EIP 2935

Smart Contract Security Assessment

December 13, 2024

ethereum foundation

ABSTRACT

Dedaub was commissioned to perform a security audit of the system contracts for EIP2935. In total only one advisory issue was found, related to an inconsistency between the buffer size being used in the contract and the buffer size defined in the EIP.

BACKGROUND

Historically, Ethereum's execution model assumed that clients store and serve recent block hashes from local node history. However, as the ecosystem moves toward "stateless clients," which do not maintain full local state, this assumption has become problematic.

Stateless clients need a way to verify block hashes without depending on a fully synchronized node. One solution is to include the necessary historical block hashes directly in the state accessible by the EVM, allowing them to be easily retrieved and verified against cryptographic proofs like Merkle or Verkle trees.

EIP-2935 proposes storing and serving historical block hashes from a specially designated system contract. By writing the last 8192 block hashes into a ring buffer inside the state, the EVM itself can then query these hashes directly. This approach does not alter the existing `BLOCKHASH` opcode semantics, but rather provides an additional way to access older hashes through contract calls.

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 2 days (each) on the following contracts:

```
src/
└── execution_hash/
    ├── ctor.eas
    └── main.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

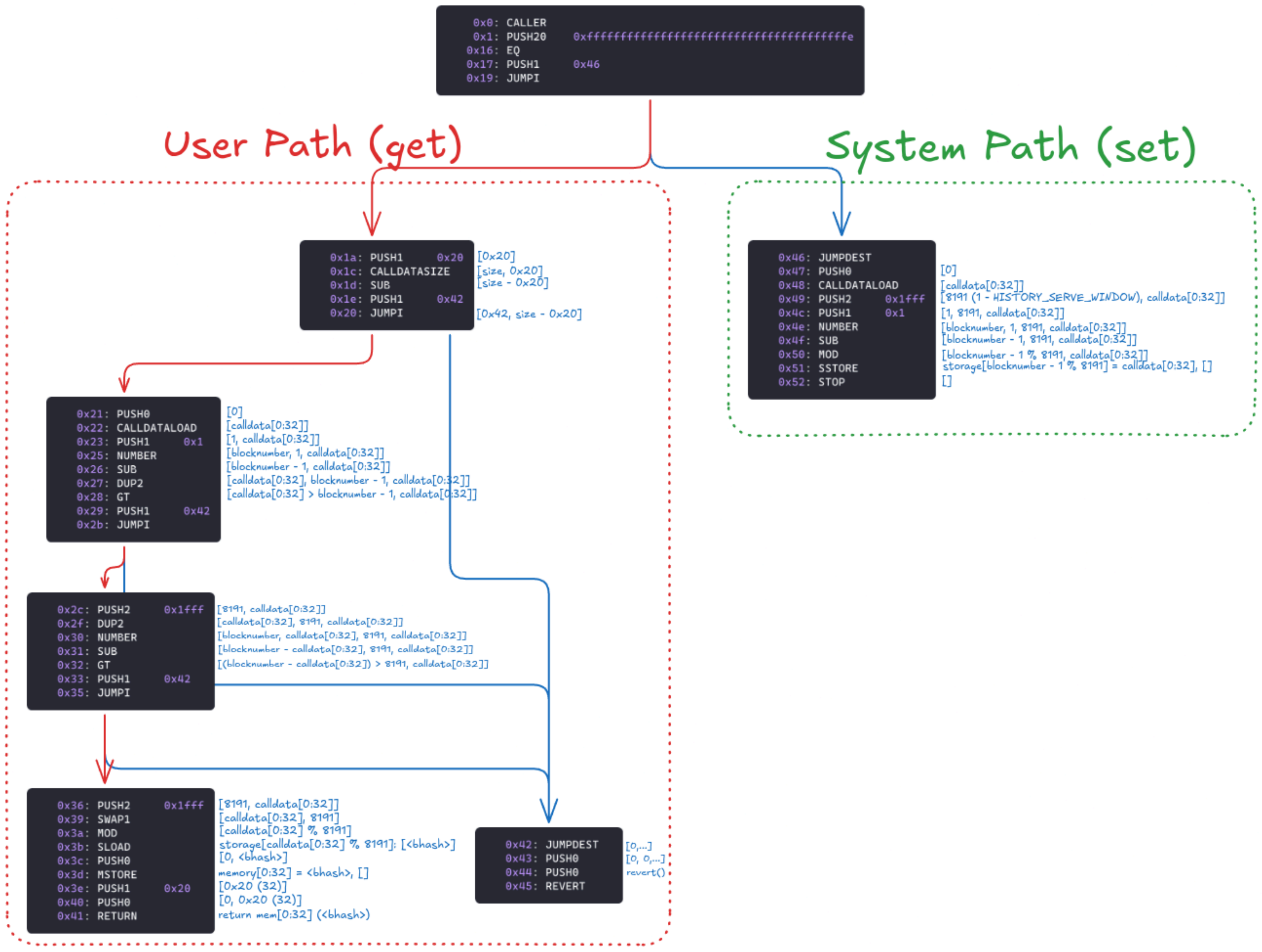
The audited smart contract includes 2 execution flows: **`set()`** which can only be called during the block's processing using the special system address (`0xfffffffffffffffffffffffffffffffffffffffe`) and stores the block hash of the previous block (`block.number - 1`), and **`get()`** which is callable by all other addresses and returns the block hash of a block, given its block `number` with `block.number - 8191 <= number <= block.number - 1`.
As the smart contract does not adhere to the ABI specification, the called method is chosen based on the contract's caller.
To bind the amount of storage used by the contract, block hashes are stored in a ring buffer of 8191 slots. Given a valid, recent block number, its block hash is stored in the storage slot at index `number % 8191`.

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

```
contract EIP2935{

  function __function_selector__(uint256 calldata_0_32) public
payable {
       if (0xfffffffffffffffffffffffffffffffffffffffe == msg.sender) {
       // set()
       STORAGE[(block.number - 1) % 8191] = bytes32(calldata_0_32);
       exit;
       } else {
       // get()
       require(!(msg.data.length - 32));
       require(calldata_0_32 <= block.number - 1);
       require(block.number - calldata_0_32 <= 8191);
       return STORAGE[calldata_0_32 % 8191];
       }
  }
}
```

The following diagram annotates the control flow of the contract, showing the stack values at each point.



```
 0x0: CALLER
 0x1: PUSH20    0xfffffffffffffffffffffffffffffffffffffffe
0x16: EQ
0x17: PUSH1     0x46
0x19: JUMPI
```

```
0x1a: PUSH1        0x20   [0x20]
0x1c: CALLDATASIZE        [size, 0x20]
0x1d: SUB                 [size - 0x20]
0x1e: PUSH1        0x42
0x20: JUMPI               [0x42, size - 0x20]
```

```
0x21: PUSH0               [0]
0x22: CALLDATALOAD        [calldata[0:32]]
0x23: PUSH1      0x1      [1, calldata[0:32]]
0x25: NUMBER              [blocknumber, 1, calldata[0:32]]
0x26: SUB                 [blocknumber - 1, calldata[0:32]]
0x27: DUP2                [calldata[0:32], blocknumber - 1, calldata[0:32]]
0x28: GT                  [calldata[0:32] > blocknumber - 1, calldata[0:32]]
0x29: PUSH1      0x42
0x2b: JUMPI
```

```
0x2c: PUSH2      0x1fff   [8191, calldata[0:32]]
0x2f: DUP2                [calldata[0:32], 8191, calldata[0:32]]
0x30: NUMBER              [blocknumber, calldata[0:32], 8191, calldata[0:32]]
0x31: SUB                 [blocknumber - calldata[0:32], 8191, calldata[0:32]]
0x32: GT                  [(blocknumber - calldata[0:32]) > 8191, calldata[0:32]]
0x33: PUSH1      0x42
0x35: JUMPI
```

```
0x36: PUSH2      0x1fff   [8191, calldata[0:32]]
0x39: SWAP1               [calldata[0:32], 8191]
0x3a: MOD                 [calldata[0:32] % 8191]
0x3b: SLOAD               storage[calldata[0:32] % 8191]: [<bhash>]
0x3c: PUSH0               [0, <bhash>]
0x3d: MSTORE              memory[0:32] = <bhash>, []
0x3e: PUSH1      0x20     [0x20 (32)]
0x40: PUSH0               [0, 0x20 (32)]
0x41: RETURN              return mem[0:32] (<bhash>)
```

```
0x42: JUMPDEST   [0,...]
0x43: PUSH0      [0, 0,...]
0x44: PUSH0      revert()
0x45: REVERT
```

```
0x46: JUMPDEST
0x47: PUSH0               [0]
0x48: CALLDATALOAD        [calldata[0:32]]
0x49: PUSH2      0x1fff   [8191 (1 - HISTORY_SERVE_WINDOW), calldata[0:32]]
0x4c: PUSH1      0x1      [1, 8191, calldata[0:32]]
0x4e: NUMBER              [blocknumber, 1, 8191, calldata[0:32]]
0x4f: SUB                 [blocknumber - 1, 8191, calldata[0:32]]
0x50: MOD                 [blocknumber - 1 % 8191, calldata[0:32]]
0x51: SSTORE              storage[blocknumber - 1 % 8191] = calldata[0:32], []
0x52: STOP                []
```

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:

[No medium severity issues]

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	Inconsistent History Buffer Size	**RESOLVED**

The contract `execution_hash/main.eas` defines the history window to be 8191, whilst the EIP defines `HISTORY_SERVE_WINDOW` to be 8192. Looking at another system contract EIP 4788, this uses a buffer size of 8191.

Comments:
The team communicated that this is in fact an issue with the EIP spec, and not the code

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.