Daisy: Subscription Contracts Audit

1 Summary

ConsenSys Diligence conducted a security audit on the Daisy Subscription contract.

Daisy is a smart contract for managing paid subscriptions, including recurring payments and plan management.

- **Project Name:** Daisy: Subscription Contracts
- **Client Name:** Daisy
- **Lead Auditor:** Steve Marx
- **Co-auditors:** Shayan Eskandari
- **Date:** 2019-08-16

2 Audit Scope

This audit covered the following files:

File	SHA-1 hash
contracts/SubscriptionManager.sol	173cae20d294e09b7dfac922777331fc5dcf19f5
contracts/IndexedArrayLib.sol	ebf9ad40ab2c123da8844248cfdd3874ae8db173
contracts/Delegated.sol	a246efe078bade1b1a9761e7be2573ffd03edf48
contracts/interfaces/ISubscriptionManager.sol	0293d4cb558ec1e2e9811bdb6e4fedd80c957c64
contracts/interfaces/IEnumerableSubscriptionManager.sol	b992d2cd727bd4bcdcebff14675c57c9d0a63b4e

Frozen commit hash (develop branch): `8dcb96fd8a76464c5f3ec8577344ea687921293a`.

This is a follow up audit to the previous one that can be found here. There are design changes and additional functions that require a new audit.

The audit team evaluated that the system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three broad categories:

1. **Security:** Identifying security related issues within the contract.
2. **Architecture:** Evaluating the system architecture through the lens of established smart contract best practices.
3. **Code quality:** A full review of the contract source code. The primary areas of focus include:
    - Correctness
    - Readability
    - Scalability
    - Code complexity
    - Quality of test coverage

3 Key Observations/Recommendations

- We didn't find any vulnerabilities related to delegation, but we believe the mechanism is risky and recommend a safer approach in issue 4.2.
- The subscription contract is using `ownable` library which means the owner can be changed, intentionally or due to key compromise. The owner can change the fee recipient address, fee rates and subscription wallet address.
- For more please refer to the previous report.

4 Issues

Each issue has an assigned severity:

- **Minor** issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- **Medium** issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- **Major** issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

The following table contains all the issues discovered during the audit, ordered based on their severity.

Chapter	Issue Title	Issue Status	Severity
4.1	When replacing a subscription, credits from the previous subscription are lost	Closed	Medium
4.2	`Delegated.delegate()` allows calling any function with the smart contract as `msg.sender`	Closed	Medium
4.3	Consider removing `usesNonce`	Closed	Minor
4.4	Consider requiring existence in `SubscriptionManager.nextPaymentTimestamp()`	Closed	Minor
4.5	Fee recipient or amount can be changed via front running	Won't Fix	Minor
4.6	Optimization: avoid updating `Meta.credits` when it's already zero	Closed	Minor
4.7	Consider merging `Subscription` and `Meta`	Closed	Minor
4.8	Consider renaming `SubscriptionManager.MAX_FEE` and	Closed	Minor

	SubscriptionManager.fee		
4.9	Use IERC20 type where appropriate	Closed	Minor

4.1 When replacing a subscription, credits from the previous subscription are lost

Severity	Status	Remediation Comment
Medium	Closed	This is fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5 by adding `prev.credits` to the new subscription's credits.

Description

When a new subscription replaces an old one, the new subscription is credited for the amount of service unused on the previous subscription. Any *credits* remaining in the old subscription, however, are lost. They are not applied to the new subscription, and the previous subscription is deleted.

This could happen if, for example, a previous subscription started with 100 credits and consumed only 50 of them so far. The remaining 50 should be applied to the new subscription, but they are not.

Recommendation

Add the previous subscription's remaining credits to the new subscription in `SubscriptionManager._create()`:

```
credits = credits.add(prev.credits);
```

4.2 `Delegated.delegate()` allows calling any function with the smart contract as `msg.sender`

Severity	Status	Remediation Comment
Medium	Closed	This has been fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5 by using `delegatecall` instead of `call` to preserve the original `msg.sender`. Correspondingly, checks for delegation ignore `msg.sender` and exclusively rely on `_delegatedData`.

Description

`delegate()` is used to allow anyone to call a restricted function on the smart contract as long as they've obtained the right signature. The signature checking takes place in the target function, not in `delegate()` itself:

code/contracts/Delegated.sol:L55-L84

```
function delegate(
    bytes calldata data,
    bytes32 nonce,
    uint256 signatureExpiresAt,
    bytes calldata signature
)
    external
{
```

```
    // Reentrancy guard
    require(
        _delegatedData.signatureExpiresAt == 0,
        "Not possible to delegate more than once in the same transaction"
    );

    require(now < signatureExpiresAt, "Signature expired");

    // _delegatedData will exist during the delegated call
    _delegatedData = DelegatedData(nonce, signatureExpiresAt, signature);

    (bool success, bytes memory returnData) = address(this).call(data);

    // If the call failed, use assembly to propagate the error
    if (!success) {
        assembly {
            revert(add(0x20, returnData), returndatasize)
        }
    }

    delete _delegatedData;
}
```

A function expecting to be called this way uses the `_delegatedCheck()` function to determine whether a call contains a valid signature:

code/contracts/Delegated.sol:L86-L101

```
/**
 * @dev Internal function used by functions that support delegated
 * execution.
 * @param typeHash EIP712 type hash used for verifying the signature.
 * @param data Data to be used for the EIP712 hash (without the typeHash
 * and delegated data).
 * @param usesNonce True if a nonce is needed.
 * @return The address of the signer.
 */
function _delegatedCheck(
    bytes32 typeHash,
    bytes memory data,
    bool usesNonce
)
    internal
    returns(address caller)
```

Functions that *don't* expect to be called this way would presumably just use `msg.sender` to determine the caller, but `delegate()` effectively allows callers to spoof `msg.sender` to be the contract itself.

Although the audit team was unable to find a specific exploitable vulnerability, this pattern is concerning because it places a high burden on the development and auditing teams to make sure things are done correctly everywhere:

1. Each function called by `delegate()` must take care to call `_delegatedCheck()` rather than rely on `msg.sender`.
2. *Any* function can be called via `delegate()`, but only some functions are written to explicitly handle that.

Recommendation

A few recommendations, in descending order of preference:

1. In a previous iteration of the contract, delegation was handled in each function on a case-by-case basis. Thus functions had to "opt in" to delegation, and other functions could not be called with a misleading `msg.sender`. It may be safer to return to that delegation mechanism.
2. As an alternative, perhaps it's possible to add signature checking directly to `delegate()` so only calls with proper signatures will be forwarded at all.
3. Finally, `delegatecall` would be a way to preserve `msg.sender` during delegation. This method carries some general risks, but this is an option if the other recommendations aren't feasible.

4.3 Consider removing `usesNonce`

Severity	Status	Remediation Comment
Minor	Closed	This is fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5 by removing the `usesNonce` option and requiring a nonce everywhere.

Description

`Delegated._delegatedCheck()` accepts a boolean parameter `usesNonce` that determines whether a nonce is included in the data that is signed:

code/contracts/Delegated.sol:L112-L125

```
if (usesNonce) {
    d = abi.encodePacked(
        typeHash,
        data,
        _delegatedData.nonce,
        _delegatedData.signatureExpiresAt
    );
} else {
    d = abi.encodePacked(
        typeHash,
        data,
        _delegatedData.signatureExpiresAt
    );
}
```

Only `SubscriptionManager._delete()` and `SubscriptionManager.cancel()` choose `usesNonce == false`, and there's no clear benefit. In both cases, they already refuse to delete or cancel the same subscription for a second time.

Removing `usesNonce` might be a good way to simplify the code.

Recommendation

Remove `usesNonce` and instead always use a nonce.

4.4 Consider requiring existence in `SubscriptionManager.nextPaymentTimestamp()`

Severity	Status	Remediation Comment
Minor	Closed	This is fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5 as per the recommendation.

Description

`nextPaymentTimestamp()` just returns 0 for non-existent subscriptions:

code/contracts/SubscriptionManager.sol:L506-L517

```
function nextPaymentTimestamp(bytes32 subscriptionId)
    public
    view
    returns(uint256)
{
    Subscription storage sub = _subscriptions[subscriptionId];
    uint256 executions = _meta[subscriptionId].executions;

    // Periods from the start of the subscription for the next payment
    uint256 periodsFromStart = executions.mul(sub.periods);
    return _addPeriods(sub.createdAt, sub.periodUnit, periodsFromStart);
}
```

Reverting may be better so callers don't have to do this check themselves. Notably, `_create()` uses details of a previous subscription without explicitly checking for existence:

code/contracts/SubscriptionManager.sol:L848-L853

```
// If previous id was specified, we are replacing the subscription
if (previousSubscriptionId != 0) {
    Subscription memory prev = _subscriptions[previousSubscriptionId];
    uint256 nextPayment = nextPaymentTimestamp(
        previousSubscriptionId
    );
```

This particular example seems harmless, but reverting in the case of a non-existent subscription would remove all doubt.

Recommendation

Add an explicit check to `nextPaymentTimestamp()`:

```
require(_subscriptionExists(subscriptionId), "Subscription doesn't exist");
```

4.5 Fee recipient or amount can be changed via front running

Severity	Status	Remediation Comment
Minor	Won't Fix	Per the recommendation, the client team has decided not to make any change to address this.

Description

Transaction relayers are incentivized by a fee system. At any point in time, the contract's state dictates what address will receive a fee and what that fee will be (as a fraction of the amount transacted). Both the recipient and amount can be adjusted at any time by calling `SubscriptionManager.setFee()` :

code/contracts/SubscriptionManager.sol:L276-L280

```
function setFee(
    address _feeRecipient,
    uint256 _fee
)
    external
```

This makes the following front-running sequence possible:

1. Relayer checks the contract to make sure the expected fee will be paid to them.
2. Relayer submits a transaction.
3. The contract owner changes the fee recipient or amount with a higher gas price.
4. The relayer's transaction is processed, but no fee is paid to them.

Recommendation

Likely no change is required here. This sort of attack can only be done by the contract owner and presumably only once (because it's readily observed by the relayer). Relayers can simply refuse to relay future transactions to that contract.

If a change is desired, perhaps a two-step process with a delay can be required on fee changes so relayers have a chance to see the change coming and stop relaying transactions before it takes effect.

4.6 Optimization: avoid updating `Meta.credits` when it's already zero

Severity	Status	Remediation Comment
Minor	Closed	This is fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5 as per the recommendation.

Description

In `SubscriptionManager.execute()` , the following code checks for available credits, applies them, and updates the number of credits remaining:

code/contracts/SubscriptionManager.sol:L670-L675

```
uint256 credits = meta.credits;

// If subscription has credits, substract from price
if (credits < price) {
```

```
    price = price - credits;
    meta.credits = 0;
```

In the typical case, where there are no credits to apply, this code does an unnecessary write to storage. For gas efficiency, it would be better to skip that write.

Recommendation

Consider only updating `price` and `meta.credits` when there's a credit to apply:

```
if (credits < price) {
    if (credits > 0) {
        price = price - credits;
        meta.credits = 0;
    }
    ...
```

4.7 Consider merging `Subscription` and `Meta`

Severity	Status	Remediation Comment
Minor	Closed	This is fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5, where the two structs have been merged into one (`Subscription`).

Description

In `SubscriptionManager`, every subscription consists of both a `Subscription` struct and a `Meta` struct.

code/contracts/SubscriptionManager.sol:L34-L60

```
// Immutable subscription data
struct Subscription {
    // Address of the subscriber
    address subscriber;
    // Address of the token
    address token;
    // Number of tokens to transfer
    uint256 price;
    // Unit used for billing periods
    PeriodUnit periodUnit;
    // Number of PeriodUnits between payments
    uint256 periods;
    // Number of times that it can be executed (0 means no limit)
    uint256 maxExecutions;
    // Plan on chain id
    bytes32 planIdHash;
    // Timestamp (seconds)
    uint256 createdAt;
}

// Additional data that changes over the lifetime of a subscription.
```

```
struct Meta {
    // Tokens to be substracted from next execution's price
    uint256 credits;
    // Number of times it has been executed
    uint256 executions;
}
```

Although this helps keep the mutable state of the subscription separate from the immutable state, it leads to complications in the code and the potential for mistakes.

Examples

`SubscriptionManager.getSubscription()` exists solely to merge the two structs together. Otherwise the Solidity-generated getter would be sufficient:

code/contracts/SubscriptionManager.sol:L365-L396

```
function getSubscription(bytes32 subscriptionId)
    external
    view
    returns(
        address subscriber,
        address token,
        uint256 price,
        PeriodUnit periodUnit,
        uint256 periods,
        uint256 maxExecutions,
        bytes32 planIdHash,
        uint256 createdAt,
        uint256 credits,
        uint256 executions
    )
{
    Subscription storage sub = _subscriptions[subscriptionId];
    Meta storage meta = _meta[subscriptionId];

    return (
        sub.subscriber,
        sub.token,
        sub.price,
        sub.periodUnit,
        sub.periods,
        sub.maxExecutions,
        sub.planIdHash,
        sub.createdAt,
        meta.credits,
        meta.executions
    );
}
```

In `SubscriptionManager._delete()`, care must be taken to delete the subscription from both mappings:

code/contracts/SubscriptionManager.sol:L768-L770

```
// Set all subscription fields to 0
delete _subscriptions[subscriptionId];
delete _meta[subscriptionId];
```

Recommendation

Merge the fields from `Meta` into `Subscription` itself and use that struct exclusively.

4.8 Consider renaming `SubscriptionManager.MAX_FEE` and `SubscriptionManager.fee`

Severity	Status	Remediation Comment
Minor	Closed	This has been fixed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5, where the two state variables have been renamed to `feeNumerator` and `FEE_DENOMINATOR`.

Description

Although `MAX_FEE` *is* the maximum allowed value for `fee`, it's not the primary purpose of that variable, which is to serve as the denominator for `fee`, as seen here:

code/contracts/SubscriptionManager.sol:L706

```
uint256 feeAmount = price.mul(fee) / MAX_FEE;
```

`MAX_FEE` is a maximum only in that a fee greater than 100% is not allowed.

Similarly, `fee` is not a fee but rather the numerator of the fee fraction.

Recommendation

Consider using `feeNumerator` and `feeDenominator` or the like to better describe what these variables represent.

4.9 Use `IERC20` type where appropriate

Severity	Status	Remediation Comment
Minor	Closed	This has been addressed in https://github.com/ConsenSys/daisy-subscription-contracts/pull/5, where the stricter type `IERC20` is used in most places.

Description

In a couple places in the code, the type `address` is being used for a value that will eventually be cast to the type `IERC20`. To simplify the code and make slightly better use of Solidity's type system, it's better to just use the `IERC20` type from the start.

Examples

code/contracts/SubscriptionManager.sol:L35-L39

```
struct Subscription {
    // Address of the subscriber
    address subscriber;
    // Address of the token
    address token;
```

code/contracts/SubscriptionManager.sol:L335-L336

```
function withdraw(
    address[] calldata tokens,
```

Remediation

Use the specific variable/parameter type `IERC20` wherever possible.

Appendix 1 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") -- on its GitHub account (https://github.com/ConsenSys). CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may

use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.