Smart Contract Security Audit Report for FatCats

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

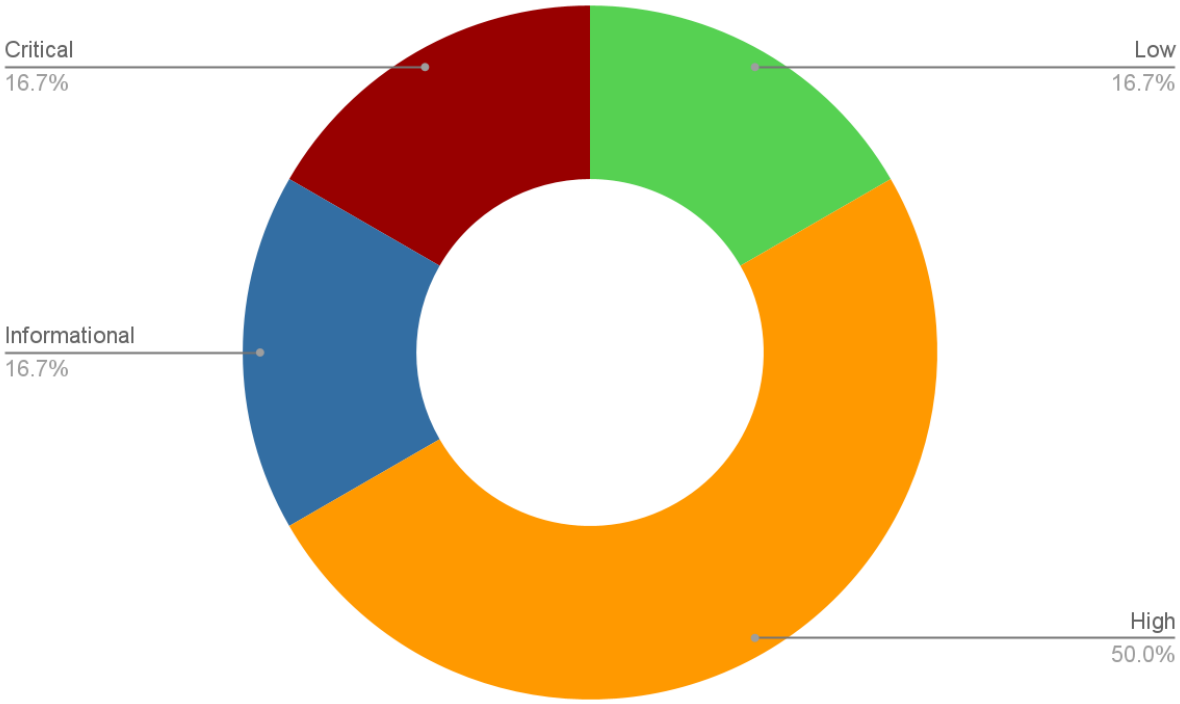
Management Summary

FatCats contacted Sayfer Security in order to perform smart contract auditing on their NFT contract on the Ethereum network

Over the research period of 3 weeks we discovered 6 vulnerabilities in the contract. One vulnerability was classified as high which let an attacker access the NFT metadata before it was revealed and make smart predictions about airdrops or specific minting transactions for rare NFTs.

Most vulnerabilities are relevant only when minting new NFTs so we highly recommend fixing the vulnerabilities found in this report before minting new ones.

Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	1	0	3	1	1

- **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

FatCats contacted Sayfer in order to perform a security audit on FatCats 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 FatCats smart contracts.

Scope Overview

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

- ./FatCats.sol - 0xedF6d3C3664606Fe9EE3a9796d5CC75E3B16e682

Our tests were performed between May 24 to June 5, 2022

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. The second highest risk to the platform is the ability of attack to steal NFT from the contract.

Protocol Overview

Protocol Introduction

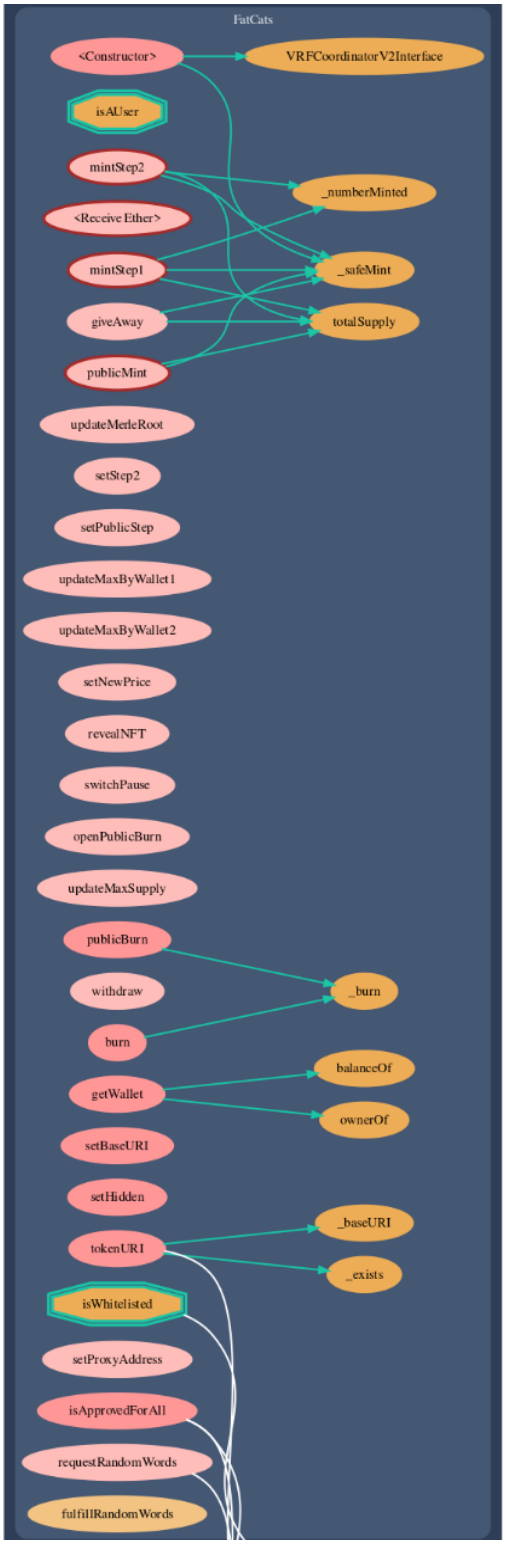
Fat Cats is a collection of 5,000 unique NFTs that double as a membership token and share of all Fat Cats holdings. Owning a Fat Cats NFT gives you access to a share of the entire DAO's holdings at an affordable price.

All liquid funds will be held in a basket of stable coins and other crypto assets as the Members see fit. In the case of time-sensitive decisions, the Council may spend up to 20% of all escrowed funds.

FatCats Contract is an NFT token contract having minting in steps - step1, step2, and public mint. Minting steps are set by the owner. Whitelisted users can mint NFT in step1 and step2. Whitelisted users are validated by Merkle proof.

FatCats contract inherits the MerkleProof, Ownable, Address, VRFCoordinatorV2Interface, VRFConsumerBaseV2, IERC721Receiver, Context, IERC721Metadata , Strings, ERC165, IERC721, standard smart contracts from the OpenZeppelin library. These OpenZeppelin contracts are considered community-audited and time-tested, and hence are not part of the audit scope.

Protocol Graph



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 to borrow 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 vise versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract).

Audit Findings

Guessable Token URIs

Status	Open
Risk	Critical
Location	FatCats.sol
Tools	Manual testing

Description	Having a head start of knowledge about the NFT's metadata could give an attacker to know which NFT he should mint before the public reveal. This could potentially let the attacker gain control of the most valuable NFTs in the project.
	
	The following vulnerability enables an attacker to decide which NFT to buy based on metadata before the public reveal. This vulnerability is relatively easy to exploit due to other vulnerabilities we found. In the time between the `shuffle` flag turning true in `requestRandomWords()` and the NFTs being revealed in `revealNFT()`, an attacker can guess the token's URIs.
	
	While the `revealed` flag is still false the `tokenURI()` returns `hideUri`, which means the average user will see only the dummy/hidden URI without the full metadata.
	
	To fully construct the token's URI string another transaction is being made to execute the `requestRandomWords()` method which sets the `s_randomWords` via `fulfillRandomWords()`. This is done via Chianlink's VRF.
	
	Only then the following code is being executed and returns the full token URI string:
	
	`string memory URI = _baseURI();`
	`uint256 randomId = ((s_randomWords + tokenId) % maxSupply) + 1;`
	`return`
	`bytes(URI).length > 0`
	`? string(abi.encodePacked(URI, randomId.toString(), ".json"))`
	`: "";`
	
	The vulnerability is that `baseURI`, `maxSupply` and `s_randomWords` are public and an attacker can predict the token's URI without the need of the `revealed` variable

In V1 of the project we have seen that the first transaction that executes `requestRandomWords()`:

	Txn Hash	Method ⓘ	Block	Age
👁	0xaa34c687444c602a42…	Request Random W...	14818982	11 days 15 hrs ago

And then the second transaction that reveals the NFT by executing the `revealNFT()`:

	Txn Hash	Method ⓘ	Block	Age
👁	0x7a42f7065faa12b2637…	Reveal NFT	14819665	11 days 12 hrs ago

We can see that during 3 hours, an attacker could have viewed any NFT metadata before it was revealed and make smart predictions about airdrops or specific minting transactions for rare NFTs.

Mitigation	Revealing an NFT has no one size fits all. It is highly dependent on the strategy and the user experience the end-user should have. For better security we encourage to reveal, set the base URI and change the state of `s_randomWords` state variable with the shortest time difference between them. While not mitigating the vulnerability completely, reducing the time between these actions can reduce the risk of bad actors exploiting it. A better way is to do all of the state changes in the same transaction while revealing, this is many times not technically possible. For deeper dive into how to implement a random NFT airdrop and more secure ways we recommend reading Randomization strategies for NFT drops

Whitelisted Addresses Can Be Contracts

Status	Open
Risk	High
Location	FatCats.sol
Tools	Manual testing
Description	When minting an NFT the minting contract could block or allow minting to contract addresses. Allowing minting to contract or having a vulnerability that allows contract minting could result in a situation where the attacker can revert the transaction if he has prior knowledge about which NFT he should mint. The modifier `isAUser()` is only used for `publicMint()`. `function publicMint(uint256 amountToMint) external payable isAUser {` For `mintStep1()` and `mintStep2()` methods the `isWhitelisted()` modifier is being used. This means the code does not check for non-contract addresses in the whitelist Merkle proof hashes. While we can not know what was the processes behind the whitelist generating and how secured it is, the code itself is vulnerable to this attack. Checking for non-contract addresses off-chain or not in the same transaction is discouraged as it is vulnerable to multiple attack vectors An attacker could obtain an NFT and revert the transaction if the NFT is not rare enough. In conjunction with the Guessable Token URI, an attacker can easily gain prior knowledge about the NFT's metadata and revert transactions based on the rarity.
Mitigation	Add the `isAUser` modifier to the `mintStep1()` and `mintStep2()` methods

Weak Insecure Random Implementation of Chainlink's VRF

Status	Open
Risk	High
Location	FatCats.sol
Tools	Manual testing

Description	Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context.
	
	Smart contracts have to rely on off-chain solutions to generate secured random numbers. Fatcats uses Chainlink's VRF system to generate a random number which later being used as a random id for the token's URI using the `fulfillRandomWords` method:
	
	`function fulfillRandomWords(`
	`uint256, /* requestId */`
	`uint256[] memory randomWords`
	`) internal override {`
	`s_randomWords = (randomWords[0] % maxSupply) + 1;`
	`}`
	
	The method saves the random number to a state variable called `s_randomWords` which by the name implies it is a random word. In practice, the number is modulo with `maxSupply`(5000) so it is a pseudo weak random number between 1 to 5000. This can lead developers to think they can rely on this number for secured randomness, which it isn't.

Mitigation	Assign the returned value coming from VRF to the global state variable

Centralization of the Contract and Team Wallet

Status	Open
Risk	High
Location	FatCats.sol
Tools	Manual testing
Description	Projects that rely on one key are vulnerable to key losses, phishing attacks, inside actors manipulations, death of the owner, and more. When the project is dependent only on one key. These kinds of attacks are the most common of the biggest hacks. The project checks the `isOwner` in multiple places. Losing the keys to this deployment address will compromise the entire project. In addition, the team wallet that will get the eth using the `withdraw` function is also subjected to the same attack vector If the above wallets are using multisig, this finding will be resolved.
Mitigation	Depending on the use case, level of security and the user experience the project wants to apply there are multiple ways to mitigate this attack, for instance using multisig wallet, smart wallet or applying multiple owners to the project.

openPublicBurn Function Switches Rather than Opens

Status	Open
Risk	Low
Location	FatCats.sol
Tools	Manual testing

Description	The method `openPublicBurn` does not just open the public burn, it is switching it on and off depending on the current state of the variable.
	
	`function openPublicBurn() external onlyOwner {`
	`publicBurnFlag = !publicBurnFlag;`
	`}`
	
	This could lead a developer or an operator of the project to think they are opening the public burn step but actually switching it off causing a bad reputation to the project.

Mitigation	Set the state variable `publicBurnFlag` to true or change the method name to `switchPublicBurn`

Step Mechanism Is Hard to Maintain

Status	Open
Risk	Info
Location	FatCats.sol
Tools	Manual testing
Description	The code uses boolean variables for each step, this means that any time the developer adds another step he/she has to add the logic to toggle the reset of the flags. This could lead to confusion and potential security bugs. A better approach would be to use enum as the current step if, this gives the advantages of a clear step name like `EARLY_ADAPTERS`, `PUBLINK_MINT,` etc If this is too verbose or confusing, it could be implemented via a simple integer, this has the mathematical advantage of `if step > 2` or `require(newStep < oldStep, "can not go back"`
Mitigation	Use solidity built-in enum or simple integer syntax