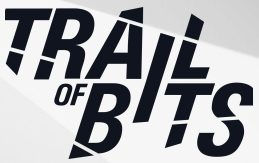

Plume

Security Assessment (Summary Report)

October 3, 2024

Prepared for:
Eugene Y. Q. Shen
Plume Network

Prepared by: **Simone Monica**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mike Shelton, Project Manager
mike.shelton@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Simone Monica, Consultant
simone.monica@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 12, 2024	Pre-project kickoff call
September 25, 2024	Delivery of report draft
September 25, 2024	Report readout meeting
October 3, 2024	Delivery of comprehensive report

Project Targets

The engagement involved a review and testing of the targets listed below.

go-ethereum

Repository	https://github.com/plumenetwork/go-ethereum
Version	93c572d3fb3b371fcaecd18f16079a8214eb4aaa
Type	Go
Platform	Ethereum

Contracts

Repository	https://github.com/plumenetwork/contracts
Version	from `cffad6f` to `10f5752`
Type	Solidity
Platform	Plume

Executive Summary

Engagement Overview

Plume engaged Trail of Bits to review the security of a change to go-ethereum and their smart contracts. Plume is a Arbitrum Nitro fork, and this change supports smart contract wallets at the protocol level; the smart contracts implement the smart wallet and a yield token.

A team of one consultant conducted the review from September 16 to September 20, 2024, for a total of one engineer-week of effort. With full access to source code and documentation, we performed static testing of the codebase, using automated and manual processes.

Observations and Impact

We reviewed the change made to the go-ethereum `Call` and `StaticCall` behavior, and we found that it does not alter the execution in unexpected ways that would allow arbitrary execution of an address or lead to possible incompatibilities with environment opcodes such as `extcodesize`.

The smart wallets contracts implement a double proxy pattern in which the first proxy is the code loaded when an externally owned account (EOA) address is called. We checked that powerful functions, like upgrading the implementation, have appropriate access controls in place, and that the chain of proxies is working as expected. Current smart wallet extensions include an `AssetVault` contract and a `SignedOperations` contract:

- `AssetVault` allows users to lock yield-bearing tokens and redistribute that yield to beneficiaries; we reviewed that the tokens are actually locked and present in the wallet and looked for ways that beneficiaries could either cause a denial of service (DoS) to the giver or to obtain more yields than they are due.
- `SignedOperations` allows the EOA owner to sign multiple operations that can be executed by anyone in a single transaction. We reviewed if it is possible to bypass the signature validation and whether signature replays attacks are possible.

The `AssetToken` contract inherits from the `YieldDistributionToken` contract; the latter is an ERC20 token that receives yield deposits and distributes that yield to token holders. We reviewed the `YieldDistributionToken` contract for the correct calculation of the yield, and that state variables are updated accordingly when transferring the tokens or depositing the yield. The accrue yield function could benefit from additional coverage after testing has been added.

The $P contract, which represents the governance token, is an ERC20 token that inherits most of the implementation from OpenZeppelin contracts; it can be paused, updated, and

burned by authorized addresses. We checked that the contract implementation inherits from the upgradeable contract, that an upgrade does not cause issues, and that the added functions have the correct access controls.

We found two high-severity issues, including one that would give an attacker full control of every smart wallet (TOB-PLUME-3) and another that makes it impossible to transfer the asset token (TOB-PLUME-6). We also found one medium-severity issue where a beneficiary could temporarily DoS its giver (TOB-PLUME-7); three low-severity issues; and three informational issues.

The codebase is still in development; it contains important TODO comments and completely lacks testing. This small-but-critical change to go-ethereum does not appear to follow an accurate design and a clear plan for implementation (TOB-PLUME-1, TOB-PLUME-2).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Plume take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Create more documentation.** Create high-level documentation that explains the project's goals and enumerates users' possible actions (described through diagrams). Given the system's high level of centralization, the documentation should also clarify who is the owner and what their privileges are. Additionally, there should be more technical documentation for developers who want to integrate their protocol with the smart wallet system; it should clearly describe every publicly accessible function with pre- and post conditions.
- **Implement tests.** We absolutely recommend having tests for the positive and negative flows that every function in the smart contracts can take. Note that this change to go-ethereum complicates testing with common tools.
- **Stay updated with Arbitrum Nitro updates**. Since Plume is a fork of Arbitrum Nitro, we advise constantly looking at the latest updates in case security issues are fixed; if they are, analyze them to determine if they also affect this project and fix them accordingly.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The arithmetic used is mostly simple, and no overflows or underflows are possible. The most complex calculation accruing yield, which is documented with inline comments; however, this calculation would benefit from higher-level documentation with different scenarios explained through examples. Additionally, this calculation is completely untested.	**Moderate**
Auditing	Most of the state-changing functions emit events. However, one event is missing from a critical function, and another function uses incorrect arguments. It is unclear if an off-chain monitoring system and an incident response plan are present.	**Moderate**
Authentication / Access Controls	The access controls are applied correctly where needed for privileged functions. The only roles are Owner and Wallet. A two-step process to transfer ownership is not currently implemented, and it is unclear if the Owner is an EOA or a multi signature wallet.	**Moderate**
Complexity Management	In general, the functions are easy to follow, do not have code duplication, and follow a clear naming scheme. Some functions fail to validate arguments, which led to some of the issues we identified. The design of EOA code overwrites in go-ethereum and the double proxy chain make it more difficult to follow the transaction execution flow.	**Moderate**
Decentralization	The owner has full control of the smart wallet implementation in the WalletFactory contract used by every EOA, it would benefit from having the upgrade	**Weak**

	functionality behind a timelock. The `AssetToken`'s owner can update its value, mint the token, and whitelist which addresses that can interact with it.	
Documentation	The only documentation available is NatSpec comments throughout the code; while these are generally good and helpful, they are not enough. A high-level documentation of the system and why this design has been chosen is missing, including possible user flows explained with diagrams. Additionally, the go-ethereum change, even if small, has important consequences throughout the system, and the intended behavior is not specified.	**Weak**
Low-Level Manipulation	Inline assembly and low-level calls are not used other than to support the proxy pattern.	**Satisfactory**
Testing and Verification	Smart contracts are untested. We highly recommend developing tests for both positive and negative behaviors of a function, and running a testnet where users can interact with the same exact system that will be deployed on the mainnet.	**Missing**
Transaction Ordering	We did not find a way to perform transaction-ordering attacks against the system.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	CodeSize and CodeCopy does not behave as if no code is present for EOA	Undefined Behavior	Low
2	EOA overwritten contract is set with a wrong code hash	Undefined Behavior	Informational
3	Anyone can execute arbitrary calls from a smart contract wallet	Data Validation	High
4	Custom error emitted with incorrect arguments	Data Validation	Informational
5	Missing event in WalletFactory's upgrade function	Data Validation	Informational
6	AssetToken's transfer always reverts	Data Validation	High
7	Yield beneficiary can temporarily DoS the giver for a specific token through acceptYieldAllowance function	Denial of Service	Medium
8	acceptYieldAllowance does not check that the amount is actually available	Data Validation	Low
9	ClaimedYield considers only currently whitelisted users	Undefined Behavior	Low

Detailed Findings

1. CodeSize and CodeCopy does not behave as if no code is present for EOA	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PLUME-1
Target: `go-ethereum/core/vm/instructions.go`	

Description

Plume supports smart contract wallets at the protocol level by overwriting the code executed for `Call` and `StaticCall` when an account exists in storage but does not have code. However, for an externally owned account (EOA), the execution of `CodeSize` and CodeCopy (figure 1.1) return the overwritten code, even though the intended behavior is for those addresses to still behave as EOA. This is because the execution uses `scope.Contract.Code`, which is overwritten with actual code and will not be empty for EOA addresses.

```
func opCodeSize(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext)
([]byte, error) {
	scope.Stack.push(new(uint256.Int).SetUint64(uint64(len(scope.Contract.Code))))
	return nil, nil
}

func opCodeCopy(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext)
([]byte, error) {
	var (
		memOffset  = scope.Stack.pop()
		codeOffset = scope.Stack.pop()
		length     = scope.Stack.pop()
	)
	uint64CodeOffset, overflow := codeOffset.Uint64WithOverflow()
	if overflow {
		uint64CodeOffset = math.MaxUint64
	}
	codeCopy := getData(scope.Contract.Code, uint64CodeOffset, length.Uint64())
	scope.Memory.Set(memOffset.Uint64(), length.Uint64(), codeCopy)

	return nil, nil
}
```

Figure 1.1: The `opCodeSize` and `opCodeCopy` functions (`go-ethereum/core/vm/instructions.go#L353-L372`)

Exploit Scenario

Alice uses her custom smart wallet implementation, which uses `codeSize` opcode and unexpectedly returns a non-zero result.

Recommendations

Short term, add an `eoa` flag to the Contract structure that is set to `true` if the current address is an EOA in `Call` and `StaticCall`. When executing the opcodes, check this flag with `scope.Contract.eoa` and return `0` if it is `true`.

Long term, when adding custom modifications to an EVM implementation, consider and document their deep consequences.

2. EOA overwritten contract is set with a wrong code hash

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PLUME-2
Target: `go-ethereum/core/vm/evm.go`	

Description

In the `Call` and `StaticCall` functions, the code of an EOA is overwritten with the code of an already-deployed contract. However, the code hash is set to the code hash of the EOA address, which is the empty code hash because the EOA does not have code stored in the state.

```
func (evm *EVM) Call(...) (...) {
   ...
   // If it's an EOA, we load the code from the Plume Gateway
   if len(code) == 0 && evm.StateDB.Exist(addr) {
       code =
evm.StateDB.GetCode(common.HexToAddress("0x38F983FcC64217715e00BeA511ddf2525b8DC692"
))
   }
   addrCopy := addr
   // If the account has no code, we can abort here
   // The depth-check is already done, and precompiles handled above
   contract := NewContract(caller, AccountRef(addrCopy), value, gas)
   contract.SetCallCode(&addrCopy, evm.StateDB.GetCodeHash(addrCopy), code)
   ret, err = evm.interpreter.Run(contract, input, false)
   ...
```

Figure 2.1: Snippet of the `Call` function (go-ethereum/core/vm/evm.go#L256-L268)

The only place that uses the code hash set in the contract instance is the jump dest analysis, which is used as the key to save the analysis. The analysis is saved for the empty code hash instead of the code hash for the corresponding code.

Recommendations

Short term, when the code is overwritten for an EOA in the `Call` and `StaticCall` functions, use the code hash of the address where the code resides: `0x38F983FcC64217715e00BeA511ddf2525b8DC692`.

3. Anyone can execute arbitrary calls from a smart contract wallet

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PLUME-3
Target: `contracts/smart-wallets/src/extensions/SignedOperations.sol`	

Description

The `executeSignedOperations` function allows anyone to execute arbitrary calls in a smart contract wallet due to the incorrect implementation of EIP-1271. The function tries to validate the signature by calling the `isValidSignature` of the `msg.sender`; when the `msg.sender` is a contract, it can return `true` and bypass the signature validation.

```
function executeSignedOperations(
     address[] calldata targets,
     bytes[] calldata calls,
     uint256[] calldata values,
     bytes32 nonce,
     bytes32 nonceDependency,
     uint256 expiresAt,
     uint8 v,
     bytes32 r,
     bytes32 s
 ) external {
         ...

         try IERC1271(msg.sender).isValidSignature(hash, abi.encodePacked(r, s, v))
returns (bytes4 magicValue) {
             if (magicValue != MAGICVALUE) {
                 revert InvalidSigner(nonce, msg.sender);
             }
         } catch {
             address signer = ECDSA.recover(hash, v, r, s);
             if (signer != address(this)) {
                 revert InvalidSigner(nonce, signer);
             }
         }
         $.nonces[nonce] = 1;
     }

     for (uint256 i = 0; i < length; i++) {
         {
             (bool success,) = targets[i].call{ value: values[i] }(calls[i]);
             if (success) {
                 continue;
             }
```

```
            }
            revert FailedCall(nonce, targets[i], calls[i], values[i]);
        }

        emit SignedOperationsExecuted(msg.sender, nonce);
    }
```

Figure 3.1: Snippet of code for the `executeSignedOperations` function (`contracts/smart-wallets/src/extensions/SignedOperations.sol#L164-L202`)

Exploit Scenario

Eve deploys a smart contract with a `isValidSignature` implemented that always returns `true`. She then arbitrarily calls the `executeSignedOperations` function of Alice's smart contract wallet to transfer all of her tokens to Eve, bypassing the signature validation.

Recommendations

Short term, remove the implementation of EIP-1271, as it is not necessary since the smart contract wallets are implemented at the protocol level.

Long term, when designing the protocol accurately, consider what makes sense to implement and for what reasons. If no strong reasons are present, it is always better not to overcomplicate the protocol and to reduce the attack surface.

4. Custom error emitted with incorrect arguments

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PLUME-4
Target: `smart-wallets/src/extensions/AssetVault.sol`	

Description

In the `acceptYieldAllowance` function, the `MismatchedExpiration` custom error is emitted with first argument `allowance.expiration` and second argument `expiration` (figure 4.1). However, the error definition (figure 4.2) has the argument swapped, where the invalid expiration is first and actual expiration is second.

```
function acceptYieldAllowance(AssetToken assetToken, uint256 amount, uint256
expiration) external {
     ...
     if (allowance.expiration != expiration) {
         revert MismatchedExpiration(allowance.expiration, expiration);
     }
     ...
```

Figure 4.1: Snippet of code for the `acceptYieldAllowance` function (`contracts/smart-wallets/src/extensions/AssetVault.sol#L299-L301`)

```
/**
  * @notice Indicates a failure because the given expiration does not match the
actual one
  * @param invalidExpiration Expiration timestamp that does not match the actual
expiration
  * @param expiration Actual expiration timestamp at which the yield expires
  */
 error MismatchedExpiration(uint256 invalidExpiration, uint256 expiration);
```

Figure 4.2: `MismatchedExpiration` error definition (`contracts/smart-wallets/src/extensions/AssetVault.sol#L138-L143`)

Recommendations

Short term, swap the variables used when emitting the `MismatchedExpiration` error.

5. Missing event in WalletFactory's upgrade function

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PLUME-5
Target: `contracts/smart-wallets/src/WalletFactory.sol`	

Description

The `upgrade` function in the `WalletFactory` contract allows the owner to update the smart wallet address implementation, which is a critical operation; however, this does not emit an event.

```
function upgrade(SmartWallet smartWallet_) public onlyOwner {
     smartWallet = smartWallet_;
 }
```

Figure 5.1: The `upgrade` function (`contracts/smart-wallets/src/WalletFactory.sol#L37-L39`)

Recommendations

Short term, emit an event in the `upgrade` function with the new smart wallet address.

Long term, consider always emitting events for critical operations and implementing a monitoring system.

6. AssetToken's transfer always reverts

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PLUME-6
Target: `contracts/smart-wallets/src/token/AssetToken.sol`	

Description

The `getBalanceAvailable` function reverts due to the high-level call to the `getBalanceLocked` function of a smart wallet user. A smart wallet in Plume is an EOA where the code executed is overwritten at runtime; however, the `extcodesize` opcode still returns 0. The Solidity compiler adds a check that the target address has code for high-level calls, so in this case the check fails, causing the function to revert.

```
 function getBalanceAvailable(address user) public view returns (uint256
balanceAvailable) {
    return balanceOf(user) - SmartWallet(payable(user)).getBalanceLocked(this);
}
```

Figure 6.1: The `getBalanceAvailable` function (contracts/smart-wallets/src/token/AssetToken.sol#L287-L289)

The `getBalanceAvailable` function is used in the overridden `_update` function, which is called every time a transfer occurs. This makes every transfer of the token revert.

```
function _update(address from, address to, uint256 value) internal
override(YieldDistributionToken) {
     ...

     if (from != address(0)) {
         if (getBalanceAvailable(from) < value) {
             revert InsufficientBalance(from);
         }
     }

     ...
 }
```

Figure 6.2: Snippet of code for the `_update` function (contracts/smart-wallets/src/token/AssetToken.sol#L125-L147)

Exploit Scenario

Users try to transfer tokens, but the transaction always reverts. As a result, the system is unusable.

Recommendations

Short term, use a low-level call to call the `getBalanceLocked` function of `SmartWallet`. Document that when interacting with a `SmartWallet`, only low-level calls must be used.

Long term, implements testing of the smart contracts with the modified go-ethereum environment.

7. Yield beneficiary can temporarily DoS the giver for a specific token through acceptYieldAllowance function

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-PLUME-7
Target: `contracts/smart-wallets/src/extensions/AssetVault.sol`	

Description

A yield beneficiary can DoS the redistribution of yield by repeatedly accepting a minimum amount of tokens. This can cause the linked list that is iterated during that process to become too long and revert with an out-of-gas error.

After some validation has been performed, the `acceptYieldAllowance` function creates a new item in the `yieldDistributions` mapping that simulates a linked list for each `assetToken`.

```
function acceptYieldAllowance(AssetToken assetToken, uint256 amount, uint256
expiration) external {
     ...

     // Append the newly created yield distribution to the end of the linked list
     YieldDistributionListItem storage distribution =
$.yieldDistributions[assetToken];
     while (distribution.yield.amount > 0) {
         distribution = distribution.next[0];
     }
     distribution.beneficiary = beneficiary;
     distribution.yield.amount = amount;
     distribution.yield.expiration = expiration;
```

Figure 7.1: Snippet of code for the `acceptYieldAllowance` function (`contracts/smart-wallets/src/extensions/AssetVault.sol#L288-L318`)

The `yieldDistributions` linked list is iterated in various places. In the `renounceYieldDistribution` and `clearYieldDistributions` functions, if the full iteration can not be completed in a single transaction, it is not a problem and the function can be called a second time to complete the action. However, in the `acceptYieldAllowance`, `redistributeYield`, and `getBalanceLocked` functions, this prevents the process from executing for every other user. More interestingly, the `getBalanceLocked` function (figure 7.2) is called when transferring a token to check the user's available amount, which effectively makes the transfer of that token impossible.

```
function getBalanceLocked(AssetToken assetToken) public view returns (uint256
balanceLocked) {
       // Iterate through the list and sum up the locked balance across all yield
distributions
       YieldDistributionListItem storage distribution =
_getAssetVaultStorage().yieldDistributions[assetToken];
       uint256 amountLocked = distribution.yield.amount;
       while (amountLocked > 0) {
           if (distribution.yield.expiration > block.timestamp) {
               balanceLocked += amountLocked;
           }
           distribution = distribution.next[0];
           amountLocked = distribution.yield.amount;
       }
   }
```

Figure 7.2: The `getBalanceLocked` function (contracts/smart-wallets/src/extensions/AssetVault.sol#L268-L279)

Note that the DoS is temporary because the item can be removed with the `clearYieldDistributions` function once it expires in the distribution list.

Exploit Scenario

Eve gets a yield allowance of 100 tokens TK1 (18 decimals) from Alice. She then starts calling the `acceptYieldAllowance` function with 1 wei of token each time until the transaction reverts, which effectively means that Alice cannot transfer TK1 nor redistribute the yield to the beneficiaries.

Recommendations

Short term, in the `acceptYieldAllowance` function, increase the amount of the corresponding item associated with the current beneficiary and the expiration if it is present while iterating them instead of always adding a new item. Track the balance locked in a state variable, since this balance is used when transferring the token to reduce the gas cost.

Long term, remove the linked list data structure. If this is not possible, document the behavior that functions get increasingly more expensive as more yield is given.

8. acceptYieldAllowance does not check that the amount is actually available

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PLUME-8
Target: `contracts/smart-wallets/src/extensions/AssetVault.sol`	

Description

The `acceptYieldAllowance` function does not validate that the amount allowed to a user who is accepting it is actually available. This could allow a user to lock more tokens than the current balance.

```
function acceptYieldAllowance(AssetToken assetToken, uint256 amount, uint256
expiration) external {
     AssetVaultStorage storage $ = _getAssetVaultStorage();
     address beneficiary = msg.sender;
     Yield storage allowance = $.yieldAllowances[assetToken][beneficiary];

     if (amount == 0) {
         revert ZeroAmount();
     }
     if (expiration <= block.timestamp) {
         revert InvalidExpiration(expiration, block.timestamp);
     }
     if (allowance.expiration != expiration) {
         revert MismatchedExpiration(allowance.expiration, expiration);
     }
     if (allowance.amount < amount) {
         revert InsufficientYieldAllowance(assetToken, beneficiary,
allowance.amount, amount);
     }

     allowance.amount -= amount;
     ...
 }
```

Figure 8.1: Snippet of code for the `acceptYieldAllowance` function (`contracts/smart-wallets/src/extensions/AssetVault.sol#L288-L318`)

As a result, the `getBalanceLocked` function could return a value greater than the actual balance. This value is used in the `getBalanceAvailable` function (figure 8.2), which would revert in this case and make the token transfer impossible, since `getBalanceAvailable` is used to check if the amount to transfer is valid. The transfer would fail anyway, but with a correct “not enough balance” error instead of a simple revert.

```
function getBalanceAvailable(address user) public view returns (uint256
balanceAvailable)
{
        return balanceOf(user) - SmartWallet(payable(user)).getBalanceLocked(this);
}
```

Figure 8.2: The `getBalanceAvailable` function (contracts/smart-wallets/src/token/AssetToken.sol#L287-L289)

Another case where the transaction would revert the `redistributeYield` function were correctly implemented is when redistributing the yield (figure 8.3) because the amount to send is based on the amount locked. However, there is currently a TODO comment here, and the transfer of tokens is not executed.

```
function redistributeYield(
     AssetToken assetToken,
     ERC20 currencyToken,
     uint256 currencyTokenAmount
 ) external onlyWallet {
     ...
     uint256 amountLocked = distribution.yield.amount;
     while (amountLocked > 0) {
         if (distribution.yield.expiration > block.timestamp) {
             uint256 yieldShare = (currencyTokenAmount * amountLocked) /
amountTotal;
             // TODO: transfer yield from the user wallet to the beneficiary
             emit YieldRedistributed(assetToken, distribution.beneficiary,
currencyToken, yieldShare);
         }

         distribution = distribution.next[0];
         amountLocked = distribution.yield.amount;
     }
 }
```

Figure 8.3: Snippet of code for the `redistributeYield` function (contracts/smart-wallets/src/extensions/AssetVault.sol#L236-L260)

Note that the denial of service is temporary, as once the item expires in the distribution list, it can be removed with the `clearYieldDistributions` function.

Exploit Scenario

Alice has 100 TK1 tokens. She then gives a yield allowance of 60 TK1 to Bob and 50 TK1 to Charlie. They accept the allowance, and Alice tries to transfer some TK1 token, but the transaction reverts without returning an insufficient balance message.

Recommendations

Short term, validate in the `acceptYieldAllowance` function that the amount the user is trying to accept is available.

Long term, make unit tests for the intended positive and negative flows of every function.

9. ClaimedYield considers only currently whitelisted users

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PLUME-9
Target: `contracts/smart-wallets/src/extensions/AssetToken.sol`	

Description

The `claimedYield` function should represent the claimed yield by all users; however, it sums only the yield claimed by the current whitelisted users, missing the users who were whitelisted in the past. If the whitelisting is not enabled since the start, this function would return 0. Note that the `claimedYield` function is also used to compute the unclaimed yield.

```
/// @notice Claimed yield across all AssetTokens for all users
function claimedYield() public view returns (uint256 amount) {
    AssetTokenStorage storage $ = _getAssetTokenStorage();
    uint256 length = $.whitelist.length;
    for (uint256 i = 0; i < length; ++i) {
        amount +=
_getYieldDistributionTokenStorage().yieldWithdrawn[$.whitelist[i]];
    }
}
```

Figure 9.1: The `claimedYield` function (`contracts/smart-wallets/src/token/AssetToken.sol#L298-L304`)

Exploit Scenario

A third-party contract relies on the total unclaimed yield, but the value returned is incorrect, causing the contract to behave incorrectly.

Recommendations

Short term, sum the yield withdrawn by iterating all of the `holders` storage variables, which contain all the users who ever had tokens.

Long term, implements tests to check the correct behavior of the system. For this specific issue, test for different scenarios such as with whitelist, without whitelist, disabling it, and enabling it again.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.