MetaMask USD Token

Date	August 2025
Auditors	George Kobakhidze, Heiko Fisch

1 Executive Summary

This report presents the results of our engagement with **MetaMask** to review the contract for the **MetaMask USD (mUSD)** token, implemented by **M0**.

The review was conducted in the week of **August 11–15, 2025**, by **Heiko Fisch** and **George Kobakhidze**.

The focus of this security audit is the `mUSD` token contract, designed for integration with MetaMask's ecosystem. The `mUSD` token is built upon the M0 framework, which leverages real-world assets (RWA) to mint a rebasing `M` token on-chain, with subsequent yield from RWAs distributed through `M` token rebasing mechanisms.

The M0 framework wraps the rebasing `M` token into non-rebasing `MExtension` tokens that feature customizable configurations for yield distribution. The `mUSD` token represents one such implementation, utilizing the `MYieldToOne` extension to direct yield to a designated recipient. The broader M0 framework and `MExtension` system architecture is outside the audit scope, and the focus is specifically on the `mUSD` token contract implementation.

Beyond the standard token and M0 framework functionality, the `mUSD` token incorporates additional administrative controls, including pausability mechanisms, account freezing capabilities, and forced transfer functionality for frozen assets. The audit scope specifically covered the administrative management systems governing contract pausing behavior, along with two specialized functions enabling the movement of `mUSD` tokens from frozen accounts, both individually and through batch operations.

The security assessment of the `mUSD` token contract revealed no significant security vulnerabilities. The codebase demonstrated high quality standards with clean organization, comprehensive documentation, and logical structure. Several minor and informational findings were identified, including refinements to approval management during paused states and small code quality improvements.

It should be noted that `mUSD` has privileged roles that can freeze and seize assets at any time. Moreover, the contract is upgradeable, meaning its logic can be changed by the proxy admin owner. Extremely high standards of operational security are required for such powerful roles.

After the report delivery, fixes or acknowledgements were provided for each issue found. Individual commits and PRs are reflected in the resolutions of the associated issues. As requested by the M0 team, the final commit for revision was `b5594cd5c8a0b27534bf20f09ff08fd74bb18f7f` which contained two changes on top of fixes:

- "METAMASK USD" ASCII art
- changes to the usage of `_revertIfInsufficientBalance()`.

2 Scope

This review focused on the following repository and hash:

- M0 mUSD token contract repository at b62fab7c3e867b700bd81dad2ab140e074d98f32

Following the fix review, the last commit of the mUSD repository that we reviewed was `b5594cd5c8a0b27534bf20f09ff08fd74bb18f7f`.

The detailed list of files in scope can be found in the Appendix.

3 System Overview

The `mUSD` token operates within the broader M0 ecosystem as a wrapped version of the underlying `M` token. While the complete M0 framework is outside the scope of this review, understanding the core mechanics of `mUSD` token creation and management is essential for evaluating its security properties.

The `mUSD` token lifecycle begins when approved users interact with the `SwapFacility` contract. These users deposit `M` tokens into the `SwapFacility`, which then calls the `mUSD` contract to mint an equivalent amount of `mUSD` tokens for the recipient. The process is fully reversible - users can burn their `mUSD` tokens through the same `SwapFacility` to retrieve their original `M` tokens, completing the unwrapping process.

Beyond this core wrapping and unwrapping functionality, the `mUSD` contract implements standard ERC-20 token operations, including transfers, approvals, and delegated transfers through `transferFrom`.

However, `mUSD` extends beyond typical token functionality with several administrative controls:

- **Pause Functionality**: The pause mechanism provides an emergency brake that halts transfers, wrapping, and unwrapping. Approvals are halted too, but we discuss this in finding 5.1. Forced transfers, yield claiming, freezing and unfreezing, as well as not directly token-related operations like role management, are exempt from the pause. When the contract is paused, this effectively freezes all "regular" `mUSD` activity across the network.
- **Account Freezing**: Individual accounts can be frozen, preventing them from sending and receiving funds, as well as initiating any token transfers. This granular control allows administrators to isolate potentially compromised or non-compliant accounts without affecting the broader system operation. Yield can still be claimed to the yield recipient even if that account is frozen; M0 has confirmed that this is a deliberate exception.
- **Forced Transfers**: This administrative function enables the recovery of `mUSD` tokens from frozen accounts.

4 Security Specification

4.1 Actors

The relevant actors are listed below with their respective abilities:

- **Default Admin** - Able to change all the privileged roles in the system.
- **Proxy Admin Owner** - The `mUSD` token contract is upgradeable, and the Proxy Admin Owner is the address that can perform such an upgrade.
- **M0 Operational Administrators** - Able to modify the greater M0 framework properties that are defined outside the `mUSD` token, such as who is an approved swapper and what extensions can be swapped between each other.
- **Approved swappers** - Able to swap (or "wrap"/"unwrap") the `mUSD` tokens into `M` tokens and vice-versa, reducing and increasing the supply of `mUSD` tokens respectively.
- **Yield Recipient** - Receives any yield generated from `M` token rebasing.
- **Yield Recipient Manager** - Able to modify who receives the yield.
- **Pause Manager** - Able to set the contract in the pause state, which stops `mUSD` wrapping, unwrapping, transfers, and even approvals.
- **Freeze Manager** - Can freeze and unfreeze accounts, disabling their ability to interact with the `mUSD` token contract.
- **Forced Transfer Manager** - Able to move out `mUSD` tokens from frozen accounts.

4.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this review, we established the following trust model:

- **Default Admin** - We trust this entity fully, as they essentially are able to change all other privileged functionality by modifying all privileged roles and assigned addresses.
- **Proxy Admin Owner** - We trust this entity entirely as they can change the contracts as they want. The ability to upgrade the token contract gives the Proxy Admin Owner complete control over the code as well as the data stored in the contract.
- **M0 Operational Administrators** - We trust them to diligently perform any configuration changes as they could affect the economics of the `mUSD` token, such as if an unauthorized entity is able to swap the tokens.
- **Approved Swappers** - They are privileged to swap `mUSD` tokens as they'd get access to the `M` token liquidity, but are not particularly trusted to do anything special within the `mUSD` context alone. Them acting rationally, such as buying `mUSD` at a discount to swap into M, does support the economics of `mUSD`.
- **Yield Recipient** - They are likely to be an entity trusted by the owners of the `mUSD` token to then transfer the yield to them.
- **Yield Recipient Manager** - They are trusted not to misappropriate the yield by changing who receives it.
- **Pause Manager** - If the pause managers are compromised, the pauses could affect economic activity around the `mUSD` token by stopping all token operations.
- **Freeze Manager** - We trust them to freeze and unfreeze diligently, as a compromised freeze manager could even freeze DeFi contracts holding `mUSD`.
- **Forced transfer manager** - They are trusted not to abuse this capability to take control of funds without proper authorization.

4.3 Other Security-Related Aspects

- RWA. RWA assets are what's behind the value of `mUSD` tokens, so the value, and therefore some risk, is offchain. The onchain economics of `mUSD`, such as the users' willingness to hold and trade it at stable prices, depend on the state of RWA assets and access to them.
- Operational difficulties of `mUSD`. The greater M0 framework around `mUSD` requires diligence and care for the system to function correctly. For example, the `SwapFacility` contract needs to have appropriate flags set on it, like approved swappers, and the other `MExtension` tokens, like `mUSD`, affect the underlying `M` token, which is where the yield comes from. All moving pieces of the greater M0 system need to be managed carefully for `mUSD` to work.

5 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- 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.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- 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.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

5.1 Consider Allowing `approve` Even When Contract Is Paused Minor ✓ Fixed

Resolution
Fixed in commit abe4331467aaee24c64670e05f96f198b0bf6036 by removing the `_beforeApprove()` hook from the `MUSD` contract. Note: The behavior of `evm-m-extensions` discussed in the "Remark" section below remains unchanged, i.e., (1) accounts – even if unfrozen – can't revoke approvals for frozen accounts, and (2) frozen accounts can't revoke approvals at all. Since `evm-m-extensions` is out of scope for this review, we set the finding to "Fixed", but the "Remark" part is really only "Acknowledged".

Description

The `MUSD` contract is pausable. More specifically, it inherits from OpenZeppelin's `PausableUpgradeable` and calls `_requireNotPaused();` in `_beforeApprove`, `_beforeWrap`, `_beforeUnwrap`, and `_beforeTransfer`. This means that allowance changes, wrapping and unwrapping, as well as transfers are not possible when the contract is paused.

For example, `_beforeApprove` looks as follows:

src/MUSD.sol:L101-L111

```
/**
 * @dev   Hook called before approval of mUSD.
 * @param account The sender's address.
 * @param spender The spender's address.
 * @param amount  The amount to be approved.
 */
function _beforeApprove(address account, address spender, uint256 amount) internal view override {
    _requireNotPaused();

    super._beforeApprove(account, spender, amount);
}
```

As a side note, forced transfers, claiming yield (which entirely goes to a dedicated account), freezing, and unfreezing do not respect the paused status, since these actions are only possible from privileged addresses. M0 has confirmed that this is intentional.

We'd like to call into question whether changing the allowance – and in particular decreasing it or revoking an approval entirely – should be affected by the paused status of the contract: First of all, in OpenZeppelin's pausable ERC-20 contract, `approve` calls are exempt from the pause, so `MUSD`'s behavior would be different from a standard OZ `ERC20Pausable` or `ERC20PausableUpgradeable`. Second, the ability to revoke approvals even when the contract is paused might be beneficial from a security perspective. It is conceivable, for instance, that the contract was paused because of an ongoing hack; in this case users could revoke their approvals during a pause and, thereby, secure their funds. Or even without a hack, a user might just want to be on the safe side and revoke an approval regardless of whether the contract is paused or not.

In conclusion, we believe it is better to allow users to revoke approvals even when the contract is paused (or in other exceptional situations; see the remark below).

Recommendation

As discussed above, there are valid reasons why it might make sense to allow `approve` calls – in general or at least those that reduce the allowance or set it to zero – even when the contract is paused. There are two ways to address this:

1. *Allow `approve` calls also when the contract is paused.* This is the same behavior as a standard OpenZeppelin pausable ERC-20 contract and has the security benefits mentioned. It is also very easy to implement, as `_beforeApprove` can just be removed from `MUSD`. The downside is that increasing the allowance is also possible, which has no security benefit – but we don't see a problem with this either.
2. *Only allow `approve` calls that reduce the allowance or maybe even only calls that set it to zero.* This would bring the discussed security benefits, without unnecessarily permitting an allowance increase. The downside is that it is more difficult to implement and that this behavior is different from the well-known OpenZeppelin codebase's.

Remark

There is a similar situation regarding the `_beforeApproved` hook in `MYieldToOne`: that it can be undesirable to be unable to revoke or reduce approvals in some situations, even if a frozen account is involved. For instance, assume a contract C is deemed malicious and, therefore, frozen. Now even legit (i.e., unfrozen) users can't revoke their approval for that contract. Of course, it seems likely that C never gets unfrozen (so no transfers to or initiated by C could ever happen) and users have to trust the `FREEZE_MANAGER_ROLE` *anyway*, but it's still unfortunate if users can't revoke approvals at will. Similarly, there are conceivable (if a bit contrived) scenarios, where it would be important for an erroneously frozen – and later to be unfrozen – account to be able to revoke approvals even while the account is still frozen.

Since this is technically out of scope for the current review, we're not discussing this topic extensively in this report. We have, however, discussed it with M0 separately, as this behavior is inherited by – and therefore relevant for – `mUSD`.

5.2 Consider Making Forced Transfers Revert for Zero Amount Minor Acknowledged

Resolution

M0 acknowledged this with the following comment:

> Since the ERC20's `Transfer` event must be emitted even if the amount is `0`, we will keep the same behavior for the `ForcedTransfer` event.

Description

The `MUSD` contract introduces forced transfers, which allow an address with the `FORCED_TRANSFER_MANAGER_ROLE` to move funds from a frozen account to a different address. Assuming all other conditions are met, forced transfers with amount zero succeed:

src/MUSD.sol:L166-L178

```
function _forceTransfer(address frozenAccount, address recipient, uint256 amount) internal {
    _revertIfInvalidRecipient(recipient);
    _revertIfNotFrozen(frozenAccount);

    emit Transfer(frozenAccount, recipient, amount);
    emit ForcedTransfer(frozenAccount, recipient, msg.sender, amount);

    if (amount == 0) return;

    _revertIfInsufficientBalance(frozenAccount, balanceOf(frozenAccount), amount);

    _update(frozenAccount, recipient, amount);
}
```

This behavior mimics regular transfers, which also succeed for amount zero – which is, in fact, mandated by the ERC-20 specification:

> *Note* Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

However, it is debatable whether forced transfers should indeed behave in the same way regarding zero-amount transfers: Unlike regular transfers, forced transfers move someone else's funds without permission and should, therefore, be considered exceptional occurrences. Initiating a forced transfer without effect is most likely a mistake in the first place, and reverting might be considered beneficial in such a scenario.

Recommendation

Consider making forced transfers with amount zero revert, even if all other conditions for a forced transfer are met.

5.3 Version Mismatch for `evm-m-extensions` in `mUSD` Minor Acknowledged

Resolution

Deployment to mainnet addressed as per M0's comment:

> We used commit hash `50de1be2d971131c09a417cc5e359ad60de1b84b` when deploying to Mainnet.
>
> That being said, no change impacted mUSD between `90f144deee35071e02b1ff0b62004b8d2435ddfe` and `50de1be2d971131c09a417cc5e359ad60de1b84b`.

Note: The version of `evm-m-extensions` that is currently used in `mUSD` (where "currently" means precisely: in revision b5594cd5c8a0b27534bf20f09ff08fd74bb18f7f, which is the final version we considered in this review - but not the version that was originally deployed) is 909c536deac54de5a5bc3305f57e310c327fb441 and, therefore, one commit ahead of 50de1be2d971131c09a417cc5e359ad60de1b84b.

Description

`MUSD` inherits extensively from contracts in the `evm-m-extensions` repository. While this code itself was not in scope for this audit, we had to check some of it for compatibility with `MUSD` and were asked to consider revision `011f84f0f6a701a9796fcac1ad29896c60b65344` for this.

It should be noted, however, that the submodule that is actually used in the `mUSD` repository is at revision `90f144deee35071e02b1ff0b62004b8d2435ddfe` and therefore behind the commit hash we were given.

Recommendation

Since the `evm-m-extensions` are not in scope for this audit, we cannot give a recommendation as to which version to use. From the limited perspective of this review, the revision that is currently utilized in `mUSD` at the relevant commit hash is behind the one we were told was going to be used for `mUSD`.

5.4 ERC-20 Token Name Not Correct ✓ Fixed

Resolution

Fixed in commit 6afec02b0ee166ec26c91412afa68d75a49a89e7 by changing the name to "MetaMask USD" from "MUSD" before deploying to Mainnet.

Description

According to the MetaMask team, the token name should be "MetaMask USD". However, the ERC-20 token name – which is hard-coded in the source file – is "MUSD":

src/MUSD.sol:L56

```
__MYieldToOne_init("MUSD", "mUSD", yieldRecipient, admin, freezeManager, yieldRecipientManager);
```

Recommendation

In the line above, change `MUSD` to `MetaMask USD`.

5.5 Cumulative Hooks Should Always Call Their `super`, Even if Currently Empty Acknowledged

Resolution

Acknowledged by M0 with the following comment:

> We will consider including it in future token extension models if the contract hierarchy extends beyond one level or once we add logic to any base contracts.

Description

The `_beforeClaimYield` hook in `MUSD` is implemented as follows:

src/MUSD.sol:L148-L152

```
/**
 * @dev    Hook called before claiming yield.
 * @notice MUST only be callable by the YIELD_RECIPIENT_MANAGER_ROLE.
 */
function _beforeClaimYield() internal view override onlyRole(YIELD_RECIPIENT_MANAGER_ROLE) {}
```

This hook ensures that the caller has the `YIELD_RECIPIENT_MANAGER_ROLE`, but it doesn't call `super._beforeClaimYield();`. Specifically, this means that `MYieldToOne._beforeClaimYield` won't be called when yield is claimed on `MUSD`. Crucially, as this function has an empty body and no modifiers, this won't make a behavioral difference.

Nevertheless, we think it is good style for simple, "cumulative" hooks (i.e., hooks that add conditions that must be fulfilled, and revert if any of these must not) to always call the same hook on `super`, even if it is currently empty. While any code change should be thoroughly reviewed, this pattern is generally quite robust against changes in the code such as adding a condition to a previously empty hook or changes in the inheritance relationships.

Recommendation

We recommend adding `super._beforeClaimYield();` to the body of `MUSD._beforeClaimYield`. In general, for cumulative hooks, we believe it makes sense to always call the same hook on `super` – even if it is currently empty.

Remark

This pattern of not calling the same hook on `super` if it's empty is also present in out-of-scope code. For example, in `MYieldToOne`, the hooks `_beforeApprove`, `_beforeWrap`, `_beforeUnwrap`, and `_beforeTransfer` don't call their corresponding hook on super. Our general recommendation would be to change this consistently throughout the entire codebase, but we don't recommend doing this before a deployment/upgrade and would rather wait with this until the next audit of the affected code.

5.6 Several Public Functions Could Be External ✓ Fixed

Resolution

Fixed in commit f3454f8fec6e327d34cc1aaf96db7ee213ad4d72 and commit 964bee40a1874c51f6bde09981d7af7628cf987b by changing the visibility on the `initialize()` and `pause()` / `unpause()` functions respectively.

Description

The `initialize`, `pause`, and `unpause` functions are `public`:

src/MUSD.sol:L45-L52

```
function initialize(
    address yieldRecipient,
    address admin,
    address freezeManager,
    address yieldRecipientManager,
    address pauser,
    address forcedTransferManager
) public virtual initializer {
```

src/MUSD.sol:L66

```
function pause() public onlyRole(PAUSER_ROLE) {
```

src/MUSD.sol:L71

```
function unpause() public onlyRole(PAUSER_ROLE) {
```

However, none of them is called internally. As it seems unlikely that `MUSD` is meant to be inherited from (so they could be called internally in a derived contract), it would be more accurate to make these functions `external`.

Recommendation

Consider making the `initialize`, `pause`, and `unpause` functions `external` instead of `public`.

5.7 `initialize` Function Should Probably Not Be Virtual ✓ Fixed

Resolution

Fixed in commit f3454f8fec6e327d34cc1aaf96db7ee213ad4d72 by removing `virtual` from `initialize()`.

Description

There is only one virtual function in `MUSD`, and it is `initialize`.

src/MUSD.sol:L45-L52

```
function initialize(
    address yieldRecipient,
    address admin,
    address freezeManager,
    address yieldRecipientManager,
    address pauser,
    address forcedTransferManager
) public virtual initializer {
```

Especially the fact that functions from the `_before*` family – which are frequently overridden in derived contracts – are not virtual suggests that `MUSD` is not intended to be inherited from. In that case, it makes sense to remove the `virtual` keyword from `initialize` in order to avoid a contradictory impression.

Recommendation

If `MUSD` is not intended to be inherited from, the `virtual` keyword can be removed from `initialize`, so no function in the contract is virtual. Otherwise, it seems that more functions should be virtual, in particular the `_before*` functions.

5.8 Inconsistent Order of Functions in Interface and Contract ✓ Fixed

Resolution

Fixed in commit 5131ea952adea0fa506a8e4d5ffe587142f27d7f by moving `forceTransfer` before `forceTransfers`.

Description

Ideally, the function declarations in the interface and the corresponding function definitions in the contract are in the same order, just for the sake of consistency and readability. In `MUSD`, `forceTransfer` (single force transfer) occurs before `forceTransfers` (multiple force transfers), which most would probably say is also the more natural order:

src/IMUSD.sol:L51-L71

```
/**
 * @notice Forcefully transfers tokens from frozen accounts to recipients.
 * @dev    Can only be called by an account with the FORCED_TRANSFER_MANAGER_ROLE.
 * @param  frozenAccounts The addresses of the frozen accounts.
 * @param  recipients The addresses of the recipients.
 * @param  amounts The amounts of tokens to transfer.
 */
function forceTransfers(
    address[] calldata frozenAccounts,
    address[] calldata recipients,
    uint256[] calldata amounts
) external;

/**
 * @notice Forcefully transfers tokens from a frozen account to a recipient.
 * @dev    Can only be called by an account with the FORCED_TRANSFER_MANAGER_ROLE.
 * @param  frozenAccount The address of the frozen account.
 * @param  recipient The address of the recipient.
 * @param  amount The amount of tokens to transfer.
 */
function forceTransfer(address frozenAccount, address recipient, uint256 amount) external;
```

In `IMUSD`, however, `forceTransfers` occurs before `forceTransfer`:

src/MUSD.sol:L75-L97

```
/// @inheritdoc IMUSD
function forceTransfer(
    address frozenAccount,
    address recipient,
    uint256 amount
) external onlyRole(FORCED_TRANSFER_MANAGER_ROLE) {
    _forceTransfer(frozenAccount, recipient, amount);
}

/// @inheritdoc IMUSD
function forceTransfers(
    address[] calldata frozenAccounts,
    address[] calldata recipients,
    uint256[] calldata amounts
) external onlyRole(FORCED_TRANSFER_MANAGER_ROLE) {
    if (frozenAccounts.length != recipients.length || frozenAccounts.length != amounts.length) {
        revert ArrayLengthMismatch();
    }

    for (uint256 i; i < frozenAccounts.length; ++i) {
        _forceTransfer(frozenAccounts[i], recipients[i], amounts[i]);
    }
}
```

Recommendation

In `IMUSD`, consider moving `forceTransfer` before `forceTransfers`.

Appendix 1 - Files in Scope

This review covered the following files:

Initial version at revision b62fab7c3e867b700bd81dad2ab140e074d98f32:

File	SHA-1 hash
src/IMUSD.sol	`83cd624256e79a09e907d1e6950a3a109fad0121`
src/MUSD.sol	`adebf886b3e63586837b78847037b43d643ace19`

Final version at revision b5594cd5c8a0b27534bf20f09ff08fd74bb18f7:

File	SHA-1 hash
src/IMUSD.sol	`17f2a5d26814638b584dd985afe36edbf8cdc312`
src/MUSD.sol	`535981657d98a46681cc3403a9874d0b1e379546`

Appendix 2 - Document Changelog

Version	Date	Description
1.0	2025-08-15	Initial report
2.0	2025-08-21	Reviewed fixes, added client comments, and extended “Executive Summary” and “Scope” sections accordingly

Appendix 3 - 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 bug-free 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.

A.3.1 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 code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as 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 specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. 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.

A.3.2 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.

A.3.3 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.