CD SECURITY

AUDIT REPORT

Smart Invoice
August 2025

Prepared by
zeroXchad
ArnieSec
0xluk3

Introduction

A time-boxed security review of the **Smart Invoice** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Smart Invoice**

SmartInvoiceEscrow

Serves as the main logic and implementation contract. It is a system compromised of 3 main actors, the client, the provider, and the resolver.

The client is responsible for providing the funds and will pay out these funds to the provider after certain milestones have been achieved. Clients have the ability to add milestones or lock the funds for any reason they deem necessary, such as a dispute that needs to be resolved by the resolver.

The provider is in charge of completing said work/ milestones. After providing proof of milestones, he may receive funds that are released by the client. Similar to the client, the provider also has the ability to add milestones and lock the funds in the escrow. This ensures that both the client and the provider are given certain assurances that funds are likely to be paid out if the work/ milestones are reached.

The resolver is a trusted 3rd party that exists to resolve any potential disputes. This can allow the provider to be paid if he does work but the client refuses to pay. The resolver can also protect the client and allow him to withdraw funds if no work or milestones are achieved. The resolver can either be an EOA, or an arbitration service that follows ERC-792 such as kleros.

SmartInvoiceFactory

The SmartInvoiceFactory contract is a factory that is used to deploy clones of the SmartInvoiceEscrow implementation contract. The contract has many ways of deploying the clones, these include simple create, createDeterministic, createAndDeposit, createDeterministicAndDeposit. The contract also includes multiple helper functions that help with predicting a deterministic address as well as a function that allows resolvers to update their resolution fee.

SafeSplits

the safe splits contracts include SafeSplitsEscrowZap, SafeSplitsDaoEscrowZap, and SplitV2Lib. The SplitV2Lib is a library with helper functions that help with calculations of distributions and allocated amounts.

The SafeSplitsEscrowZao contract allows a user to deploy a safe, a splits v2 splitter, and a smartInvoice escrow all in a single transaction. The SafeSplitsDaoEscrowZap contract inherits and extends the functionality in safeSplitsEscrowZap by adding built-in DAO fee splitting via spoilsBPS.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 85d02dc2b15a1ff5254aef009089b5c99cb27398**

Scope

The following folders were in scope of the audit:

- `SmartInvoiceEscrow.sol`
- `SmartInvoiceFactory.sol`
- `SafeSplitsDaoEscrowZap.sol`
- `SafeSplitsEscrowZap.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 6 issues
- Low & Info: 11 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	The `resolve` and `rule` can revert due to ERC20 blacklisting functionality	High	Fixed

ID	Title	Severity	Status
[M-01]	SmartInvoiceEscrow's `init` can be front-run with SmartInvoiceFactory's `updateResolutionRateBPS`	Medium	Fixed
[M-02]	Donation attack will result in dos	Medium	Fixed
[M-03]	DOS of adding milestones by client	Medium	Acknowledged
[M-04]	`create` is susceptible to re orgs	Medium	Fixed
[M-05]	Unlocking depends solely on the third party action	Medium	Fixed
[M-06]	Provider update can redirect payouts via stale `providerReceiver`	Medium	Fixed
[L-01]	Client must wait until termination time in order to withdraw unrelated tokens	Low	Fixed
[L-02]	Release with milestone does not check for termination	Low	Fixed
[L-03]	Clients and providers should be allowed to change receivers when locked	Low	Fixed
[L-04]	Lock assertion in `receive` and `wrapETH` can be too strict	Low	Fixed
[L-05]	The AccessControlDefaultAdminRules is not in use	Low	Fixed
[L-06]	The terminationTime assertion can be mutually exclusive0	Low	Fixed
[L-07]	Released accounting not updated in dispute paths	Low	Fixed
[I-01]	The verify can be repeated	Informational	Fixed
[I-02]	Redundant check within the `_autoVerify` function	Informational	Fixed
[I-03]	Redundant calculation within the isFunded function	Informational	Fixed
[I-04]	Zero-value milestones allowed can cause confusing state and event spam	Informational	Fixed

Detailed Findings

[H-01] The `resolve` and `rule` can revert due to ERC20 blacklisting functionality

Severity

Impact: High

Likelihood: Medium

Description

The `SmartInvoiceEscrow` contract makes use of push over pull pattern within the following functions:

- `resolve`
- `rule` among the others. Such implementation might be problematic, especially when ERC20 token with blacklisting functionality is in use, such as USDT or USDC. Whenever one of the party addresses becomes blacklisted, it would not be possible to resolve or rule a dispute between the parties. Additionally, the `lock` mechanism prevents usage of any other functionality in the protocol, thus it is likely that e.g. when client or provider addresses were blacklisted, the tokens will be permanently or temporarily locked within the contract.

Recommendations

It is recommended to apply pull over push pattern within the codebase. Such an approach requires robust accounting to be implemented. Finally, additional functionality must be implemented that allows the party to transfer tokens on demand.

[M-01] SmartInvoiceEscrow's `init` can be front-run with SmartInvoiceFactory's `updateResolutionRateBPS`

Severity

Impact: Medium

Likelihood: Low

Description

Within the SmartInvoiceEscrow's `init` function a call to the factory is being made to retrieve resolution rate of a particular resolver

```
    function _handleData(
        address _provider,
        uint256[] calldata _amounts,
        bytes calldata _data
    ) internal virtual {
...
        uint256 _resolutionRateBPS = FACTORY.resolutionRateOf(
            initData.resolver
        );
...
```

Simultaneously, the resolution rate can be updated by the resolver at any moment.

```
    function updateResolutionRateBPS(
        uint256 _rateBPS,
        string calldata _details
    ) external {
        if (_rateBPS < 1 || _rateBPS > 1000) revert
InvalidResolutionRate(); // must be between 1-1000 BPS
        _resolutionRateBPS[msg.sender] = _rateBPS;
        emit UpdateResolutionRate(msg.sender, _rateBPS, _details);
    }
```

Such implementation can pose a risk, that a resolver can attract clients and providers with a low resolution rate and lure them to select such resolver as prime option. However, whenever the new invoice is going to be deployed, the resolver can front run such transaction and call the `updateResolutionRateBPS` to set the maximal rate possible. With such approach the resolver can attempt to maximise the profit of future possible disputes to be resolved.

Recommendations

It is recommended to prevent the possibility of the aforementioned scenario from occurring. It can be prevented in several ways, e.g.:

- A cool down period can be applied before which the resolver is unable to update the resolution fee.
- Within the new invoice deployment, users can provide additional input parameter defining maximal resolution fee they are willing to accept in the future disputes.

[M-02] Donation attack will result in dos

Severity

Impact: High

Likelihood: Low

Description

In the `resolve` function, a strict comparison is used

```
        if (_clientAward + _providerAward != balance - resolutionFee)
            revert ResolutionMismatch();

        if (_providerAward > 0) {
            _transferPayment(token, _providerAward);
        }
        if (_clientAward > 0) {
            _withdrawDeposit(token, _clientAward);
        }
        if (resolutionFee > 0) {
            IERC20(token).safeTransfer(resolver, resolutionFee);
        }
```

If the client award + the provider award does not equal the balance minus the resolution fee, then the tx will revert. This will allow a malicious user, including a malicious provider or client , to dos the execution of the resolve function by donating funds and frontrunning the call to `resolve`. this will result in the call to resolve reverting due to the strict inequality checks.

Recommendations

Consider not making the inequality strict and change it to

```
        if (_clientAward + _providerAward > balance - resolutionFee)
            revert ResolutionMismatch();
```

[M-03] DOS of adding milestones by client

Severity

Impact: Medium

Likelihood: Low

Description

The `addMilestones` allows the provider or client to add milestones, however this functionality may be DOSed by a malicious client.

```
        uint256 newTotal = total;
        for (uint256 i = 0; i < _milestones.length; ) {
            amounts.push(_milestones[i]);
            newTotal += _milestones[i];
            unchecked {
                ++i;
            }
        }

        total = newTotal;
```

This can be done simply by adding a milestone with an amount that would result in uint256 max value when added with the previous total. This will cause any additional additions of milestones to revert due to the overflow because of the large value.

Additionally view function such as `isFullFunded` will also stop working.

```
    function isFullyFunded() external view returns (bool) {
        return IERC20(token).balanceOf(address(this)) + released >= total;
    }
```

Recommendations

The fix is not obvious as restricting the value of the amounts in milestones may hurt the service.

[M-04] `create` is susceptible to re orgs

Severity

Impact: High

Likelihood: Low

Description

```
    function create(
        address _recipient,
        uint256[] calldata _amounts,
        bytes calldata _data,
        bytes32 _escrowType
    ) public override returns (address) {
        uint256 _version = currentVersions[_escrowType];
        address _implementation = implementations[_escrowType][_version];
        if (_implementation == address(0)) revert
ImplementationDoesNotExist();

        address invoiceAddress = Clones.clone(_implementation);
        _init(
            invoiceAddress,
            _recipient,
            _amounts,
            _data,
            _escrowType,
            _version
        );

        return invoiceAddress;
    }
```

Above, the create function is one of the ways a user can deploy a safe, the above function will not prefund the escrow so separate txs must be sent after the creation in order to fund the escrow. This is an issue as a re org can lead to funds being lost for the client and provider.

Step by step on how a re org can lead to loss of funds.

1. bob calls create in block 1
2. Alice calls create to create another escrow in block 2
3. bob funds the escrow by sending funds to the escrow on block 3
4. a re org happens and flips block 1 and 2 therefore block 2 now occurs first

5. since there is no salts, alice will now have bobs old escrow address and block 3 will still be bob sending funds to that address.
6. bob now has an empty escrow and the funds are now in alice's escrow.

Recommendations

Although the risk is low, we recommend to not allow the ability to deploy an escrow without some sort of differentiating salt system. The contract already allows the deployment of escrow via createDeterministic which does not suffer from this attack. Additionally the createAndDeposit function, although not differentiated with salt, it is resistant to reorgs because the funding of the escrow and the deployment happen in the same tx and are not separated into 2 different tx's. Ultimately the issue can be acknowledged or the `create` function can be removed to avoid any potential loss of funds due to reorgs.

[M-05] Unlocking depends solely on the third party action

Severity

Impact: Medium

Likelihood: Medium

Description

Whenever the `SmartInvoiceEscrow` contract is locked, two functions can be called to unlock it:

- `resolve`
- `rule`

In both cases, the action must by initiated by the `resolver`. This poses a risk, that whenever a `resolver` decides not to resolve an issue, the contract remains locked permanently and tokens are locked within it as well. It must also be considered, that the most probable scenario is that `resolver` can simply lose access to the private keys.

Recommendations

It is recommended to review the business rules implemented within the protocol and consider the aforementioned scenario. It can be considered to implement an additional fallback mechanism, which can be e.g. unlocking of the contract after grace period.

[M-06] Provider update can redirect payouts via stale `providerReceiver`

Severity Medium

Impact: Medium

Likelihood: Medium

Description

In SmartInvoiceEscrow.sol, the contract allows the current provider to hand over the provider role to a new address, intending to change who is the service performer.

However, it leaves the prior providerReceiver unchanged, and payouts continue to go to providerReceiver != 0 ? providerReceiver : provider. As a result, when the old provider had set a custom providerReceiver, subsequent client-initiated release calls will still pay that old receiver even after the provider role has been transferred, enabling the prior provider to keep receiving payments while a new provider should be receiving it instead.

```
function updateProvider(address _provider) external {
    if (msg.sender != provider) revert NotProvider(msg.sender);
    if (_provider == address(0)) revert InvalidProvider();
    if (locked) revert Locked();

    provider = _provider;
    emit UpdatedProvider(_provider);
}
```

Recommendations

Clear or force-update providerReceiver in updateProvider (e.g., set it to address(0)).

[L-01] Client must wait until termination time in order to withdraw unrelated tokens

Description

```
    function withdrawTokens(address _token) external override nonReentrant
{
        if (_token == token) {
            _withdraw();
        } else {
            if (locked) revert Locked();
            if (block.timestamp <= terminationTime) revert
NotTerminated();
            uint256 balance = IERC20(_token).balanceOf(address(this));
            if (balance == 0) revert BalanceIsZero();

            _withdrawDeposit(_token, balance);
        }
    }
```

From the snippet above we can see that even if _token != token, the client must still wait until after the termination time in order to withdraw. This does not make sense since only the main `token` should be restricted to these rules since it is the token used for the escrow. However other random tokens sent by the client on accident should not be subject to the same rules since they will not be of importance for the escrow provider.

Recommendations

Consider removing termination check for tokens that are not the main token.

[L-02] Release with milestone does not check for termination

Description

The `release` function that includes milestone parameter does not check that the termination time has not passed. This is in difference to the other `release` function which reverts if the termination time has passed.

```
    function release(
        uint256 _milestone
    ) external virtual override nonReentrant {
        if (locked) revert Locked();
        if (msg.sender != client) revert NotClient(msg.sender);
        if (_milestone < milestone) revert InvalidMilestone();
        if (_milestone >= amounts.length) revert InvalidMilestone();
```

No check for termination.

```
    function _release() internal virtual {
        if (locked) revert Locked();
        if (msg.sender != client) revert NotClient(msg.sender);
        if (block.timestamp > terminationTime) revert Terminated();
```

includes termination check.

Recommendations

Consider adding the same termination check to `release` with milestone parameter.

[L-03] Clients and providers should be allowed to change receivers when locked

Description

Currently clients and providers are disallowed to change their receivers during the lock == true state.

```
    function updateClientReceiver(address _clientReceiver) external {
        if (msg.sender != client) revert NotClient(msg.sender);
        if (_clientReceiver == address(0) || _clientReceiver ==
address(this))
            revert InvalidClientReceiver();
        if (locked) revert Locked();

        clientReceiver = _clientReceiver;
        emit UpdatedClientReceiver(_clientReceiver);
    }

    function updateProviderReceiver(address _providerReceiver) external {
        if (msg.sender != provider) revert NotProvider(msg.sender);
        if (
            _providerReceiver == address(0) ||
            _providerReceiver == address(this)
        ) revert InvalidProviderReceiver();
        if (locked) revert Locked();

        providerReceiver = _providerReceiver;
        emit UpdatedProviderReceiver(_providerReceiver);
    }
```

This should not be the case because during resolution funds are transferred to the receivers before the locked state is lifted not allowing users to ever change their receivers at any period during the locked state.

```
        if (_providerAward > 0) {
            _transferPayment(token, _providerAward);
        }
        if (_clientAward > 0) {
            _withdrawDeposit(token, _clientAward);
        }
        if (resolutionFee > 0) {
            IERC20(token).safeTransfer(resolver, resolutionFee);
        }

        // Complete all milestones
        milestone = amounts.length;

        // Reset locked state
        locked = false;
```

Recommendations

Consider allowing a change of receivers during the locked state as it imposes no risk and enhances UX.

[L-04] Lock assertion in `receive` and `wrapETH` can be too strict

Description

The `lock` functionality within the `receive` and `wrapETH` functions can prevent these actions from being executed. However, this can be considered too strict, especially, in contrast to the invoices that make use of standard ERC20 tokens instead of native token and wrapped ETH. In such contract instances, there is no mechanism preventing tokens transfer. Eventually, the end user may decide to simply transfer the wrapped ETH instead of native token. However, in extreme edge case, the call to the `wrapETH` can be prevented by lock mechanism, for already transferred tokens (e.g. by means of `selfDestruct`).

```
receive() external payable nonReentrant {
        if (locked) revert Locked();
        if (token != address(WRAPPED_ETH)) revert InvalidWrappedETH();
        WRAPPED_ETH.deposit{value: msg.value}();
        emit Deposit(msg.sender, msg.value, token);
    }
...
    function wrapETH() external nonReentrant {
        if (locked) revert Locked();
        uint256 bal = address(this).balance;
        if (bal == 0) revert BalanceIsZero();
        WRAPPED_ETH.deposit{value: bal}();
        emit WrappedStrayETH(bal);
        if (token == address(WRAPPED_ETH)) {
            // Log address(this) as depositor since it was obtained via
self-destruct
            emit Deposit(address(this), bal, token);
        }
        // Handle release of WETH as per `releaseTokens` or
`withdrawTokens` as needed
    }
```

Recommendations

It is recommended to reconsider the implementation and remove `locked` assertions from the aforementioned functions.

[L-05] The AccessControlDefaultAdminRules is not in use

Description

The `SmartInvoiceFactory` and `SafeSplitsEscrowZap` implement AccessControl library. However, the AccessControlDefaultAdminRules extension is considered superior over it.

This contract implements the following risk mitigations on top of AccessControl:

- Only one account holds the DEFAULT_ADMIN_ROLE since deployment until it's potentially renounced.
- Enforces a 2-step process to transfer the DEFAULT_ADMIN_ROLE to another account.
- Enforces a configurable delay between the two steps, with the ability to cancel before the transfer is accepted.
- The delay can be changed by scheduling.
- It is not possible to use another role to manage the DEFAULT_ADMIN_ROLE.

Recommendations

It is recommended to use AccessControlDefaultAdminRules instead of the AccessControl library.

[L-06] The terminationTime assertion can be mutually exclusive

Description

The `terminationTime` application can be mutually exclusive in some instances. E.g. when comparing the `withdrawTokens` and `lock` functions, there is a single instance when both functions can be called at once, when `terminationTime` is equal `block.timestamp`. However, it must be stated that this is extremely unlikely that one function could front-run the other, yet it is still possible.

```
    function withdrawTokens(address _token) external override nonReentrant
{
        if (_token == token) {
            _withdraw();
        } else {
            if (locked) revert Locked();
            if (block.timestamp <= terminationTime) revert
NotTerminated();
            uint256 balance = IERC20(_token).balanceOf(address(this));
            if (balance == 0) revert BalanceIsZero();

            _withdrawDeposit(_token, balance);
        }
    }
```

```
    function lock(
        string calldata _disputeURI
    ) external payable override nonReentrant {
        if (locked) revert Locked();
        uint256 balance = IERC20(token).balanceOf(address(this));
        if (balance == 0) revert BalanceIsZero();
        if (block.timestamp >= terminationTime) revert Terminated();
...
```

Recommendations

It is recommended to review the applications of all `terminationTime` usages and unify them, so they are not mutually exclusive in any case.

[L-07] Released accounting not updated in dispute paths

Description

In `SmartInvoiceEscrow.sol`, in normal releases, the contract increments `released` by the gross amount paid. In dispute resolution flows, funds are distributed to client and provider, but released is not updated. However, the impact is only observability / reported value, not loss of funds.

Recommendations

After distributing awards in `resolve()` and `rule()`, set `released = total;` // or `released + balance` before distribution to reflect that all remaining escrowed funds were finalized.

[I-01] The verify can be repeated

Description

The `verify` function can be called any number of instances. In the on-chain realm there is no incentive to call it more than once. However, the emitted events may have an impact on the off-chain processing. Finally, similar functionality can be observed within `_autoVerify` and here the multiple calls are prevented.

```
    function verify() external override {
        if (msg.sender != client) revert NotClient(msg.sender);
        verified = true;
        emit Verified(client, address(this));
    }
```

```
    function _autoVerify() internal {
        if (!verified && msg.sender == client) {
            verified = true;
            emit Verified(client, address(this));
        }
    }
```

Recommendations

It is recommended to allow calling `verify` function only once.

[I-02] Redundant check within the `_autoVerify` function

Description

The `_autoVerify` function has a check whether `msg.sender` is a client. However, this check is redundant as a similar assertion is done in the parent functions: `_release`, `release` and `releaseTokens`.

```
    function _autoVerify() internal {
        if (!verified && msg.sender == client) {
            verified = true;
            emit Verified(client, address(this));
        }
    }
```

Recommendations

It is recommended to remove redundant assertions to save some Gas.

[I-03] Redundant calculation within the isFunded function

Description

The `isFunded` function has a redundant calculation as it includes the `released` amount in both `requiredAmount` and when adding it to the contract's balance.

```
    function isFunded(uint256 _milestoneId) external view returns (bool) {
        if (_milestoneId >= amounts.length) revert InvalidMilestone();

        uint256 requiredAmount = released;
        for (uint256 i = milestone; i <= _milestoneId; i++) {
            requiredAmount += amounts[i];
        }

        return
            IERC20(token).balanceOf(address(this)) + released >=
requiredAmount;
    }
```

Recommendations

It is recommended to simplify the calculations to reduce Gas usage.

[I-04] Zero-value milestones allowed can cause confusing state and event spam

Description

In `SmartInvoiceEscrow.sol`, the intended flow is milestone-based releases that reflect real payments. However, the implementation does not enforce `amounts[i] > 0` either during initialization or when adding milestones, allowing sequences of zero-value milestones.

This may produce noisy events and forces extra client calls to advance milestone counters without moving funds, which can confuse dashboards and off-chain accounting. This also requires user to set such meaningless milestones. Overall this might be unwanted design pattern.

Recommendations

Validate input amounts by requiring all milestone amounts to be greater than zero.