Crypto Pisis

02.06.2021

Sabancı University CS48001 Course
Nidanur Günay
Furkan Çelik
Görkem Görkey

Project Outline

Gamification of cryptocurrencies with the flavor of the Turkish culture by developing Turkish Crypto Pisi inspired by the proven cryptokitties concept.

Cognitive empathy is more like a skill: Humans learn to recognize and understand others' emotional states as a way to process emotions and behavior. We tend to use this skill unconsciously, even with virtual entities; we care for them and like to show them off. After all, we all played Tamagotchi when we were little. Turkish Crypto Currencies will be digital Pisi cat characters in order to motivate that emotional connection.

Currently, Turkish people are hyped towards cryptocurrencies and new technologies under blockchain like non-fungible tokens (NFTs), and we believe that the addition of a cultural flavor to that hype may incentivize more people towards blockchain technology and can show the usability of Ethereum outside of being a cryptocurrency. This project may also convey some sort of understanding of blockchain technology to upcoming coin traders. Crypto Pisis will also have collectible nature, you can have multiple cats, build a themed collection, sell them and buy the ones you like. You must feed them in order to keep them healthy and appealing to other Pisis. Breeding will add more interaction so that people will try to collect various cats.

Goals

1. Coding a smart contract to realize the features we intend to add such as NFT of Pisis, feeding, breeding.
2. Doing security analysis on the smart contract and fixing the issues.
3. Building a website to visualize the Pisis created by the smart contract.
4. Making the website easy to use, also cute.

Specifications

Attributes & Actions of Pisi

Each Pisi has a set of attributes, these are split into appearance-related and functionality-related traits. We called a combination of these attributes a gene. You can find explanations of these below:

- Eye color

    - Specifies color of the eye in 6 digit RGB hex color representation
- Eye size
    - Specifies the size of the eye in terms of radius in float. Size holds an integer between 0-256 but we cast this value to float in the frontend.
- Head size
    - Specifies the size of the head in terms of radius in float. Size holds an integer between 0-256 but we cast this value to float in the frontend.
- Cat beard size
    - Specifies the size of the cat beard in terms of length in float. Size holds an integer between 0-256 but we cast this value to float in the frontend.
- Tail size
    - Specifies the size of the tail in terms of length in float. Size holds an integer between 0-256 but we cast this value to float in the frontend.
- Tail color
    - Specifies color of the tail in 6 digit RGB hex color representation
- Tail accent color
    - Specifies accent color of the tail in 6 digit RGB hex color representation. This will be activated or deactivated by stripe type.
- Cat color
    - Specifies color of the cat in 6 digit RGB hex color representation
- Cat accent color
    - Specifies accent color of the cat in 6 digit RGB hex color representation. This will be activated or deactivated by stripe type.
- Stripe type
    - The specified type of stripe in terms of mode number. This again can be an integer between 0-256 which we cast to a stripe type in the frontend.
- Hungeryness
    - This gene specifies the hungriness of the cat. Unlike what it says, this is not how hungry this cat is but rather can be better-called appetite. This is later used in the calculation of fertility and feeding updates.
- Fragility
    - Fragility is how fragile that Pisi is as the name suggests. This is not used in this implementation of our project but it is available for future implementation.
- Fertility
    - Fertility specifies how fertile that Pisi is and adjusted by using feeding instead of randomly initializing it like others, this will be initialized as a max value of 255.
- Appeal
    - The appeal is used in breeding and used to specify whether Pisi is suited for breeding. The appeal is not changed in the course of Pisi's life unless the

owner of the Pisi does not feed that Pisi for long times. If that Pisi is left unfeed, upon the depletion of fertility, we deplete appeal.

This gene set gives us over 2^{40} unique Pisi which is almost over 10^{12} Pisi. However, in the current state of our project, we could not support that much of a change and limited our appearance of Pisis to preconstructed parts instead of using that detailed color and size variables. These can be supported by a better Pisi generator module in React.

Minting New Pisi

Any user of the crypto pisi can mint any number of Pisi as they want, without a cost. Minting a new pisi is done by creating random traits and decode those traits into a unified hash. This hash value is a 64 byte hex representation of that particular pisi and can act as an identifier of that pisi at some points. This identifier then parsed into two different sets of variables.

The first set is responsible for generating the appearance of that Pisi and can be reachable to anyone. These are mostly 6 digit color values and stored in string format, or some type/size specifications which are again hex-based uint8 represented as a string. Representing these values as strings decreased our gas usage since manipulating a uint number to part it into sub-variables was harder to implement.

Second set is responsible for the genetics of the Pisi. These attributes are not effective at appearance of that Pisi but they are used in the internal logic of this system, which will be discussed later. This genetic information is only accessible by the owner of the Pisi.

Minted Pisis are added to existing mapping and have the ability to be sold or not which is decided by the owner of that Pisi at any price the owner wishes to sell.

Breeding Pisis

However, minting is not the only way to generate new Pisis. The other way is breeding existing Pisis someone has. This cannot be done with two parties but can be done by a single party with multiple Pisis. This method is especially useful when minted Pisis has good attributes which one wants to protect or improve upon to create the ultimate Pisi.

The breeding process is inspired by the reproduction of human genes and a crossover mechanism is adopted. Each Pisi is evaluated by their appeal and fertility, by which influence over the offspring is calculated. To lower the gas use, we split each gene of each Pisi into only 2 parts and select the correct part by influence assessment. Upon that assessment, offspring can either fully or partially inherit both parents.

Unlike many popular implementations, we opted not to include any gender-related attribute to Pisis and the breeding process. However, in the hash of the Pisis, we have

approximately 6 digits remaining among which one of them can be used for that purpose. We left that for future work.

Even though each Pisi at the breeding period has the perfect genes, there is a chance factor that can prevent the creation of any offspring. This is achieved by having a random value generation and comparing it to the appeal of both Pisis. To prevent consistently trying breeding of two good Pisi, we put a requirement of paying a certain amount of ether to the contract itself.

Feeding a Pisi

Even though there is an involvement of chance in the breeding process, an experienced player can optimize their Pisis by supporting them. This support comes from feeding them. Each Pisi has an associated Hungerness value which determines the interval of being infertile due to hunger.

This interval is necessary time needed to pass to make that Pisi infertile. Each Pisis appeal and fertility decreases over time as they get hungrier. To prevent that every now and then, the owner of that Pisi needs to feed their Pisi. In the event of for example not feeding that Pisi for 10 consecutive days, that Pisi will be infertile forever.

The feeding process requires some amount of fixed ether to be paid. We believe that this can create enough money to make further developments. In addition, the feeding feature can increase excitement amongst the users.

Control of change of the attributes of appeal and fertility has been done eagerly. We had two approaches for that: one is developing a smart contract that can make the overall system a time-locked contract. However, even though this idea sounds good, gas requirements are higher than making eager adjustments. Each function which in one way or another will reach fertility and/or appeal, first adjusts values relative to the time passed since the last feeding.

Challenges

I. Random Number Generation

Our main limitation and challenge at the smart contract was the random number generator. Unlike most programming languages, solidity does not have any inherent RNG, which leaves us to implement our own system. Since generating a cryptographically secure and truly random generation is hard to achieve and not necessary in the score of our project, we decided to implement a pseudo-random algorithm.

One of the well-known pseudo-random algorithms is using the current time of execution in nanoseconds. Knowing that we searched for possible variables we can use for that purpose and decided to use “block. timestamp” as our randomizer. However, since this value can be stable for transactions in a single block and hence resulting in the same randomized Pisi for example, we added additional values like block.difficulty, msg.sender and total Pisis on sale alongside with the number of Pisis the caller has. These additional values ensure that in a block, only the same caller will get the same random value. To achieve randomness, we used the keccak256 value of these variables.

Even with these techniques, this randomized variable is predictable beforehand, we believe that this randomness is enough for ensuring that predicting this number will have limited damage to the intended security and functionality of the application.

II. Visualization of the Pisies

One of the challenges that we have encountered during the development phase of our project is how to visualize2^{40} unique Pisi which is almost over 10^{12} Pisi. As a result of productive research on the internet, we have found some svg folders for Pisi body, Pisi beard and Pisi eye. Pisibody folder includes 35, Pisibeard folder includes 8 and Pisi Eye folder includes 8 distinct svg files. We have also created a name array that includes 58 distinct possible kitty names. Then we have summed the decimals of unique bodyColor values and taken a modulo of 35 since there are 35 different svg files. And applied the same process for eyeColor, beardSize and hash values that come from Pisi smart contract. That approach helps to create 35*8*8*58 which is 129920 different combinations of Pisi appearance. We, as a team thought that that number of combination is sufficient for the scope of this phase of the project

Security Analysis

For security analysis, we used 3 different analysis tools: Remix’s Static Code Analyzer [1], Mythril[2], and Slither[3]. None of these returned an unintended feature as an issue. They have detected some reentrancy errors at feed and transfer functions but neither of those is internally vulnerable to reentrancy attack. Besides that, the main complication of all of those analyzers was our use of block.timestamp and how it is not only predictable but also influenced by the miner. However, we believe that this influence cannot fully affect our system due to the aforementioned reasons.

[1] https://remix-ide.readthedocs.io/en/latest/static_analysis.html
[2] https://github.com/ConsenSys/mythril
[3] https://github.com/crytic/slither

Even though none of these analyzers detected, one additional possible weakness we have is the possible infinite loop and depletion of the contract. In the removal of a Pisi from the marketplace or from the arsenal of any owner, we chose a strategy to shift elements of the array to not leave any empty space in the memory. However, this requires us to do a full loop over the memory array and can cause a possible infinite loop with a huge number of Pisi's in the market. On a possible approach to this changing last element of the array and element we want to remove and then remove the last element from the array. This will solve the looping problem but can still cause the same problem in the worst case. Another measure can be to put a small fee for putting a Pisi up for sale. This will prevent an attacker from continuously putting/removing elements of the marketplace array.

We could also change the ordering of the codes in the breed, feed and transfer functions since an issue at the transfer function will revert the whole function and not cause any issues. This will prevent reentrancy attack analyzers warned us about. Detailed outputs of Remix's static analyzer and Mythril can be found at the appendix.

Gas Analysis

We used Remix to make gas analyses for us. Unfortunately, many of the functions have been found vulnerable in terms of gas cost. However, we believe that none of those are serious issues and hard to happen in real life due to the limited variable space we had at integers of loops. However, some of them are necessary like iterating over pisis on the sales list and removing elements from it. In our tests, lots of Pisi being on sale and deleting a Pisi at the start of the list, there has not been any major issues and operation had successfully happened with the set gas limit of 600k. Put down from the sale function does not vary too much from the specified usage below. One can inspect each functions transaction gas cost below:

- Deployment = 5391615 Gas
- Minting = 621284 Gas
- Put down from sale = 69148 Gas
- Put to sale = 207672 Gas
- Feeding = 37200 Gas + 2 finney
- Breeding = 327921 Gas + 2 finney
- Get Appeal/Fertility = 47160 Gas
- Transfer = 337536 Gas

Remaining functions are call functions. The only non call getter we have are get appeal and get fertility because of the aforementioned eager execution of the feed functionality.

Overall cost of the deployment and transfer is relatively big and can cause possible problems. We can solve issues at the deployment by using a more constrained approach than the ERC721 version we have used. We used base class but there are many additional functionalities which we don't need like allowance or safe transfer etc. For example switching from the Full version of ERC721 implementation to Metadata version allowed us to cut gas usage to almost half of what it was. Transfer can be solved by changing element removal strategy as discussed before.

Future Work & Possible Developments

I. Random Number Generation

As discussed earlier, we opted to use a basic pseudo random algorithm as our random number generator. The initial worry we had was implementing a real random number generator would have been very costly gas wise but it seems like there are better alternatives.

Main implementation of the cryptokitties used blockhash instead of the timestamp we used. Timestamp is a problematic variable since miners can have an influence over it very easily, however it is not possible to know future blockhash value. A miner can still make an adjustment to run a given transaction after a very specific block but it may not be worth the effort economically. Creators of the crypto kitties estimated this value to be 0.25 ETH which will rarely happen in our platform[4].

However, this implementation still may not be feasible enough but will definitely be stronger than our current approach. In addition to that, there is no visible cost overhead of this implementation. For example, a miner still can compute the previous block's hash very carefully to influence the next block but computing both of the two consecutive blocks is less likely to have an influence or at least harder to achieve than timestamp.

II. Removal of an Element

As discussed above, one ather development we could have made at the smart contract was using a more clever, more resilient and less gas consuming approach of the element removal process. This can be removing only the last bit of an array and placing that last element to the place of the element we wish to remove. Even though this approach will shorten our runtime and number of iterations in the loop, the worst case is still problematic.

[4] https://github.com/axiomzen/eth-random

Another approach is to switch from removing elements from an array to precomputing the index of elements we wish to remove and remove our element at a constant time and at a constant gass fee. In addition, instead of deleting elements, we could have used the pop function of the array which will refund some amount of gas.

Another possible solution is using an additional mapping to record the index of each entry. We believe this approach too will be costly to implement since mappings require more space and gas but SU-Squares[5] used this approach in their algorithm of NFT.

Appendix

CryptoPisies Source Code .zip folder drive link

https://drive.google.com/drive/folders/1ooosxQebO8w3DZlFQ2me5x7gp4rf6FCB?usp=sharing

Remix's Solidity Static Analysis Results

[5] https://github.com/su-squares/ethereum-contract/blob/master/contracts/SuNFT.sol

Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in Pisi.transferPisi(string): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.

more

Pos: 115:4:

Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in Pisi.breed(string,string): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.

more

Pos: 138:4:

Inline assembly:

The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.

more

Pos: 244:8:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 51:58:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 85:12:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 151:62:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 183:53:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 188:63:

Mythril's Security Analysis

==== Integer Arithmetic Bugs ====

SWC ID: 101

Severity: High

Contract: Pisi

Function name: getEyeColor(string)

PC address: 10046

Estimated Gas Usage: 3632 - 7723

The arithmetic operator can underflow.

It is possible to cause an integer overflow or underflow in the arithmetic operation.

In file: Pisi.sol:277

memory) {

return _pisiCollection[

Initial State:

Account: [CREATOR], balance: 0x100000001, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0

Caller: [CREATOR], function: getEyeColor(string), txdata: 0x5aee433000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000002001ffc9a700000000000000000000000000000000000000000000000000000000, value: 0x0

==== Integer Arithmetic Bugs ====

SWC ID: 101

Severity: High

Contract: Pisi

Function name: getEyeColor(string)

PC address: 10098

Estimated Gas Usage: 3632 - 7723

The arithmetic operator can underflow.

It is possible to cause an integer overflow or underflow in the arithmetic operation.

In file: Pisi.sol:277

memory) {

return _pisiCollection[

Initial State:

Account: [CREATOR], balance: 0x100000001, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0

Caller: [CREATOR], function: getEyeColor(string), txdata: 0x5aee433000000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000000000000000000000000000000000000002001ffc9a700000000000000000000000000000000000000000000000000000000, value: 0x0

==== Exception State ====

SWC ID: 110

Severity: Medium

Contract: Pisi

Function name: _pisiHashesToSell(uint256)

PC address: 13389

Estimated Gas Usage: 1268 - 1363

An assertion violation was triggered.

It is possible to trigger an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

In file: Pisi.sol:44

t256 public onSaleCount;

con

Initial State:

Account: [CREATOR], balance: 0x1, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0

Caller: [SOMEGUY], function: _pisiHashesToSell(uint256), txdata: 0xafc8136c0000000000000000000000000000000000000000000000000000000000000000, value: 0x0

==== Dependence on predictable environment variable ====

SWC ID: 116

Severity: Low

Contract: Pisi

Function name: mint()

PC address: 17739

Estimated Gas Usage: 1000 - 1849

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

In file: Pisi.sol:239

```
- 1;
        while (_i != 0) {
            bstr[k-
```

Initial State:

Account: [CREATOR], balance: 0x10000000000001, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0

Caller: [CREATOR], function: mint(), txdata: 0x1249c58b, value: 0x0

==== Dependence on predictable environment variable ====

SWC ID: 116

Severity: Low

Contract: Pisi

Function name: mint()

PC address: 17816

Estimated Gas Usage: 1038 - 1887

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

In file: Pisi.sol:242

```
0;
        }
        return string(bstr);
    }

    function str2Hex
```

Initial State:

Account: [CREATOR], balance: 0x1, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0

Caller: [CREATOR], function: mint(), txdata: 0x1249c58b, value: 0x0

Screenshots of the CryptoPisies DAPP

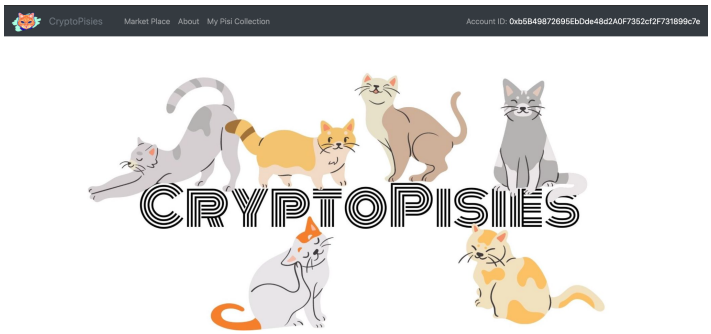

Figure 1: Landing Page of the CryptoPisies DAPP.

Figure 2: Minting pisies by clicking on Mint Pisi button.

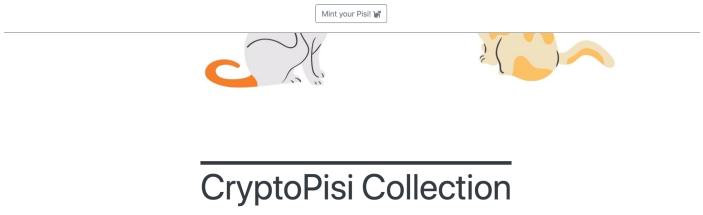

Figure 3: Screenshot of the minted pisi

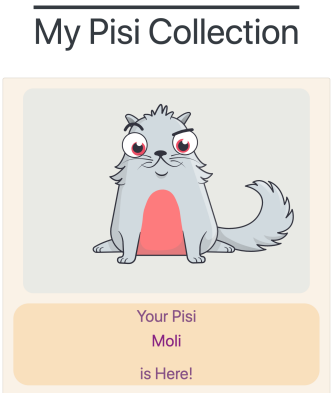

Figure 4: My Pisi Collection Page that displays the user's pisies.

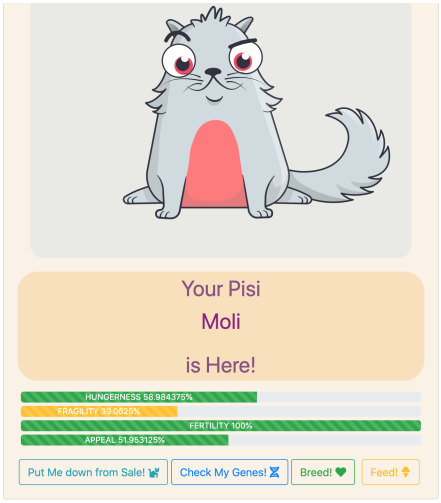

Figure 5: Screen of the Pisi attributes

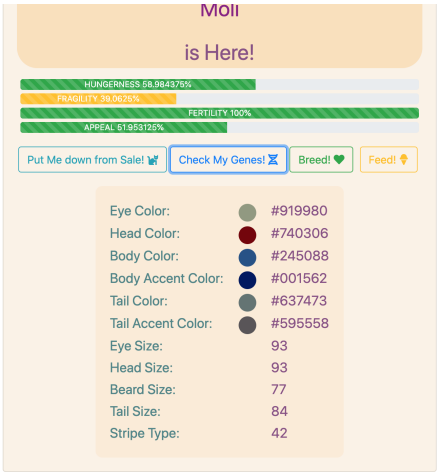

Figure 6: Screen of the Pisi Genes

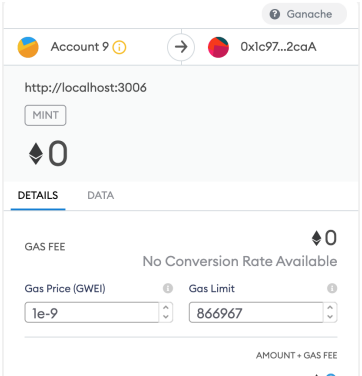

Figure 7: Screen of Metamask after call of Mint function.

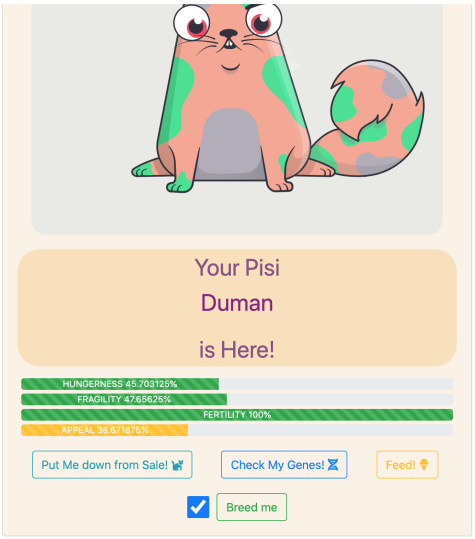

Figure 8: Breed function and Selection of the first parent.

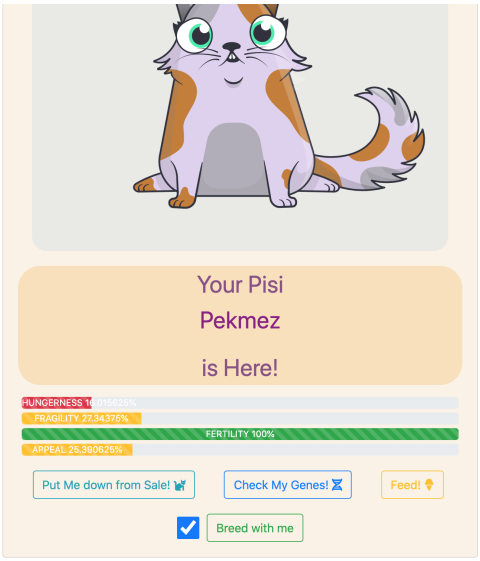

Figure 9: Selection of the second parent.

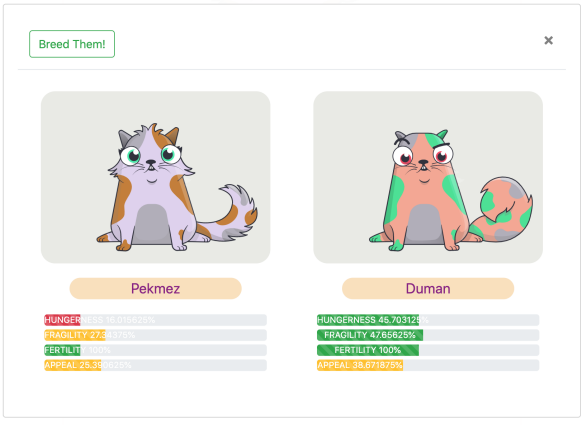

Figure 10: Parent pisies

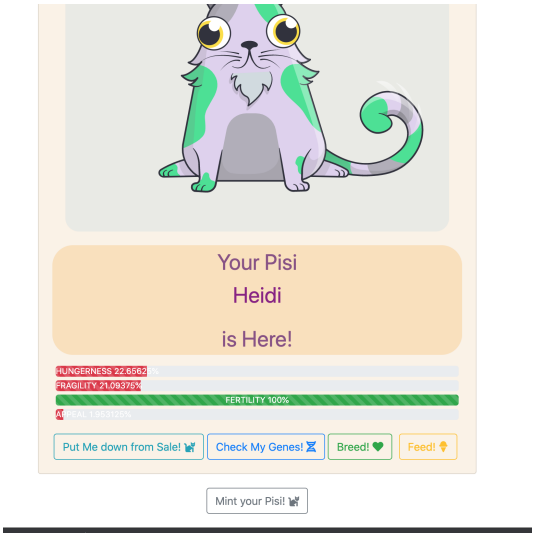

Figure 11: Baby pisi