Smart Contract Audit Report for Petcoin

Testers

1. Or Duan
2. Avigdor Sason Cohen
3. Ido Shdaimah

Table of Contents

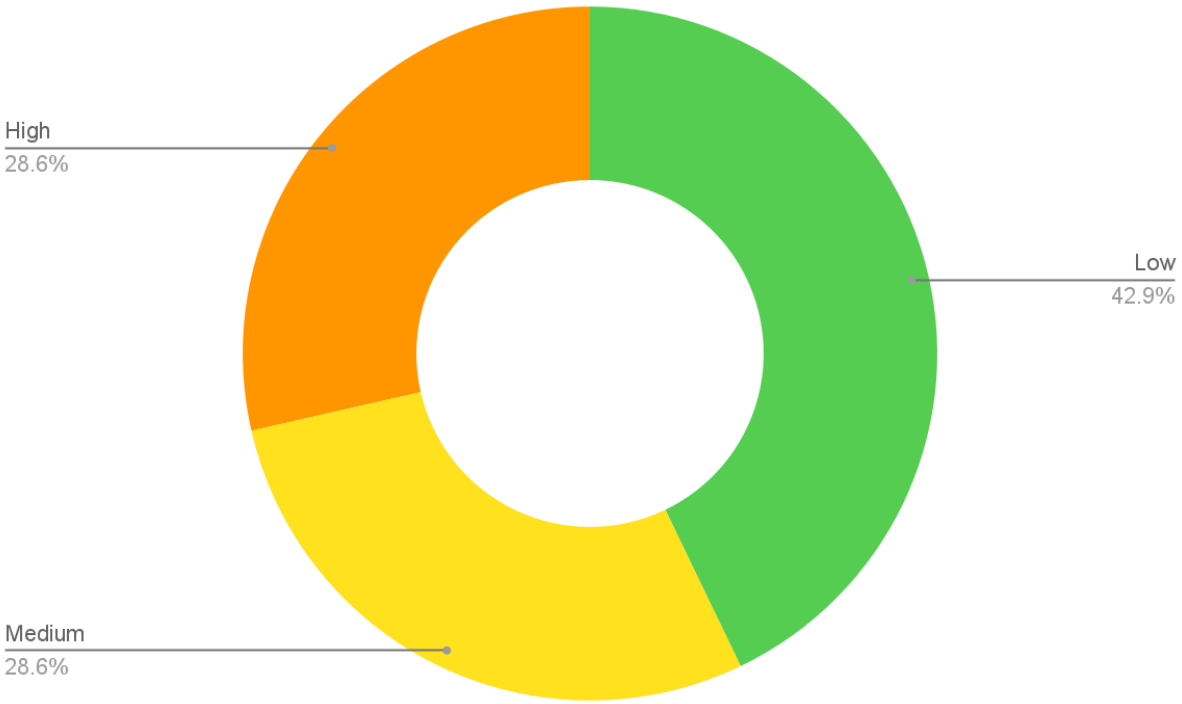
Management Summary

Petcoin contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the Petcoin smart contracts.

Over the research period of 40 research hours, we discovered 7 vulnerabilities in the contracts.

Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	3	2	2	0	0

- **Critical** - Immediate or ongoing part of the business being exploited with direct key business losses.
- **High** – Direct threat to key business processes.
- **Medium** – Indirect threat to key business processes or partial threat to business processes.
- **Low** – No direct threat exists. The vulnerability may be exploited using other vulnerabilities.
- **Informational** – This finding does not indicate vulnerability, but states a comment that notifies about design flaws and improper implementation that might cause a problem in the long run.

Approach

Introduction

Petcoin contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the client team we defined the following contract as the scope of the project:

petsForwarder.sol	48c9518e7ba6d7faecf73bb4cb199bb664a76a767121f76e9ed65c931c118248
pets.sol	f8a36aa1ffae8c41c25c4e22cdba86306d89a884f64a87a3767ca77eb0108d97

Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

Threat Model

We defined that the largest current threat to the system is the ability of malicious users to steal funds from the contract.

Protocol Overview

Protocol Introduction

PetCoin is a digital currency with a cause. A portion of each transaction is automatically donated to shelters and rescues in desperate need of assistance.

With low fees, high transparency, and blockchain-powered technology, PetCoin is ready to make a significant impact on our furry friends in need.

Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the SCSVS v1.2, with improved grammar, clarity, conciseness and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

Architecture, Design and Threat Modeling	Test Name
G1.2	Every introduced design change is preceded by threat modeling.
G1.3	The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
G1.4	The SCSVS, security requirements or policy is available to all developers and testers.
G1.5	The events for the (state changing/crucial for business) operations are defined.
G1.6	The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens).
G1.7	The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.
G1.8	If the fallback function can be called by anyone, it is included in the threat model.
G1.9	Business logic is consistent. Important changes in the logic should be applied in all contracts.
G1.10	Automatic code analysis tools are employed to detect vulnerabilities.
G1.11	The latest major release of Solidity is used.
G1.12	When using an external implementation of a contract, the most recent version is used.
G1.13	When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality.
G1.14	The order of inheritance is carefully specified.
G1.15	There is a component that monitors contract activity using events.
G1.16	The threat model includes whale transactions.
G1.17	The leakage of one private key does not compromise the security of the entire project.

Policies and Procedures	Test Name
G2.2	The system's security is under constant monitoring (e.g. the expected level of funds).
G2.3	There is a policy to track new security vulnerabilities and to update libraries to the latest secure version.
G2.4	The security department can be publicly contacted and that the procedure for

	handling reported bugs (e.g., thorough bug bounty) is well-defined.
G2.5	The process of adding new components to the system is well defined.
G2.6	The process of major system changes involves threat modeling by an external company.
G2.7	The process of adding and updating components to the system includes a security audit by an external company.
G2.8	In the event of a hack, there's a clear and well known mitigation procedure in place.
G2.9	The procedure in the event of a hack clearly defines which persons are to execute the required actions.
G2.10	The procedure includes alarming other projects about the hack through trusted channels.
G2.11	A private key leak mitigation procedure is defined.

Upgradability	**Test Name**
G2.2	Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy.
G2.3	The upgrade process is executed by a multisig contract where more than one person must approve the operation.
G2.4	Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult).
G2.5	*initialize()* can only be called once.
G2.6	*initialize()* can only be called by an authorized role through appropriate modifiers (e.g. *initializer*, *onlyOwner*).
G2.7	The update process is done in a single transaction so that no one can front-run it.
G2.8	Upgradeable contracts have reserved gap on slots to prevent overwriting.
G2.9	The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added.
G2.10	There are no changes in the order in which the contract state variables are declared, nor their types.
G2.11	New values returned by the functions are the same as in previous versions of the contract (e.g. *owner()*, *balanceOf(address)*).
G2.12	The implementation is initialized.
G2.13	The implementation can't be destroyed.

Business Logic	**Test Name**
G4.2	The contract logic and protocol parameters implementation corresponds to the documentation.
G4.3	The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed.
G4.4	The contract has correctly enforced business limits.
G4.5	The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow).
G4.6	The business logic does not rely on the contract's balance (e.g., *balance == 0*).

G4.7	Sensitive operations do not depend on block data (e.g., *block hash*, *timestamp*).
G4.8	The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes).
G4.9	The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead.

Access Control	**Test Name**
G5.2	The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization.
G5.3	New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted.
G5.4	The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation.
G5.5	The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed.
G5.6	Calls to external contracts are only allowed if necessary.
G5.7	Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts.
G5.8	All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized.
G5.9	the access controls fail securely, including when a revert occurs.
G5.10	If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible.

Communication	**Test Name**
G6.2	Libraries that are not part of the application (but the smart contract relies on to operate) are identified.
G6.3	Delegate call is not used with untrusted contracts.
G6.4	Third party contracts do not shadow special functions (e.g. revert).
G6.5	The contract does not check whether the address is a contract using *extcodesize* opcode.
G6.6	Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the *send* function unless it is a must.
G6.7	The result of low-level function calls (e.g. *send*, *delegatecall*, *call*) from other contracts is checked.
G6.8	Contract relies on the data provided by the right sender and does not rely on tx.origin value.

Arithmetic	**Test Name**
G7.2	The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
G7.3	the unchecked code snippets from Solidity ≥ 0.8.* do not introduce integer

	under/overflows.
G7.4	Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract.
G7.5	Non-strict inequality is used for balance equality.
G7.6	Correct orders of magnitude are used in the calculations.
G7.7	In calculations, multiplication is performed before division for accuracy.
G7.8	The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator.

Denial of Service	Test Name
G8.2	The contract does not iterate over unbound loops.
G8.3	Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
G8.4	The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent.
G8.5	The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit).
G8.6	Expressions of functions assert or require have a passing variant.
G8.7	If the fallback function is not callable by anyone, it is not blocking contract functionalities.
G8.8	There are no costly operations in a loop.
G8.9	There are no calls to untrusted contracts in a loop.
G8.10	If there is a possibility of suspending the operation of the contract, it is also possible to resume it.
G8.11	If whitelists and blacklists are used, they do not interfere with normal operation of the system.
G8.12	There is no DoS caused by overflows and underflows.

Blockchain Data	Test Name
G9.2	Any saved data in contracts is not considered secure or private (even private variables).
G9.3	No confidential data is stored in the blockchain (passwords, personal data, token etc.).
G9.4	Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack.
G9.5	Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number).

Gas Usage and Limitations	Test Name
G10.2	Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion.
G10.3	Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary).

Clarity and Readability	Test Name
G11.2	The logic is clear and modularized in multiple simple contracts and functions.
G11.3	Each contract has a short 1-2 sentence comment that explains its purpose and functionality.
G11.4	Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract.
G11.5	The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions.
G11.6	Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like *ownable*) instead of implementing their own.
G11.7	Consistent naming patterns are followed throughout the project.
G11.8	Variables have distinctive names.
G11.9	All storage variables are initialized.
G11.10	Functions with specified return type return a value of that type.
G11.11	All functions and variables are used.
G11.12	*require* is used instead of *revert* in *if* statements.
G11.13	The *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition in input from users and external contracts.
G11.14	Assembly code is only used if necessary.

Test Coverage	Test Name
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis.
G12.5	Contract specification has been formally verified.
G12.6	The specification and results of the formal verification is included in the documentation.

Decentralized Finance	Test Name
G14.1	The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
G14.2	Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.
G14.3	Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back.

G14.4	If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig).
G14.5	Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10^18 / divisor).
G14.6	Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares.
G14.7	Governance contracts are protected from flash loan attacks. One possible mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
G14.8	When using on-chain oracles, contracts are able to pause operations based on the oracles' result (in case of a compromised oracle).
G14.9	External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
G14.10	Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack).
G14.11	Complex math operations that consist of both multiplication and division operations first perform multiplications and then division.
G14.12	When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract).

Audit Findings

Broken Transfers

Status	Fixed
Risk	High
Location	Pets.unsetExistingFee & Pets.unsetLiquidityPool
Tools	Manual testing
Description	In these functions, the array elements are not removed, but only set to *address(0)*. In the case of *unsetExistingFee*, this is very problematic: *_transfer* will call *ERC20Upgradeable._transfer* with the recipient set to *address(0)*: `super._transfer(sender, masterFeeList[i], feeAmount);` However, OpenZeppelin's *_transfer* reverts when *address(0)* is passed as the recipient. Therefore, calling *Pets.unsetExistingFee* currently breaks transfers completely.
Mitigation	You should use *pop()* instead of *delete* so the length is also updated and there is no default value.

Impossible Transfers

Status	Fixed
Risk	High
Location	Pets._transfer
Tools	Manual testing

Description	In *_transfer*, the number of blocks since the last receipt of tokens is checked:
	
	`uint256 lastReceive = lastUserReceive[userAddress];`
	`uint256 checkLastReceive = block.number - lastReceive;`
	`require(`
	`checkLastReceive >= timeLimitReceive,`
	`"PETS: Transaction Denied: Time Limit Enabled - Check timeLimitReceive for required blocks"`
	`);`
	
	This feature can be abused by an attacker to prevent any transfers for an user. The attacker can regularly transfer a very small amount to the user (e.g., 1 wei) so that *lastUserReceive[recipient]* is set to the current block number.

Mitigation	This one is tricky to mitigate. You could either change the logic of the function or set a minimum amount for which *lastUserReceive[recipient]* is updated for example.

Roles Impossible To Revoke

Status	Fixed
Risk	Medium
Location	Pets.initialize:188-192
Tools	Manual testing

Description	*msg.sender* is used directly there (instead of *_msgSender()*). This can become problematic when initialize is called over the forwarder.
	In that case, the caller will get all roles (lines 165 - 169), but they will not be revoked in lines 188-192 because *msg.sender* will refer to the forwarder address.
	
	`if(msg.sender != walletPetsMultiSig){`
	`_revokeRole(SNAPSHOT_ROLE, msg.sender);`
	`_revokeRole(PAUSER_ROLE, msg.sender);`
	`_revokeRole(MINTER_ROLE, msg.sender);`
	`_revokeRole(UPGRADER_ROLE, msg.sender);`
	`_revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);`
	`}`

Mitigation	Use *_msgSender()* instead of *msg.sender*.

Wrong Function Action

Status	Fixed
Risk	Medium
Location	Pets.setSymbol
Tools	Manual testing

Description	This function is wrong.
	Instead of updating the symbol, it updates the name. It is therefore currently not possible to update the symbol.
	
	`function setSymbol(string memory newValue)`
	`external`
	`onlyRole(DEFAULT_ADMIN_ROLE)`
	`returns (bool)`
	`{`
	`tokenName = newValue;`
	`emit UpdateSymbol(newValue);`
	`return true;`
	`}`

Mitigation	Modify the function.

Wrong Function Signature

Status	Fixed
Risk	Low
Location	PetsForwarder - Line 24
Tools	Manual testing
Description	We suspect that *_TYPEHASH* is wrong. The last argument is “bytes data” but because the data is hashed in *verify*, it should be “bytes32 data”. This can cause problems because bytes is an alias to bytes1. `bytes32 private constant _TYPEHASH = keccak256("ForwardRequest(address from,address to,uint256 value,uint256 gas,uint256 nonce,bytes data)");`
Mitigation	Modify the function signature.

Possible Non-Existing Destination Chain

Status	Fixed
Risk	Low
Location	Pets.bridgePETS
Tools	Manual testing

Description	I assume that these events are consumed by some off-chain components for bridging funds to another chain. However, the string *destinationChain* is not validated.
	A user can therefore mistakenly pass a non-existing destination chain there, in which case the off-chain components probably would ignore the event and the funds would be lost.
	
	`emit Bridge(`
	`  _msgSender(),`
	`  tokenAmount,`
	`  destinationChain,`
	`  destinationWallet`
	`);`

Mitigation	It might be a good idea to have a few (configurable) whitelisted destination chain strings there.

Possible Duplicates

Status	Fixed
Risk	Low
Location	Pets.setLiquidityPool & Pets.setNewFee
Tools	Manual testing
Description	It is not checked if the passed address is already contained in the list (*liquidityPoolList* or *masterFeeList*). It is therefore possible that these lists contain duplicates. This would lead to undesired behavior. For instance in the case of *liquidityPoolList*, the balance of the pool would be counted two times in *liquidityPoolBuyMaxTransfer*.
Mitigation	Add a require that checks that the passed address is not already contained in the list.