Report

Smart Contract Audit

Peccala Group s.r.o. - 15.01.2022 - Version v1.0

turingpoint.

Changelog

1.0	14.01.2022	Final Version
0.9	14.01.2022	QA
0.2	13.01.2022	Vulnerabilities added
0.1	13.01.2022	Initial Draft

tp.

turingpoint GmbH
Rödingsmarkt 9
20459 Hamburg
Deutschland

Tel.: +49 40 52477883
Mail: hello@turingpoint.de
Web: turingpoint.de

Handelsregister: 157676
USt-IdNr: DE325011218

Table of Contents

1 Dashboard

1.1 Metadata

Analyst	Jan Kahmen
Interviewpartner	Laura Arcade
Type	Smart Contract Audit
Method	White-Box
Timeframe	11.01.2022 - 14.01.2022

1.2 Targets

Smart Contract Audit	PeccalaUser.sol#4723f4e5be5180ba3bd88f6acc#sha256sum
Smart Contract Audit	PeccalaHigh.sol#925f91ff701b2be187dcd3123d#sha256sum

1.3 Risiks

Critical Risk	0
High Risik	0
Medium Risik	0
Low Risik	3
Info	0
Total	**3**

1.4 Vulnerabilities (Top 5)

Read of persistent state following external call	Low
A floating pragma is set	Low
Write of persistent state following external call	Low

2 Management summary

2.1 Scope and Limitations

The objective of this review was to identify and assess any safety deficiencies in the application. Due to the high complexity of the smart contracts, a test period of 4 days was set.

2.2 Vulnerabilities

The security audit identified 3 vulnerabilities of low. No vulnerabilities in the critical, medium or high risk categories could be identified. To further increase the security level of the smart contract code, it is recommended to implement the proposed measures.

2.3 Strategic Recommendations

In the short term, all hardening measures described should be implemented to ensure the system security of the tested code. In addition, security checks should already be implemented in the development process in order to be able to detect vulnerabilities at an early stage. Furthermore, it is very advisable to have the SWC Registry (Smart Contract Weakness Classification and Test Cases) in view.

3 Objective

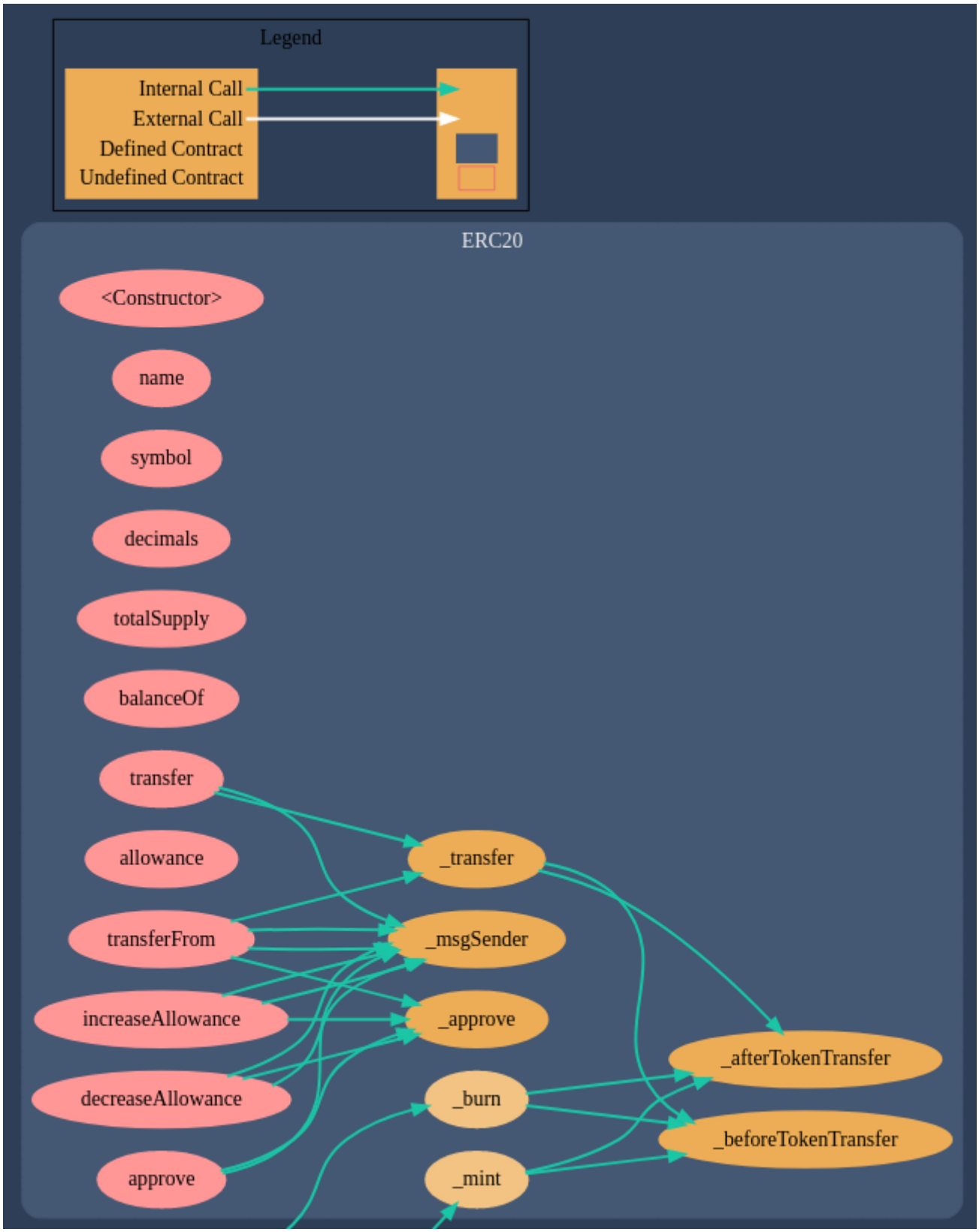
On behalf of **Peccala Group s.r.o.**, **turingpoint GmbH** conducted a security audit of the smart contracts in january 2022. The aim of this audit was to determine the security level of the applications used, as well as to identify existing vulnerabilities and document measures that could eliminate vulnerabilities.

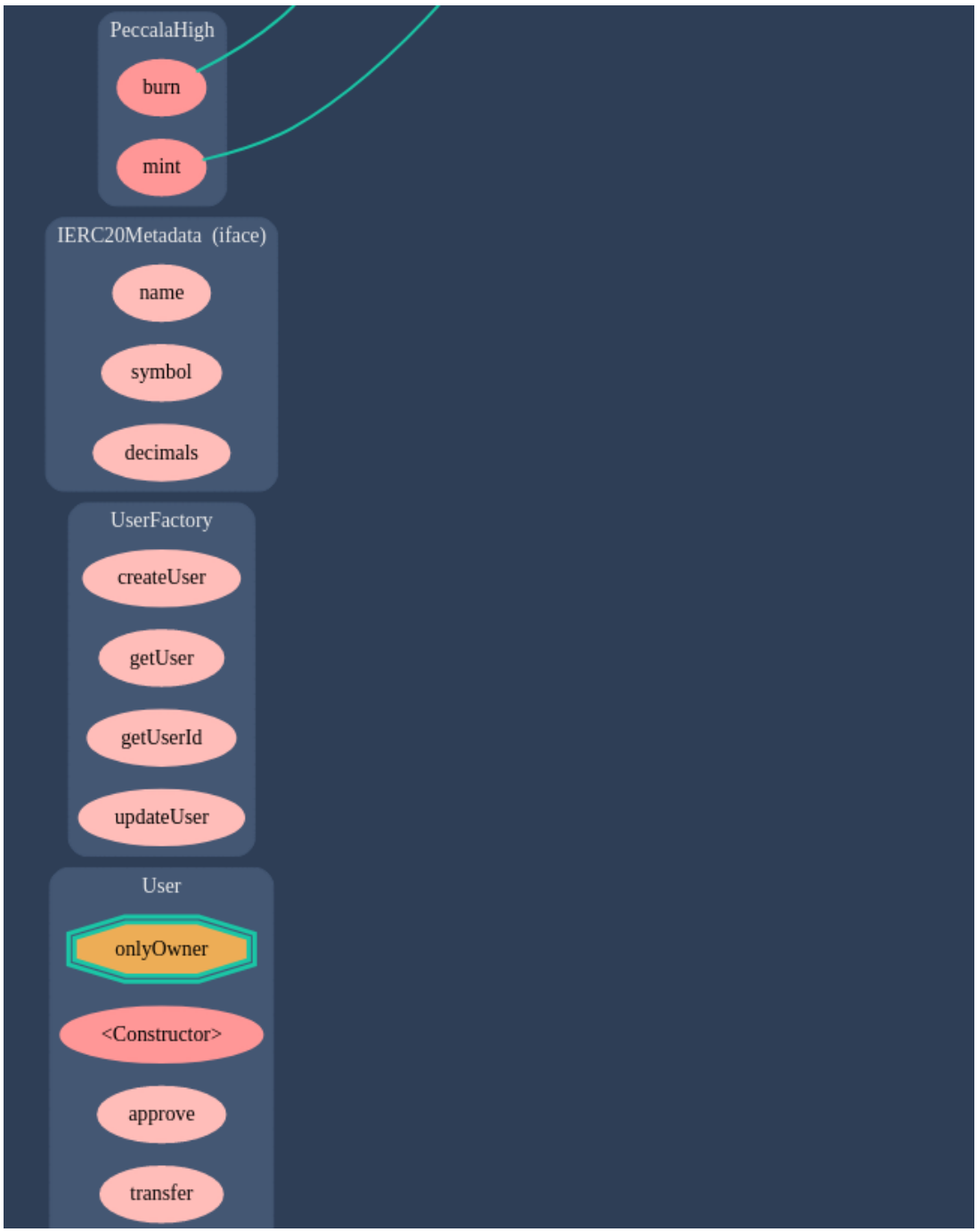
3.1 Delimitation

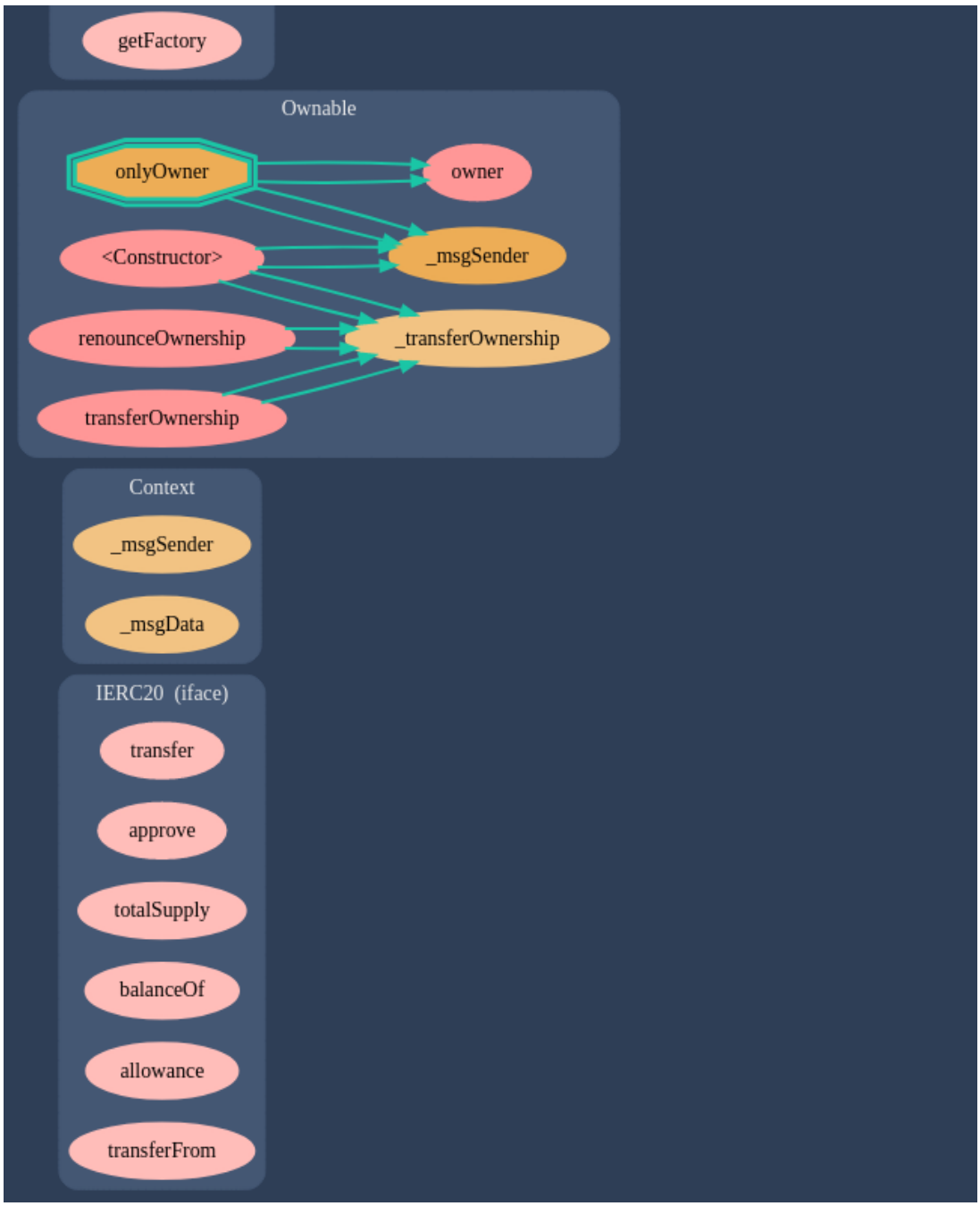
A technical security analysis is naturally a random analysis in which an attempt is made to find as many vulnerabilities as possible within a test object with a finite amount of effort. Normally only a part of the system properties found can be clearly classified as vulnerabilities, the other part requires more extensive testing. This involves investigating to what extent the identified properties of a system are functions that were implemented as a requirement and to what extent this implementation created vulnerabilities that could not be created by an alternative implementation. Due to the number of possible combinations that arise here, a balance must be struck between the number of properties to be evaluated and the depth of the examination to be performed, within the available time quota.

3.2 Graphical Representation of Logic

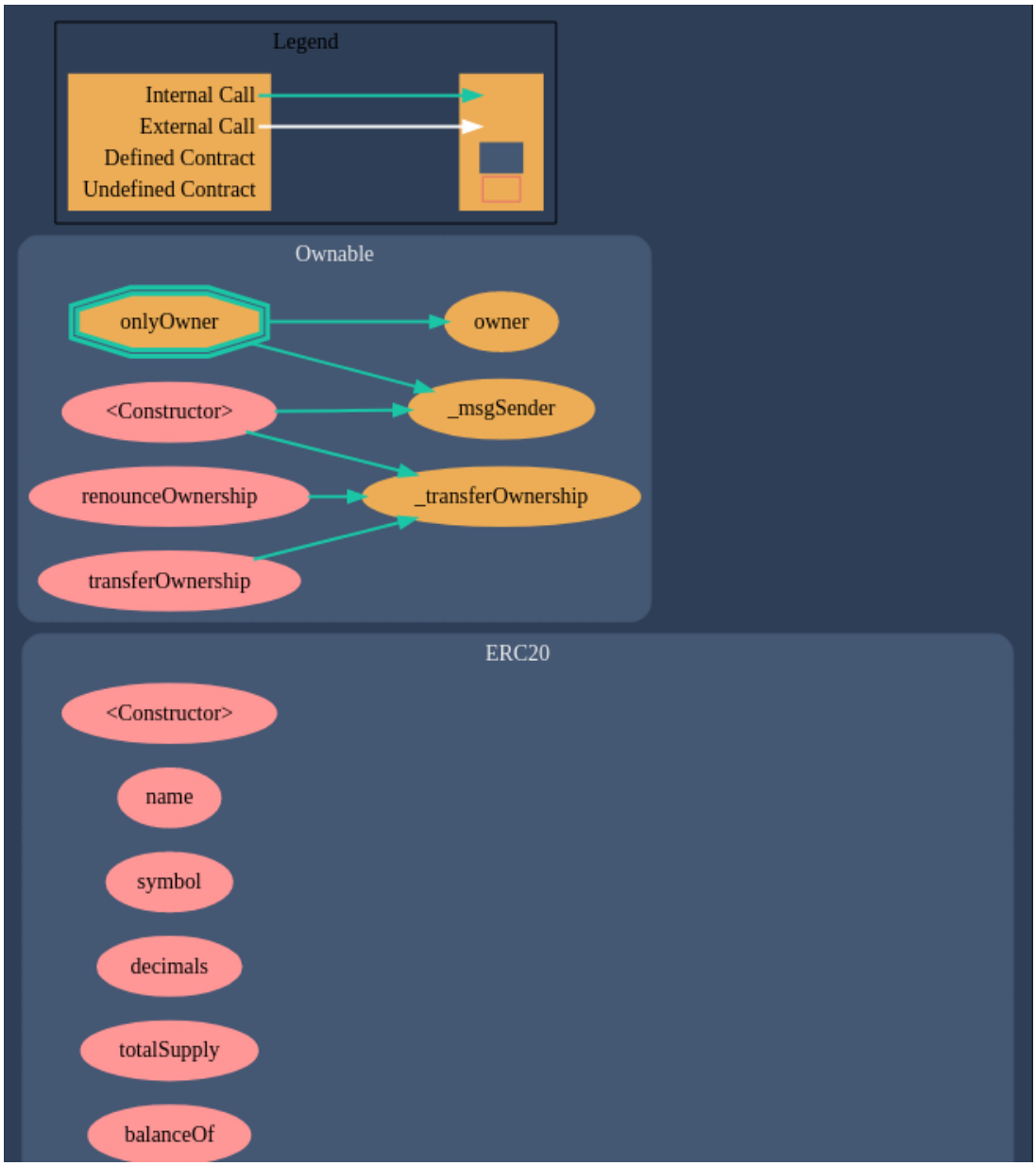
3.2.1 PeccalaUser.sol

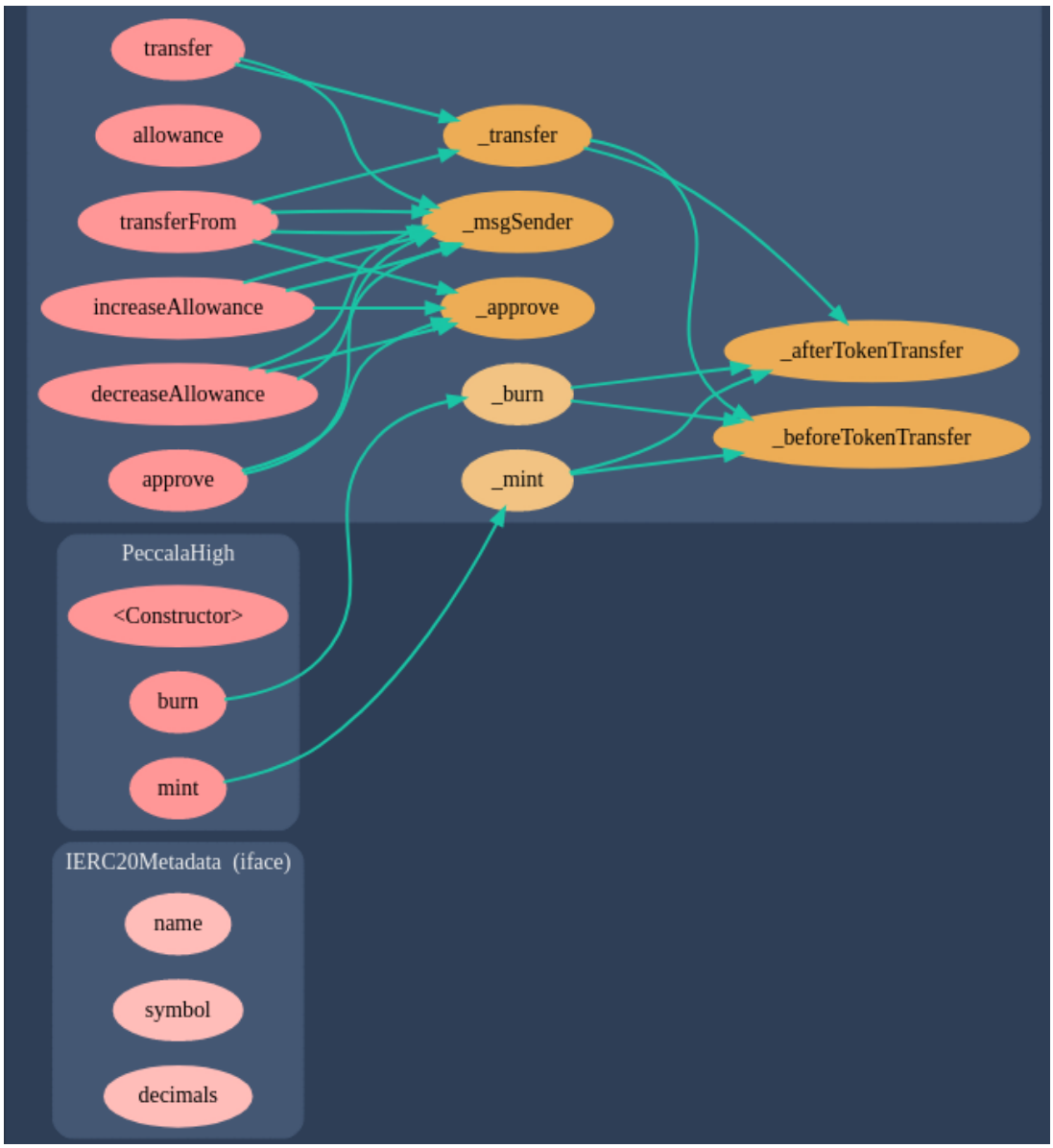


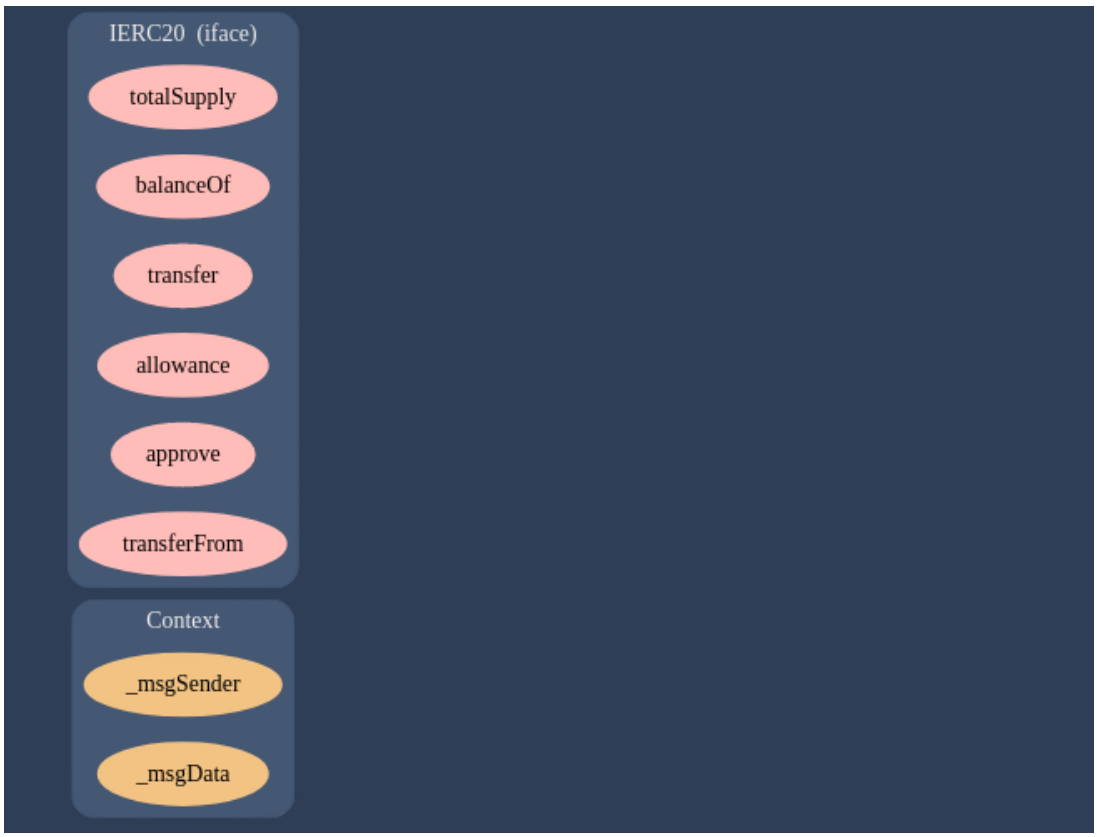

PeccalaHigh
burn
mint
IERC20Metadata (iface)
name
symbol
decimals
UserFactory
createUser
getUser
getUserId
updateUser
User
onlyOwner
<Constructor>
approve
transfer


getFactory
Ownable
onlyOwner
owner
<Constructor>
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
Context
_msgSender
_msgData
IERC20 (iface)
transfer
approve
totalSupply
balanceOf
allowance
transferFrom

3.2.2 PeccalaHigh.sol




transfer
allowance
transferFrom
increaseAllowance
decreaseAllowance
approve
_transfer
_msgSender
_approve
_burn
_mint
_afterTokenTransfer
_beforeTokenTransfer
PeccalaHigh
<Constructor>
burn
mint
IERC20Metadata (iface)
name
symbol
decimals



3.3 Read of persistent state following external call

Risik Low
Path PeccalaUser.sol#86

Description

One of the major dangers of calling external contracts is that they can take over the control flow.

Risk

In the reentrancy attack (a.k.a. recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.

Solution

The best practices to avoid Reentrancy weaknesses are:

- Make sure all internal state changes are performed before the call is executed. This is known as the Checks-Effects-Interactions pattern.
- Use a reentrancy lock (ie. OpenZeppelin's ReentrancyGuard.

Proof

```
    createUser(bytes32 userId) external onlyOwner returns(address) {
        User data = new User();
        userById[userId] = address(data);
        userByAddress[address(data)] = userId;
        emit UserCreated(address(data), userId);
        return address(data);
    }
```

3.4 A floating pragma is set

Risik Low
Path PeccalaHigh.sol#1-4
Path PeccalaUser.sol#1-4

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Risk

Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Solution

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

Deploy with any of the following Solidity versions:

```
* 0.5.16 - 0.5.17
* 0.6.11 - 0.6.12
* 0.7.5 - 0.7.6 Use a simple pragma version that allows any of these versions.
Consider using the latest version of Solidity for testing.
```

Proof

```
pragma solidity ^0.8.6;
```

3.5 Write of persistent state following external call

Risik Low
Path PeccalaUser.sol#87

Description

One of the major dangers of calling external contracts is that they can take over the control flow.

Risk

In the reentrancy attack (a.k.a. recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.

Solution

The best practices to avoid Reentrancy weaknesses are:

- Make sure all internal state changes are performed before the call is executed. This is known as the Checks-Effects-Interactions pattern.
- Use a reentrancy lock (ie. OpenZeppelin's ReentrancyGuard.

Proof

```
    createUser(bytes32 userId) external onlyOwner returns(address) {
        User data = new User();
        userById[userId] = address(data);
        userByAddress[address(data)] = userId;
        emit UserCreated(address(data), userId);
        return address(data);
    }
```