Smart Contract Audit

Date	March 2021

UMBRA SMART CONTRACTS

This report contains the results of ScopeLift's engagement to review the Umbra Protocol smart contract smart contracts.

1 | EXECUTIVE SUMMARY

This report contains the results of ScopeLift's engagement to review the Umbra Protocol smart contract smart contracts.

Nicholas Ward conducted the review between March 22nd, 2021 and March 26th.

2 | SCOPE

The review focused on the commit hash fa2e17367d66a85f20c77299ded5942d9ab64fe0. A cursory review of the ENS Resolver contract for stealth keys was also performed at commit hash 2d7795082308d303eb23c66490579a5b21a1bac9. The Appendix contains a list of files within scope.

3 | SECURITY SPECIFICATION

ScopeLift's documentation is provided in the project README as well as in the code. This section should not be used in place of documentation. This is the summary:

Umbra allows semi-private payments to be made between two parties by using "stealth address". `Umbra` forwards ETH payments using these stealth addresses and emits an encrypted message from the sender. The recipient can use this encrypted message to identify the intended recipients and recover forwarded funds.

`Umbra` contracts accept token payments from senders and allow the recipient to withdraw tokens. A signed message from a "sponsor" can be sent to the `Umbra` contract to allow the recipient to withdraw their tokens. Token payment withdrawals may also include a call from a "sponsor" to the contract. This can be used to forward payments into another privacy solution, interact on-chain protocol, or take any other arbitrary action after receiving a payment. The Umbra contract does not provide nonce-based signature replay protection due to the expectation that stealth addresses will be used only once.

It is important to note that privacy gained through the Umbra Protocol may be affected by users' privacy hygience and that of their counterparties.

4 | FINDINGS

Each issue is assigned a severity:

- **Minor** problems are subjective. These are usually suggestions about best practices or readability. These issues should be addressed by code maintainers.
- **Medium** issues are objective, but they are not security vulnerabilities. These issues should be addressed, unless there are compelling reasons not to.
- Security vulnerabilities are critical issues that can't be exploited directly or require special conditions to be exploited. All of these **Major** problems should be addressed.
- **Critical**Security vulnerabilities that could be exploited to cause **Critical**issues need to be addressed.

4.1 | Reuse of `CHAINID` from contract deployment

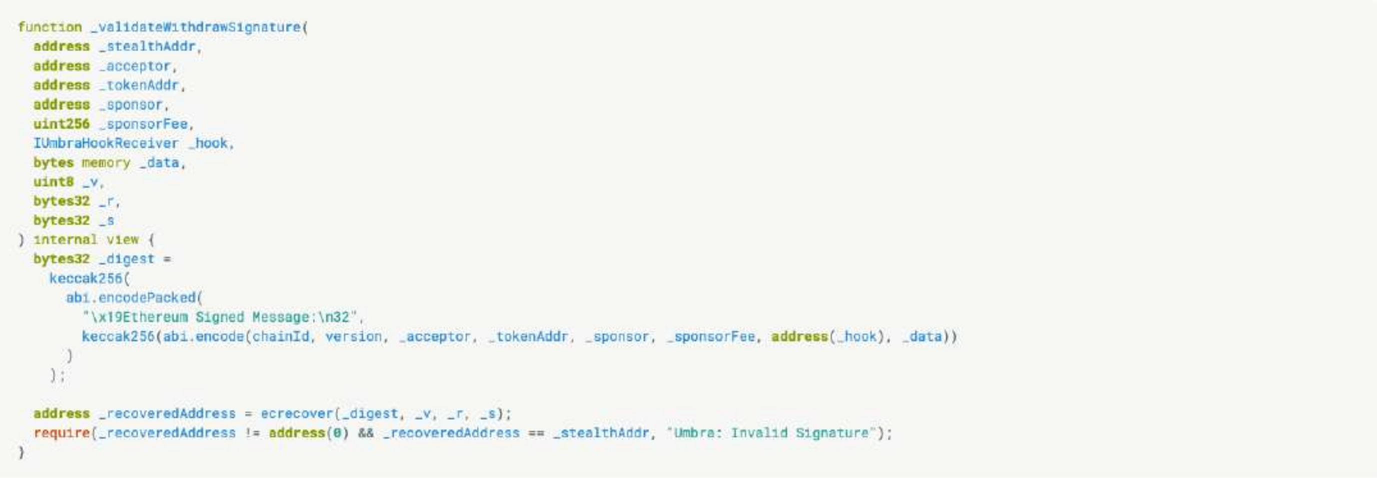
Minor Fixed

This is addressed in ScopeLift/umbra-protocol@7cfdc81.

Description

The internal function _validateWithdrawSignature() is used to check whether a sponsored token withdrawal is approved by the owner of the stealth address that received the tokens. The chain ID is signed to prevent signature replays on other EVM compatible chains.

contracts/contracts/Umbra.sol:L307-L329



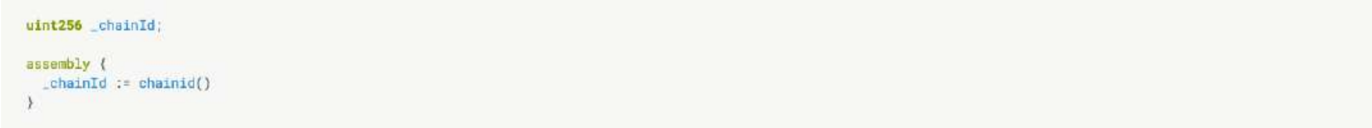
```
function _validateWithdrawSignature(
  address _stealthAddr,
  address _acceptor,
  address _tokenAddr,
  address _sponsor,
  uint256 _sponsorFee,
  IUmbraHookReceiver _hook,
  bytes memory _data,
  uint8 _v,
  bytes32 _r,
  bytes32 _s
) internal view {
  bytes32 _digest =
    keccak256(
      abi.encodePacked(
        "\x19Ethereum Signed Message:\n32",
        keccak256(abi.encode(chainId, version, _acceptor, _tokenAddr, _sponsor, _sponsorFee, address(_hook), _data))
      )
    );

  address _recoveredAddress = ecrecover(_digest, _v, _r, _s);
  require(_recoveredAddress != address(0) && _recoveredAddress == _stealthAddr, "Umbra: Invalid Signature");
}
```

This chain ID is however set in the contract constructor as an immutable value. The Umbra contract will exist on both the resulting chains in the event of an upcoming contentious hard fork to the Ethereum network. The Umbra contracts wouldn't be aware that one of these chains would change the network's ID. Signing the Umbra contract on one chain would allow you to replay your signature on the other.

Recommendation

Replace the use of the chainId immutable value with the CHAINID opcode in _validateWithdrawSignature(). CHAINID can only be accessed using Solidity's Inline Assembly. This means that it would have to be accessed the same way it is currently accessed by the contract's builder.

contracts/contracts/Umbra.sol:L307-L329



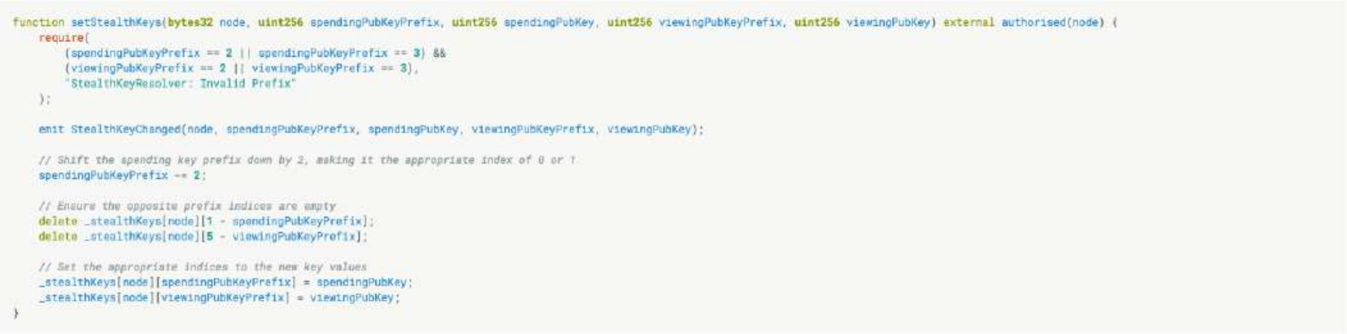
```
uint256 _chainId;

assembly {
  _chainId := chainid()
}
```

5 | RECOMMENDATIONS

Description

The StealthKeyResolver stores keys in a mapping bytes32=>uint256=>uint256 that maps nodes=>prefixes=>keys. To distinguish between spending public keys and viewing them, the prefixes in setStealthKeys() are offset. These offsets can be reversed by stealthKeys() view.

contracts/profiles/StealthKeyResolver.sol:L37-L56



```
function setStealthKeys(bytes32 node, uint256 spendingPubKeyPrefix, uint256 spendingPubKey, uint256 viewingPubKeyPrefix, uint256 viewingPubKey) external authorised(node) {
    require(
        (spendingPubKeyPrefix == 2 || spendingPubKeyPrefix == 3) &&
        (viewingPubKeyPrefix == 2 || viewingPubKeyPrefix == 3),
        "StealthKeyResolver: Invalid Prefix"
    );

    emit StealthKeyChanged(node, spendingPubKeyPrefix, spendingPubKey, viewingPubKeyPrefix, viewingPubKey);

    // Shift the spending key prefix down by 2, making it the appropriate index of 0 or 1
    spendingPubKeyPrefix -= 2;

    // Ensure the opposite prefix indices are empty
    delete _stealthKeys[node][1 - spendingPubKeyPrefix];
    delete _stealthKeys[node][5 - viewingPubKeyPrefix];

    // Set the appropriate indices to the new key values
    _stealthKeys[node][spendingPubKeyPrefix] = spendingPubKey;
    _stealthKeys[node][viewingPubKeyPrefix] = viewingPubKey;
}
```

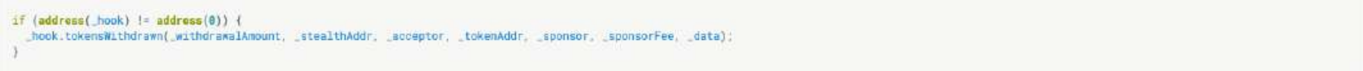
Manual prefix adjustment adds complexity to a function that is otherwise very simple. This can be avoided by splitting the mapping into two distinct ones - one for viewing keys, and one for spending keys. Make sure to specify which mappings are visible.

5.2 | Document potential edge cases for hook receiver contracts

Description

The functions withdrawTokenAndCall() and withdrawTokenAndCallOnBehalf() make a call to a hook contract designated by the owner of the withdrawing stealth address.

contracts/contracts/Umbra.sol:L289-L291



```
if (address(_hook) != address(0)) {
  _hook.tokensWithdrawn(_withdrawalAmount, _stealthAddr, _acceptor, _tokenAddr, _sponsor, _sponsorFee, _data);
}
```

The Umbra contract has very few restrictions on these calls. Anybody can make a call to a Hook contract by sending tokens to an address they control, and then withdrawing the tokens. The target address will be the hook receiver. These UmbraHookReceiver contract developers should validate the caller and the parameters of the tokensWithdrawn() function. Edge cases can arise and should be addressed when possible. The following are some examples:

| The amount may not have been transferred directly to the hook receiver.
| TokensWithdrawn() may have received four addresses that could all be identical. These address parameters can also be used to address any address. This includes the token contract address and the address for the hook receiver.
| The token may not be of any value.
| It is possible that the token received was malicious. Only requirements are that the token contract address contain code and accepts calls from the ERC20 methods transfer() or transferFrom().

It is hard to find a viable exploit without knowing what future hook receiver contracts might do, but this is a somewhat contrived example.

Let's say that a user creates a hook receiver contract which accepts an arbitrary token TOK and provides liquidity immediately to the ETH -TOK Uniswap pairing when tokensWithdrawn() gets called by Umbra. An attacker could create a malicious token which cannot be transferred from its own Uniswap Pair contracts and force Umbra to call the hook receiver contract. The hook receiver could provide liquidity to the pool, but it would not be able to remove it. This would result in any Ethereum that was lost.

5.3 | Document token behavior restrictions

It is crucial to clearly identify which tokens will be supported by any protocol that interacts directly with ERC20 tokens. This is often done by providing a description of the expected behavior of ERC20 tokens. After careful consideration of particular tokens and their interactions with protocol, you can relax this specification.

The Umbra Protocol does not support the following behavior.

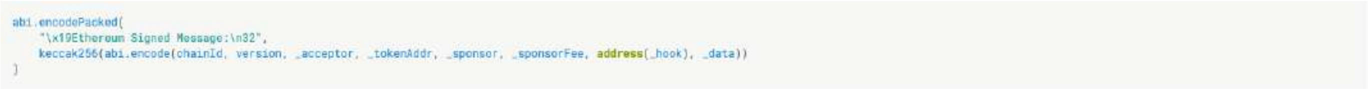
- Fee-on-transfer or deflationary tokens: Tokens where the amount of a transfer does not affect the balance of the recipient. Balances may be unexpectedly reduced by another mechanism. These tokens can be sent using the sendToken() function. However, Umbra's internal accounting will not match the balance recorded in the token contract. This could lead to funds being lost.
- Inflationary tokens: A form of deflationary coins. The Umbra contract does not provide any mechanism to claim positive balance adjustments.
- Rebasing tokens: This is a combination of the cases above. These tokens are tokens in that an account's balance changes with supply expansions and contractions. Unexpected balance adjustments can cause funds to be lost, as the contract does not provide any mechanism for updating its internal accounting.

5.4 | Add an address parameter to withdrawal signatures Fixed

This is addressed in ScopeLift/umbra-protocol@d6e4235, which replaces the version parameter with address(this) in the signature encoding.

Description

As discussed above, the _validateWithdrawSignature() function checks the signer of a digest consisting of the keccak-256 hash of the following preimage:



```
abi.encodePacked(
    "\x19Ethereum Signed Message:\n32",
    keccak256(abi.encode(chainId, version, _acceptor, _tokenAddr, _sponsor, _sponsorFee, address(_hook), _data))
)
```

You might add the address to the signed message. It is possible to deploy multiple contracts with the same version to a single chain. Signatures can be replayed across all contracts. Although users will likely only have balances for one stealth address in these contracts, an address parameter adds replay protection. The contract cannot be self-destructed so a given address can only contain one version of Umbra.

APPENDIX 1 - FILES IN SCOPE

The following files were included in the audit:

File Name	SHA-1 hash
Umbra.sol	bb9fc1f58c7c1246aa8533161153533392042 0b8
IUmbraHookReceiver.sol	f8c1835a62a82c9129318aa05f77cee6e4176d93
StealthKeyResolver.sol	f27bf5e6c29bfd3b516352ca15d0704c3899b65c

APPENDIX 2 DISCLOSURE

ConsenSys Dialigence ("CD") receives compensation from clients (the Clients) for the analysis performed in these reports (the Reports). Reports can be distributed via ConsenSys publications or other distributions.

Reports are not intended to endorse or indict any project or team. They also do not guarantee security for any project. This Report doesn't consider or have any bearing on the economics of token sales, token tokens, or any other product, services, or assets. Cryptographic tokens, which are emerging technologies, carry high technical risks and uncertainties. Any Report does not provide any representation or warranty to Third-Parties in any way. This includes regarding the bug-free nature of code, any business model or proprietors, or the legal compliance of such businesses. The Reports should not be relied upon by any third party, even if it is used to make decisions about buying or selling tokens, products, services, or assets. This Report is not intended as investment advice and should not be relied on as such. It is also not an endorsement of this team or project, and is not a guarantee of absolute security. CD is not obligated to any Third-Party for publishing these Reports.

PURPOSE OF THE REPORTS Reports and analysis contained therein are only for Clients. They can be published with their permission. Our review will only cover Solidity code. We are limited to reviewing the Solidity codes we have identified as being included in this report. Solidity language is still under development. It may have flaws and risks. The review does NOT cover the compiler layer or any other areas that could pose security risks beyond Solidity. Cryptographic tokens, which are emerging technologies, carry high technical risk and uncertainty.

CD makes the Reports accessible to clients and other parties (i.e. "third parties") via its website. CD hopes that the public availability of these analyses will help the blockchain ecosystem to develop best practices in this rapidly changing area of innovation.

LINKS TO OTHER WEBSITES FROM THIS WEB site You can, via hypertext or other computer hyperlinks, gain access web sites owned by people other than ConsenSys. These hyperlinks are provided only for your convenience and are not intended to replace the owners of these web sites. ConsenSysys or CD are not responsible or liable for any content or operation of these Web sites. You also agree that ConsenSysys or CD will not be liable for any third-party Web site. Except as stated below, linking from this Web Site to another site does not mean or imply that ConsenSysys or CD endorses that site's content or its operator. It is up to you to decide whether or not you can use content from any other websites to which the Reports link. ConsenSys or CD will not be responsible for third-party software used on the Web Site. They also assume no liability for any errors or inaccuracies of any output generated by such software.

TIMELINESS CONTENT. The Reports are current as of the Report's date. However, they can be modified at any time. ConsenSys or CD are the only sources of information, unless otherwise indicated.