0x-Settler

CrossChainReceiverFactory

Smart Contract Security Assessment

June 10, 2025

ABSTRACT

Dedaub was commissioned to perform a security audit of the cross-chain receiver functionality of **0x-Settler** protocol. The audit identified only one security issue with medium severity corresponding to a possible front-running attack when native tokens are used. The impact of such an attack is limited to disabling the intended functionality of the protocol (performing the swap of the tokens), but does not lead to any loss of funds. Moreover, the report raises a few protocol considerations and advisory issues. In addition, the developers requested some in-depth analysis of the protocol to answer a few technical questions to which we include answers in this report.

BACKGROUND

The 0x Settler protocol is designed to facilitate secure and gas-efficient asset swaps across a range of blockchain protocols. This is achieved by offering users the ability to execute meta-transactions, wherein users can specify a list of actions they want the protocol to execute on their behalf (generally different swaps). Dedaub was tasked to audit a PR which aimed to introduce cross-chain receiver functionality.

The functionality allows swaps of user funds received from bridges through wallets that store some merkle tree of possible swaps. This is encoded as an `EIP712` signing hash of an object with Permit2's domain separator and of type `PermitWitnessTransferFrom` or `PermitTransferFrom`. The aim is to achieve this gaslessly, without on-chain user interaction. The funds are assumed to have arrived at a counterfactual address. A cross-chain receiver factory contract is introduced to deploy `ERC7739` minimal proxies at deterministic addresses using the `CREATE2` instruction. The address where this proxy is deployed is the same address where these funds are received and it is derived from the merkle tree root and the address of the owner of these funds.

For the swap to be executed, a 0x-Relayer deploys the minimal proxy and calls a function `approvePermit2` to give allowance for Permit2 to transfer the funds. Then, it

executes one of the allowed actions encoded in the merkle tree, by passing in a signature that contains the merkle tree proof for the requested swap.

For Permit2 to perform the transfer, it verifies the signature using the `ERC7739` functionality of the minimal proxy. Therefore, `isValidSignature()` implements a verification of a merkle proof by computing the root, then deriving the address of the minimal proxy and checking its correctness. Additionally, an alternative execution is enabled where the users opt to execute a different swap than the one encoded in the merkle tree (for example, when the market moves away and none of the encoded swaps in the merkle tree are possible or satisfying). In such a case, the user provides (off-chain) an `ERC7739` signature of the desired swap object to the relayer. The relayer executes the requested swap by using the `ERC7739` signature instead of a merkle proof. For this to be possible (and also to have the minimal proxy complaint with `ERC7739`), `isValidSignature()` also implements the standard `ERC7739` signature verification. Signatures based on merkle tree proofs are distinguished from normal `ERC7739` signatures by the first 12 bytes being zero representing an address of the original owner of the proxy in the first case. A high-level description of the protocol is presented in the diagram below.

To optimize for gas, the deployment of the proxy does not store the owner of the proxy when merkle proof signatures are used but instead the owner is passed as part of the signature. However, when `ERC7739` signatures are used, the owner has to be stored for the verification.

The most critical part of this mechanism is the verification of the signature. In particular, the `isValidSignature()` function should implement correct verification of the hash in both cases. Therefore, a thorough analysis of these steps were conducted as part of the audit.

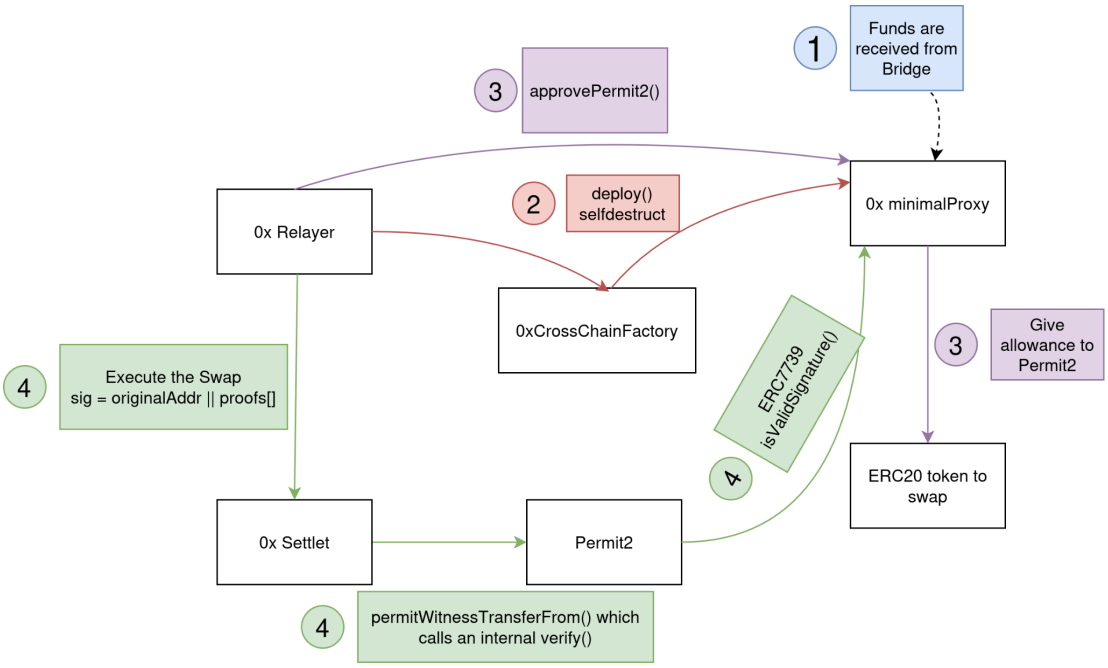

Fig 1: High-level description of the execution flow for merkle proof signatures

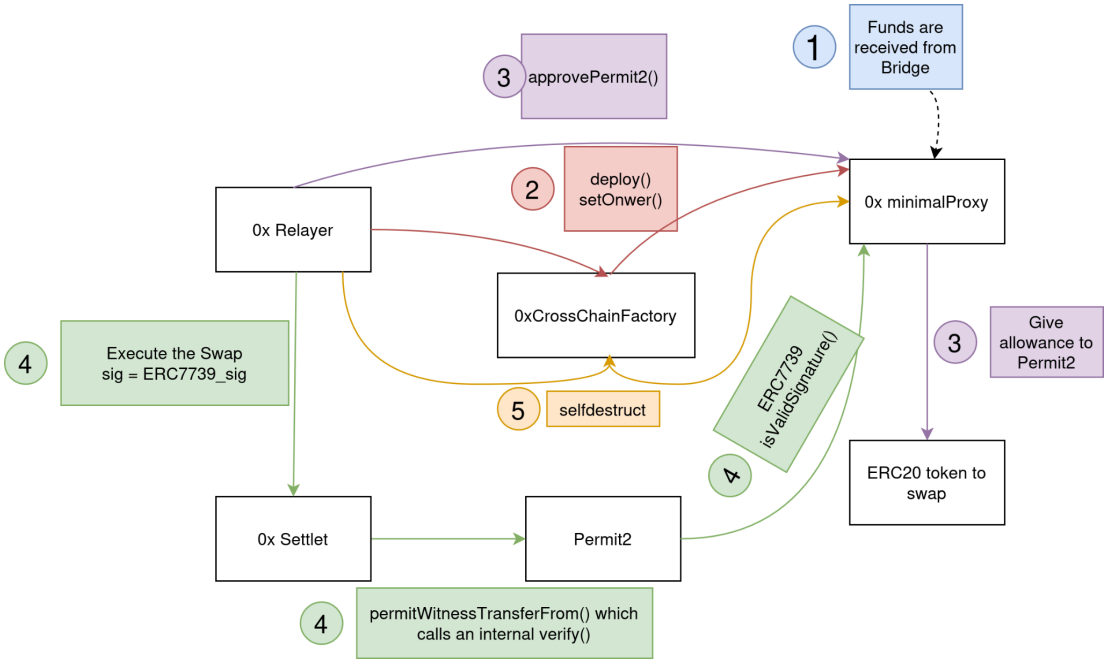

Fig 2: High-level description of the execution flow for ERC7739 signatures

SETTING & CAVEATS

This audit report mainly covers the changes at PR#338 of the **0x-Settler** repository.
Two auditors worked on the following contracts:

```
src
├── CrossChainReceiverFactory.sol
└── multicall
    └── MultiCallContext.sol
```

Audit Start Date:	**June 4, 2025**
Report Submission Date:	**June 11, 2025**

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

TECHNICAL ANALYSIS

We address a few questions posed by the developers.

ID	Description	STATUS
Q1	Is there a scheme for nested, rehashed (ERC7739) smart contract (ERC1271) signatures? In other words, how can we make the signature scheme work when owner() is itself a smart contract?	**INFO**

When the proxy owner is itself another contract, the `ERC7739` verification cannot be performed as the signer is another contract and not an EOA account. There is currently no ERC standard that can support such nested `ERC7739` signatures. However, such a standard is interesting so we describe at a high level how this can be standardized. Given the path of the n contracts that are going to verify the signature, a signature should be constructed in the following format `sig = originalSignature || APP_DOMAIN_SEPARATOR[1] || ... || APP_DOMAIN_SEPARATOR[n] || contents || contentsDescription || uint16(contentsDescription.length) || n || <special_magic_number_indicating_this_format>`.

Where `APP_DOMAIN_SEPARATOR[i]` is the domain separator of the caller's `i`'th verifying contract. The `originalSignature` would then be constructed as the signature of the `finalTypedDataSignHash` constructed as follows.

First the following type hashes are computed:

```
typedDataSignTypehash =
    keccak256(
          abi.encodePacked(
              "TypedDataSign(",
                  contentsName, " contents,",
                  "string name,",
                  "string version,",
                  "uint256 chainId,",
                  "address verifyingContract,",
                  "bytes32 salt"
```

```
                ")",
                contentsType
            )
        );

struct NestedTypedDataSignHash {
    bytes32 typedDataSignHash;
}


nestedTypedDataSignTypehash =
    keccak256(
          abi.encodePacked(
              "TypedDataSign(",
                  "NestedTypedDataSignHash contents,",
                  "string name,",
                  "string version,",
                  "uint256 chainId,",
                  "address verifyingContract,",
                  "bytes32 salt"
              ")",
              "NestedTypedDataSignHash(bytes32)"
          )
      );
```

Then the following typed data sign hash are computed

For `i = 1`,

```
TypedDataSignHash[1] =
    keccak256(
        abi.encodePacked(
            hex"1901",
            bytes32(APP_DOMAIN_SEPARATOR[1]),
            keccak256(
                abi.encode(
                    typedDataSignTypehash,
```

```
                    bytes32(hashStruct(originalStruct)),
                    // `eip712Domain()` of 1st verifying contract.
                    keccak256(bytes(eip712Domain().name)),
                    keccak256(bytes(eip712Domain().version)),
                    uint256(eip712Domain().chainId),
                    uint256(uint160(eip712Domain().verifyingContract)),
                    bytes32(eip712Domain().salt)
                )
            )
        )
    );
```

Then for `i = 2, ... n`,

```
nestedTypedDataSignTypehash[i] =
NestedTypedDataSignHash(TypedDataSignHash[i-1]);

TypedDataSignHash[i] =
    keccak256(
        abi.encodePacked(
            hex"1901",
            bytes32(APP_DOMAIN_SEPARATOR[i]),
            keccak256(
                abi.encode(
                    nestedTypedDataSignTypehash,
                    bytes32(hashStruct(nestedTypedDataSignTypehash[i])),
                    // `eip712Domain()` of the ith verifying contract.
                    keccak256(bytes(eip712Domain().name)),
                    keccak256(bytes(eip712Domain().version)),
                    uint256(eip712Domain().chainId),
                    uint256(uint160(eip712Domain().verifyingContract)),
                    bytes32(eip712Domain().salt)
                )
            )
        )
    );
```

Finally,

```
finalTypedDataSignHash = TypedDataSignHash[n]
```

For the verification contract, each contract from `i=1,..,n-1` will be using the `APP_DOMAIN_SEPARATOR[i]` domain separator to compute the hash `TypedDataSignHash[i]` similar to how it is done in `ERC7739`. Consequently, the verifier contract calls `ERC1271(owner).isValidSignature(hash , sig)` where `hash` and `sig` are constructed as follows:
`hash = APP_DOMAIN_SEPARATOR[i] || bytes32(hashStruct(NestedTypedDataSignHash(TypedDataSignHash[i])))`
and
`sig = originalSignature || APP_DOMAIN_SEPARATOR[i+1] || ... || APP_DOMAIN_SEPARATOR[n] || bytes32(hashStruct(NestedTypedDataSignHash(TypedDataSignHash[i]))) || "NestedTypedDataSignHash(bytes32)" || uint16(32) || n-i || <special_magic_number_indicating_this_format>`.
The final verifying contract (detected when the `n` field is equal 1) performs a normal `ERC7739` verification after stripping out the `n` field and the magic number.

Note: Our proposal is not intended as a concrete description of an ERC but rather just give a high level idea of how such a standard can be defined. The current suggestion is not tested and requires more in depth analysis and potential refinements.

Q2	Is this compatible with ERC6492?	**INFO**

The ERC6492 “proposes a standard way for any contract or off-chain actor to verify whether a signature on behalf of a given counterfactual contract (that is not deployed yet) is valid”. This is performed by wrapping the signature with some auxiliary information that allows the `ERC6492` verifier to deploy and initialise the `ERC1271` signing contract before passing the signature to its `isValidSignature()` function.

To differentiate wrapped ERC6492 signatures, a magic number "0x6492649264926492649264926492649264926492649264926492649264926492649264926492" is appended to the end of the signature. According to the ERC6492 standard, the ERC6492 verifying contract should take care of unwrapping the signature and stripping out the magic number from its tail. Hence, the ERC1271 contract (i.e., the minimal proxy in our case) need not be aware of such wrapping.

On the other hand, an ERC7739 signature has the format of `originalSignature || APP_DOMAIN_SEPARATOR || contents || contentsDescription || uint16(contentsDescription.length)`. So for such a signature to be compatible with ERC6492 standard, the tail byte32 of this signature should not be confused with a magic number specific for this standard. Fortunately, for this to happen, `contentsDescription.length` should match the value `0x6492 = 25746` which is a very large number unlikely to appear for typical `contents`. Moreover, the `contentsDescription` should also include a `0x92` char which is not an ASCII character thus it can never appear in valid ERC7739 signature. Therefore, valid ERC7739 signatures can correctly be passed to an ERC6492 contract and correctly forwarded to `isValidSignature()` of the minimal proxy concluding that the current implementation is compatible with ERC6492 signatures.

Q3	This contract has been written for chains supporting Shanghai or higher hardfork. Are there noteworthy chains that require London support, once Linea has upgraded to (roughly) Prague?	**INFO**

We are not aware of such chains that currently require London support apart from the one mentioned (i.e., Linea). However, it's worth noting that the Scroll chain does not support the `selfdestruct` instruction so the `deploy()` cannot be called with

`setOwnerNotCleanup = false` on that chain and the minimal proxy cannot be cleaned up after deployment.		
Q4	Can you think of a good reason why we ought to expose DOMAIN_SEPARATOR() in addition to the ERC5267-mandated fields?	**INFO**
We currently do not see any need for exposing the `DOMAIN_SEPARATOR()` for the current contract as it does not serve any use. However, with the suggested nested ERC7739 format in Q1, this value should be exposed to compute the nested signature.		

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	A front-runner can perform a DoS by removing the allowance to `Permit2` contract	**INFO**
The current implementation is expected to be used by deploying the minimal proxy contract, then calling the `approvePermit2()` function on the proxy, and finally executing the settler swap in a single bundled transaction using the `MultiCall` contract. However, if this pattern is not respected, for example, calling the `deploy()` (with `setOwnerNotCleanup = true`) and `approvePermit2()` in a one transactions, and the swap execution in another transaction, then the second transaction (i.e., the one that performs the swap) can be frontrun by a malicious transaction that removes the allowance to `Permit2` preventing the swap from happening. This is possible because `approvePermit2()` is not a permissioned function. Therefore, we point out this protocol-level consideration, to caution the developers that all the needed calls should always be bundled in a single transaction.		

P2	The current mechanism creates state bloat on the blockchain	**INFO**
In the case of proxy deployment with `setOwnerNotCleanup = true`, the `cleanup()` function is not called upon deployment and thus the minimal proxy will not get self destructed upon deployment. To destroy the proxy, only its owner is permitted to call `cleanup()` later which will succeed in deleting the proxy contract only for chains with pre-cancun hard fork. Even for such chains, the owner has no obligation to perform the clean up as he has to deliberately call the `cleanup()` function after the relayer executes his swap. Therefore it's worth noting that the current mechanism can leave bloat in the blockchain state.		

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	A front-runner can perform a DoS by deploying the minimal proxy	**RESOLVED**
Resolution:		

The developers modified the `cleanup()` function to perform the wrapping of the native token before calling `selfdestruct`. This way, a frontrunner who deploys the proxy can now only perform the wrapping of the native tokens and destroy it. The relayer can simply redeploy the proxy. So this neutralized the impact of front running. The changes are provided in a set of commits visible through a `git diff origin/pr/338 b52801adf56efed0df877fc12cc4882f61c949cf`.

When the funds received from the bridge are native tokens, an attacker can front-run the deployment transaction by deploying the minimal proxy themselves with a call to `deploy(root, initialOwner, false)`. This call will deploy the proxy, `selfdestruct` it, and thus send the funds to the owner, preventing him from performing the swap thus disabling the functionality of this contract.

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	Unnecessary `mstore` operation in `deploy()` function	**INFO**
The `mstore(0x40, ptr)` instruction (line 276) in `deploy()` function is not needed as the memory address `0x40` is not touched by this function.		
A2	Confusion of different types of signatures in `isValidSignature()` function	**INFO**

The function `isValidSignature()` accepts two different formats of signatures. One format is an `ERC7739` signature given in the form of `ECDSA_COMPACT_SIG || APP_DOMAIN_SEPARATOR || contents || contentsDescription || uint16(contentsDescription.length)` and a second format that represents a merkle proof in the format of `originalOwner || proofs[]`.

Confusion between these two is a possible issue, but requires an `r` value starting with `12` bytes of zeros. In reality, an honest signer (i.e., choosing the scalar `k` randomly) will never get such an `r` value, except with negligible probability. So, the confusion of the `r` value with an `address` leading to parsing the `ERC7739` signature as a merkle proof signature format can never happen in an honest interaction except with negligible probability.

For dishonest interactions, this implementation does not pose any evident security risks. A malicious user can pass an invalid signature that gets parsed as a merkle proof signature, but they can only make the signature verification pass if they pass the correct `hash` value for the merkle proof. The possibility of inputting two different types of signatures then does not pose any new security risks.

However, it is worth mentioning the following consequences of this implementation. If one can possibly find a valid signature with an `r` value that starts with `12` bytes of zeros, then `isValidSignature()` may return `0xffffffff` for that valid signature, contradicting what the standard mandates. We did an analysis to understand if this is computationally possible. Our analysis shows that this problem is not solvable in polynomial time. A simple brute force technique has the complexity of `O(2^96)`. On the other hand, using Pollard's Rho method with multiple targets is also as hard as using it for a single target when the number of the targets is relatively small to the number of security bits as shown in this article.

Alternatively, another possibility for finding an `r` with small value is exploiting some hidden structures in the secp256k1 curve. One such known structure is that the scalar `k=2^-1` generates a point with x-coordinate with 11 bytes of zeros. While this is not sufficient to break the implementation of `isValidSignature()`, one may fear other unknown secp256k1 structures that render the assumption on `r` having always less than 12 bytes of leading zeros false.

As we are not aware currently of any feasible way to generate a signature with small `r` value, we cannot argue that this implementation does not follow the `ERC7739` standard. However, our general recommendation is to avoid such highly optimized approaches that are based on battle tested assumptions (i.e., there is no known DL of a secp256k1 point with more than 11 bytes of zeros).

A2	Compiler bugs	**INFO**

The code is compiled with Solidity `0.8.28`. Version `0.8.28`, in particular, has no known bugs.

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.