SwapVM + Aqua Security Audit Report

Version 1.0

Vadim Fadeev

January 7, 2026

SwapVM + Aqua Security Audit Report

Vadim Fadeev

January 6, 2026

SwapVM + Aqua Security Audit Report

Prepared by: Vadim Fadeev
Lead Auditors:

- Vadim Fadeev

Assisting Auditors:

- None

Table of contents

See table

About Vadim Fadeev

Disclaimer

The Vadim Fadeev team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

Risk Classification

Likelihood	High Impact	Medium Impact	Low Impact
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
89cedfa608aa9fdc60d8cd936b76a540e4b9346f
```

Scope

This report covers the SwapVM + Aqua integration codebase.

SwapVM

In scope - `src/*.sol` - `src/routers/AquaSwapVMRouter.sol` - `src/opcodes/AquaOpcodes.sol` - `src/instructions/Balances.sol` - `src/instructions/Controls.sol` - `src/instructions/Decay.sol` - `src/instructions/Extruction.sol` - `src/instructions/Fee.sol` - `src/instructions/PeggedSwap.sol` - `src/instructions/XYCConcentrate.sol` - `src/instructions/XYCSwap.sol` - `src/libs/MakerTraits.sol` - `src/libs/PeggedSwapMath.sol` - `src/libs/TakerTraits.sol` - `src/libs/VM.sol`

Methodology

- Manual review of instruction contracts and shared libraries
- Reasoning about invariants and state transitions (especially: balances bookkeeping, fee layering, swap math)
- Threat modeling around “maker-programmed” execution and “taker-executed” swaps
- Cross-reference with Solodit Security Checklist
- Identification of:
  - panic reverts / implicit underflow/overflow reliance
  - initialization sentinels and state ambiguity

- external call surfaces (fee providers, extruction targets)
- denial-of-service vectors (gas griefing / malformed programs)

Threat Model and Trust Assumptions

SwapVM is explicitly a **programmable execution environment**: - Makers define a program (bytecode-like instruction stream). - Takers execute it (often via routers) and must assume the maker program may be adversarial.

Therefore: - Many issues are **"order-level misconfiguration or malicious program risks"** rather than protocol-wide vulnerabilities. - Wherever possible, the protocol should fail with **clear errors**, enforce **basic numeric bounds**, and prevent **state ambiguity** that can be exploited across calls.

External call surfaces: - `Fee._dynamicProtocolFeeAmountInXD` calls an untrusted fee provider via `staticcall`. - `Extruction._extruction` calls an arbitrary target contract (view or non-view depending on context). These are inherently risky and should be treated as part of the trusted computing base *for that specific program/order*.

System Overview

Aqua (Shared Liquidity Layer)

Aqua is a shared liquidity layer that manages balances/allowances between makers and "apps/strategies", allowing apps to pull from maker wallets while tracking balances.

SwapVM (Programmed Swaps)

SwapVM provides: - A **VM** (`Context`, `VM`, `ContextLib::runLoop`) executing an instruction stream from calldata. - A set of **instructions** (e.g., `Balances`, `Fee`, `XYCSwap`, `PeggedSwap`, `Decay`, `Controls`, `Extruction`). - Routers/opcode registries (e.g., `AquaOpcodes`, `AquaSwapVMRouter`) assembling instruction tables.

Stateful instructions: - `Balances` maintains per-order token balances. - `Decay` maintains per-order decaying offsets. - `XYCConcentrate` tracks per-order liquidity scaling.

Executive Summary

Issues found

Severity	Number of issues found
High	0
Medium	5
Low	5
Gas Optimizations	0
Total	**10**

Key Takeaways

- Two **Medium severity** division-by-zero vulnerabilities were identified in `Fee.sol` and `XYCConcentrate.sol` that can permanently brick orders.
- Several core instructions rely on **implicit panics** (division-by-zero, underflow) rather than explicit validation with custom errors, which harms:
  - debuggability
  - integrator UX
  - security monitoring / incident response
- External call surfaces (`dynamic fee provider`, `extruction`) are legitimate features and should be clearly documented as **trust boundaries**.

Findings

Medium

[M-1] Division-by-zero if `feeBps == BPS` in ExactOut fee calculation

File: `src/instructions/Fee.sol:196`

Description:
In the ExactOut path, `_feeAmountIn()` computes:

```
1  feeAmountIn = ctx.swap.amountIn * feeBps / (BPS - feeBps);
```

If `feeBps == BPS` (100% fee), the denominator becomes zero, causing a panic revert.

Root Cause:

- `FeeArgsBuilder.buildFlatFee()` validates `feeBps <= BPS` (allows equality) - `parseFlatFee()` has no runtime validation - Programs can encode `feeBps = BPS` directly

Impact:

- Orders with 100% fee permanently bricked in ExactOut mode - Panic revert instead of clear custom error

Verification:

```
// Fee.sol:196
feeAmountIn = ctx.swap.amountIn * feeBps / (BPS - feeBps);
// If feeBps = 1e9 (BPS), then (BPS - feeBps) = 0 → division by zero
```

Recommended Mitigation:

```
function _feeAmountIn(Context memory ctx, uint256 feeBps) internal
    returns (uint256 feeAmountIn) {
    require(ctx.swap.amountIn == 0 || ctx.swap.amountOut == 0, ...);
+   require(feeBps < BPS, FeeBpsCannotBe100PercentForExactOut(feeBps));
    // ...
}
```

[M-2] Division-by-zero if `initialLiquidity == 0` in XYCConcentrate

File: `src/instructions/XYCConcentrate.sol:113`

Description:

The `concentratedBalance()` function divides by `initialLiquidity`:

```
function concentratedBalance(..., uint256 initialLiquidity) public view
    returns (uint256) {
    uint256 currentLiquidity = liquidity[orderHash];
    return currentLiquidity == 0
        ? balance + delta
        : balance + delta * currentLiquidity / initialLiquidity;  // ← Division!
}
```

Attack Scenario: 1. Maker creates order with `initialLiquidity = 0` in program args 2. First trade succeeds (`currentLiquidity == 0` takes first branch) 3. `_updateScales()` sets `liquidity[orderHash]` to non-zero value 4. Second trade: `currentLiquidity != 0`, divides by `initialLiquidity = 0` → **PANIC**

Impact:

- Order permanently bricked after first trade - Maker loses access to remaining liquidity - No recovery mechanism

Verification:

```
// XYCConcentrate.sol:111-114
function concentratedBalance(bytes32 orderHash, uint256 balance, uint256 delta, uint256 initialLiquidity) public view returns (uint256) {
    uint256 currentLiquidity = liquidity[orderHash];
    return currentLiquidity == 0 ? balance + delta : balance + delta * currentLiquidity / initialLiquidity;
}
// Line 157: liquidity[ctx.query.orderHash] = Math.sqrt(newInv); // Sets non-zero after first trade
```

Recommended Mitigation:

```
function parse2D(bytes calldata args, address tokenIn, address tokenOut) internal pure returns (...) {
    // ... existing parsing ...
    liquidity = uint256(bytes32(args.slice(64, 96, ...)));
+   require(liquidity > 0, ConcentrateInitialLiquidityMustBeNonZero());
}
```

[M-3] `Decay::_decayXD` underflow when offset exceeds balance

File: `src/instructions/Decay.sol:87`

Description:

```
ctx.swap.balanceOut -= _offsets[ctx.query.orderHash][ctx.query.tokenOut][false].getOffset(period);
```

If the decayed offset exceeds `balanceOut`, subtraction underflows → panic.

How this happens: 1. Initial `balanceOut = 100` 2. Large swap adds offset of 80 to tokenOut 3. Another swap reduces `balanceOut` to 50 via Balances instruction 4. Offset decays to 60, but `50 - 60` underflows

Impact: - Orders become temporarily or permanently unusable - Depends on interaction between Decay and balance-modifying instructions

Recommended Mitigation:

```
uint256 offsetOut = _offsets[ctx.query.orderHash][ctx.query.tokenOut][
    false].getOffset(period);
ctx.swap.balanceOut = ctx.swap.balanceOut > offsetOut ? ctx.swap.
    balanceOut - offsetOut : 0;
```

[M-4] `VM::runLoop` lacks bounds checking for malformed programs

File: `src/libs/VM.sol:97-107`

Description:

```
for (uint256 pc = ctx.vm.nextPC; pc < programBytes.length; ) {
    unchecked {
        uint256 opcode = uint8(programBytes[pc++]);      // Read byte 1
        uint256 argsLength = uint8(programBytes[pc++]);  // Read byte 2
            - OOB if only 1 byte left!
        uint256 nextPC = pc + argsLength;
        bytes calldata args = programBytes[pc:nextPC];   // OOB if
            argsLength exceeds remaining
        // ...
    }
}
```

Problem scenarios: 1. **Single trailing byte**: Loop enters with 1 byte remaining, reading `argsLength` panics 2. **argsLength exceeds remaining bytes**: Slice operation panics

Impact: - Malformed programs cause opaque panic reverts - No way for takers to validate program well-formedness before execution

Recommended Mitigation:

```
for (uint256 pc = ctx.vm.nextPC; pc < programBytes.length; ) {
+   require(pc + 1 < programBytes.length, RunLoopMalformedInstruction(
    pc));

    uint256 opcode = uint8(programBytes[pc]);
    uint256 argsLength = uint8(programBytes[pc + 1]);
    uint256 nextPC = pc + 2 + argsLength;

+   require(nextPC <= programBytes.length, RunLoopArgsExceedProgram(pc,
     argsLength));
+   require(opcode < ctx.vm.opcodes.length, RunLoopInvalidOpcode(opcode
    ));

    bytes calldata args = programBytes[pc + 2:nextPC];
```

```
12      // ...
13  }
```

[M-5] Invalid opcode causes array out-of-bounds panic

File: `src/libs/VM.sol:105`

Description:

```
1  ctx.vm.opcodes[opcode](ctx, args);
```

If `opcode >= ctx.vm.opcodes.length`, this causes an array index out-of-bounds panic.

Impact: - Malformed programs with invalid opcodes revert with opaque panics - No clear error message for debugging

Recommended Mitigation:

```
1  require(opcode < ctx.vm.opcodes.length, RunLoopInvalidOpcode(opcode,
       ctx.vm.opcodes.length));
2  ctx.vm.opcodes[opcode](ctx, args);
```

Low

[L-1] Fee does not enforce `feeBps <= BPS` at runtime for flat fees

File: `src/instructions/Fee.sol:42-44`

Description:
`parseFlatFee()` has no bounds check; only `buildFlatFee()` validates. Programs can bypass the builder with arbitrary `feeBps` values.

Impact: - `feeBps > BPS` could cause underflow in fee subtraction - Opaque panics instead of clear errors

Recommended Mitigation:

```
1  function _feeAmountIn(Context memory ctx, uint256 feeBps) internal
       returns (uint256 feeAmountIn) {
2  +    require(feeBps <= BPS, FeeBpsOutOfRange(feeBps));
```

```
3       // ...
4   }
```

[L-2] `Decay::decayPeriod == 0` should be validated

File: `src/instructions/Decay.sol:45`

Description:
While analysis shows division-by-zero is practically unreachable due to timestamp logic, a `decayPeriod = 0` value is clearly invalid and should be rejected at parse time for defense-in-depth.

Recommended Mitigation:

```
1  function parse(bytes calldata args) internal pure returns (uint16
       period) {
2      period = uint16(bytes2(args.slice(0, 2, DecayMissingPeriodArg.
           selector)));
3  +   require(period > 0, DecayPeriodMustBeNonZero());
4  }
```

[L-3] `Controls::_onlyTakerTokenSupplyShareGte` has potential overflow

File: `src/instructions/Controls.sol`

Description:
The share calculation multiplies balance by 1e18:

```
1  require(totalSupply > 0 && balance * 1e18 >= minShareE18 * totalSupply,
       ...);
```

If `balance > type(uint256).max / 1e18` (~1.15e59), multiplication overflows.

Impact: - Tokens with extremely high decimals (27+) or supply could cause unexpected reverts - Panic error instead of clear message

Recommended Mitigation:

```
1  uint256 share = Math.mulDiv(balance, 1e18, totalSupply);
2  require(share >= minShareE18, ...);
```

[L-4] Unused imports

Files: - `src/instructions/Fee.sol:7` - `src/instructions/FeeExperimental.sol:12` - `src/libs/TakerTraits.sol:11`

Description:
Redundant import statements that are not used in the contracts:

```
// Fee.sol
import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";

// FeeExperimental.sol
import { IProtocolFeeProvider } from "./interfaces/IProtocolFeeProvider.sol";

// TakerTraits.sol
import { ITakerCallbacks } from "../interfaces/ITakerCallbacks.sol";
```

Impact: - Slightly increases bytecode size - Reduces code clarity - May confuse auditors about expected functionality

Recommended Mitigation: Remove the unused import statements.

[L-5] State changes without event emission in Invalidators

File: `src/instructions/Invalidators.sol:52-60`

Description:
Three functions modify state but emit no events:

```
function invalidateBit(uint256 bitIndex) external {
    // State change, no event
}

function invalidateTokenIn(bytes32 orderHash, address tokenIn) external {
    // State change, no event
}

function invalidateTokenOut(bytes32 orderHash, address tokenOut) external {
    // State change, no event
```

```
11 }
```

Impact: - Off-chain indexers cannot track invalidation changes - Reduced transparency for order lifecycle monitoring - Harder to debug invalidation-related issues

Recommended Mitigation: Add events for each state change:

```
1 event BitInvalidated(address indexed maker, uint256 bitIndex);
2 event TokenInInvalidated(address indexed maker, bytes32 indexed
     orderHash, address tokenIn);
3 event TokenOutInvalidated(address indexed maker, bytes32 indexed
     orderHash, address tokenOut);
```

Appendix

Hardening Checklist

1. **Add explicit initialization tracking** for `Balances::_dynamicBalancesXD` and `XYCConcentrate`

2. **Add runtime bounds checks** for:
   - `feeBps < BPS` for ExactOut paths
   - `feeBps <= BPS` for all fee instructions
   - `decayPeriod > 0`
   - `initialLiquidity > 0`

3. **Add program encoding validation** in `VM::runLoop`:
   - Instruction header bounds
   - Args bounds
   - Opcode bounds

4. **Cap returndata handling** for dynamic fee providers

5. **Use `Math.mulDiv`** for overflow-safe calculations in:
   - `Controls::_onlyTakerTokenSupplyShareGte`
   - `XYCConcentrate::concentratedBalance`

6. **Add fuzz tests** for:

- Fee extremes (0, near BPS, BPS)
- Malformed programs (odd lengths, invalid opcodes)
- Zero-amount swaps with `allowZeroAmountIn`
- Multi-swap sequences with Decay
- Concentrate liquidity scaling over multiple trades

References

- Solodit Security Checklist
- OpenZeppelin Math Library