EIP 7002

Smart Contract Security Assessment

December 13, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the system contracts for EIP7002. In total one medium severity Denial of Service related issue was found, in addition to two advisory issues related to potential gas optimisations.

BACKGROUND

In Ethereum's staking model, a validator is managed by two separate credentials: a "hot" key and a "cold" key. The hot key is a BLS key used by the validator to sign messages and perform its consensus duties, while the cold key is an Ethereum address (either an externally owned account or a smart contract) that receives withdrawals.

This division can create problems when the hot key and cold key are controlled by different parties. Since only the hot key holder can trigger withdrawals at the consensus layer, the party holding that key can effectively hold staked ETH and accrued rewards hostage, preventing the rightful owner (the cold key holder) from accessing their funds. Whilst this is not an issue for home stakers, it poses significant risks due to the proliferation of liquid staking protocols.

EIP-7002 addresses this issue by allowing the withdrawal credential holder to directly trigger validator exits and partial withdrawals. By shifting initiation control to the credential that actually owns the staked funds, validators can trustlessly manage and recover their stakes.

SETTING & CAVEATS

This audit report mainly covers the contracts of the **sys-asm** repository at commit `1d679164e03d73dc7f9a5331b67fd51e7032b104`, as well as the code, pseudocode, and specifications in the EIP (considering the EIP definition at commit `28ee5e8562a1d4437032718dab1b8fa5b031bda4` of the **ethereum/EIPs** repository).

2 auditors worked on the codebase for 4 days (each) on the following contracts:

```
src/
├── withdrawals/
│   ├── ctor.eas
│   └── main.eas
└── common/
    └── fake_expo.eas
```

Throughout the audit the auditors reviewed the code at numerous levels of abstraction, including the Geas code itself, the disassembled bytecode, as well as the three-address-code (TAC) and decompilation produced by our decompiler. This was done to ensure both human and programmatic understanding of the code, as well as isolating any potential bugs introduced by Geas.

The audit was primarily focused on the security of the smart contracts, and their adherence to the given specification. How the rest of the EIP is implemented into the protocol is outside of the scope of this audit, however the auditors did attempt to identify potential cross cutting issues based on the code in scope.

CONTRACT LOGIC OVERVIEW

The contract's logic is divided into two main flows based on the caller's address: a "system" path and a "user" path. If the caller is the system address (`0xfffffffffffffffffffffffffffffffffffffffe`) the contract runs the system path; otherwise, it runs the user path.

On the user path, if no call data and no value are provided, the contract returns the current request fee. If valid call data and sufficient ETH are sent, the contract adds the user's request to a queue. The required ETH is determined by an exponential function based on the number of excess requests (i.e., those beyond the target). Each successfully added request increments a counter tracking the total requests for the current block.

On the system path, the contract dequeues up to `MAX_CONSOLIDATION_REQUESTS_PER_BLOCK` requests (the maximum is currently set at 16). It should be noted that removing requests from the queue does not delete their used storage slots. After processing these, it resets the request counter and updates the "excess" value, which records how many requests exceeded the target (specified by the `TARGET_CONSOLIDATION_REQUESTS_PER_BLOCK` parameter, currently set at 2). Finally, the contract returns the collected requests.

Fee Calculation

Submitting a new request is associated with a fee, which is computed based on the number of excess requests over the contract's target . The motivation behind this fee is DoS protection.
Specifically the fee calculation approximates the following formula:
`MIN_WITHDRAWAL_REQUEST_FEE * e**(excess / WITHDRAWAL_REQUEST_FEE_UPDATE_FRACTION)`
with the current values of `WITHDRAWAL_REQUEST_FEE_UPDATE_FRACTION` and `MIN_WITHDRAWAL_REQUEST_FEE` being 17 and 1 respectively.

The number of excess requests is updated once per block, during the system call.

The following table presents the fee cost in ETH, for different excess request numbers:

# Excess Requests	Fee (ETH)
0	1.00e-18
50	1.80e-17
100	3.57e-16
150	6.78e-15
200	1.29e-13
250	2.44e-12
300	4.61e-11
350	8.73e-10
400	1.65e-08
450	3.13e-07
500	5.93e-06
550	0.0001123830375
600	0.002128277913
650	0.04030500032
700	0.7632918597
750	14.4559702
800	273.7625199
850	5184.695758

Decompiled Contract

The annotated decompiled code expresses the entirety of the contract's logic:

User Path

```
require(uint256.max != excess_request_count);
// fake_exponential starts here
v12 = 17;
v14 = 1;
v16 = 0;
while (v12 > 0) {
    v16 += v12;
    v12 = excess_request_count * v12 / (v14 * 17);
    v14 += 1;
}
// fake_exponential ends here, result is "v16 / 17"

if (56 == msg.data.length) {
    // add_consolidation_request starts here
    require(msg.value >= v16 / 17);
    request_count += 1;
    STORAGE[4 + 3 * queue_tail_index] = msg.sender;
    // stores first 32 bytes of validator_pubkey
    STORAGE[5 + 3 * queue_tail_index] = calldata_0_32;
    // stores last 16 bytes of validator_pubkey
    // and 8 bytes of amount (big endian) last 8 bytes are empty
    STORAGE[6 + 3 * queue_tail_index] = calldata_32_64;

    // stores msg.sender (20 bytes) to memory at offset 0
    MEM[0] = msg.sender << 96;
    // copies the public key and amount (56 bytes) to memory at
offset 20
```

```
    CALLDATACOPY(20, 0, 56);
    log MEM[0:76] // logs [msg.sender ++ calldata_0_56];

    queue_tail_index += 1;
    exit;
    // add_consolidation_request ends here
} else {
    // fee_getter starts here
    require(!msg.data.length);
    require(!msg.value);
    return v16 / 17;
    // fee_getter ends here
}
```

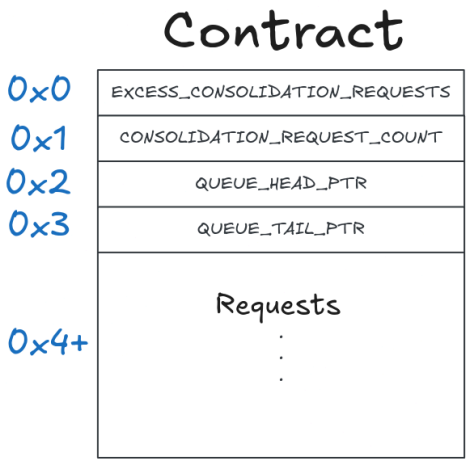
System Path

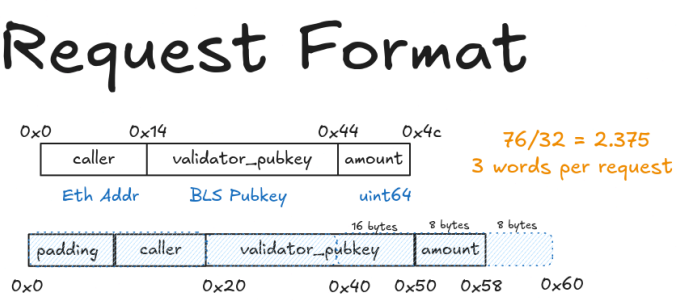
```
v0 = v1 = queue_tail_index - queue_head_index;
if (16 <= v1) {
    v0 = 16;
}

v3 = 0;
while (v3 != v0) {
    // copy source_address into the first 20 bytes in memory
    MEM[76 * v3] = STORAGE[4 + 3 * (v3 + queue_head_index)] << 96;

    // copy first 32 bytes of validator_pubkey into memory
    MEM[20 + 76 * v3] = STORAGE[5 + 3 * (v3 + queue_head_index)];

    // copy next 16 bytes of validator_pubkey into memory
    MEM[52 + 76 * v3] = bytes16(STORAGE[6 + 3 * (v3 +
queue_head_index)]);
    v5 = STORAGE[6 + 3 * (v3 + queue_head_index)] >> 64;
```

```
    // little_endian_to_uint64, copy 8 bytes into memory
    MEM8[75 + 76 * v3] = v5 >> 56 & 0xFF;
    MEM8[74 + 76 * v3] = v5 >> 48 & 0xFF;
    MEM8[73 + 76 * v3] = v5 >> 40 & 0xFF;
    MEM8[72 + 76 * v3] = v5 >> 32 & 0xFF;
    MEM8[71 + 76 * v3] = v5 >> 24 & 0xFF;
    MEM8[70 + 76 * v3] = v5 >> 16 & 0xFF;
    MEM8[69 + 76 * v3] = v5 >> 8 & 0xFF;
    MEM8[68 + 76 * v3] = v5 & 0xFF;
    v3 += 1;
}

if (queue_tail_index == queue_head_index + v0) {
    queue_head_index = 0;
    queue_tail_index = 0;
} else {queue_head_index = queue_head_index + v0;}

// update_excess_withdrawal_requests starts here!
v6 = v7 = excess_request_count;
if (uint256.max == v7) {
    v6 = 0;
}
if (request_count + v6 > 2) {
    v9 = request_count + v6 - 2;
} else {
    v9 = 0;
}
excess_request_count = v9;
// update_excess_withdrawal_requests ends here!
// reset_withdrawal_requests_count starts here
request_count = 0;
// reset_withdrawal_requests_count ends here
return MEM[0:76 * v3];
```

Diagrams

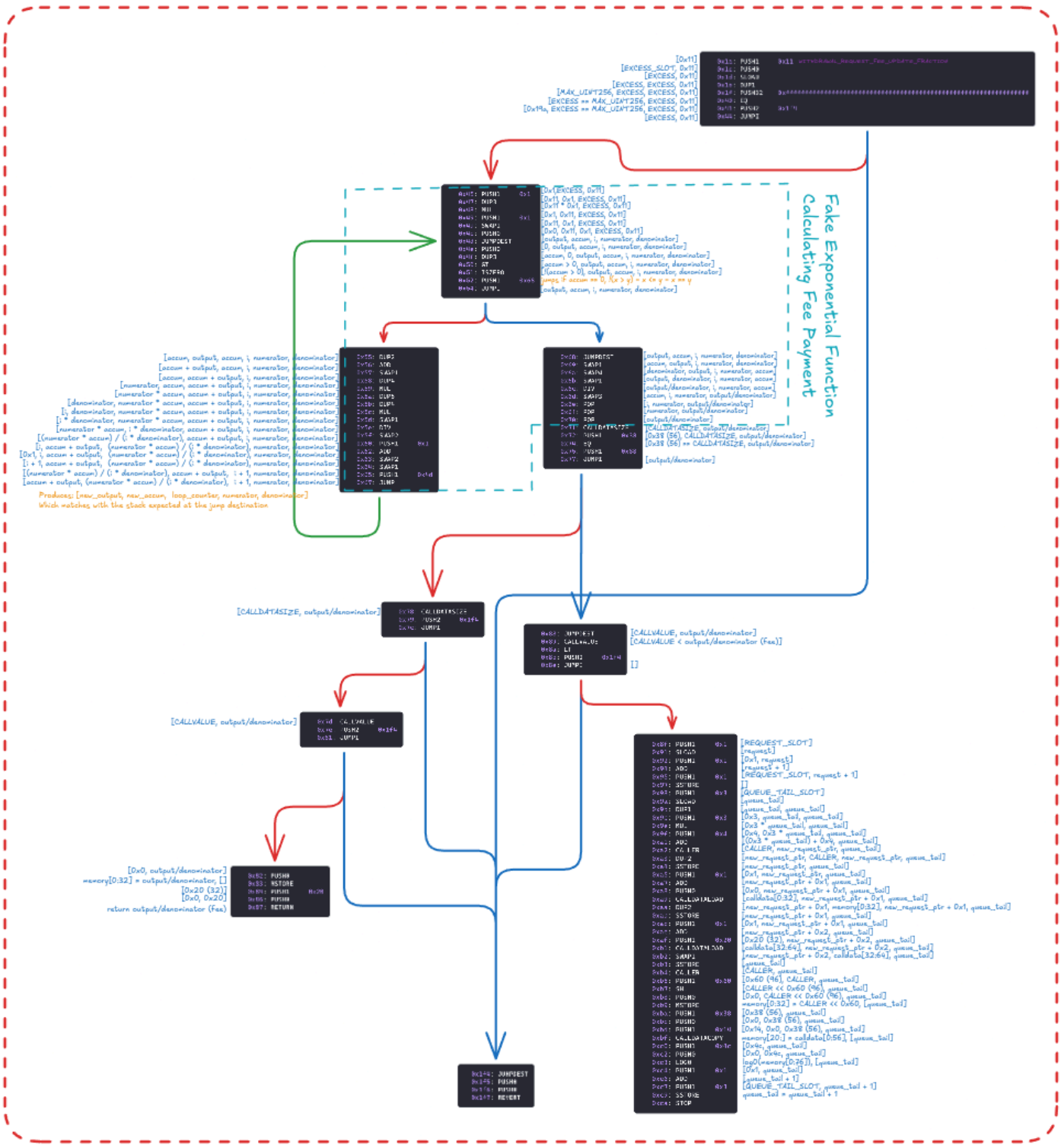
To assist with the understanding of the contract we have produced a series of diagrams, the following outlines the storage layout of the contract and the layout of requests in storage.



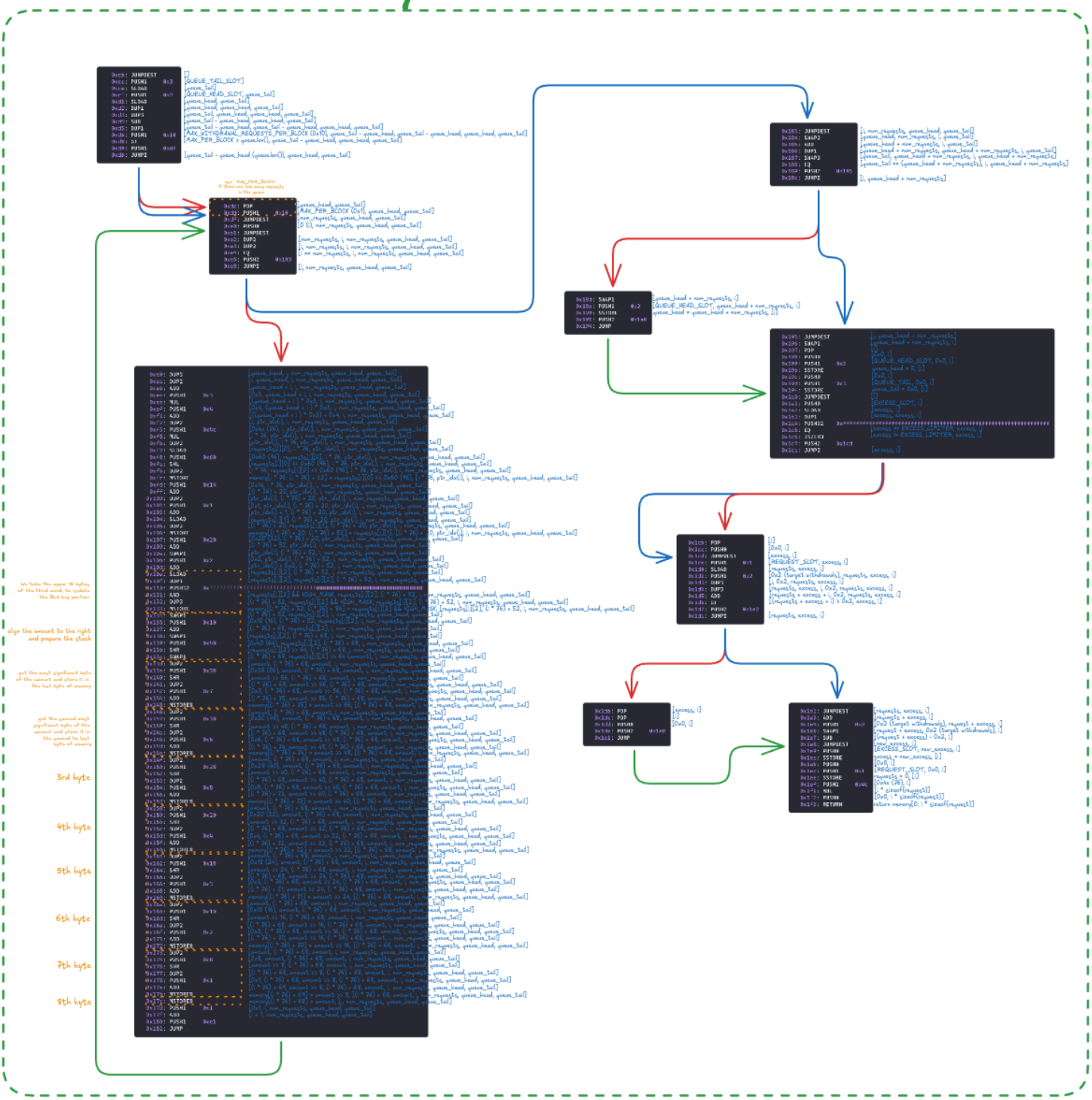


We have also produced a fully annotated control flow diagram for the contract, which can be seen below. A full interactive diagram can be found here: https://link.excalidraw.com/readonly/PctJXquwFa52eRPBez9g

User Path



System Path



USER LEVEL CONSIDERATIONS

The fee required to submit a new withdrawal request is dynamic and can potentially change between transaction submission and inclusion. As the overpaid fees are not returned by the contract, EIP authors suggest submitting requests via a wrapper contract that first fetches the required fee and then submits a request with the acquired fee. However, as the fee change can be drastic, we suggest contracts tasked with initiating withdrawal requests are called with an additional parameter containing the maximum fee the contract wants to pay to submit a request.

Therefore, we suggest editing the example provided in the EIP text to the following:

```
function addWithdrawal(bytes memory pubkey, uint64 amount, uint256
maxFee) private {
    assert(pubkey.length == 48);

    // Read current fee from the contract.
    (bool readOK, bytes memory feeData) =
WithdrawalsContract.staticcall('');
    if (!readOK) {
        revert('reading fee failed');
    }
    uint256 fee = uint256(bytes32(feeData));
    require(fee <= maxFee, "Maximum fee exceeded!")
    // Add the request.
    bytes memory callData = abi.encodePacked(pubkey, amount);
    (bool writeOK,) = WithdrawalsContract.call{value: fee}(callData);
    if (!writeOK) {
        revert('adding request failed');
    }
}
```

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Fee update logic can lead to DoS	**OPEN**

Given that the number of excess requests is updated once per block (at the end of its processing), it allows a user to submit multiple requests for the same fee without experiencing the rate limiting effects of the fee calculation function.

Taking this to the extreme an attacker could utilise an entire block to fill the contract with “bogus” requests for the sole purpose of driving up the fee for the following blocks. This would result in the contract being unusable until the excess came down to a reasonable level (a few hours “downtime”). This attack can easily be mitigated by updating the number of excess requests every time a new request is submitted, instead of once per block.

Although maintaining the attack will be costly and complex, we believe opportunities to DoS the contract for a few hours will be frequent and, given the simple mitigation, believe it should be fixed.

Attack specifics:

To optimize the gas consumption of the attack the 56 bytes of call data provided will all be zeros. This will mean that each new request will only write one storage slot, containing the caller's address. Additionally, if the current queue tail points to slots that have been written previously, the attack will end up freeing storage space, further increasing our attack's efficiency.

To demonstrate this we consider 2 scenarios:

- When writing on a part of the queue that has not been written it is possible to create 800 requests in a single transaction. This will mean that, without any additional requests, the excess will return to its previous position after 1.3 hours.
- If all written storage slots have been used before, the number of requests that can fit in a single transaction rises to 1900, introducing a 3.1 hour delay.

Looking at the earlier table containing the fee values for different numbers of excess requests one can see that, if the attack is performed given favorable conditions (e.g starting with around 500 excess requests), the system will be unusable due to extremely high fees for the majority of the added delay.

LOW SEVERITY:

[No low severity issues]

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Fee calculation can be optimized	**INFO**

Although our earlier suggestion for the fee calculation was to update the number of excess requests at each request submission, if that change is not made the contracts can be optimized by assigning the calculated fee value once per block, instead of calculating it twice per request submission. The rationale behind this is that any action performed on the system side is "free", and pre-calculating the fee acts like memoization for the users.

ID	Description	STATUS
A2	Constant operation can be optimized away	**INFO**

The multiplication at the beginning of the "fake exponentiation" function, is one between 2 constants and could be optimized away.
As seen in the following geas snippet the multiplication is between the `WITHDRAWAL_REQUEST_FEE_UPDATE_FRACTION` and `MIN_WITHDRAWAL_REQUEST_FEE` constant parameters and could be optimized away:

```
;; fake exponentiation
;; input stack = [factor, numerator, denominator]

dup3       ;; [denom, factor, numer, denom]
mul        ;; [accum, numer, denom]
...
```

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.