert_eq!(
        small.gas_used() + 1,
        large.gas_used(),
        "gas accounting should not depend on the copied input size"
    );

    // Now do timings
    const WARMUP_ITERS: u32 = 10;
    const MEASURE_ITERS: u32 = 20;
    for _ in 0..WARMUP_ITERS {
        let _ = call_with_len(&mut ctx, contract, SMALL_LEN);
    }
    let start_small = Instant::now();
    for _ in 0..MEASURE_ITERS {
        let _ = call_with_len(&mut ctx, contract, SMALL_LEN);
    }
    let elapsed_small = start_small.elapsed() / MEASURE_ITERS;

    for _ in 0..WARMUP_ITERS {
        let _ = call_with_len(&mut ctx, contract, LARGE_LEN);
    }
    let start_large = Instant::now();
    for _ in 0..MEASURE_ITERS {
        let _ = call_with_len(&mut ctx, contract, LARGE_LEN);
    }
    let elapsed_large = start_large.elapsed() / MEASURE_ITERS;

    println!("small call: {:?}, large call: {:?}", elapsed_small, elapsed_large);
}
```

5.1.2 V-FNT-VUL-002: Expect in decompress causes node to panic

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/[...]/edwards_decompress.rs:44`		
Confirmed Fix At	https://github.com/fluentlabs-xyz/fluentbase/pull/213, f68cc91		

The Edwards decompression syscall calls `decompress(...).expect(...)`. Any invalid input makes decompress return `None`, which triggers a panic in the syscall when `expect()` is called.

```
1 let compressed_y = CompressedEdwardsY(compressed_edwards_y);
2 let decompressed = decompress(&compressed_y).expect("curve25519 Decompression failed"
      );
```

Snippet 5.2: Snippet from `syscall_ed25519_decompress_impl()`

Impact Untrusted callers can trigger a panic and abort the VM execution, leading to denial of service for the node or runtime.

Recommendation Eliminate panics. Return a defined `ExitCode` on failure.

Developer Response The developers implemented the recommendation.

5.1.3 V-FNT-VUL-003: Panic on out-of-bounds slicing

Severity	Critical	**Commit**	e88ea57
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/host/` ▶ `forward_output.rs:31` ▶ `read_input.rs:27`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/223`, 2efcbee		

Multiple host-side syscall handlers accept guest-controlled parameters used as lengths and offsets. Before indexing into an array, a bounds check is performed. As shown in the below snippet, the syscall checks that `offset + len <= return_data.len() as u32`. However, `offset + len` can overflow, passing the check as a `u32`, but triggering a panic when used as a `usize` during indexing.

```
1 if offset + len <= ctx.execution_result.return_data.len() as u32 {
2         let ret_data = &ctx.execution_result.return_data
3             [(offset as usize)..(offset as usize + len as usize)]
4             .to_vec();
```

Impact Denial of Service: the host process can panic on out-of-bounds slices, terminating the execution environment. As these inputs are controlled by the smart contract application, any user may abuse this panic to shut down nodes on the network.

Recommendation Validate bounds in the `usize` domain using checked arithmetic.

Developer response The developers implemented the bounds check using checked arithmetic, adding tests to validate the error case does not trigger a panic.

5.1.4 V-FNT-VUL-004: Write FD syscall can crash node

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/host/write_fd.rs`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/231`, b2e9bf5		

Description The syscall write-fd hook dispatcher and cryptographic hooks allow untrusted callers to pass arbitrary input. Multiple sites use `assert!()`, `unwrap()`, `expect()`, unchecked slicing, and `unimplemented!()`, all of which will panic on malformed inputs. A malicious transaction can therefore crash any node by executing these hooks.

Usage patterns which may cause panics include:

1. Indexing/slicing into buffers without length checks:
   - `hook_ed_decompress()`:
     - `let mut bytes: [u8; 32] = buf[..32].try_into().unwrap();` indexes into a buffer with no length check.
   - `bls::hook_bls12_381_sqrt()` / `bls::hook_bls12_381_inverse()`:
     - `let field_element = BigUint::from_bytes_be(&buf[..48]);` indexes into a buffer with no length check.
2. Unwrapping/expecting values which may return `None`:
   - `ecrecover::handle_secp256k1`:
     - `K256FieldElement::from_bytes(...).unwrap();` appears twice, unwrapping a value which will be `None` if the bytes are not canonical.
     - `K256Scalar::from_repr(...).unwrap();` also assumes bytes are canonical.
   - `ecrecover::handle_secp256r1`:
     - `P256FieldElement::from_bytes(...).unwrap();` appears twice, each time assuming bytes are canonical.
     - `P256Scalar::from_repr(...).unwrap();` also assumes bytes are canonical.
     - `handle_secp256k1()` / `handle_secp256r1()`:
       - `invert().expect("Non zero r scalar")` both functions panic if the user-provided parameter `r` is zero.
3. Panicking arithmetic preconditions
   - `handle_secp256k1()`: `assert!(!bool::from(alpha.is_zero()))` panics if the user-provided parameter `alpha` is zero.
   - `fp_ops::hook_fp_inverse`: `assert!(!element.is_zero())` also panics if the user-provided value is zero.
   - `fp_ops::legendre_symbol`: `assert!(!element.is_zero())` also panics if the user-provided value is zero, although this code point is not reached directly since `hook_fp_sqrt()` handles the zero-case specially.
4. Unknown curve-id
   - `hook_ecrecover()`: `unimplemented!()` triggers a panic on an unknown (user-supplied) curve id.

5. Divide-by-zero
   - `hook_rsa_mul_mod`: `prod.div_rem(&m)` will panic when the user-provided `m` is zero.
   - `fp_ops::{hook_fp_inverse, sqrt_fp, tonelli_shanks, legendre_symbol}`:
     - multiple `modpow(..., &modulus)` and divisions will panic if the user-provided `modulus` is zero.
6. Assumption `nqr` is not a quadratic residue:
   - `hook_fp_sqrt()` assumes that `nqr` is a non-quadratic residue, unwrapping the square root of `nqr * element` in the case that `element` is not a quadratic residue. However, `nqr` is user-supplied, and could be a quadratic residue, leading to a panic on the `unwrap()`.
7. Vector API misuse
   - `ecrecover::handle_secp256k1` and `handle_secp256r1`:
     - `let mut result = vec![0x1];` followed by `result.copy_from_slice(...)` will trigger a panic, as the copy method aborts when there is a length mismatch
       - It should instead use `extend_from_slice`.

Impact Any user can submit a crafted syscall that triggers a panic, halting the node's execution thread. This enables chain-wide denial of service, as validators or full nodes crash when executing malicious transactions that reach these hooks.

Recommendation Apply a consistent "no-panic" policy for all untrusted inputs and propagate `ExitCode::MalformedBuiltinParams` (or a specific error) on failure.

Concrete hardening steps must include:

Input length and parsing

- Check buffer lengths up front; return an error if insufficient.
  - `ed_decompress`: require `buf.len() >= 64`.
  - `bls12-381` hooks: require `buf.len() >= 48`.
  - `rsa_mul_mod`: require `buf.len() == 768`.
  - `fp_ops` hooks: require `buf.len() == 4 + 2*len` or `4 + 3*len` as applicable.
- Replace all `.unwrap()`/`.expect()` on parsing with `.ok_or(ExitCode::MalformedBuiltinParams)?`.

Field element canonicality

- Before `from_bytes`/`from_repr`, verify the integer is `< modulus/order`.
  - For k256/p256 field elements and scalars, reject non-canonical encodings.

Arithmetic preconditions

- Replace `assert!`/`expect` with checked branches returning an error:
  - `alpha.is_zero()` -> error.
  - `r == 0` before `invert()` -> error.
  - `element == 0` in `fp_inverse`/`legendre_symbol` -> error.
- Validate modulus:

- `modulus > 1`, odd (and prime if required by algorithm).
- Reject `modulus == 0` early to avoid `modpow` panics.

Sqrt / NQR handling

- Never `unwrap()/expect()` on square roots. Use:
  - If `sqrt(x)` returns `None`, compute `sqrt(nqr*x)` only if `nqr` is verified to be a non-residue; if that also fails, return error.
- For generic FP ops, either:
  - Require a proven `nqr` (protocol constant), or
  - Derive a valid non-residue deterministically and verify it at runtime; otherwise return error.

Result construction

- Replace `copy_from_slice` into `result` with `extend_from_slice` to avoid length mismatch panics in `handle_secp256k1/r1`.

Control flow

- Replace `unimplemented!()` in `hook_ecrecover` with `return Err(ExitCode::MalformedBuiltinParams)` on unknown curve id.

RSA modular reduction

- Validate `m != 0` before `div_rem`.
- Optionally require `m` to be odd and set a maximum bit length.
- Return error on zero or oversized modulus.

General approach

- Audit all `assert!`, `unwrap()`, `expect()`, and unchecked slices; replace with explicit validation + error returns.
- Add tests for each hook to ensure malformed inputs return errors without panics.
- Document accepted encodings and bounds for each hook input.

Developer Response The developers removed the `write_fd` syscall.

5.1.5 V-FNT-VUL-005: Secp256r1 decompress syscall uses wrong function

Severity	Critical	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler.rs:94`		
Confirmed Fix At	65d9433		

The `invoke_runtime_handler` function dispatches syscalls based on the `SysFuncIdx` enum and maps them to the corresponding handler. However, for the `SysFuncIdx::SECP256R1_DECOMPRESS` case, it incorrectly routes to `syscall_secp256k1_decompress_handler` instead of `syscall_secp256r1_decompress_handler`.

```
1 SysFuncIdx::SECP256R1_DECOMPRESS => syscall_secp256k1_decompress_handler(caller,
      params, result),
```

This is a misrouting bug that causes secp256r1 decompression operations to be handled by logic meant for a different curve (secp256k1), leading to invalid point interpretation or failure of cryptographic operations that rely on secp256r1.

Impact Requests to decompress secp256r1 points are handled incorrectly using the secp256k1 decompression logic. This can cause:

- Invalid or failed decompression of public keys
- Inability to verify secp256r1 signatures or perform associated operations
- Misleading execution results without explicit error signaling

This may affect any protocol logic relying on correct support for secp256r1.

Recommendation Route `SysFuncIdx::SECP256R1_DECOMPRESS` to the correct handler function (`syscall_secp256r1_decompress_handler`) to ensure curve-specific operations are processed accurately.

Developer Response The developers implemented the recommendation.

5.1.6 V-FNT-VUL-006: Users can force panic in Weierstrass decompression syscall

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/[...]/weierstrass_decompress.rs:151-155`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/222`, 7fd40e9		

The Weierstrass elliptic curve decompression syscalls in the Fluent blockchain invoke `syscall_weierstrass_decompress_impl`, which performs elliptic curve point decompression based on compressed x-coordinates and sign bits. These syscalls include:

- `syscall_secp256k1_decompress_handler`
- `syscall_secp256r1_decompress_handler`
- `syscall_bn254_decompress_handler`
- `syscall_bls12381_decompress_handler`

Each of these ultimately dispatches to a generic handler:

```
fn syscall_weierstrass_decompress_impl<...>(
    x_bytes: [u8; COMPRESSED_SIZE],
    sign_bit: u32,
) -> Result<[u8; DECOMPRESSED_SIZE], ExitCode>
```

Inside this function, the decompression implementation performs a decompression based on the curve type:

```
let decompress_fn = match E::CURVE_TYPE {
    CurveType::Secp256k1 => secp256k1_decompress::<E>,
    CurveType::Secp256r1 => secp256r1_decompress::<E>,
    CurveType::Bls12381 => bls12381_decompress::<E>,
    _ => panic!("unsupported curve: {}", E::CURVE_TYPE),
};
```

This code path explicitly panics if an unsupported curve type is passed. In particular, the `CurveType::Bn254` type is not supported. Additionally, downstream decompression functions can also panic when decompressing an invalid point-e.g., if the point is not on the curve, if `sqrt` fails, or if `x_bytes` is malformed. This can be seen, for example, in the definition of `secp256k1_decompress()`:

```
pub fn secp256k1_decompress<E: EllipticCurve>(bytes_be: &[u8], sign: u32) ->
    AffinePoint<E> {
    let computed_point =
        k256::AffinePoint::decompress(bytes_be.into(), Choice::from(sign as u8)).
    unwrap();
```

Snippet 5.3: Snippet from `secp256k1_decompress()`

Impact Panic in a syscall leads to a host-level panic in the WASM runtime, which terminates the entire node process. This allows malformed or malicious inputs (such as an invalid compressed point) to crash the node.

This results in a Denial-of-Service (DoS) vector against Fluent nodes.

Recommendation Ensure that all elliptic curve decompression logic is panic-free. Refactor decompression implementations (`*_decompress`) to return `Result` instead of panicking. Avoid unchecked unwraps or matches on critical paths and handle unsupported curves gracefully.

Introduce comprehensive input validation for:

- Sign bit values
- x-coordinate byte length and range
- Curve membership

Add test cases with invalid inputs to confirm syscall robustness.

Developer Response The developers implemented the recommendations, though now return `(y,x)` rather than `(x,y)`.

5.1.7 V-FNT-VUL-007: Users can crash node via Weierstrass curve addition syscall

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/[...]/weierstrass_add.rs:99`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/224/`, 9b7ddde		

The elliptic curve addition syscalls in `syscall_weierstrass_add_handler` and `syscall_weierstrass_add_impl` do not handle exceptional cases, and instead rely on panicking behavior from the underlying arithmetic library.

Specifically, curve addition may panic when:

1. The input points are identical, and doubling logic is not invoked.
2. The coordinates are not reduced mod the field modulus (due to an underflow in intermediate computations).

These panics can occur at runtime in the following function:

```
let result_affine = p_affine + q_affine;
```

Which calls:

```
impl<E: WeierstrassParameters> AffinePoint<SwCurve<E>> {
    pub fn sw_add(&self, other: &AffinePoint<SwCurve<E>>) -> AffinePoint<SwCurve<E>>
    {
        if self.x == other.x && self.y == other.y {
            panic!("Error: Points are the same. Use sw_double instead.");
        }
        ...
    }
}
```

As these functions are used inside syscall handlers like `syscall_secp256k1_add_handler` and related variants, any panic will terminate execution of the WASM runtime, resulting in a full node crash.

Impactful handlers include:

- `syscall_secp256k1_add_handler`
- `syscall_secp256r1_add_handler`
- `syscall_bn254_add_handler`
- `syscall_bls12381_add_handler`

These handlers operate in a context where panics should be mapped to `TrapCode` errors instead of propagating uncontrolled.

Impact Unvalidated or malicious input to the syscall interface can cause panics during curve addition, resulting in a complete node crash. This constitutes a denial of service (DoS) vector against the Fluent runtime. A malformed point or a point doubling input (where `P == Q`) is sufficient to trigger this.

Any contract which calls these syscalls-either directly or indirectly- can crash the node.

Recommendation Ensure all arithmetic operations within syscalls are guarded against panics. Specifically:

- Perform a pre-check for equality of points to handle doubling properly or reject.
- Validate input coordinates are reduced modulo the field prime before deserialization.
- Replace panic-prone operations with Result-returning alternatives, propagating errors as `TrapCode`.

The syscall boundary should handle all malformed input as a recoverable failure, not a fatal panic.

Developer Response The developers now validate that the point coordinates are properly reduced, and that the points are not identical.

5.1.8 V-FNT-VUL-008: Users can crash node via underflow in tower subtraction

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/tower/` ▶ `tower_fp1_add_sub_mul.rs:143`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/225`, 8ac3556		

Description The Fluent runtime exposes several syscalls for performing arithmetic over Fp2 elements in the BN254 and BLS12-381 curve families. These syscalls, such as `syscall_tower_fp2_bn254_sub_handler` and `syscall_tower_fp2_bls12381_sub_handler`, provide subtraction over Fp2 values. Internally, they delegate to a shared generic handler function:

```
syscall_tower_fp2_add_sub_mul_handler::<NUM_BYTES, P, FP_FIELD_SUB>
```

The core logic of subtraction within this shared implementation is implemented as shown below:

```
(ac0 + modulus - bc0) % modulus
```

This expression assumes that `bc0` is a valid field elements, i.e., that it is less than the field modulus. However, no bounds check enforces this. If `bc0 > modulus`, the subtraction can underflow, causing a panic. Since these syscalls are accessible from arbitrary smart contracts, any contract can trigger this condition by supplying non-canonical `bc0`.

The affected syscall handlers are:

- `syscall_tower_fp2_bn254_sub_handler`
- `syscall_tower_fp2_bls12381_sub_handler`
- and the shared:
  - `syscall_tower_fp2_add_sub_mul_handler`
  - `syscall_tower_fp2_add_sub_mul_impl`

These do not validate that inputs are less than the field modulus before performing arithmetic.

The same applies to `bc1`, and to the analogous function in the Fp1 tower function.

Impact If a contract supplies a `bc0` or `bc1` input which is greater than the modulus, the subtraction logic may panic. This can crash the host runtime. Because these syscalls are user-accessible, this creates a denial of service risk: any contract can submit malicious inputs that crash the node.

Recommendation Add explicit input validation before arithmetic to ensure all field elements are less than the modulus. Reject non-canonical values early, returning a trap code.

Developer Response The developers now return an error when the right-hand side of a subtraction would cause na underflow.

PoC The below unit tests passes with the error shown below, indicating the underflow causes a panic.

```
thread 'syscall_handler::tower::tower_fp2_add_sub_mul::tests::
    test_tower_fp2_bn254_sub_panics_on_non_canonical_bc1' panicked at .cargo/registry
    /src/index.crates.io-1949cf8c6b5b557f/num-bigint-0.4.6/src/biguint/subtraction.rs
    :69:5:
Cannot subtract b from a because b is larger than a.
```

```
#[test]
#[should_panic]
fn test_tower_fp2_bn254_sub_panics_on_non_canonical_bc1() {
    const MODULUS: &str =
        "
    21888242871839275222246405745257275088696311157297823662689037894645226208583";
    let modulus = BigUint::from_str(MODULUS).unwrap();
    let non_canonical = modulus.clone() + BigUint::from(1u32);

    let _ = syscall_tower_fp2_bn254_sub_impl(
        big_uint_into_bytes(&BigUint::ZERO),
        big_uint_into_bytes(&BigUint::ZERO),
        big_uint_into_bytes(&BigUint::ZERO),
        big_uint_into_bytes(&non_canonical),
    );
}
```

5.1.9 V-FNT-VUL-009: Square root syscall can trigger infinite loop

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/[...]/write_fd.rs:521-524`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/231`, b2e9bf5		

Description The `tonelli_shanks` function is intended to compute the square root of a quadratic residue in a finite field using the Tonelli-Shanks algorithm. A critical step in the algorithm involves decomposing `modulus - 1` into the form `Q * 2^S`. This is performed using a loop:

```
let mut q = modulus - BigUint::one();
while &q % &BigUint::from(2u64) == BigUint::zero() {
    s += BigUint::from(1u64);
    q /= BigUint::from(2u64);
}
```

However, this loop can become infinite if the `modulus` is `1`. In such a case, `modulus - 1` evaluates to `0`, and `q` will remain `0` throughout the loop. Dividing 0 by 2 repeatedly will always yield 0, causing an infinite loop.

This edge case violates the function's implicit assumption that the modulus is a valid prime greater than 2.

Impact Supplying `modulus = 1` causes an infinite loop, leading to denial of service when `modulus == 1` is passed to the square root hook in the `WRITE_FD` syscall. This input is technically invalid, but it is not explicitly rejected, which leaves room for unhandled edge-case behavior.

Recommendation Add a validation check at the beginning of the function to ensure `modulus > 2` before proceeding. If the modulus is not greater than 2, return `None` or raise an error, depending on the desired API behavior.

Developer Response The developers removed the `write_fd` syscall.

5.1.10 V-FNT-VUL-010: Weierstrass decompressor returns malformed affine points

Severity	Critical	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/[...]/weierstrass_decompress.rs:160-162`		
Confirmed Fix At	`pull/234through3caf886`, `pull/222through7fd40e9`		

Function `syscall_weierstrass_decompress_impl` returns only the recovered `Y` limb padded to the field size, so callers receive `Y || 0...0` instead of the expected `X || Y` layout. The helpers that consume decompressed points (addition, doubling, etc.) split inputs in half and assume the first half is `X`, so they silently operate on malformed data.

Impact Any syscalls `syscall_secp256k1_decompress_handler`, `syscall_secp256r1_decompress_handler`, `syscall_bn254_decompress_handler` which rely on `syscall_weierstrass_decompress_handler` to decompress Weierstrass points, receive incorrect coordinates. This leads to wrong curve arithmetic (e.g., doubling, addition) and invalid cryptographic checks or proofs. This breaks parity with the expected output `X || Y` and may result in verification failures.

Recommendation Update the Weierstrass decompression flow to mirror the Edwards implementation: after recovering the affine point, populate both `X` and `Y` limbs, little-endian and zero-padded, before returning to callers.

Developer Response The developers have implemented the recommendation. They additionally added naming conventions to make clear that the returned point is `(y,x)` in little-endian, rather than `(x,y)`.

5.1.11 V-FNT-VUL-011: Weierstrass decompression syscall always panics

Severity	Critical	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/[...]/weierstrass_decompress.rs:142, 162`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/234`, 3caf886		

In the function `syscall_weierstrass_decompress_impl`, the return value is a vector of length `DECOMPRESSED_SIZE`. During the execution, the code allocates a vector of length `<E::BaseField as NumLimbs>::Limbs::USIZE * 4` and returns this value. However, the `DECOMPRESSED_SIZE != <E::BaseField as NumLimbs>::Limbs::USIZE * 4`. As a result, the return vector does not fit into the returned value and the call ends with panic.

```
fn syscall_weierstrass_decompress_impl<
    E: EllipticCurve,
    const COMPRESSED_SIZE: usize,
    const DECOMPRESSED_SIZE: usize,
>(
    ...
) -> Result<[u8; DECOMPRESSED_SIZE], ExitCode> {
    let num_bytes = <E::BaseField as NumLimbs>::Limbs::USIZE * 4;
    ...
    decompressed_y_bytes.resize(num_bytes, 0u8);
    Ok(decompressed_y_bytes.try_into().unwrap())
}
```

Impact The function `syscall_weierstrass_decompress_impl` always panics.

Recommendation Align the lengths of returned vector to declared value.

PoC The following code outputs `weierstrass_decompress | curve=Secp256k1 | field_bytes=64 | vec_len=128 | expected_raw_affine=64`, which demonstrates a mismatch of lengths between `DECOMPRESSED_SIZE` and `num_bytes`.

```
println!(
        "weierstrass_decompress | curve={:?} | field_bytes={} | vec_len={} |
    expected_raw_affine={} ",
        E::CURVE_TYPE,
        num_bytes / 2,
        decompressed_y_bytes.len(),
        DECOMPRESSED_SIZE,
    );
```

Developer Response The developers updated the Weierstrass implementation to use the correct length, inlining the SP1 implementations and also adding error handling. See also related issue Weierstrass decompressor returns malformed affine points.

5.1.12 V-FNT-VUL-012: Unsafe unwrap in resume

Severity	Critical	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/executor.rs:329`		
Confirmed Fix At	`pull/217throughbb791bf`		

A logic flaw in `RuntimeFactoryExecutor::resume` can trigger a panic when resuming execution of a runtime with insufficient fuel. The function subtracts `fuel_consumed` from the previously recorded `fuel_remaining` value without verifying that the subtraction will not underflow. When `fuel_remaining < fuel_consumed`, the use of `checked_sub(...).unwrap()` causes a panic.

The logic is shown in the below snippet. As the developer comment suggests, if the fuel consumption was checked inside the call to `resume_inner()`, then this unwrap would be safe. However, the `Result::Err` returned by `try_consume_fuel()` in this case is propagated, leading to a panic when unwrapped.

```
let resume_inner = |runtime: &mut ExecutionMode| {
    // [VERIDISE]: elided...
    if let Some(charge_fuel) = charge_fuel {
        runtime.try_consume_fuel(charge_fuel)?;
    }
    // [VERIDISE]: elided...
};
// We need to adjust the fuel limit because `fuel_consumed` should not be included into spent.
// We can safely unwrap here, because `OutOfFuel` check we did in `resume_inner`.
let fuel_remaining = fuel_remaining.map(|v| v.checked_sub(fuel_consumed).unwrap());
```

Snippet 5.4: Snippet from `resume()`

If a contract call consumes more fuel than available during a syscall, this unchecked subtraction triggers a panic, resulting in a denial-of-service condition. Since this panic occurs inside the main executor loop, the entire runtime is terminated, affecting subsequent transactions processed by the node.

Impact The practical impact is a denial-of-service (DoS) against Fluent blockchain executors, halting transaction processing for the affected node or validator. Attackers can exploit this by deploying a malicious contract that exhausts fuel and forces a panic, potentially leading to chain stalls or validator downtime.

Modifying the proof of concept in Fuel undercharge leading to stall by changing `FUEL_THRESHOLD` to be 25k will provoke this panic.

Recommendation Instead of using `unwrap()`, convert an error (if any) into an out of fuel execution result. In general, be extremely cautious with the use of dangerous APIs like `unwrap()`.

Developer Response The developers now guard against `Err(TrapCode::OutOfFuel)` before unwrapping.

5.1.13 V-FNT-VUL-013: Code copy ignores offset/length

Severity	High	**Commit**	e88ea57
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/revm/src/syscall.rs:649-697`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/208`, 74c77a3		

Inside the `SYSCALL_ID_CODE_COPY` branch, the handler parses `address`, `offset`, and `length`, but the parsed `offset` is explicitly unused, and the parsed `length` is used only to compute gas `gas::extcodecopy_cost(spec_id, code_length as usize, account.is_cold)`.

The implementation selects the full bytecode via `EthereumMetadata::code_copy(...)` for ownable account or `Bytecode::original_bytes(...)` otherwise.

The developers were already aware of this bug at the beginning of the audit, marking it in the code as an issue which needed resolution.

Impact

- Gas undercharge: Gas is charged as if `code_length` bytes are copied, but the handler returns `|code|` bytes. An attacker can set `code_length = 1` and receive the entire bytecode.
- Denial of service: Returning the whole code forces unnecessary allocation/copy, creating a low-cost way to stress memory and bandwidth on the host/engine boundary.

Recommendation Use slicing instead of cloning the entire bytecode; only allocate exactly `code_length` bytes to return.

Developer Response The developers now extract the requested number of bytes from the code, padding out with zeros as required by the specification. This is all done after charging for the requested length.

5.1.14 V-FNT-VUL-014: Contract output wiped during interrupt

Severity	High	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/executor.rs:175`		
Confirmed Fix At	https://github.com/fluentlabs-xyz/fluentbase/pull/218, 26380d2		

The `call std::mem::take` replaces `ctx.execution_result` with `Default::default()` and moves its current value into a local variable execution_result before matching `next_result`. The interruption path delegates to `handle_resumable_state`, but the moved value is not placed back into the context. Hence the buffers disappear from the suspended runtime and only `Default::default()` can be accessed after resumption.

```
fn handle_execution_result(
        &mut self,
        next_result: Result<(), TrapCode>,
        fuel_consumed: Option<u64>,
        ctx: &mut RuntimeContext,
    ) -> RuntimeResult {
        let mut execution_result = take(&mut ctx.execution_result);
        ...
        match next_result {
            ...
            Err(TrapCode::InterruptionCalled) => {
                return self.handle_resumable_state(execution_result, ctx);
            }
            ...
        }
        RuntimeResult::Result(execution_result)
    }

fn handle_resumable_state(
        &mut self,
        execution_result: ExecutionResult,
        ctx: &mut RuntimeContext,
    ) -> RuntimeResult {
        let ExecutionResult {
            fuel_consumed,
            fuel_refunded,
            ..
        } = execution_result;
        // Take resumable context from execution context
        let resumable_context = ctx.resumable_context.take().unwrap();
        if resumable_context.is_root {
            unimplemented!("validate this logic, might not be ok in STF mode");
        }
        ...
                let output = resumable_context.params.encode();
        RuntimeResult::Interruption(ExecutionInterruption {
            fuel_consumed,
            fuel_refunded,
            output,
        })
    }
```

Impact When the execution subsequently routes into `handle_resumable_state`, that function consumes the taken `ExecutionResult`, repackaging only the fuel metrics and interrupt payload. The original output never makes it back into the context. Since `execution_result.output` is not restored by `handle_resumable_state`, any output written earlier is lost.

Recommendation Restore `execution_result` in `ctx.execution_result` so that nested callers can still observe the execution artifacts.

Developer Response The developers now ensure `execution_result.output` and `execution_result.return_data` are unchanged by `handle_resumable_state()` when there is a resumable context. This ensures that, during any interruption, the output and return data are not cleared.

5.1.15 V-FNT-VUL-015: Memory leak in module caching allows DoS

Severity	High	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/module_factory.rs`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/216`, 85eb6e5		

The ModuleFactoryInner struct maintains two unbounded HashMap caches: `cached_modules` and (conditionally) `wasmtime_modules`-keyed by the code hash. These maps grow indefinitely, as no eviction or memory-bound mechanism exists. Each time a module is loaded, a new entry is created. The Fluent team is aware of this issue, and explicitly acknowledges the absence of an eviction policy.

This creates a memory leak vulnerability, since the system continues to accumulate entries for every distinct code hash processed. In environments that load arbitrary or attacker-supplied WASM code, such as smart contract or plugin systems, this behavior allows untrusted users to cause unbounded memory growth.

Since entries are never removed, the process heap grows until exhaustion.

Impact This issue leads to denial of service (DoS) through resource exhaustion. Attackers can upload or trigger loading of many unique WASM modules, consuming memory and eventually crashing the node or degrading performance.

This will require them to pay the cost of multiple cold loads. However, with the size of an RWasm contract set to 2 MiB, after a few thousand calls to, e.g. `CODESIZE`, multiple GiB may be forced into application memory.

Recommendation Implement a bounded cache with eviction. Use an LRU or LFU (Least Frequently Used) cache structure to limit memory usage-for example, capping the number of cached modules or total memory size. Modules should be released or garbage-collected once unused or after reaching cache limits.

Developer Response The developers now use an LRU cache to store recently used modules.

5.1.16 V-FNT-VUL-016: Syscall OOG not propagated

Severity	High	**Commit**	e88ea57
Type	Denial of Service	**Status**	Fixed
Location(s)	`crates/revm/src/` ▶ `executor.rs:236-325`		
Confirmed Fix At	c41bc44, 4823943		

When a state-modification is required, the exec syscall is used. Syscalls are responsible for charging their own gas, returning early from `execute_rwasm_interruption()` when they have insufficient gas.

```
macro_rules! charge_gas {
   ($value:expr) => {{
       if !local_gas.record_cost($value) {
           return_result!(OutOfFuel);
       }
   }};
}
```

Snippet 5.5: Snippet from `execute_rwasm_interruption()`.

In the case where local_gas has insufficient gas, the call to `record_cost()` has no effect on state, simply returning false.

```
pub fn record_cost(&mut self, cost: u64) -> bool {
    if let Some(new_remaining) = self.remaining.checked_sub(cost) {
        self.remaining = new_remaining;
        return true;
    }
    false
}
```

Snippet 5.6: Definition of `Gas::record_cost()`, showing that the spent gas is not recorded when an out-of-gas error occurs.

This causes an issue when recording the gas expenditure in the resuming execution. `execute_rwasm_resume()` computes the fuel_consumed by taking the `result.gas.spent()` and forwarding it to the resume function. Importantly, this *excludes* the expenditure of all remaining gas caused by a syscall. The out-of-gas error is passed as an argument via the exit code, but not handled by the RWasm engine. Instead, only fuel up to *just before* the out-of-gas error is charged to the transaction signer.

```
let fuel_consumed = result
        .gas
        .spent()
// [VERIDISE] elided...

let exit_code: ExitCode = match result.result {
    // [VERIDISE] elided...

    // out of gas error codes
    InstructionResult::OutOfGas
    | InstructionResult::OutOfFuel
    | InstructionResult::MemoryOOG
    | InstructionResult::MemoryLimitOOG
```

```
    | InstructionResult::PrecompileOOG
    | InstructionResult::InvalidOperandOOG
    | InstructionResult::ReentrancySentryOOG => ExitCode::OutOfFuel,
    // don't map other error codes
    _ => ExitCode::UnknownError,
};

// [VERIDISE] elided...
let (fuel_consumed, fuel_refunded, exit_code) = syscall_resume_impl(
    &mut runtime_context,
    inputs.call_id,
    result.output.as_ref(),
    exit_code.into_i32(),
    fuel_consumed,
    fuel_refunded,
    inputs.syscall_params.fuel16_ptr,
);
```

Snippet 5.7: Snippet from execute_rwasm_resume()

Impact Users may invoke syscalls which trigger an out-of-gas error without causing state to revert or paying out the remainder of their gas. For example, the proof-of-concept attached to this issue demonstrates a smart contract which invokes 1,812 `SSTORE`s for only 249,883 gas. A cold `SSTORE` should cost [at least 20,000 gas](`https://www.evm.codes/?fork=cancun#55`), so 1,812 should cost over 36M gas, well above the current block gas limit. Additionally, since `SSTORE` gas is charged by syscall before `charge_gas!` is invoked, this means all 1,812 slots will have been warmed and the store successfully performed in the journal, for less than one one-hundredth of the intended cost.

In general, users may use this to bypass gas limits, submitting large transactions which slow down or DoS the system.

Recommendation Out-of-gas should be handled by the system, ensuring a revert occurs and the gas is spent.

Care should be taken that the un-resumed frame's state is cleared. See related issue Failure before resume could lead to DoS or arbitrary code execution.

Developer Response The developers resolved the issue. The fix ensures that execution halts properly on any OOG or halt-related error code, rather than passing the error back to the VM. Other refactors were made to rely on only a single gas counter and to ensure cached VM states do not lead to memory leaks.

Note also that the charge fuel instruction which allows users to set refunds is now disabled for applications, rather than allowing users to call it but enforcing an error when invoked from a non-system contract.

Proof of Concept The below test demonstrates the issue.

```
use crate::EvmTestingContextWithGenesis;
use fluentbase_sdk::{calc_create_address, Address, Bytes};
use fluentbase_testing::{EvmTestingContext, TxBuilder};
use wat::parse_str;

const FUEL_LIMIT_PER_CALL: i64 = 10_000; // ~10 gas

const EXEC_STORAGE_GAS_BOMB_WAT: &str = r#"
    (module
        (import "fluentbase_v1preview" "_exec"  (func $_exec  (param i32 i32 i32 i32 i32) (result i32)))
        (import "fluentbase_v1preview" "_write" (func $_write (param i32 i32)))
        (import "fluentbase_v1preview" "_fuel"  (func $_fuel  (result i64)))
        (import "fluentbase_v1preview" "_exit"  (func $_exit  (param i32)))
        (memory 2)
        (data (i32.const 64)
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\02")
        (data (i32.const 128)
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\00"
            "\00\00\00\00\00\00\00\01")
        (func $main (local $count i32) (local $fuel i64)
            ;; zero output area
            i32.const 0
            i64.const 0
            i64.store
            i32.const 8
            i32.const 0
            i32.store

            block $exit
                loop $bomb
                    ;; Stop once remaining fuel dips below the per-call budget.
                    call $_fuel
                    local.tee $fuel
                    i64.const 100000
                    i64.le_u
                    br_if $exit

                    ;; Refresh fuel buffer and invoke STORAGE_WRITE with tiny allowance.
                    i32.const 200
                    i64.const 0
                    i64.store
                    i32.const 208
                    i64.const 0
                    i64.store

```

```
                    i32.const 64      ;; SYSCALL_ID_STORAGE_WRITE
                    i32.const 128     ;; slot || value
                    i32.const 64      ;; input length
                    i32.const 200     ;; fuel buffer
                    i32.const 0       ;; STATE_MAIN
                    call $_exec
                    drop

                    local.get $count
                    i32.const 1
                    i32.add
                    local.set $count
                    br $bomb
                end
            end

            ;; Publish remaining fuel and number of iterations executed.
            i32.const 0
            local.get $fuel
            i64.store
            i32.const 8
            local.get $count
            i32.store
            i32.const 0
            i32.const 12
            call $_write
            i32.const 0
            call $_exit
        )
        (export "main" (func $main))
        (export "memory" (memory 0))
    )
"#;

fn deploy_balance_bomb(ctx: &mut EvmTestingContext) -> Address {
    let wasm = parse_str(EXEC_STORAGE_GAS_BOMB_WAT).expect("invalid WAT");
    let deployer = Address::ZERO;
    let deploy = TxBuilder::create(ctx, deployer, wasm.into())
        .gas_price(0)
        .gas_limit(60_000_000)
        .exec();
    assert!(
        deploy.is_success(),
        "failed to deploy storage gas bomb contract: {deploy:?}"
    );
    calc_create_address(&deployer, 0)
}

#[test]
fn exec_balance_undercharges_on_syscall_oog() {
    let mut ctx = EvmTestingContext::default().with_full_genesis();
    let contract = deploy_balance_bomb(&mut ctx);

    let result = TxBuilder::call(&mut ctx, Address::ZERO, contract, None)
        .gas_price(0)
        .gas_limit(250_000)
        .input(Bytes::default())
```

```
        .exec();

    assert!(result.is_success(), "attack call failed: {result:?}");

    let output = result.output().cloned().unwrap_or_default();
    assert!(
        output.len() >= 12,
        "unexpected output length: {}",
        output.len()
    );

    let remaining = {
        let mut fuel = [0u8; 8];
        fuel.copy_from_slice(&output.as_ref()[0..8]);
        i64::from_le_bytes(fuel)
    };
    let iterations = {
        let mut count = [0u8; 4];
        count.copy_from_slice(&output.as_ref()[8..12]);
        u32::from_le_bytes(count) as u64
    };

    assert!(
        iterations > 0,
        "invalid iteration count: {iterations}"
    );

    // With 'iterations' SSTORE-equivalent syscalls we should be charged iterations *
     20_000 gas.
    let expected_charge = 21_000 + iterations * 20_000;
    let gas_used = result.gas_used();
    let exploitation_ratio = expected_charge / gas_used.max(1);
    let msg = format!(
        "iterations={iterations}, expected_charge={expected_charge}, gas_used={
    gas_used}, remaining_fuel={remaining}, exploitation={exploitation_ratio}x"
    );
    assert!(
        exploitation_ratio >= 100,
        "{}", &msg
    );
    println!("Exploit successful: {}", &msg);

    assert!(
        remaining >= FUEL_LIMIT_PER_CALL / 2,
        "expected to retain fuel, got {remaining}"
    );
}
```

5.1.17 V-FNT-VUL-017: Wrong index in syscall_write_fd_handler params

Severity	High	**Commit**	e88ea57
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/host/write_fd.rs:35`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/229`, 48debed		

The parameters for `syscall_write_fd_handler` contains a typo. Parameter `params[1]` is used twice while `params[2]` is never used.

```
pub fn syscall_write_fd_handler(
    caller: &mut impl Store<RuntimeContext>,
    params: &[Value],
    _result: &mut [Value],
) -> Result<(), TrapCode> {
    let (fd, slice_ptr, slice_len) = (
        params[0].i32().unwrap() as u32,
        params[1].i32().unwrap() as u32,
        params[1].i32().unwrap() as u32,
    );
```

Impact Parameter `slice_len` is retrieved incorrectly.

Recommendation Use `params[2]` for initialization of `slice_len`.

Developer Response The developers updated the parameter to use the correct index.

5.1.18 V-FNT-VUL-018: Elliptic curve x-coordinate interpreted as scalar

Severity	High	**Commit**	e88ea57
Type	Cryptographic Vulnerability	**Status**	Fixed
Location(s)	`crates/runtime/src/[...]/write_fd.rs:117-118, 154-155`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/231`, b2e9bf5		

The `hook_ecrecover` function parses its input as a compressed elliptic curve point, with the first 32 bytes representing the x-coordinate (`r`). According to the documentation, `r` is interpreted as a field element in the *curve field* and corresponds to the x-coordinate of a point on the elliptic curve.

However, in the implementation of `handle_secp256k1` and `handle_secp256r1`, the same `r` is reused both as a field element (to check whether the point is valid and reconstruct the y-coordinate) and as a scalar in the group scalar field when computing its inverse:

```
let r = K256FieldElement::from_bytes(K256FieldBytes::from_slice(&r)).unwrap();

// [VERIDISE] elided...
if let Some(mut y_coord) = alpha.sqrt().into_option().map(|y| y.normalize()) {
    let r = K256Scalar::from_repr(r.to_bytes()).unwrap();
    let r_inv = r.invert().expect("Non zero r scalar");
```

This is inconsistent and may produce incorrect results because the scalar fields and base fields are different. `r` in the base field does not necessarily correspond to a valid scalar in the scalar field. Converting a field element directly to a scalar representation may trigger a panic, causing the node to abort if `r` does not lie in the scalar field when `.unwrap()` is called.

This misuse occurs in the following locations:

- `ecrecover::handle_secp256k1`
- `ecrecover::handle_secp256r1`

More pernicious, however, is the computation of the inverse. The inverse is a value `r_inv` such that `r * r_inv` is equivalent to zero modulo *the order of the group*, not the order of the base field. This means usage of the inverse may lead to incorrect results by applications.

Impact By calling this syscall with appropriately chosen `r`, a user may crash the node. This and other means to crash the node via this syscall are discussed in Write FD syscall can crash node.

This particular issue adds an additional vector: usage of `r_inv` as an inverse over the base field will introduce application bugs into the system.

Recommendation Avoid interpreting the x-coordinate (`r`) as both a field element and a scalar. If a scalar inverse is needed for protocol purposes, define a separate 32-byte input field and interpret it explicitly as a scalar in the group's scalar field, with validation. Do not attempt to coerce field elements into scalars via serialization unless range-checked.

If the usage of `r` as a scalar is intentional, clearly document the semantics of `r_inv`. Additionally, clearly document the field type expectations for all inputs and enforce strict type separation in the implementation.

Developer Response The developers disabled the `write_fd` syscall.

Proof of Concept The below test panics inside the invocation of `handle_secp256r1()`, inside the call to `P256Scalar::from_repr()`

```
#[test]
#[should_panic]
fn handle_secp256r1_panics_when_x_not_in_scalar_field() {
    let r = [
        0xff, 0xff, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff
    , 0xff,
        0xff, 0xff, 0xbc, 0xe6, 0xfa, 0xad, 0xa7, 0x17, 0x9e, 0x84, 0xf3, 0xb9, 0xca
    , 0xc2,
        0xfc, 0x63, 0x25, 0x54,
    ];
    let alpha = [
        0xbf, 0xb5, 0x77, 0x2f, 0x12, 0x6f, 0x31, 0xf8, 0xc7, 0x78, 0x08, 0xc9, 0x21
    , 0x01,
        0x23, 0x3b, 0xfe, 0xd2, 0xb3, 0xda, 0x1d, 0xd4, 0xd4, 0x24, 0x35, 0xa8, 0x47
    , 0x90,
        0xae, 0x7e, 0x0a, 0x7f,
    ];
    let y = [
        0x48, 0x4f, 0x0c, 0x0f, 0xda, 0x43, 0x4e, 0xf0, 0xa8, 0x08, 0x45, 0x89, 0x14
    , 0xf3,
        0x28, 0x71, 0x5d, 0x7a, 0x54, 0x5e, 0x19, 0x8a, 0xc7, 0xee, 0xe3, 0x1d, 0xff
    , 0xe8,
        0x61, 0xb5, 0xd2, 0x3f,
    ];

    // Sanity-check that (x, y) lies on the secp256r1 curve.
    let p = BigUint::from_str(
        "
    115792089210356248762697446949407573530086143415290314195533631308867097853951",
    )
    .unwrap();
    let a = &p - BigUint::from(3u8); // a = -3 mod p
    let b = BigUint::from_str(
        "
    41058363725152142129326129780047268409114441015993725554835256314039467401291",
    )
    .unwrap();

    let x = BigUint::from_bytes_be(&r);
    let y_val = BigUint::from_bytes_be(&y);
    let alpha_bn = BigUint::from_bytes_be(&alpha);

    let x_sq = (&x * &x) % &p;
    let rhs = ((&x_sq * &x) % &p + (&a * &x) % &p + &b) % &p;
    assert_eq!(rhs, alpha_bn, "alpha should match curve equation");

    let lhs = (&y_val * &y_val) % &p;
    assert_eq!(lhs, alpha_bn, "y^2 should equal alpha");

    // The input describes a valid curve point, but x >= group order, so converting
    // the coordinate into a scalar fails and unwrap() panics.
```

```
44      handle_secp256r1(r, alpha, true);
45  }
```

5.1.19 V-FNT-VUL-019: Fuel undercharge leading to stall

Severity	High	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/executor.rs:329`		
Confirmed Fix At	`pull/239through7379862`		

For CALLs that pass large input slices, the metering function is effectively constant in payload length `L` while actual work - byte copies across the host boundary, page materialization/zeroing, and memory growth - scales linearly in `L`.

This violates EVM-style expectations that memory expansion costs will effectively cover the computational effort of performing a CALL.

An adversary can therefore submit very large CALL inputs that force nodes to perform `O(L)` work while paying almost no additional gas, creating a griefing/DoS vector and severe fee mispricing.

Empirical signal:

```
solidity payload=4096 bytes, gas_used=92440, gas_per_byte=22.568359375, sec=26 ms
    payload=65536 bytes, gas_used=92866, gas_per_byte=1.417022705078125, sec=59 ms
    payload=1048576 bytes, gas_used=99197, gas_per_byte=0.09460163116455078, sec=306
    ms payload=4194304 bytes, gas_used=119294, gas_per_byte=0.028441905975341797, sec
    =1170 ms payload=16777216 bytes, gas_used=193835, gas_per_byte
    =0.011553466320037842, sec=4984 ms payload=33554432 bytes, gas_used=283645,
    gas_per_byte=0.008453279733657837, sec=9524 ms
```

- Gas/byte decreases by a factor of **793.72** (22.5 -> 0.008) from 4 KiB -> 32 MiB.
- Wall-time grows by a factor of ***3000x** (26 ms -> 9.5 s).

Impact

- **Denial of Service:**
  An adversary can submit very large payloads that consume CPU/memory while paying almost no additional gas.
- **Economic mismatch:**
  Consensus metering no longer reflects actual resource use, inviting griefing and fee manipulation.

Recommendation Increase the charge for memory expansion to be more consistent with EVM costs.

Developer Response The developers now charge quadratically for memory expansion.

PoC The below test was used to gather the included timing data.

```
use crate::EvmTestingContextWithGenesis;
use core::{convert::TryInto, fmt::Write};
use std::time::{Duration, Instant};
use fluentbase_sdk::{calc_create_address, syscall::SYSCALL_ID_CALL, Address, Bytes};
use fluentbase_testing::{EvmTestingContext, TxBuilder};
use wat::parse_str;

const PAGE_SIZE: u32 = 64 * 1024;
const CALL_INPUT_PTR: u32 = 32;
const CALL_METADATA_BYTES: u32 = 20 + 32; // target address + value slot
const OUTPUT_TRAILER_BYTES: u32 = 4;
const FUEL_THRESHOLD: i64 = 250_000; // 25k fuel units

fn encode_bytes_for_wat(bytes: &[u8]) -> String {
    let mut out = String::with_capacity(bytes.len() * 4);
    for byte in bytes {
        let _ = write!(&mut out, "\\{:02x}", byte);
    }
    out
}

fn build_call_contract(payload_bytes: u32, current_address: Address) -> Vec<u8> {
    let payload_start = CALL_INPUT_PTR + CALL_METADATA_BYTES;
    let total_bytes = payload_start + payload_bytes + OUTPUT_TRAILER_BYTES;
    let required_pages = (total_bytes + PAGE_SIZE - 1) / PAGE_SIZE;
    let memory_grow_pages = required_pages.saturating_sub(1);
    let call_input_len = CALL_METADATA_BYTES + payload_bytes;
    let syscall_hash = encode_bytes_for_wat(&SYSCALL_ID_CALL.0);
    let address_bytes = current_address.as_slice();
    let call_target_lo = u64::from_le_bytes(address_bytes[0..8].try_into().unwrap());
    let call_target_mid = u64::from_le_bytes(address_bytes[8..16].try_into().unwrap()
    );
    let call_target_hi = u32::from_le_bytes(address_bytes[16..20].try_into().unwrap()
    );

    let wasm_source = format!(
        r#"
        (module
            (import "fluentbase_v1preview" "_exec"  (func $_exec  (param i32 i32 i32
    i32 i32) (result i32)))
            (import "fluentbase_v1preview" "_write" (func $_write (param i32 i32)))
            (import "fluentbase_v1preview" "_fuel"  (func $_fuel  (result i64)))
            (import "fluentbase_v1preview" "_exit"  (func $_exit  (param i32)))
            (memory 1)
            (data (i32.const 0) "{syscall_hash}")
            (func $main (local $out_ptr i32) (local $old_pages i32) (local
    $remaining_fuel i64)
                ;; Make a big stack to avoid underflow errors due to rwasm bug
                i32.const 0
                ...
                ;; issue CALL with the giant input slice
                i32.const 0
                i32.const {call_input_ptr}
                i32.const {call_input_len}
                i32.const 0
```

```
                i32.const 0
                call $_exec
                drop
                ;; return the payload length as proof of execution
                local.get $out_ptr
                i32.const {output_trailer}
                call $_write
                i32.const 0
                call $_exit
            )
            (export "main" (func $main))
            (export "memory" (memory 0))
        )
        "#,
        memory_grow_pages = memory_grow_pages,
        payload_start = payload_start,
        payload_bytes = payload_bytes,
        call_input_ptr = CALL_INPUT_PTR,
        call_input_len = call_input_len,
        call_input_ptr_mid = CALL_INPUT_PTR + 8,
        call_input_ptr_hi = CALL_INPUT_PTR + 16,
        call_target_lo = format!("{:#018x}", call_target_lo),
        call_target_mid = format!("{:#018x}", call_target_mid),
        call_target_hi = format!("{:#010x}", call_target_hi),
        fuel_threshold = FUEL_THRESHOLD,
        output_trailer = OUTPUT_TRAILER_BYTES,
        syscall_hash = syscall_hash,
    );

    parse_str(&wasm_source).expect("invalid WAT")
}

fn measure_call_gas(payload_bytes: usize) -> (u64, f64, Duration) {
    let deployer = Address::ZERO;
    let contract = calc_create_address(&deployer, 0);
    let wasm = build_call_contract(payload_bytes.try_into().unwrap(), contract);
    let mut ctx = EvmTestingContext::default().with_full_genesis();

    let deploy = TxBuilder::create(&mut ctx, deployer, wasm.into())
        .gas_price(0)
        .gas_limit(80_000_000)
        .exec();
    assert!(deploy.is_success(), "deploy failed: {deploy:?}");

    let start = Instant::now();
    let call = TxBuilder::call(&mut ctx, Address::ZERO, contract, None)
        .gas_price(0)
        .gas_limit(20_000_000)
        .input(Bytes::default())
        .exec();
    let elapsed = start.elapsed();
    assert!(call.is_success(), "call failed: {call:?}");

    let output = call.output().cloned().unwrap_or_default();
    assert_eq!(output.len(), OUTPUT_TRAILER_BYTES as usize, "unexpected output size:
    {}", output.len());
    let mut buf = [0u8; OUTPUT_TRAILER_BYTES as usize];
```

```
108     buf.copy_from_slice(&output);
109     let echoed_len = u32::from_le_bytes(buf) as usize;
110     assert_eq!(echoed_len, payload_bytes, "payload mismatch");
111 
112     let gas_used = call.gas_used();
113     let gas_per_byte = gas_used as f64 / payload_bytes as f64;
114     (gas_used, gas_per_byte, elapsed)
115 }
116 
117 #[test]
118 fn exec_call_undercharges_large_payload() {
119     let payload_sizes = [
120         1usize << 12,  // 4 KiB
121         1usize << 16,  // 64 KiB
122         1usize << 20,  // 1 MiB
123         4usize << 20,  // 4 MiB
124         16usize << 20, // 16 MiB
125         32usize << 20, // 32 MiB
126     ];
127 
128     let mut observations = Vec::with_capacity(payload_sizes.len());
129     for &payload in &payload_sizes {
130         let (gas_used, gas_per_byte, elapsed) = measure_call_gas(payload);
131         let millis = elapsed.as_millis();
132         println!(
133             "payload={payload} bytes, gas_used={gas_used}, gas_per_byte={gas_per_byte
    }, sec={millis} ms"
134         );
135         observations.push((payload, gas_used, gas_per_byte));
136     }
137 
138     let &(_, gas_used, gas_per_byte) = observations.last().unwrap();
139     println!(
140         "largest payload gas summary: bytes={}, gas_used={gas_used}, gas_per_byte={
    gas_per_byte}",
141         payload_sizes.last().unwrap()
142     );
143     assert!(gas_per_byte < 0.01, "gas per byte too high: {gas_per_byte}");
144 }
```

5.1.20 V-FNT-VUL-020: CREATE syscall may be DoS'ed

Severity	Medium	**Commit**	9ed9bd5
Type	Frontrunning	**Status**	Fixed
Location(s)	`crates/revm/src/syscall.rs:821-823`		
Confirmed Fix At	`pull/204through1522a9a`		

Description The `SYSCALL_ID_METADATA_CREATE` implementation allows creation of derived ownable accounts at a deterministic address calculated via CREATE4-like semantics:

```
let derived_metadata_address =
    calc_create4_address(&account_owner_address, &salt, |v| keccak256(v));
let account = ctx.journal_mut().load_account_code(derived_metadata_address)?;
// make sure there is no account create collision
if !account.is_empty() {
    return_result!(CreateContractCollision);
}
```

The collision check only ensures that the account is empty. However, in Ethereum, non-empty is defined as having either non-zero nonce, balance, or code. This means an attacker could send a minimal balance (e.g., 1 wei) to the derived address before its intended creation. This makes the account non-empty and causes the syscall to fail with a `CreateContractCollision`. This is a front-running vector. An adversary can DoS contract deployments at predictable addresses by pre-funding them with trivial balances.

Impact Any participant can prevent the successful creation of ownable accounts at deterministic addresses by pre-funding them with a small transfer. If a deployment transaction is observed in the mempool, an attacker can preemptively send wei to the derived address, blocking the deployment.

As a consequence, deployments that depend on predictable addresses (for system contracts or factories) can be permanently bricked, requiring protocol redesign or migration.

Recommendation Align the collision semantics with Ethereum's CREATE and CREATE2 rules: enforce that both the account nonce is zero and code is empty before allowing contract creation.

Developer Response The developers followed the recommendation and resolved the issue. They modified collision detection in `SYSCALL_ID_METADATA_CREATE` to check only `code_hash` and `nonce`, excluding balance from emptiness check. This allows creation even if the address has received funds. This now matches the check performed by revm in the call to `create_account_checkpoint()` made from `make_create_frame()`.

5.1.21 V-FNT-VUL-021: EIP-7702 delegation to ownable accounts not resolved in call path

Severity	Medium	**Commit**	42b1b4c
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/handler/src/frame.rs:247`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/revm-rwasm/pull/38`, `pull/38through78b57d3`		

In `make_call_frame` the bytecode selection uses a single `if ... else if ...` chain:

- First, it checks `if let Bytecode::Eip7702(..) = bytecode { ... }` and, on match, replaces `bytecode/code_hash` with those of the delegated address.
- Otherwise (only if the first `if` did not match), it checks `else if let Bytecode::OwnableAccount(..) = bytecode { ... }` to unwrap to the owner's code and set `interpreter_input.account_owner`.

This control flow short-circuits by construction: if the first `if` matches (i.e., the callee is a 7702 delegator), the `else if` branch is never evaluated. Even though the code inside the first block mutates `bytecode` to point at the delegated account's code, the `else if` does not "re-check" the new `bytecode`. Therefore, if the delegated target's code is itself `OwnableAccount`, that second unwrap never happens. Concretely, for a 7702 -> Ownable chain, we end up with `bytecode/code_hash` taken from the *wrapper* (the `0xef44 || 0 || <metadata>` blob) rather than the owner's actual runtime, and `account_owner` remains unset.

Impact Calls routed through a 7702 delegator to an `OwnableAccount` will fail to execute. This will prevent EIP-7702 from functioning correctly, preventing users from delegating accounts. EOAs will effectively be prevented from using EIP-7702.

Recommendation

1. Replace the `if ... else if ...` with two independent checks. First run the 7702 unwrap, then check again for `OwnableAccount` on the resulting bytecode.
2. Implement a small resolver loop that follows 7702 delegation and `OwnableAccount` ownership until reaching plain code, with a hop limit and a cycle guard.

Add tests for: (1) plain 7702, (2) plain Ownable, (3) 7702 -> Ownable, (4) Ownable -> 7702, (5) multi-hop and cycle cases.

Developer Response The developers implemented both recommendations.

5.1.22 V-FNT-VUL-022: Static-call bypass in metadata syscalls

Severity	Medium	**Commit**	2d3517a
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/revm/src/syscall.rs:771-801, 831-851, 893-921`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/211`, 1623654		

Multiple metadata-related syscalls mutate state without guarding against `STATICCALL`. In contrast, other state-changing syscalls (e.g., storage write, transient write, selfdestruct) explicitly reject static context.

- `SYSCALL_ID_METADATA_CREATE` rewrites account code via `set_code(..)`
- `SYSCALL_ID_METADATA_WRITE` grows and overwrites the Ownable metadata, then calls `set_code(..)`
- `SYSCALL_ID_METADATA_STORAGE_WRITE` writes to the owner's storage via `sstore(..)`

Impact `STATICCALL` must not alter state. The listed metadata syscalls can

- rewrite account code (`set_code(..)`)
- mutate Ownable metadata (which is persisted inside code)
- write to the owner's storage.

Contracts frequently use `STATICCALL` to safely query helpers. A callee that internally invokes these syscalls can persist changes during a static frame, surprising callers and tooling. Divergence from expected EVM semantics increases risk of subtle consensus bugs across environments integrating this runtime.

Recommendation Add the standard guard to every state-changing metadata syscall.

Developer Response The developers now check that the context is not static in the `SYSCALL_ID_METADATA_CREATE`, `SYSCALL_ID_METADATA_WRITE`, and `SYSCALL_ID_METADATA_STORAGE_WRITE` cases.

5.1.23 V-FNT-VUL-023: BLOCKHASH may return zero on host execution error

Severity	Medium	**Commit**	2d3517a
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/revm/src/syscall.rs:959`		
Confirmed Fix At	`pull/209through0ae7b49`		

The `BLOCK_HASH` syscall starts by determining whether or not the requested block is within the `[current-256, current-1]` window. Within this branch, the call to `ctx.block_hash(requested_block)` may fail due to an error in the host code. In this case, `None` is returned and coerced to `B256::ZERO`. This conflates a host/availability *bug* (missing recent hash, bookkeeping/reorg gap, uninitialized ring buffer, etc.) with the legitimate "out of range" zero result.

```
let hash = match current_block.checked_sub(requested_block) {
    Some(diff) if diff > 0 && diff <= 256 => {
        ctx.block_hash(requested_block).unwrap_or(B256::ZERO)
    }
    _ => B256::ZERO,
};
return_result!(hash, Ok);
```

Impact For an in-window block, a compliant EVM must return the actual hash; zero is not a permissible substitute. This means a concrete zero will be encoded into the trace where the canonical run would contain a non-zero hash. This turns an operational fault into a seemingly valid state transition and makes problems invisible to the guest.

Recommendation Do not default to zero for in-window misses. Return a deterministic, guest-visible error/trap code so the failure is part of the trace, and fix/guarantee the host's 256-deep recent-hash buffer.

Developer Response Developers now propagate errors instead of defaults and added a test suite for this case.

5.1.24 V-FNT-VUL-024: Failure before resume could lead to DoS or arbitrary code execution

Severity	Medium	**Commit**	e88ea57
Type	Logic Error,Denial of Service	**Status**	Fixed
Location(s)	`crates/runtime/src/executor.rs`		
Confirmed Fix At	https://github.com/fluentlabs-xyz/fluentbase/pull/217, 4823943		

When a contract triggers a syscall that interrupts execution (e.g., `EXEC`), the runtime instance for that call is cached in the `RuntimeFactoryExecutor`'s `recoverable_runtimes` field. That cache entry is removed on a successful call to `resume()`. However, several terminal error paths between the interruption and the resume never perform the corresponding cleanup, leaving the cached runtime orphaned. An attacker can repeatedly force these paths to grow recoverable_runtimes unbounded within a single transaction and eventually exhaust memory or blow per-transactions limits.

More concretely:

- During frame execution, RWasm code runs via `syscall_exec_impl(...)` and may request an interruption. The engine returns a positive `exit_code` to indicate an interruption has been requested.
- `process_exec_result(...)` decodes the syscall params and decides whether to handle the interruption or to fail fast. There are multiple early-return error paths where no call to `resume()` occurs, for example, if there is insufficient gas before attempting the interruption, an out-of-fuel error will be returned.

```
// process_exec_result(...)
if !is_gas_free && syscall_params.fuel_limit / FUEL_DENOM_RATE > gas.remaining() {
    return Ok(NextAction::error(ExitCode::OutOfFuel, gas)); // cleanup never runs
}
```

- In these cases, the cached runtime created at interruption time is never evicted, because eviction happens during a call to `resume()`.

However, there is an even more concerning possibility. Under the hood, the `resume()` function invokes `ExecutionEngineInner::resume()` inside of the rwasm crate. This resume implementation takes the *most recent* value and call stack and resumes execution with those.

```
/// Resumes the execution of a WASM (WebAssembly) function that was previously
    interrupted.
pub fn resume<T: Send + Sync>(
    &mut self,
    store: &mut RwasmStore<T>,
    module: &RwasmModule,
    params: &[Value],
    result: &mut [Value],
) -> Result<(), TrapCode> {
    let (value_stack, call_stack) = (
        self.value_stack.last_mut().unwrap(),
        self.call_stack.last_mut().unwrap(),
    );
```

Snippet 5.8: Snippet from `src/vm/engine.rs` in the rwasm crate.

This means that, if an attacker can successfully cause a failure between interruption and resumption, they can potentially swap out the value and call stack to be one maliciously generated by a contract of their choosing.

Impact Currently, the Veridise analysts have been unable to trigger an error between interruption and resumption. The situations which trigger an early return in the line shown above are currently unreachable. However, any possible error before resumption could lead not only to denial of service by an attacker exploiting the memory leak to cause an out-of-memory error in Fluent nodes, but could also allow them to swap out the value stack and call stack for maliciously crafted ones.

Recommendation It is important to take careful precautions against this situation. Consider adding a `cleanup()` function which handles all the cleanup necessary, and is guaranteed to be invoked across *every* program path before a frame is returned from.

In general, stronger reliance on RAII patterns, rather than globals, may avoid this type of error in the future.

Developer Response The developers resolved this with the fix to Syscall OOG not propagated. A new `forget_runtime()` method is used, called whenever a frame's execution ends for any reason after an interrupt.

5.1.25 V-FNT-VUL-025: Write FD assumes elements are reduced

Severity	Medium	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/host/write_fd.rs`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/231`, b2e9bf5		

Description Several field operations accept non-canonical inputs and rely on `is_zero()` checks that assume the input has already been reduced modulo the field modulus. This causes inconsistencies across hooks:

Affected locations:

- `fp_ops::hook_fp_sqrt`
  - The range check returns early `if element > modulus || nqr > modulus`, allowing unreduced values `element == modulus` or `nqr == modulus`.
    Later logic treats non-zero inputs as possibly quadratic residues.
- `fp_ops::hook_fp_inverse`
  - There is an assertion that the element is non-zero, but no range check that `0 < element < modulus`. This means that the inverse of `modulus`, for example, will be returned as zero.
- `bls::hook_bls12_381_sqrt`
  - This function accepts any 48-byte value without enforcing that `element < modulus`. It then returns early in the zero-case only if the element is exactly zero.
- `bls::hook_bls12_381_inverse`
  - This function accepts any 48-byte value without enforcing that `element < modulus`. It then returns early with an error in the zero-case only if the element is exactly zero, ignoring values which are equivalent to zero modulo `modulus`.

Many of these functions use operations which reduce modulo the modulus at play. This means that unreduced values will still lead to values being returned, but these values may be incorrect or inconsistent based on the representative chosen.

Impact Incorrect acceptance of non-canonical inputs leads to inconsistent status bytes and outputs, making callers or constraints rely on values that do not match the documented semantics.

Callers who are unaware that they need to reduce values may receive incorrect results from the field operations.

Recommendation Ensure that any user-provided inputs which claim to be field elements are canonical.

Document input canonicality requirements in each hook and mirror them with precise checks at the API boundary to avoid relying on downstream reductions.

Developer Response The developers removed the `write_fd` syscall.

5.1.26 V-FNT-VUL-026: Floor gas may be bypassed

Severity	Low	**Commit**	42b1b4c
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/handler/src/validation.rs:301-304`		
Confirmed Fix At	https://github.com/fluentlabs-xyz/revm-rwasm/pull/54, 32da81e		

Description EIP-7623 introduces calldata pricing changes where a minimum floor gas is always charged to ensure a transaction cannot bypass resource costs by posting large calldata with low intrinsic gas. In revm, this is enforced through eip7623_check_gas_floor() during post_execution(). If the transaction spends less than the required floor gas, any unused gas refund is cleared.

In Fluent, the developers intend to charge floor-gas in terms of fuel for account-creation transactions which are creating large WASM binaries. This is done by adding the below snippet to `validate_initial_tx_gas()`:

```
let mut floor_gas = gas.floor_gas;
if tx.input().starts_with(&WASM_MAGIC_BYTES) {
    floor_gas /= FUEL_DENOM_RATE;
}
```

Snippet 5.9: Snippet from `validation.rs`:

The intended behavior is to reduce floor gas for contract creation transactions deploying WASM code, since WASM bytecode size is significantly larger than EVM init code and its average opcode cost is cheaper. However, the check only inspects whether `tx.input()` starts with `WASM_MAGIC_BYTES`, without validating that the transaction is actually a `CREATE`. This means *any transaction*, including `CALL`, can trigger the lowered floor gas check, allowing cheaper data-posting than in the EVM specification.

Impact This introduces a semantic mismatch between EVM gas accounting and the modified WASM execution path.

An attacker can craft a `CALL` with input data beginning with `WASM_MAGIC_BYTES` to artificially reduce the billed floor gas. This allows attackers to submit very large calldata payloads while paying below the EIP-7623 mandated floor gas. This effectively bypasses EVM semantics around calldata pricing, enabling gas griefing, denial-of-service style bloating, or state rent exploitation at discounted cost.

In other words, calldata that should be priced under EVM rules (unconverted gas units) is instead charged at a reduced "fuel" rate intended only for WASM deployment, leading to underpayment.

In general, even if this lower rate were applied only to data in transactions which will be included as WASM bytecode, this allows bypassing EIP-7623. For example, a user wishing to post a large amount of data on-chain could include the data in the data-section of the WASM bytecode at a lower cost.

Recommendation As mentioned above, simply restricting the floor gas denomination adjustment to WASM contract creation transactions will be insufficient to prevent bypassing EIP-7623 in its intention, although this would ensure pure-EVM transactions are consistent with gas costs on Ethereum.

However, this could be easily bypassed in practice by creating a WASM contract whose constructor then passes along data to the desired EVM contracts. Consider instead explicitly disabling EIP-7623, or charging uniformly for data costs without regard to the transaction input.

This, and any other differences from the EVM, should be clearly documented in a central location.

Developer Response The developers followed the recommendations and disabled EIP-7623.

5.1.27 V-FNT-VUL-027: Unsafe Offset/Length Handling

Severity	Low	**Commit**	2d3517a
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/revm/src/syscall.rs:860-863`		
Confirmed Fix At	`pull/208through74c77a3`		

The code limits `length` using the total metadata size instead of the remaining bytes after `offset`.

```
let offset = LittleEndian::read_u32(&inputs.syscall_params.input[20..24]);
let length = LittleEndian::read_u32(&inputs.syscall_params.input[24..28]);
// take min
let length = length.min(ownable_account_bytecode.metadata.len() as u32);
let metadata = ownable_account_bytecode
               .metadata
               .slice(offset as usize..(offset + length) as usize);
```

This creates two cases which may trigger a panic:

1. `offset` past end: if `offset > metadata.len()`, the start index is already out of bounds and the slice will panic.
2. `offset + length` past end: even when `offset` is in-bounds, using `length.min(metadata.len())`still allows `offset + length > metadata.len()`, producing an out-of-bounds end index and a panic.

If truncation is intended, the syscall must:

- Return `[]` when `offset >= metadata.len()` (nothing left to copy).
- Restrict by the remaining window: use `length.min(metadata.len() - offset)`, not `length.min(metadata.len())`.

Additionally, `offset/length` are added in `u32` (`offset + length`) and only then cast to `usize`. That addition can overflow/wrap in `u32` space before the cast, yielding an incorrect end index even when the post-cast value appears "small".

Impact Out-of-bounds access triggers a panic at runtime, breaking execution reliability and potentially unwinding higher-level logic.

Recommendation

- Guard `offset`: if `offset >= metadata.len()`, return an empty slice.
- Bound by remaining: compute `len_out = min(requested_len, metadata.len() - offset)` before forming the range.
- Unify arithmetic width: convert inputs to `usize` immediately and avoid mixed-width math; perform all index calculations in `usize` before slicing.

Developer Response The developers updated the logic to perform a full metadata rewrite.

Updated Veridise Response The analysts recommend removing the code offset parameter as it is essentially unused.

Updated Developer Response The developers cannot remove the `offset` param right now, as it affects the syscall input layout and can break compatibility with testnet.

5.1.28 V-FNT-VUL-028: Unbounded metadata write via unchecked length arithmetic

Severity	Low	**Commit**	2d3517a
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/revm/src/syscall.rs:841`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/210`, 7c9eab7		

In the `SYSCALL_ID_METADATA_WRITE` branch of `syscall.rs`, the handler takes an `offset` and an implicit `length` derived from the remainder of the syscall input, then grows a `Vec<u8>` to `offset + length` and slices `metadata[offset..offset + length]` without additional checks. Notably, if `offset+length` is shorter than the original length, all bytes after `offset + length` will be dropped.

```
match inputs.syscall_params.code_hash {
    SYSCALL_ID_METADATA_WRITE => {
        assert_return!(
            inputs.syscall_params.input.len() >= 20 + 4,
            MalformedBuiltinParams
        );
        let offset =
            LittleEndian::read_u32(&inputs.syscall_params.input[20..24]) as usize;
        let length = inputs.syscall_params.input[24..].len();
        // TODO(dmitry123): "figure out a way how to optimize it"
        let mut metadata = ownable_account_bytecode.metadata.to_vec();
        metadata.resize(offset + length, 0);
        metadata[offset..(offset + length)]
            .copy_from_slice(&inputs.syscall_params.input[24..]);
        // ...
    }
    // ...
}
```

Impact Attempted updates to a slice of bytes may accidentally drop large amounts of data.

There is no cap on total metadata size or per-call write size. The caller can force attempts to allocate up to `offset + length` bytes which can lead to OOM and panic. However, this syscall may only be invoked by a system contract, requiring an attacker to find a way to exploit trusted system precompiles to trigger a large data write.

Recommendation Only resize if additional space is needed, do not resize if writing to just a slice of the data.

Define caps `MAX_METADATA_BYTES` (total size limit) and perform pre-flight allocation. Return an error if reservation fails before calling `resize`.

Developer Response The developers updated the `SYSCALL_ID_METADATA_WRITE` syscall to perform a full rewrite, copying in the data directly as provided from the arguments.

5.1.29 V-FNT-VUL-029: Bytecode hash unwrap prior to validation causes hard crash

Severity	Low	**Commit**	e88ea57
Type	Data Validation	**Status**	Fixed
Location(s)	`crates/revm/src/executor.rs:161`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/232`, 717925e		

The code retrieves a code hash from `interpreter.bytecode` using `hash().unwrap()` **before** the subsequent `match` that materializes or validates the bytecode. This ordering hard-wires a precondition-that the hash already exists and is usable-into the control flow, even though the bytecode's concrete form hasn't been established yet.

```
let rwasm_code_hash = interpreter.bytecode.hash().unwrap();

let rwasm_bytecode = match &interpreter.bytecode.clone() {
    // ...
};
```

Impact Calling `.unwrap()` on the result of `interpreter.bytecode.hash()` will panic if the hash is absent. Because this happens before the bytecode is validated in a subsequent `match`, the panic becomes a process abort rather than a handled error. This creates an availability vulnerability.

Recommendation Swap the two blocks so the bytecode is established first, then confirm the hash and propagate a typed error instead of panicking.

Developer Response The developers applied the recommendation, calling `unwrap()` inside of the match block.

5.1.30 V-FNT-VUL-030: rWasm Deployment may get overwritten with legacy code

Severity	Low	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/revm/src/executor.rs:101`		
Confirmed Fix At	https://github.com/fluentlabs-xyz/revm-rwasm/pull/44, 2986c9b		

In `crates/revm/src/executor.rs`, within the CREATE path that handles delegated-runtime handoff, the engine detects an rWasm payload (0xEF prefix) and verifies the created account is an `OwnableAccount` owned by the delegated runtime. It then installs the rWasm module by journaling it as the account's code and switching the interpreter to that module:

- `ctx.journal_mut().set_code(create_frame.created_address, Bytecode::new_rwasm(...))`
- update `frame.interpreter.bytecode = ExtBytecode::new_with_hash(...)`

Afterward, the generic EVM CREATE finalization still executes. In `revm-rwasm/crates/handler/src/frame.rs::return_create(...)`, once `journal.checkpoint_commit()` runs, any **non-empty** constructor return (`interpreter_result.output`) is deposited as legacy code:

```
if !interpreter_result.output.is_empty() {
    let bytecode = Bytecode::new_legacy(interpreter_result.output.clone());
   journal.set_code(address, bytecode);
}
```

The EVM-precompile path clears or controls the return data; the rWasm path does not. Consequently, on rWasm deployments, any non-empty constructor output is subsequently deposited as legacy bytecode, overwriting the previously installed rWasm module.

Impact The rWasm module is replaced by whatever the constructor returned, potentially changing behavior. This breaks a core invariant that all accounts are either ownable or RWasm modules, allowing code which should be unreachable to be reached.

Currently, this just causes an early return whenever the legacy bytecode is loaded.

Recommendation Since Fluent runtimes are responsible for writing bytecode in all cases, in `return_create(...)`, do not overwrite the code.

Developer Response The developers added a configuration option to their revm fork, completely disabling the account code update in `return_create()` when the configuration is enabled.

5.1.31 V-FNT-VUL-031: Trusted address can overwrite code

Severity	Low	**Commit**	e88ea57
Type	Authorization	**Status**	Fixed
Location(s)	`crates/revm/src/` ▶ `evm.rs:226-251`		
Confirmed Fix At	`pull/240through1a5c5e2`		

In the CALL path, two alterations are made to the usual execution flow.

First, an attempt is made to resolve a "native precompile" by peeking at the start of the call input (the 4-byte sentinel) via `try_resolve_precompile_account_from_input(..)`. If `inputs.caller` is privileged address `UPDATE_GENESIS_AUTH`, then the `upgrade_runtime_hook_v*()` is run, allowing deployment of arbitrary code.

```
if inputs.caller == UPDATE_GENESIS_AUTH {
   let input = inputs.input.bytes(ctx);
   if input.starts_with(&UPDATE_GENESIS_PREFIX_V1) {
         return upgrade_runtime_hook_v1(ctx, inputs);
   } else if input.starts_with(&UPDATE_GENESIS_PREFIX_V2) {
       return upgrade_runtime_hook_v2(ctx, inputs);
   }
}
```

Second, if the bytes of the input happen to start with the bytes in `PRECOMPILE_NATIVE_MULTICALL`, then a specified precompile is run.

```
// TODO(dmitry123): "do we want to disable it for mainnet?"
if let Some(precompiled_address) = try_resolve_precompile_account_from_input(
    inputs.input.bytes(ctx).as_ref(),
) {
    let account = &ctx.journal_mut().load_account_code(precompiled_address)?;
    // rewrite bytecode address
    inputs.bytecode_address = precompiled_address;
    // rewrite bytecode with code hash
    new_frame.interpreter.bytecode = ExtBytecode::new_with_hash(
       account.info.code.clone().unwrap_or_default(),
       account.info.code_hash,
    );
```

Impact If the privileged key/address is ever compromised, the attacker can deploy arbitrary bytecode to any address.

Additionally, any transaction whose input begins with a certain set of bytes invokes a system precompile. Ethereum precompiles are invoked by destination address, not by calldata prefix (EIP-1352). Calldata-based dispatch deviates from consensus expectations and can surprise tooling or admin scripts that assume address-keyed precompiles.

Recommendation Remove the functionality for mainnet deployment.

Developer Response We restricted `UPDATE_GENESIS_AUTH` and calldata-based precompile dispatch to testnet-only via 'fluent-testnet' feature flag. Mainnet deployments now completely disable this functionality.

Key changes:

- Gated vulnerable code paths behind feature flag in `frame_init()`
- Restricted multicall and `update_account` tests to testnet-only
- Added security documentation

Impact: Mainnet now uses standard address-based precompile dispatch only. Compromised privileged key cannot deploy arbitrary bytecode on production.

5.1.32 V-FNT-VUL-032: RSA Quotient incorrectly truncated

Severity	Low	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/[...]/write_fd.rs:264-270`		
Confirmed Fix At	https://github.com/fluentlabs-xyz/fluentbase/pull/231, b2e9bf5		

The `hook_rsa_mul_mod` syscall in Fluent performs modular reduction on a 512-byte product and a 256-byte modulus, returning a truncated 256-byte remainder and a 256-byte quotient. However, this implementation assumes that the quotient always fits into 256 bytes, which is incorrect. Specifically, when the modulus is small (e.g., 1), the quotient can be as large as the original 512-byte product, requiring up to 512 bytes to encode accurately.

```rust
pub fn hook_rsa_mul_mod(buf: &[u8]) -> Result<Vec<u8>, ExitCode> {
    if buf.len() != 256 + 256 + 256 {
        return Err(ExitCode::MalformedBuiltinParams);
    }

    let prod: &[u8; 512] = buf[..512].try_into().unwrap();
    let m: &[u8; 256] = buf[512..].try_into().unwrap();

    let prod = BigUint::from_bytes_le(prod);
    let m = BigUint::from_bytes_le(m);

    let (q, rem) = prod.div_rem(&m);

    let mut rem = rem.to_bytes_le();
    rem.resize(256, 0);

    let mut q = q.to_bytes_le();
    q.resize(256, 0);  // Unsafe truncation

    let mut result = rem;
    result.extend_from_slice(&q);
    Ok(result)
}
```

Snippet 5.10: Implementation of `hook_rsa_mul_mod()`

The truncation of `q` to 256 bytes introduces incorrect output when the true quotient is larger. The function documentation is also inaccurate, claiming that the result contains a 512-byte remainder and 256-byte quotient, when it is the reverse.

Impact This issue causes the syscall to return incorrect results in extreme cases. It breaks soundness guarantees of any cryptographic protocol relying on precise arithmetic, especially when the zkVM assumes these values are trustworthy. The truncated quotient can cause verification mismatches, incorrect constraint assumptions, or failure in cryptographic proofs.

Recommendation Return 256 bytes for the remainder and up to 512 bytes for the quotient to preserve correctness. Adjust the syscall interface and consumer zkVM logic to handle variable-

length quotient outputs or mandate a fixed 512-byte format. Update the documentation to reflect the correct structure and size of the outputs.

Fortunately, most common applications will use large moduli, and primarily use the remainder.

Developer Response The developers removed the `write_fd` syscall.

Proof of Concept The below test demonstrates the issue.

```
#[test]
fn hook_rsa_mul_mod_truncates_large_quotient() {
    let prod = vec![0xff; 512];
    let mut modulus = vec![0u8; 256];
    modulus[0] = 1;

    let mut input = prod.clone();
    input.extend_from_slice(&modulus);

    let output = hook_rsa_mul_mod(&input).unwrap();
    assert_eq!(output.len(), 512);

    let rem = BigUint::from_bytes_le(&output[..256]);
    let q = BigUint::from_bytes_le(&output[256..]);
    let expected_prod = BigUint::from_bytes_le(&prod);
    let modulus_int = BigUint::from_bytes_le(&modulus);

    let reconstructed = &q * &modulus_int + &rem;
    assert_ne!(reconstructed, expected_prod);  // Fails due to q truncation
}
```

Snippet 5.11: Test case demonstrating the issue:

5.1.33 V-FNT-VUL-033: Return data unchanged when unrecognized FD used in write FD syscall

Severity	Low	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/syscall_handler/[...]/write_fd.rs:60-61`		
Confirmed Fix At	`pull/231throughb2e9bf5`		

In `syscall_write_fd_impl`, when an unrecognized file descriptor (`fd`) is passed, the function exits early via `return Ok(())` without modifying `ctx.execution_result.return_data`. This means that if `return_data` contained data from a prior successful syscall, it will persist across invocations even though the current syscall performed no operation.

Snippet from `syscall_write_fd_impl()`:

```
let output = match fd {
    FD_ECRECOVER_HOOK => hook_ecrecover(input),
    FD_ED_DECOMPRESS => hook_ed_decompress(input),
    FD_RSA_MUL_MOD => hook_rsa_mul_mod(input),
    FD_BLS12_381_SQRT => bls::hook_bls12_381_sqrt(input),
    FD_BLS12_381_INVERSE => bls::hook_bls12_381_inverse(input),
    FD_FP_SQRT => fp_ops::hook_fp_sqrt(input),
    FD_FP_INV => fp_ops::hook_fp_inverse(input),
    _ => return Ok(()),
}?;
```

This branch returns without clearing or resetting `ctx.execution_result.return_data`.

As a result, downstream components consuming `return_data` cannot reliably distinguish between valid output from the current syscall and stale data from previous executions. This can cause undefined behavior or misinterpretation of hook outputs, especially in environments where syscalls are invoked sequentially in a shared runtime context.

Impact Leaving stale `return_data` in the runtime context can lead to inconsistent execution semantics and erroneous state propagation between syscalls. Subsequent logic may read and act upon outdated or invalid data, potentially producing incorrect cryptographic results (e.g., invalid signature recovery) or undefined execution behavior.

Recommendation When no handler matches the provided `fd`, explicitly clear `ctx.execution_result.return_data` to ensure it does not hold prior values.

Alternatively, return an explicit error (e.g., `Err(ExitCode::InvalidFd)`) to signal unrecognized file descriptors. The latter approach improves observability and prevents silent failures.

Developer Response The developers removed the `write_fd` syscall.

5.1.34 V-FNT-VUL-034: Interpreter state carryover when invoking rWasm constructor

Severity	Warning	**Commit**	e88ea57
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/revm/src/executor.rs:95-100`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/233`, 837293b		

After the deploy-shim precompile returns the rWasm module and constructor params, the code mutates only a subset of fields on the current frame's interpreter and immediately re-enters the rWasm loop. It does **not** perform a full frame/interpreter re-initialization.

```
// Change input params
frame.interpreter.input.input = CallInput::Bytes(constructor_params_raw);
frame.interpreter.input.account_owner = None;
frame.interpreter.bytecode = ExtBytecode::new_with_hash(bytecode, bytecode_hash);
frame.interpreter.gas = interpreter_result.gas;
// Re-run deploy function using rWasm
return run_rwasm_loop(frame, ctx, inspector);
```

Impact Stateful fields such as the EVM stack, return buffer, memory context, and the frame's finished flag can persist from the precompile phase into the constructor phase.

Recommendation Call `set_finished(false)`, `stack.clear()`, `return_data.clear()`, `memory.free_child_context()`, `input.bytecode_address = None`.

Developer Response The developers implemented the recommendation.

5.1.35 V-FNT-VUL-035: call_id may overflow

Severity	Warning	**Commit**	e88ea57
Type	Logic Error	**Status**	Fixed
Location(s)	`crates/runtime/src/executor.rs:148-150`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/214`, f35419c		

The `call_id`, used to identify resumable runtimes, is a 32-bit identifier of a paused execution.

```
let call_id = self
    .transaction_call_id_counter
    .fetch_add(1, Ordering::Relaxed);
self.recoverable_runtimes.insert(call_id, runtime);
```

Snippet 5.12: Snippet from `try_remember_runtime()` getting the next `call_id` from a global counter.

It is important that each `call_id` is unique, since when execution is resumed the `call_id` is used to identify the executing engine.

In principle, 32 bits should be enough to prevent overflow. The current gas limit is around 30 million, and requiring around 131 interrupts per unit of gas to cause an overflow. However, any errors in gas metering may open this as an attack vector. For example, issues like Syscall OOG not propagated allow an attacker to execute certain syscalls for more than 100x cheaper than intended, bringing overflow into the range of possibility.

Impact This effectively converts DoS or gas metering issues into logical attacks which may allow inserting attacker-controlled code into the call stack at key locations, potentially enabling theft.

Recommendation Check the return value of `insert()` to ensure no call ID is already stored there.

Developer Response The developers now check for overflow before incrementing the `u32`. They also removed the unnecessary atomic wrapper, since only a single thread is accessing this.

5.1.36 V-FNT-VUL-036: SVM ELF Magic bytes form valid EVM code prefix

Severity	Warning	**Commit**	e88ea57
Type	Usability Issue	**Status**	Fixed
Location(s)	`crates/revm/src/evm.rs:255-256`		
Confirmed Fix At	`pull/237throughebf4da4`		

The `resolve_precompiled_runtime_from_input()` function in Fluent's `frame_init()` function determines the runtime (EVM, WASM, SVM, or Universal Token) based on the first few bytes of the contract's init code. The function determines which runtime to use based on the first few bytes of the `input`, defaulting to EVM if there is no match.

```
pub fn resolve_precompiled_runtime_from_input(input: &[u8]) -> Address {
    if input.len() > WASM_MAGIC_BYTES.len() && input[..WASM_MAGIC_BYTES.len()] ==
    WASM_MAGIC_BYTES {
        PRECOMPILE_WASM_RUNTIME
    } else if input.len() > SVM_ELF_MAGIC_BYTES.len()
        && input[..SVM_ELF_MAGIC_BYTES.len()] == SVM_ELF_MAGIC_BYTES
    {
        PRECOMPILE_SVM_RUNTIME
    } else if input.len() > UNIVERSAL_TOKEN_MAGIC_BYTES.len()
        && input[..UNIVERSAL_TOKEN_MAGIC_BYTES.len()] == UNIVERSAL_TOKEN_MAGIC_BYTES
    {
        PRECOMPILE_UNIVERSAL_TOKEN_RUNTIME
    } else {
        PRECOMPILE_EVM_RUNTIME
    }
}
```

However, these "magic bytes" can correspond to valid EVM bytecode sequences. The EVM interpretation of the bytecode is shown below.

- **WASM**
  - **Bytes:** 00 61 73 6d
  - **EVM Interpretation:** STOP, PUSH2, PUSH20, PUSH14
  - **Result:** Execution halts immediately - empty contract
- **SVM ELF**
  - **Bytes:** 7f 45 4c 46
  - **EVM Interpretation:** PUSH32 + 32-byte immediate starting with 45 4c 46...
  - **Result:** Valid EVM bytecode that deploys constant data
- **ERC20**
  - **Bytes:** 45 52 43 20
  - **EVM Interpretation:** GASLIMIT, MSTORE, NUMBER, KECCAK256
  - **Result:** Stack underflow (runtime error)

The WASM and ERC20 runtimes correspond to CREATE calls which would either deploy empty code or panic. However, because `0x7f` is a valid PUSH32 opcode, valid EVM contracts beginning with that byte sequence could be misinterpreted as SVM binaries and thus routed to the SVM runtime precompile instead of the EVM runtime.

Impact Valid EVM contracts with certain byte prefixes (notably those whose initialization code begins with `0x7f 45 4c 46`, the SVM ELF header) will not deploy under the expected EVM environment. Instead, they will be redirected to a precompile that cannot execute EVM bytecode, causing deployment failure or undefined behavior. This renders a subset of otherwise valid EVM contracts undeployable on Fluent.

Recommendation Ensure that the runtime detection logic cannot misclassify valid EVM bytecode. Potential remediations include:

1. **Using reserved non-EVM opcodes** with explicit leading markers (e.g. 0xEF) for WASM/SVM detection to avoid conflicts.
2. **Requiring precompile-based deployment** for WASM and SVM contracts, so that EVM contracts are always routed to the EVM runtime by default.

Developer Response The developers placed SVM support under feature.

5.1.37 V-FNT-VUL-037: Maintainability

Severity	Warning	**Commit**	2d3517a, e88ea57, 9ed9bd5
Type	Maintainability	**Status**	Fixed
Location(s)	`crates/` ▶ `evm/src/evm.rs:176` ▶ `revm/src/` • `evm.rs:228` • `executor.rs:391` • `result.rs:30` • `syscall.rs:137, 275, 378, 753, 782` ▶ `runtime/src/` • `context_wrapper.rs:6` • `runtime/strategy_runtime.rs:46` • `syscall_handler/` ∗ `hashing/` · `keccak256.rs` · `poseidon.rs` · `sha256.rs` · `sha256_compress.rs` ∗ `host/write_fd.rs:226, 371` • `syscall_handler.rs:112`		
Confirmed Fix At	`https://github.com/fluentlabs-xyz/fluentbase/pull/243`		

This issue aggregates multiple maintainability, correctness, and cryptographic interface problems across the `revm` and `runtime` crates.

1. Redundant Implementation in `impl EvmTr for RwasmEvm` evm.rs @ L176: The `RwasmEvm` struct wraps an `Evm` instance but re-implements many trait methods already defined on `Evm`, instead of delegating to the inner implementation. This introduces maintenance risk if the upstream `Evm` changes. Consider following the example implementation which correctly delegates to the underlying EVM.

2. Testnet-Only Logic Not Isolated evm.rs @ L228: `<RwasmEvm as <EvmTr>>::frame_init()` includes some testnet functionality allowing code upgrades to be performed by a specific account. It would be best to separate this logic into a separate function which is only invoked in the testnet interpreter to be sure no code path can accidentally reach this point on mainnet.

3. Inspector Frame Skips on Panic executor.rs @ L391: Frames which panic due to interpreter errors may not have their frames recorded in the inspector, which may make debugging crashes more difficult.
4. Commented-Out Code syscall.rs @ L137: There is commented out code for supporting the inspector.
5. Inconsistent 63/64 Gas Rule Application syscall.rs @ L275: Some sys calls aim to be backwards compatible (e.g. `SYSCALL_ID_STATIC_CALL` checks if the `TANGERINE` fork is enabled before applying the 63/64 rule) while others do not (e.g. `SYSCALL_ID_CALL` unconditionally applies the 63/64 gas rule). It would improve clarity to allow follow one design or the other so that forks/maintainers do not incorrectly assume portions of the code are backwards compatible.

---

6. Misaligned SHA256 Endianness and Documentation sha256.rs: The implementation parses input bytes as big-endian `u32`s per the SHA-256 spec, but writes those as little-endian back into memory. This mismatch is undocumented and could confuse callers or cause invalid results if consumers assume big-endian output. This behavior is only exercised via `*_impl()` in tests, which bypass the syscall interface and do not cover the full execution path.

---

7. Inconsistent LE vs BE usage for salt in syscall.rs @ 378,782

LE at L378`let salt = U256::from_le_slice(&inputs.syscall_params.input[32..64]);`

BE at L782`let salt = U256::from_be_slice(&inputs.syscall_params.input[..32]);`

---

1. Missing Address Length Check syscall.rs @ L751: The `SYSCALL_ID_METADATA_ACCOUNT_OWNER` syscall should validate that the provided address matches `Address::len_bytes()` to avoid processing malformed data.

---

1. Missing `#[must_use]` strategy_runtime.rs @ L46: Functions returning important values (like status or handles) are missing `#[must_use]`, which could result in critical logic being ignored.

---

1. Missing Defensive Enum Handling syscall_handler.rs @ L112: The dispatcher uses `unreachable!()` for unrecognized syscall values. This should be replaced with explicit matching or error-returning fallbacks.

---

1. Inconsistent Endianness in Salt and Hashing write_fd.rs: Some syscalls use little-endian formats, while others (e.g., in hashing logic) use big-endian formats. This inconsistency is error-prone, especially when serialized data is reused across syscalls.

---

1. `pad_to_be()` Over-Resizes write_fd.rs @ L371: This function always resizes output. It should only do so when `len > bytes.len()`.

---

1. Undocumented Argument Constraints: Functions like `weierstrass_add` and others assume input field sizes or constraints but do not document or enforce them.

---

1. InstructionResult Depends on Return Data Size result.rs @ L30: The result returned for `RETURN` and `STOP` instructions is conditional on the return data size, not just the opcode. This behavior is non-obvious and should be clarified or avoided.

---

1. Missing Endianness Guard keccak256_permute.rs @ L20: This syscall lacks a `#[cfg]` or runtime check to prevent incorrect behavior on big-endian platforms.

---

1. `inner.spec` should follow caller's `RwasmSpecId`. The runtime currently uses `SpecId::default()` for setting `inner.spec`, rather than mapping from the caller's `RwasmSpecId`. This can cause runtime behavior to diverge from expected fork settings. Instead, explicitly match:

```
let inner_spec = match spec {
    RwasmSpecId::PRAGUE => SpecId::PRAGUE,
    _ => unimplemented!("Unsupported RwasmSpecId"),
};
```

1. Deprecated or Unused Cryptographic Functions Still Present

- syscall_handler/hashing contains `syscall_hashing_blake3_handler`, which should be renamed to `*_impl()`.
- The `sha256`, `blake3`, `keccak` and `poseidon` functions are flagged for removal but remain in the codebase.

---

1. Unused Code

- inter_process_lock is unused.
- syscall_exec_continue is unused.
- testing_store.rs is unused.
- `RuntimeContextWrapper` includes unwraps and unreachable code and is only used in test contexts.

Impact

- Code duplication increases maintenance risk
- Testnet logic in core paths may affect mainnet stability
- Endianness inconsistencies risk cryptographic faults
- Deprecated and unused code may introduce accidental usage
- Missing validation or error handling may lead to crashes
- Spec mismatch may cause incorrect fork behavior

Recommendation

- Reuse existing Evm implementations via delegation
- Move testnet-specific code to isolated paths
- Enforce consistent endianness in hashing and memory
- Clean up deprecated and unused logic
- Validate inputs and handle all enum variants defensively
- Explicitly map RwasmSpecId to inner SpecId

Developer Response The developers implemented items 8, 9, 12-17, and 18. Note that `write_fd` is no longer a supported syscall, and only some of the unused functionality was removed. Other recommendations were not taken due to internal development considerations.

A Appendix

A.1 Intended Behavior: Non-Issues of Note

A.1.1 V-FNT-APP-VUL-001: Rwasm stack may underflow

Severity	Critical	**Commit**	e88ea57
Type	Logic Error	**Status**	Intended Behavior
Location(s)	`crates/runtime/src/syscall_handler/host/` ▶ `forward_output.rs:33` ▶ `write_output.rs:11-14`		

The trust model and assumptions throughout the audited codebase expect rWasm compilation to guarantee that stack underflow cannot occur, and all syscalls rely on this invariant without rechecking stack height. However, we observed a stack underflow when no error or a deterministic `StackUnderflow` trap was expected.

Example A - Stack pointer desynchronization after reallocation In some cases the rWasm stack underflows even when it shouldn't semantically. Logging around `visit_call` in `src/vm/executor/control_flow.rs` indicates that a transient value-stack growth triggers a reallocation; `sync_stack_ptr(self.sp)` updates only the top-of-stack pointer to the new buffer while other derived state still references the old allocation (`sp.src != base_ptr.src`). This may lead to a stack underflow as demonstrated by the PoC below.

Example B – Error in func.ref Calling `func.ref` on a function which is not initialized into a `table` using `elem` causes a failure during deployment. The attached wasm code contains an example. While this is not a stack underflow, it is also an out-of-scope behavior inconsistent with regular wasm code, and so included in this issue.

Impact Denial of service through contracts that either induce stack reallocation and pointer drift or exploit control flow to bypass argument preparation, causing underflow and a process panic in current builds.

Recommendations Improve stack pointer handling and convert any stack underflow into a deterministic trap code (for example StackUnderflow). Remove panic or unwrap paths in stack operations.

PoC

Example A The below script exhibits the error described in Example A above.

```
use crate::EvmTestingContextWithGenesis;
use core::{convert::TryInto, fmt::Write};
use fluentbase_sdk::{calc_create_address, syscall::SYSCALL_ID_CALL, Address, Bytes};
use fluentbase_testing::{EvmTestingContext, TxBuilder};
use wat::parse_str;

const PAGE_SIZE: u32 = 64 * 1024;
const CALL_INPUT_PTR: u32 = 32;
const CALL_METADATA_BYTES: u32 = 20 + 32; // target address + value slot
const OUTPUT_TRAILER_BYTES: u32 = 4;
const FUEL_THRESHOLD: i64 = 25_000_000; // 25k gas in fuel units

fn encode_bytes_for_wat(bytes: &[u8]) -> String {
    let mut out = String::with_capacity(bytes.len() * 4);
    for byte in bytes {
        let _ = write!(&mut out, "\\{:02x}", byte);
    }
    out
}

fn build_call_contract(payload_bytes: u32, current_address: Address) -> Vec<u8> {
    let payload_start = CALL_INPUT_PTR + CALL_METADATA_BYTES;
    let total_bytes = payload_start + payload_bytes + OUTPUT_TRAILER_BYTES;
    let required_pages = (total_bytes + PAGE_SIZE - 1) / PAGE_SIZE;
    let memory_grow_pages = required_pages.saturating_sub(1);
    let call_input_len = CALL_METADATA_BYTES + payload_bytes;
    let syscall_hash = encode_bytes_for_wat(&SYSCALL_ID_CALL.0);
    let address_bytes = current_address.as_slice();
    let call_target_lo = u64::from_le_bytes(address_bytes[0..8].try_into().unwrap());
    let call_target_mid = u64::from_le_bytes(address_bytes[8..16].try_into().unwrap()
    );
    let call_target_hi = u32::from_le_bytes(address_bytes[16..20].try_into().unwrap()
    );

    let wasm_source = format!(
        r#"
        (module
            (import "fluentbase_v1preview" "_exec"  (func $_exec  (param i32 i32 i32
    i32 i32) (result i32)))
            (import "fluentbase_v1preview" "_write" (func $_write (param i32 i32)))
            (import "fluentbase_v1preview" "_fuel"  (func $_fuel  (result i64)))
            (import "fluentbase_v1preview" "_exit"  (func $_exit  (param i32)))
            (memory 1)
            (data (i32.const 0) "{syscall_hash}")
            (func $main (local $out_ptr i32) (local $old_pages i32) (local
    $remaining_fuel i64)
                ;; reserve space for the call buffer and trailing output
                i32.const {memory_grow_pages}
                memory.grow
                local.tee $old_pages
                i32.const -1
                i32.eq
                if
                    i32.const 1
                    call $_exit
```

```
                end

                ;; track the pointer to the byte right after the payload
                i32.const {payload_start}
                i32.const {payload_bytes}
                i32.add
                local.tee $out_ptr

                ;; touch the last byte of the payload so pages materialize
                local.get $out_ptr
                i32.const 1
                i32.sub
                i32.const 0xaa
                i32.store8

                ;; stash payload length for the host
                local.get $out_ptr
                i32.const {payload_bytes}
                i32.store
                drop

                ;; populate target address when there is sufficient fuel to recurse
                call $_fuel
                local.set $remaining_fuel

                local.get $remaining_fuel
                i64.const {fuel_threshold}
                i64.gt_u
                if
                    i32.const {call_input_ptr}
                    i64.const {call_target_lo}
                    i64.store

                    i32.const {call_input_ptr_mid}
                    i64.const {call_target_mid}
                    i64.store

                    i32.const {call_input_ptr_hi}
                    i32.const {call_target_hi}
                    i32.store
                end

                ;; issue CALL with the giant input slice
                i32.const 0
                i32.const {call_input_ptr}
                i32.const {call_input_len}
                i32.const 0
                i32.const 0
                call $_exec
                drop

                ;; return the payload length as proof of execution
                local.get $out_ptr
                i32.const {output_trailer}
                call $_write

                i32.const 0
```

```
                call $_exit
            )
            (export "main" (func $main))
            (export "memory" (memory 0))
        )
        "#,
        memory_grow_pages = memory_grow_pages,
        payload_start = payload_start,
        payload_bytes = payload_bytes,
        call_input_ptr = CALL_INPUT_PTR,
        call_input_len = call_input_len,
        call_input_ptr_mid = CALL_INPUT_PTR + 8,
        call_input_ptr_hi = CALL_INPUT_PTR + 16,
        call_target_lo = format!("{:#018x}", call_target_lo),
        call_target_mid = format!("{:#018x}", call_target_mid),
        call_target_hi = format!("{:#010x}", call_target_hi),
        fuel_threshold = FUEL_THRESHOLD,
        output_trailer = OUTPUT_TRAILER_BYTES,
        syscall_hash = syscall_hash,
    );

    parse_str(&wasm_source).expect("invalid WAT")
}

fn measure_call_gas(payload_bytes: usize) -> (u64, f64) {
    let deployer = Address::ZERO;
    let contract = calc_create_address(&deployer, 0);


    let wasm = match BUILD {
        BuildContract::Example_Underflow => {
            build_call_contract(payload_bytes.try_into().unwrap(), contract)
        }
    };

    let mut ctx = EvmTestingContext::default().with_full_genesis();

    let deploy = TxBuilder::create(&mut ctx, deployer, wasm.into())
        .gas_price(0)
        .gas_limit(80_000_000)
        .exec();
    assert!(deploy.is_success(), "deploy failed: {deploy:?}");

    let call = TxBuilder::call(&mut ctx, Address::ZERO, contract, None)
        .gas_price(0)
        .gas_limit(120_000_000)
        .input(Bytes::default())
        .exec();
    assert!(call.is_success(), "call failed: {call:?}");

    // let output = call.output().cloned().unwrap_or_default();
    // assert_eq!(output.len(), OUTPUT_TRAILER_BYTES, "unexpected output size: {}",
    output.len());
    // let mut buf = [0u8; OUTPUT_TRAILER_BYTES];
    // buf.copy_from_slice(&output);
    // let echoed_len = u32::from_le_bytes(buf) as usize;
    // assert_eq!(echoed_len, payload_bytes, "payload mismatch");
```

```
    let gas_used = call.gas_used();
    let gas_per_byte = gas_used as f64 / payload_bytes as f64;
    (gas_used, gas_per_byte)
}




#[derive(Clone, Copy)]
enum BuildContract {
    Example_Underflow,
}

const BUILD: BuildContract = BuildContract::Example_Underflow;

#[test]
fn jump_test() {
    let payload_sizes = [
        1usize << 12,  // 4 KiB
        1usize << 16,  // 64 KiB
    ];

    let mut observations = Vec::with_capacity(payload_sizes.len());
    for &payload in &payload_sizes {
        let (gas_used, gas_per_byte) = measure_call_gas(payload);
        println!(
            "payload={payload} bytes, gas_used={gas_used}, gas_per_byte={gas_per_byte}"
        );
        observations.push((payload, gas_used, gas_per_byte));
    }

    let &(_, gas_used, gas_per_byte) = observations.last().unwrap();
    println!(
        "largest payload gas summary: bytes={}, gas_used={gas_used}, gas_per_byte={gas_per_byte}",
        payload_sizes.last().unwrap()
    );
    assert!(gas_per_byte < 0.01, "gas per byte too high: {gas_per_byte}");
}
```

Example B The below script exhibits the error described in Example B above.

```
(module
    ;; imports
    (import "fluentbase_v1preview" "_charge_fuel" (func $_charge_fuel (param i64)))
    ;; type of function
    (type $t0 (func (param i64)))
    ;; make function table, pointing to our target function
    (table 1 funcref)
    ;; Define target function to wrap $_charge_fuel
    (func $target (type $t0)
        (call $_charge_fuel (local.get 0))
```

```
    )

    (elem (i32.const 0) $_charge_fuel)
    ;; (elem (i32.const 0) $target)

    (func $main
        ;; test basic call
        i64.const 0
        i32.const 0
        call_indirect (type $t0)

        ;; ;; test indirect call which immediately invokes syscall
        i32.const 0 ;; table index
        ref.func $target;; function pointer
        table.set

    )
    (memory 1)
    (export "main" (func $main))
    (export "memory" (memory 0))
)
```

Developer Response The developers have been notified of the issue, but not yet provided a response.

A.1.2 V-FNT-APP-VUL-002: Exit code set inconsistently for syscalls

Severity	Info	**Commit**	e88ea57
Type	Maintainability	**Status**	Intended Behavior
Location(s)	`crates/runtime/src/syscall_handler/` ▶ `edwards/` • `edwards_add.rs` • `edwards_decompress.rs` ▶ `hashing/poseidon.rs` ▶ `host/write_fd.rs` ▶ `weierstrass/weierstrass_decompress.rs`		

Description The function `handle_execution_result()` is responsible for setting the exit code in case of an error.

```
match next_result {
    Ok(_) => {}
    Err(TrapCode::InterruptionCalled) => {
        return self.handle_resumable_state(execution_result, ctx);
    }
    Err(err) => {
        execution_result.exit_code = ExitCode::from(err).into_i32();
    }
}
```

Snippet A.1: Snippet from `handle_execution_result()`

However, some syscall handlers manually set the exit code as well.

```
let res = syscall_edwards_add_impl(p_bytes, q_bytes)
    .map_err(|e| syscall_process_exit_code(ctx, e))?;
```

Snippet A.2: Snippet from `syscall_edwards_add_handler()`

This setting is unnecessary, duplicating functionality from the execution result handler and leading to inconsistencies between different syscall implementations.

Impact Developers / auditors may be confused as to the intended behavior of syscalls. This may lead to users setting the exit code to custom values, which will be overridden by the system.

Recommendation Remove the `syscall_process_exit_code()` function, and any of its invocations.

Developer Response We have two structures: **TrapCode** and **ExitCode**. **TrapCode** represents halted execution at the *rWasm* level. Sometimes we need to stop execution without producing any trap codes. To support this, we use a special trap code: **ExecutionHalted**. `ExecutionHalted` is a meta-trap code. It never propagates back to the caller. During rWasm execution, the VM intercepts `Err(TrapCode::ExecutionHalted)`and replaces it with `Ok(())`.

This handling occurs inside the rWasm VM:

`https://github.com/fluentlabs-xyz/rwasm/blob/devel/src/vm/executor.rs#L439`

ExitCode is a higher-level structure used at the Fluentbase layer. It represents application-level exit codes (similar to panic codes). If a `TrapCode` occurs during execution, the system automatically maps the `TrapCode` into an appropriate `ExitCode`. In the example:

```
let res = syscall_edwards_add_impl(p_bytes, q_bytes)
    .map_err(|e| syscall_process_exit_code(ctx, e))?;
```

we use `TrapCode::ExecutionHalted` to signal that execution stops cleanly without bubbling an error upward.

dos-builtin-panics.yml

```yaml
rules:
  - id: dos-builtin-panics
    message: "‘unwrap_unchecked‘ and ‘assume_init‘ can cause UB/panics."
    languages: [rust]
    severity: WARNING
    patterns:
      - pattern-either:
          - pattern: unsafe { $EXPR.unwrap_unchecked() }
          - pattern: unsafe { $EXPR.assume_init() }
      - pattern-not-inside: |
          #[cfg(test)]
          mod $M {
            ...
          }
      - pattern-not-inside: |
          #[cfg(test)]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          #[test]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          mod tests {
            ...
          }
      - pattern-not-inside: |
          mod test {
            ...
          }
    paths:
      include:
        - "/crates/revm/**"
        - "/crates/runtime/**"
      exclude:
        - "**/tests/**"
        - "**/*_test.rs"
        - "**/test_*.rs"

```

dos-copy-from-slice.yml

```yaml
rules:
  - id: dos-copy-from-slice
    message: "‘copy_from_slice‘ panics on length mismatch; validate buffer sizes"
```

```
    languages: [rust]
    severity: WARNING
    patterns:
      - pattern: $DST.copy_from_slice($SRC)

      - pattern-not-inside: |
          let mut $DST = [$VALUE; $SIZE]
          ...
          $DEREFERENCED_SRC.resize($SIZE, $VALUE);
          ...
          $$DST.copy_from_slice(&$DEREFERENCED_SRC);

      - pattern-not-inside: |
          $DEREFERENCED_SRC.resize($SIZE, $VALUE);
          ...
          $DST_ARR[..$SIZE].copy_from_slice(&$DEREFERENCED_SRC);

      # NOTE: This should actually be implemented with semgrep:ignore comments,
      #       but for the purposes of time we just filter them out aggressively
      - pattern-not-inside: |
          $DEREFERENCED_SRC.resize($SIZE, $VALUE);
          ...
          $DST_ARR[$SIZE..].copy_from_slice(&_DEREFERENCED_SRC);

      - pattern-not-inside: |
          #[cfg(test)]
          mod $M {
            ...
          }
      - pattern-not-inside: |
          #[cfg(test)]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          #[test]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          mod tests {
            ...
          }
      - pattern-not-inside: |
          mod test {
            ...
          }
      - metavariable-regex:
          metavariable: $DST
          regex: "^(?!.*get(_mut)?\\().*$"
    paths:
      include:
        - "/crates/revm/**"
        - "/crates/runtime/**"
      exclude:
        - "**/tests/**"
        - "**/*_test.rs"
```

```
          - "**/test_*.rs"
          - "**/testing_*.rs"

```

dos-expect.yml

```yaml
rules:
  - id: dost-expect
    message: "`expect()` panics on failure; propagate errors instead."
    languages: [rust]
    severity: WARNING
    patterns:
      - pattern: $VAL.expect($MSG)
      # Ignore common fluent patterns that take wasm values for syscalls
      - pattern-not: |
          ($IGN: Value).i32().expect($MSG)
      - pattern-not: |
          ($IGN: Value).i64().expect($MSG)
      - pattern-not: |
          ($IGN: Value).f32().expect($MSG)
      - pattern-not: |
          ($IGN: Value).f64().expect($MSG)
      - pattern-not: |
          ($IGN: Value).try_into().expect($MSG)
      - pattern-not-inside: |
          #[cfg(test)]
          mod $M {
            ...
          }
      - pattern-not-inside: |
          #[cfg(test)]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          #[test]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          mod tests {
            ...
          }
      - pattern-not-inside: |
          mod test {
            ...
          }
    paths:
      include:
        - "/crates/revm/**"
        - "/crates/runtime/**"
      exclude:
        - "**/tests/**"
        - "**/*_test.rs"
        - "**/test_*.rs"
```

```
50
```

dos-indexing.yml

```yaml
rules:
  - id: dos-indexing
    message: "Indexing may triger OOB error"
    languages: [rust]
    severity: CRITICAL
    patterns:
      - pattern: |
          $VEC[$IDX]

      # Exclude safe slices
      - pattern-not-inside: |
          if $DEREFERENCED_VEC.len() > $SIZE {
              &$DEREFERENCED_VEC[..$SIZE]
          }

      # Exclude syscall unwraps
      # NOTE: This should actually be implemented with semgrep:ignore comments,
      #       but for the purposes of time we just filter them out aggressively
      - pattern-not-inside: |
          params[$IDX].i32().unwrap()
      - pattern-not-inside: |
          params[$IDX].i64().unwrap()

      # Exclude syscall output writes
      # NOTE: This should actually be implemented with semgrep:ignore comments,
      #       but for the purposes of time we just filter them out aggressively
      - pattern-not-inside: |
          result[$OUTPUT_IDX] = Value::$VALUE_TYPE($RESULT);

      # Exclude indexing into fixed-size arrays.
      # NOTE: This should actually be implemented with semgrep:ignore comments,
      #       but for the purposes of time we just filter them out aggressively
      - pattern-not-inside:
          let $VEC = [$VAL; $SIZE];
          ...
          $VEC[$IDX]

      # Filter out tests
      - pattern-not-inside: |
          #[cfg(test)]
          mod $M {
            ...
          }
      - pattern-not-inside: |
          #[cfg(test)]
          fn $F(...) {
            ...
          }
      - pattern-not-inside: |
          #[test]
          fn $F(...) {
```

```
            ...
          }
      - pattern-not-inside: |
          mod tests {
            ...
          }
      - pattern-not-inside: |
          mod test {
            ...
          }

    paths:
      include:
        - "/crates/revm/**"
        - "/crates/runtime/**"
      exclude:
        - "**/tests/**"
        - "**/*_test.rs"
        - "**/test_*.rs"

```

dos-memory-alloc.yml

```
rules:
  - id: dos-memory-alloc
    message: "Unbounded memory allocation may lead to a denial of service if not charged for."
    languages: [rust]
    severity: CRITICAL
    patterns:
      - pattern-either:
        # Macro case, need to use regex
        - patterns:
          - pattern-regex: '(?s)vec!\s*\[\s*([^\];]+)\s*;\s*([^\]]+)\]'
          #  - integer literals (with underscores and optional suffixes)
          #  - ALL_CAPS-style const idents
          - pattern-not-regex: '(?s)vec!\s*\[\s*[^\];]+\s*;\s*(?:\d[\d_]*(?:usize|isize|u64|i64|u32|i32|u16|i16|u8|i8)?|[A-Z_][A-Z0-9_]*)\s*\]'
        # Non-macro case, where we can parse the args
        - patterns:
          - pattern: |
              [$VAL; $SIZE]
          - pattern: |
              alloc::vec::from_elem($VAL, $SIZE)
          - pattern: |
              Vec::with_capacity($VAL, $SIZE)
          - metavariable-pattern:
              metavariable: $SIZE
              patterns:
                # Filter out common constants
                - pattern-not: 7
                - pattern-not: 8
                - pattern-not: 16
                - pattern-not: 32
```

```
                - pattern-not: 48
                - pattern-not: 64
                - pattern-not: 200
                - pattern-not: 256
                - pattern-not: COMPRESSED_SIZE
                - pattern-not: ED25519_POINT_COMPRESSED_SIZE
                - pattern-not: ED25519_POINT_DECOMPRESSED_SIZE
                - pattern-not: NUM_BYTES
                - pattern-not: POINT_SIZE

        # Filter out tests
        - pattern-not-inside: |
            #[cfg(test)]
            mod $M {
              ...
            }
        - pattern-not-inside: |
            #[cfg(test)]
            fn $F(...) {
              ...
            }
        - pattern-not-inside: |
            #[test]
            fn $F(...) {
              ...
            }
        - pattern-not-inside: |
            mod tests {
              ...
            }
        - pattern-not-inside: |
            mod test {
              ...
            }

      paths:
        include:
          - "/crates/revm/**"
          - "/crates/runtime/**"
        exclude:
          - "**/tests/**"
          - "**/*_test.rs"
          - "**/test_*.rs"

```

dos-panic-macros.yml

```
rules:
  - id: dos-assert
    message: "macro panics abort execution; consider error handling."
    languages: [rust]
    severity: WARNING
    patterns:
      - pattern-either:
          - pattern: panic!(...)
```

```
            - pattern: todo!(...)
            - pattern: unimplemented!(...)
            - pattern: unreachable!(...)
        - pattern-not-inside: |
            #[cfg(test)]
            mod $M {
              ...
            }
        - pattern-not-inside: |
            #[cfg(test)]
            fn $F(...) {
              ...
            }
        - pattern-not-inside: |
            #[test]
            fn $F(...) {
              ...
            }
        - pattern-not-inside: |
            mod tests {
              ...
            }
        - pattern-not-inside: |
            mod test {
              ...
            }
      paths:
        include:
          - "/crates/revm/**"
          - "/crates/runtime/**"
        exclude:
          - "**/tests/**"
          - "**/*_test.rs"
          - "**/test_*.rs"

```

dos-unwrap.yml

```
rules:
  - id: dos-unwrap
    message: "`unwrap()` panics on `Err`/`None`; ensure upstream handling."
    languages: [rust]
    severity: WARNING
    patterns:
      - pattern: $VAL.unwrap()
      - pattern-not-inside: |
          $VEC.resize($SIZE, $FILL_VAL);
          let $ARR: [$TYP; $SIZE] = $VEC.try_into().unwrap();

      # Exclude syscall unwraps
      # NOTE: This should actually be implemented with semgrep:ignore comments,
      #       but for the purposes of time we just filter them out aggressively
      - pattern-not: |
          params[$IDX].i32().unwrap()
      - pattern-not: |
```

```
            params[$IDX].i64().unwrap()

        # Exclude tests
        - pattern-not-inside: |
            #[cfg(test)]
            mod $M {
              ...
            }
        - pattern-not-inside: |
            #[cfg(test)]
            fn $F(...) {
              ...
            }
        - pattern-not-inside: |
            #[test]
            fn $F(...) {
              ...
            }
        - pattern-not-inside: |
            mod tests {
              ...
            }
        - pattern-not-inside: |
            mod test {
              ...
            }
      paths:
        include:
          - "/crates/revm/**"
          - "/crates/runtime/**"
        exclude:
          - "**/tests/**"
          - "**/*_test.rs"
          - "**/test_*.rs"

```

Glossary

revm `https://github.com/bluealloy/revm.git` . 1, 2

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1

Wasm WebAssembly, or Wasm, is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications. See `https://webassembly.org` to learn more . 1

Zero Knowledge zero-knowledge circuit. 1

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 93