Splits Smart Vaults Audit Report

Prepared by Riley Holterhus
September 3, 2024

Contents

1 Introduction **3**

1.1 About Splits Smart Vaults . . . 3

1.2 About the Auditor . . . 3

1.3 Disclaimer . . . 3

2 Audit Overview **4**

2.1 Scope of Work . . . 4

2.2 Summary of Findings . . . 4

3 Findings **5**

3.1 Low Severity Findings . . . 5

3.1.1 Trust assumptions for last signer . . . 5

3.1.2 Signature padding considerations . . . 6

3.1.3 Signature verification can reach `bytes32(0)` as merkle root . . . 6

3.2 Informational Findings . . . 8

3.2.1 Empty signers can be explicitly prevented . . . 8

3.2.2 `validateUserOp()` should revert in additional scenarios . . . 9

3.2.3 Owner recovering address considerations . . . 10

3.2.4 `WebAuthn` deferred return value . . . 11

1 Introduction

1.1 About Splits Smart Vaults

Splits Smart Vaults are ERC-4337 compliant smart accounts. In addition to traditional signature verification, Smart Vaults allow signatures to verify multiple user operations across multiple chains using a merkle tree approach. Smart Vaults also use a multi-sig threshold, where one signer approves the entire user operation while the others sign off on the core aspects through a "light" user operation. For more information about Splits, visit their website: splits.org.

1.2 About the Auditor

Riley Holterhus is an independent security researcher that focuses on Solidity smart contracts. Other than conducting independent security reviews, he works as a Lead Security Researcher at Spearbit, and also searches for vulnerabilities in live codebases. Riley can be reached by email at rileyholterhus@gmail.com, by Telegram at @holterhus and on Twitter/X at @rileyholterhus.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the authors have made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an "as-is" basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

From August 19th, 2024 through August 23rd, 2024, Riley Holterhus conducted an audit of Splits' smart-vaults smart contracts. During this period, a manual analysis was undertaken to identify various security issues and logic flaws.

This audit was conducted on the codebase found in the `0xSplits/splits-contracts-monorepo` GitHub repository, starting on commit 1db8acb. The following files were in scope for the audit:

- packages/smart-vaults/src/library/UserOperationLib.sol
- packages/smart-vaults/src/signers/AccountSigner.sol
- packages/smart-vaults/src/signers/MultiSigner.sol
- packages/smart-vaults/src/signers/Signer.sol
- packages/smart-vaults/src/signers/PasskeySigner.sol
- packages/smart-vaults/src/utils/FallbackManager.sol
- packages/smart-vaults/src/utils/Caller.sol
- packages/smart-vaults/src/utils/MultiSignerAuth.sol
- packages/smart-vaults/src/utils/ModuleManager.sol
- packages/smart-vaults/src/utils/ERC1271.sol
- packages/smart-vaults/src/vault/SmartVault.sol
- packages/smart-vaults/src/vault/SmartVaultFactory.sol

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities are subjective, but aim to capture the impact and feasibility of each potential issue. In total, 3 low-severity findings and 4 informational findings were identified.

All issues identified in the code have been either addressed or acknowledged. The resulting changes were reviewed, and all mitigations have been documented in this report.

3 Findings

3.1 Low Severity Findings

3.1.1 Trust assumptions for last signer

Description: When a userOp is validated in a Splits Smart Vault, one signer signs the full userOp hash, while the others sign a "light" version that includes only the sender, nonce, and calldata of the userOp. During the audit kickoff discussion, the Splits team mentioned a concern and a planned modification to this behavior.

Based on the current implementation, one signer has full control over the `maxPriorityFeePerGas` value (packed within the `gasFees` field). Since they can set any value they choose, the account could have its ETH drained by the one signer using an excessively high priority fee. To address this, the plan is to allow the other "light" userOp signers to set an upper bound on the allowable `maxPriorityFeePerGas` value.

It's worth noting that additional fields could also benefit from upper bounds set by the "light" userOp signers. Specifically:

1. The `preVerificationGas` value. This is the gas amount taken by the bundler for overhead transaction costs. Since none of this gas is refunded, setting an excessively high value could drain an account's ETH.
2. The `maxFeePerGas` value. This is another value packed within the `gasFees` field. While the `maxPriorityFeePerGas` accounts for the gas price above the base fee, the lack of an upper bound on the `maxFeePerGas` still allows the final signer to spend ETH during a period of high base fees that the other signers might not have approved. However, this scenario is unlikely, as it would require the "light" signers to approve a transaction despite disagreeing with the current base fee, or for their signatures to remain unused long enough for significant changes in the base fee.
3. The `accountGasLimits` field. This field contains two values - the gas allocated for the `validateUserOp()` call, and the gas forwarded to the account during the execution step. Although any unused gas is refunded at the end of the userOp, there is a 10% penalty on this refund, meaning excessively high `accountGasLimits` values could result in wasted ETH.
4. The `paymasterAndData` field. This field includes the gas limits for the `validatePaymasterUserOp()` call and the two potential `postOp()` calls. Although the paymaster covers these gas fees in ETH, it's still in the account's best interest to avoid excessive values being set. This is because the account might approve a paymaster to recoup its costs in another form, for example by taking an equivalent amount of an ERC20 token. In this case, an unexpectedly high gas limit or gas usage could still end up costing the account.

Recommendation: For each of the four points mentioned above, consider adding additional upper bounds that the "light" userOp signers can approve.

Splits: Addressed in PR 43.

Auditor: Verified. Upper bounds have been implemented for all the fields mentioned above, except for `maxFeePerGas`. After a discussion with the team, it was agreed that the `maxFeePerGas` concern is minimal and an upper bound on the `maxPriorityFeePerGas` is sufficient.

3.1.2 Signature padding considerations

Description: In some smart wallet implementations, "signature padding" can be an issue. This occurs when an attacker intercepts a valid signature before it is executed on-chain, and then pads it with extra data to increase the gas cost paid by the account. In the Splits Smart Vaults code, this is a potential problem in the following locations:

1. In the `validateUserOp()` function, the `signature` bytes are decoded into two different possible structs:

```
function validateUserOp(/* ... */) /* ... */ {
    // ...
        SingleUserOpSignature memory signature = abi.decode(userOp_.signature[1:], (
            SingleUserOpSignature));
        // ...
        MerkelizedUserOpSignature memory signature = abi.decode(userOp_.signature[1:], (
            MerkelizedUserOpSignature));
    // ...
}
```

   Note that `abi.decode()` doesn't guarantee the absence of unnecessary extra data, and there's also no guarantee that the struct hasn't been encoded suboptimally with extra padding bytes. As a result, extra bytes can be added that may go undetected during this decoding step.

2. In the `isValidSignature()` function within the `MultiSignerLib`, there is no validation on the length of the `signatures_` array. Only the signatures required by the account's `threshold` are used, allowing extra elements in the array to go undetected.

3. In the `PasskeySignerLib`, the `isValidSignature()` function decodes the signature into the `WebAuthnAuth` struct, which has the same potential for padding as mentioned in point (1) above.

4. In the `AccountSignerLib`, there are situations where the account invokes the ERC-1271 `isValidSignature()` function on the signer, forwarding the entire signature in the process. If the signer is vulnerable to signature padding, the Smart Vault account would be susceptible as well.

Recommendation: Most of the potential issues mentioned above would be difficult to fully address, since detecting all forms of unnecessary padding would require non-trivial code changes. Instead of attempting to mitigate this concern in all areas of the smart contract code, consider setting a reasonable verification gas limit when creating each userOp off-chain. This would limit the additional expense that padding could cause.

Splits: In addition to setting the verification gas limit appropriately, this issue is partially mitigated by PR 43.

Auditor: Verified. The changes in PR 43 allow the light signers to set an upper bound on the verification gas limit, which gives less room for signature padding to increase gas costs.

3.1.3 Signature verification can reach `bytes32(0)` as merkle root

Description: In the `SmartVault` code, the first `n-1` signers authorize a "light" userOp hash, while the final signer confirms the full userOp hash, with `n` being the account's required threshold. The code also allows signers to sign a

merkle root of a tree of hashes, enabling multiple userOps to be authorized with a single signature. Part of this system is implemented as follows:

```
function _validateMerkelizedUserOp(
    bytes32 lightHash_,
    bytes32 userOpHash_,
    MerkelizedUserOpSignature memory signature
)
    internal
    view
    returns (uint256 validationData)
{
    if (!MerkleProof.verify(signature.merkleProof, signature.merkleTreeRoot, userOpHash_)) {
        revert InvalidMerkleProof();
    }

    if (signature.lightMerkleTreeRoot != bytes32(0)) {
        if (!MerkleProof.verify(signature.lightMerkleProof, signature.lightMerkleTreeRoot,
            lightHash_)) {
            revert InvalidMerkleProof();
        }
    }

    return _isValidSignature(signature.lightMerkleTreeRoot, signature.merkleTreeRoot, signature.
        signatures);
}
```

Notice that when `signature.lightMerkleTreeRoot` is equal to `bytes32(0)`, the merkle verification of the "light" userOp tree is skipped, and the empty value is passed to the `_isValidSignature()` function. Since the `signature` input can be any value, it's possible for someone to intentionally do this, regardless of the account's threshold. This would cause the first `n-1` signatures to be verified against `bytes32(0)` instead of the root of a merkle tree containing the "light" userOp hash.

Fortunately, this seems very unlikely to be exploitable, as there's no reason for an attacker to have access to the signers' signatures over `bytes32(0)`, and the final signer's signature would still need to be verified as normal. However, removing this behavior entirely would be safer.

Recommendation: Consider modifying the logic of when the "light" userOp tree should be verified. This can be achieved with the following changes:

```
function validateUserOp(
    PackedUserOperation calldata userOp_,
    bytes32 userOpHash_,
    uint256 missingAccountFunds_
)
    external
    onlyEntryPoint
    payPrefund(missingAccountFunds_)
    returns (uint256 validationData)
{
        // ...
```

```
        // if threshold is greater than 1, `threshold - 1` signers will sign over the merkle tree
    root of light user
        // op hash(s). We lazily calculate light userOp hash based on value of light merkle tree
    root. If threshold
        // is 1 then light userOp hash won't be needed.
-       if (signature.lightMerkleTreeRoot != bytes32(0)) lightHash = _getLightUserOpHash(userOp_);
+       if (signature.signatures.length > 1) lightHash = _getLightUserOpHash(userOp_);

        return _validateMerkelizedUserOp(lightHash, userOpHash_, signature);
        // ...
}

function _validateMerkelizedUserOp(
    bytes32 lightHash_,
    bytes32 userOpHash_,
    MerkelizedUserOpSignature memory signature
)
    internal
    view
    returns (uint256 validationData)
{
    if (!MerkleProof.verify(signature.merkleProof, signature.merkleTreeRoot, userOpHash_)) {
        revert InvalidMerkleProof();
    }

-   if (signature.lightMerkleTreeRoot != bytes32(0)) {
+   if (signature.signatures.length > 1) {
        if (!MerkleProof.verify(signature.lightMerkleProof, signature.lightMerkleTreeRoot,
    lightHash_)) {
            revert InvalidMerkleProof();
        }
    }

    return _isValidSignature(signature.lightMerkleTreeRoot, signature.merkleTreeRoot,
    signature.signatures);
}
```

With this logic, since a threshold greater than one implies that `signature.signatures.length` must be greater than one, the "light" merkle tree root will always be used when necessary.

Splits: Addressed in PR 47.

Auditor: Verified.

3.2 Informational Findings

3.2.1 Empty signers can be explicitly prevented

Description: In the Splits Smart Vaults codebase, signer information is stored in a storage array, and indices to this array are provided during signature verification. Currently, the code doesn't explicitly check whether a provided index actually contains signer information, or if it's an unset/empty index. If an empty index is used, the code would reach the following `else` case:

```
function isValidSignature(
    Signer memory signer_,
    bytes32 hash_,
    bytes memory signature_
)
    internal
    view
    returns (bool)
{
    if (isPasskeyMem(signer_)) {
        return decodePasskeySigner(signer_).isValidSignature(hash_, signature_);
    } else {
        return decodeAccountSigner(signer_).isValidSignature(hash_, signature_);
    }
}
```

Since an empty account signer would default to `address(0)`, the subsequent `isValidSignature()` call would eventually fail, meaning there is no way to abuse the lack of explicit validation for empty indices. However, it would be safer and more explicit to prevent this behavior altogether.

Recommendation: Consider adding a check to prevent an empty index from being used in signature verification. One way to accomplish this would be by modifying the `else` case mentioned above as follows:

```
if (isPasskeyMem(signer_)) {
    return decodePasskeySigner(signer_).isValidSignature(hash_, signature_);
} else if (isEOAMem(signer_) {
    return decodeAccountSigner(signer_).isValidSignature(hash_, signature_);
} else {
    revert();
}
```

Note that `isEOAMem()` would be a new function, similar to the existing `isEOA()` but with a `memory` input instead of `calldata`.

Splits: Fixed in PR 45.

Auditor: Verified.

3.2.2 `validateUserOp()` should revert in additional scenarios

Description: In the EIP-4337 spec, the following snippet outlines when `validateUserOp()` should either revert or gracefully return `SIG_VALIDATION_FAILED`:

> If the account does not support signature aggregation, it MUST validate that the signature is a valid signature of the `userOpHash`, and SHOULD return SIG_VALIDATION_FAILED (and not revert) on signature mismatch. Any other error MUST revert.

This can be interpreted to mean that `SIG_VALIDATION_FAILED` should only be returned when the signature is valid but resolves to the wrong signer, whereas all other errors should trigger a revert.

There is one location in the Smart Vaults code where this logic is not followed. In the `MultiSignerLib`, the `isValidSignature()` function gracefully returns `false` (which eventually leads to `SIG_VALIDATION_FAILED`) when there are duplicate indices within the `signatures_` array:

```
function isValidSignature(/* ... */) internal view returns (bool isValid) {
    // ...
    for (; i < threshold - 1; i++) {
        signerIndex = signatures_[i].signerIndex;
        mask = (1 << signerIndex);

        if (/* ... */ && alreadySigned & mask == 0) {
            alreadySigned |= mask;
        } else {
            isValid = false;
        }
    }
    // ...
}
```

Since this error goes beyond simply resolving to the wrong signer, it would better align with the 4337 spec if this scenario resulted in a revert instead.

Recommendation: Consider modifying the `isValidSignature()` function above so that it reverts if an index is used more than once.

Splits: Addressed in PR 46.

Auditor: Verified.

3.2.3 Owner recovering address considerations

Description: The Splits Smart Vaults implementation is designed so that the `owner` can always recover their account if something goes wrong, such as with a malicious module, fallback, or signer. During the audit, this behavior was confirmed based on the following observations:

- The `upgradeToAndCall()` function, only callable by the `owner`, guarantees that the account can always be reinitialized with a new implementation. There doesn't appear to be any way to block the `owner` from calling this function.
- There is no way for a `delegatecall` to be reached in the code, except with the `upgradeToAndCall()` function mentioned above. This implies that the account's storage is modified only by its own code, not by any external/unknown source.
- The only intended way the `owner` storage slot can be modified is through functions in the `Ownable` mixin, all of which are protected by the `onlyOwner` modifier. There doesn't seem to be any alternative or hacky way to change the `owner` storage slot, as no arbitrary storage writes exist in the Smart Vaults code.

So, even if all modules, fallbacks, and signers on an account become malicious, it seems that the owner can still recover their account. This matches the system's intended design.

Recommendation: No recommended change - this finding is provided for informational purposes only. For future versions of the code, consider maintaining the key points mentioned above to preserve the invariant.

3.2.4 `WebAuthn` deferred return value

Description: This issue was raised by the Splits team and is added here for tracking purposes.

The `PasskeySignerLib` contract utilizes the `WebAuthn` library imported from `@web-authn/WebAuthn.sol` to validate passkey signatures. This library was introducing specific gas estimation issues, and to address this, a forked version of the library has been implemented. The primary change in this forked version is the deferral of the `bool` return value until later in the function call.

Splits: As described above, addressed in PR 50.

Auditor: Verified. The new version of the `WebAuthn` library behaves equivalently to the original (more specifically, it returns `true` if and only if the original library would have returned `true`). It's worth noting that deferring the return value can cause some inputs that would have previously resulted in an early return to now lead to a revert. Since the new code only defers cases that would have returned `false` early, and since a revert is essentially equivalent to returning `false` during validation, there is no problem with this.