// HALBORN

POLKADEX

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **March 15, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	03/15/2021	Gabi Urrutia
0.2	Document Edits	03/23/2021	Oussama Amri
0.3	Document Review and Edits	03/23/2021	Steven Walbroehl
1.0	Remediation Plan	03/30/2021	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Oussama Amri	Halborn	Oussama.Amri@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Polkadex engaged Halborn to conduct a security assessment on their Smart contracts beginning on March 15th, 2021 and ending March 22th, 2021. The security assessment was scoped to the contract dex.sol and an audit of the security risk and implications regarding the changes introduced by the development team at Polkadex prior to its production release shortly following the assessments deadline.

The most important security findings were the use of an improper key management policy and the need to correctly implement a Role-Based Access Control Policy. It has learned from the past attack on PAID network that it is important to implement a key management policy based on multi signature to perform critical actions such as deploying or upgrading. In addition, applying the principle of least privilege in privileged accounts allows that if a private key is compromised, the rest of the critical actions can be performed by other privileged users and quickly recovering the control of the contract. Moreover, the contract does NOT contain any obvious exploitation vectors that Halborn was able to leverage within the time frame of testing allotted.

Though the outcome of this security audit is that the Polkadex team needs to solve some critical issues. Due to time and resource constraints, only testing and verification of essential properties related to the Contract was performed to achieve objectives and goals set in the scope.

Halborn recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.2 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit.While manual testing is recommended

to uncover flaws in logic, process,and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart Contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions(solgraph)
- Manual Assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- Testnet deployment (Truffle,Ganache)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident, and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.3 SCOPE

IN-SCOPE:
Code related to dex.sol smart contract.

Specific commit of contract:

3f38ff05e4bae60255a126c2d2cc6df99078e130

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	2	1	1	4

LIKELIHOOD

IMPACT

		(HAL-03)		(HAL-01)
			(HAL-02)	
	(HAL-05)	(HAL-04)		
(HAL-06) (HAL-07)				
(HAL-08) (HAL-09)				

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
IMPROPER KEY MANAGEMENT POLICY	Critical	HARDWARE WALLET WILL BE USED - 03/25/2021
IMPROPER ROLE-BASED ACCESS CONTROL POLICY	High	FUNCTIONS REMOVED - 03/25/2021
USE OF SELFDESTRUCT FUNCTION	High	FUNCTION REMOVED - 03/25/2021
NO TEST COVERAGE	Medium	FUTURE RELEASE UPDATE
FLOATING PRAGMA	Low	SOLVED - 03/30/2021
USE OF BLOCK.NUMBER	Informational	SOLVED - 03/30/2021
PRAGMA VERSION	Informational	RISK ACCEPTED - 03/30/2021
POSSIBLE MISUSE OF PUBLIC FUNCTIONS	Informational	RISK ACCEPTED - 03/30/2021
DOCUMENTATION	Informational	FUTURE RELEASE UPDATE
STATIC ANALYSIS	-	-
AUTOMATED SECURITY SCAN RESULTS	-	-

FINDINGS & TECH DETAILS

3.1 (HAL-01) IMPROPER KEY MANAGEMENT POLICY - CRITICAL

Description:

The very basics of Blockchain is decentralization which should be applied as much as possible in all processes around Blockchain, such as key management. If a single private key is used to manage control over the smart contract and perform some actions such as deploy or upgrade a Smart Contract. However if some how the private key has been compromised, it could lead to devastating consequences. For example, on March 5, 2021, the PAID Network smart contract was attacked, regardless of the smart contract was audited before. Approximately $100 million of PAID token were extracted by the attacker. In that case, the private key was compromised and the attacker had successfully upgraded and replaced the original smart contract with a malicious version that allowed tokens to be burned and minted. Thus, if best practices had been implemented in the key management policy, the attacked could not upgrade the contract using a single private key because it needed one more key to perform any action over the contract with a multi-signature implementation. Including multi-signature in the key management policy avoid that any critical action over the contract can be performed by a single user.

Reference: https://halborn.com/explained-the-paid-network-hack-march-2021/

Risk Level:

Likelihood - 5
Impact - 5

Recommendations:

It is recommended to include multi-signature in the key management policy in order to avoid the risk of losing control over the smart contract because of the private key being compromised.

Remediation Plan:

After reviewing the findings and associated risks, Polkadex team doesn’t consider necessary to use a multi-signature wallet for key management policy and they will store the deployer’s private key in a hardware wallet.

3.2 (HAL-02) IMPROPER ROLE-BASED ACCESS CONTROL POLICY - HIGH

Description:

In smart contracts, implementing a correct Access Control policy is an essential step to maintain security and decentralization for permissions on a token. All the features of the smart contract , such as mint/burn tokens and pause contracts are given by Access Control. For instance, Ownership is the most common form of Access Control. In other words, the owner of a contract (the account that deployed it by default) can do some administrative tasks on it. Nevertheless,other authorization levels are required to follow the principle of least privilege, also known as least authority. Briefly, any process, user or program only can access to the necessary resources or information. Otherwise, the ownership role is useful in a simple system, but more complex projects require the use of more roles by using Role-based access control.

Therefore, there could be multiple roles such as manager, minter, admin, or pauser in contracts which use a proxy contract. In dex.sol contract, Owner is the only one privileged role. Owner can transfer the contract ownership, call selfdestruct function and disable transfers of the token. In conclusion, owner role can do too many actions in Polkadex smart contract. So, if the private key of the owner account is compromised and multi-signature was not implemented, the attacker can perform many actions such as transferring ownership or destruct the contract without following the principle of least privilege.

Code Location:

Owner role can access below functions:

dex.sol Lines #34-36

```
34      function DestructToken() public OnlyOwner {
35          selfdestruct(msg.sender);
36      }
```

dex.sol Lines #38-40

```
38      function TransferOwnerShip(address payable NewAddress) public OnlyOwner {
39          Owner = NewAddress;
40      }
41
```

dex.sol Lines #46-48

```
46      function disable() public OnlyOwner {
47          IsEnd = true;
48      }
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

It's recommended to use role-based access control based on the principle of least privilege to lock permissioned functions using different roles.

Reference: https://www.cyberark.com/what-is/least-privilege/

Remediation Plan:

Polkadex team removed the privileged functions DestructToken() and disable() in their last commit c00b1ed5d53b27f02f29cb37777bd494eb5b9c21. The only privileged function left is TransferOwnerShip. Then, the issue was

solved.

3.3 (HAL-03) USE OF SELFDESTRUCT FUNCTION - HIGH

Description:

After the Parity Bug incident in 2017, the use of the selfdestruct function began to stop being used. In addition, if selfdestruct function is implemented in a smart contract without a proper access control policy, any attacker can self-destruct the contract if the private key is compromised.

References:
- https://swcregistry.io/docs/SWC-106
- SEC554: How to lose $280 million with a single line of code

Code Location:

dex.sol Line #35

```
34      function DestructToken() public OnlyOwner {
35          selfdestruct(msg.sender);
36      }
```

Risk Level:

Likelihood - 3
Impact - 5

Recommendations:

It is recommended not to use the selfdestruct function. If the use of this function is absolutely necessary, a multi-signature address will be required to approve the action of **self destruct** the contract and one role can only execute this action.

Remediation Plan:

Polkadex team removed the `DestructToken()` function in their last commit `c00b1ed5d53b27f02f29cb37777bd494eb5b9c21`.

3.4 (HAL-04) NO TEST COVERAGE - MEDIUM

Description:

Unlike other software programs, smart contracts can not be modified or removed if deployed once into a specific address except if you deploy them with a proxy contract. Checking the code by automated testing (unit testing or functional testing) is a good practice to be sure all lines of the code work correctly. Mocha and Chai are useful tools to perform unit test in Smart Contracts functions. Mocha is a Javascript testing framework for creating both synchronous and asynchronous unit tests. Moreover, Chai is an assertions library with some interfaces such as assert, expect and should to develop custom unit tests.

References:
- https://github.com/mochajs/mocha
- https://github.com/chaijs/chai
- https://docs.openzeppelin.com/learn/writing-automated-tests

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended considering to perform as much as possible test cases to cover all possible scenarios in the smart contract.

Remediation Plan:

Pending: Polkadex team will include test cases in future release.

3.5 (HAL-05) FLOATING PRAGMA - LOW

Description:

Polkadex contract uses the floating pragma ^0.7.6. 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 another pragma, for example, either an outdated pragma version that might introduce bugs that affect the contract system negatively or a recently released pragma version which has not been extensively tested.

Code Location:

dex.sol Line #~1

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.7.6;
3  import "./ERC20.sol";
```

Reference: ConsenSys Diligence - Lock pragmas

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Consider lock the pragma version known bugs for the compiler version. Therefore, it is recommended not to use floating pragma in the production.. Apart from just locking the pragma version in the code, the caret (^) need to be removed. it is possible locked the pragma fixing the version both in truffle-config.js if you use the Truffle framework and

in hardhat.config.js if you use HardHat framework for the deployment.

Remediation plan:

Solved: Polkadex team locked the pragma in their last commit c00b1ed5d53b27f02f29cb37777bd494eb5b9c21.

3.6 (HAL-06) USE OF BLOCK.NUMBER - INFORMATIONAL

Description:

During a manual static review, the tester noticed the use of block.number. Sometimes, Contract developers should be aware that using block.timestamp and block.number does not mean current time. block.number can be also influenced (to a lesser extent) by miners, so the testers should be warned that this may have some risk if miners collude on time manipulation to influence the price oracles.

Code Location:

dex.sol Line #12

```
9       constructor() ERC20("Polkadex", "PDEX") {
10          MainHolder();
11          Owner = msg.sender;
12          InitialBlockNumber = block.number;
13      }
```

dex.sol Line #23

```
23          require(block.number > InitialBlockNumber + 25, "Time to claim vested tokens has not reached")
24          require(VestedTokens[msg.sender] > 0, "You are not eligible for claim");
25          mint(msg.sender, VestedTokens[msg.sender]);
26          VestedTokens[msg.sender] = 0;
27      }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

If possible, it is recommended to use oracles instead of block.number as a source of entropy and random number.

Remediation Plan:

Solved: In this case, the use of block.number is safe since their timescales will occur across 3 months rather than seconds.

3.7 (HAL-07) PRAGMA VERSION - INFORMATIONAL

Description:

Polkadex contract uses one of the latest pragma version (0.7.6) which was released back in December 16, 2020. The latest pragma version is (0.8.2) and it was released in March 2021. Many pragma versions have been lately released, going from version 0.6.x to the recently released version 0.8.x. in just 6 months.

Reference: https://github.com/ethereum/solidity/releases

Code Location:

dex.sol Line #2

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.7.6;
3  import "./ERC20.sol";
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

In the Solitidy Github repository, there is a json file describe all bugs reported for each compiler versions. No bugs have been found in > 0.7.3 versions but very few in 0.7.0 -- 0.7.3. So, the latest stable version is pragma 0.6.12. Furthermore, pragma 0.6.12 is widely used by Solidity developers and has been extensively tested in many security audits.

Reference: https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json

Remediation Plan:

Polkadex team accepts the use of pragma version 0.7.6.

3.8 (HAL-08) POSSIBLE MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

In public functions, array arguments are immediately copied to memory, while external functions can read directly from calldata. Reading call-data is cheaper than memory allocation. Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, the function expects its arguments being located in memory when the compiler generates the code for an internal function.

Code Location:

dex.sol Line #15

```
15      function ClaimAfterVesting() public {
16          //change 25 to require blocknumber from launching block for
17
```

dex.sol Line #34

```
34      function DestructToken() public OnlyOwner {
35          selfdestruct(msg.sender);
36      }
37
```

dex.sol Line #38

```
38      function TransferOwnerShip(address payable NewAddress) public OnlyOwner {
39          Owner = NewAddress;
40      }
```

dex.sol Line #42

```
42      function ShowOwner() public view returns (address) {
43          return Owner;
44      }
45
```

dex.sol Line #46

```
46      function disable() public OnlyOwner {
47          IsEnd = true;
48      }
49  }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider as much as possible declaring external variables instead of public variables. As for best practice, you should use external if you expect that the function will only be called externally and use public if you need to call the function internally. To sum up, all can access to public functions while external functions only can be accessed externally.

Remediation Plan:

Polkadex team considers proper the use of public functions in the smart contract.

3.9 (HAL-09) DOCUMENTATION - INFORMATIONAL

Description:

Documentation provided by Polkadex team is not complete. For instance, the documentation included in the GitHub repository should include a walkthrough to deploy and test the smart contracts.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider updating the documentation in Github for greater ease when contracts are deployed and tested. Have a Non-Developer or QA resource work through the process to make sure it addresses any gaps in the set-up steps due to technical assumptions.

Remediation Plan:

Pending: Polkadex team will include documentation in future release.

3.10 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.

Results:

```
INFO:Detectors:
ERC20._transfer(address,address,uint256) (contracts/ERC20.sol#283-315) compares to a boolean constant:
        -IsRegisted[sender] == false (contracts/ERC20.sol#291)
ERC20._transfer(address,address,uint256) (contracts/ERC20.sol#283-315) compares to a boolean constant:
       -require(bool,string)(IsEnd == false,transfer date ended) (contracts/ERC20.sol#290)
ERC20._transfer(address,address,uint256) (contracts/ERC20.sol#283-315) compares to a boolean constant:
        -IsRegisted[recipient] == false (contracts/ERC20.sol#296)
ERC20._approve(address,address,uint256) (contracts/ERC20.sol#589-600) compares to a boolean constant:
       -require(bool,string)(IsEnd == false,transfer date ended) (contracts/ERC20.sol#596)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
INFO:Detectors:
Pragma version^0.7.4 (contracts/ERC20.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version^0.7.4 (contracts/ERC20.sol#108) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version^0.7.4 (contracts/ERC20.sol#194) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version^0.7.4 (contracts/ERC20.sol#217) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version^0.7.4 (contracts/dex.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.7.4 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Struct ERC20.info (contracts/ERC20.sol#271-274) is not in CapWords
Function ERC20.MintRegister(address,uint256) (contracts/ERC20.sol#255-259) is not in mixedCase
Parameter ERC20.MintRegister(address,uint256)._Addr (contracts/ERC20.sol#255) is not in mixedCase
Parameter ERC20.MintRegister(address,uint256)._Amount (contracts/ERC20.sol#255) is not in mixedCase
Function ERC20.TotalTokenHolders() (contracts/ERC20.sol#279-281) is not in mixedCase
Function ERC20.MainHolder() (contracts/ERC20.sol#633-936) is not in mixedCase
Variable ERC20.VestedTokens (contracts/ERC20.sol#261) is not in mixedCase
Variable ERC20.IsEnd (contracts/ERC20.sol#270) is not in mixedCase
Variable ERC20.GetUserBalanceByIndex (contracts/ERC20.sol#275) is not in mixedCase
Variable ERC20.IsRegisted (contracts/ERC20.sol#276) is not in mixedCase
Variable ERC20.GetIdByAddress (contracts/ERC20.sol#277) is not in mixedCase
Function PolkaDex.ClaimAfterVesting() (contracts/dex.sol#15-27) is not in mixedCase
Function PolkaDex.DestructToken() (contracts/dex.sol#34-36) is not in mixedCase
Function PolkaDex.TransferOwnerShip(address) (contracts/dex.sol#38-40) is not in mixedCase
Parameter PolkaDex.TransferOwnerShip(address).NewAddress (contracts/dex.sol#38) is not in mixedCase
Function PolkaDex.ShowOwner() (contracts/dex.sol#42-44) is not in mixedCase
Variable PolkaDex.Owner (contracts/dex.sol#6) is not in mixedCase
Variable PolkaDex.InitialBlockNumber (contracts/dex.sol#7) is not in mixedCase
Modifier PolkaDex.OnlyOwner() (contracts/dex.sol#29-32) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

```
INFO:Detectors:
TotalTokenHolders() should be declared external:
        - ERC20.TotalTokenHolders() (contracts/ERC20.sol#279-281)
name() should be declared external:
        - ERC20.name() (contracts/ERC20.sol#336-338)
symbol() should be declared external:
        - ERC20.symbol() (contracts/ERC20.sol#344-346)
decimals() should be declared external:
        - ERC20.decimals() (contracts/ERC20.sol#361-363)
totalSupply() should be declared external:
        - ERC20.totalSupply() (contracts/ERC20.sol#368-370)
balanceOf(address) should be declared external:
        - ERC20.balanceOf(address) (contracts/ERC20.sol#375-383)
transfer(address,uint256) should be declared external:
        - ERC20.transfer(address,uint256) (contracts/ERC20.sol#393-401)
allowance(address,address) should be declared external:
        - ERC20.allowance(address,address) (contracts/ERC20.sol#406-414)
approve(address,uint256) should be declared external:
        - ERC20.approve(address,uint256) (contracts/ERC20.sol#423-431)
transferFrom(address,address,uint256) should be declared external:
        - ERC20.transferFrom(address,address,uint256) (contracts/ERC20.sol#446-461)
increaseAllowance(address,uint256) should be declared external:
        - ERC20.increaseAllowance(address,uint256) (contracts/ERC20.sol#475-486)
decreaseAllowance(address,uint256) should be declared external:
        - ERC20.decreaseAllowance(address,uint256) (contracts/ERC20.sol#502-516)
ClaimAfterVesting() should be declared external:
        - PolkaDex.ClaimAfterVesting() (contracts/dex.sol#15-27)
DestructToken() should be declared external:
        - PolkaDex.DestructToken() (contracts/dex.sol#34-36)
TransferOwnerShip(address) should be declared external:
        - PolkaDex.TransferOwnerShip(address) (contracts/dex.sol#38-40)
ShowOwner() should be declared external:
        - PolkaDex.ShowOwner() (contracts/dex.sol#42-44)
disable() should be declared external:
        - PolkaDex.disable() (contracts/dex.sol#46-48)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

All relevant findings were founded in the manual code review.

3.11 AUTOMATED SECURITY SCAN RESULTS

Description:

Halborn used automated security scanners to assist with detection of well known security issues, and identify low-hanging fruit on the scoped contract targeted for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the testers machine, and sent the compiled results to MythX to locate any vulnerabilities. Security Detections are only in scope, and the analysis was pointed towards issues with the dex.sol.

Results:

SWC-000	Medium	Function could be marked as external.	dex.sol	L: 15 C: 4
SWC-000	Medium	Function could be marked as external.	dex.sol	L: 34 C: 4
SWC-000	Medium	Function could be marked as external.	dex.sol	L: 38 C: 4
SWC-000	Medium	Function could be marked as external.	dex.sol	L: 42 C: 4
SWC-000	Medium	Function could be marked as external.	dex.sol	L: 46 C: 4
SWC-100	Low	Function visibility is not set.	dex.sol	L: 9 C: 4
SWC-103	Low	A floating pragma is set.	dex.sol	L: 2 C: 0
SWC-108	Low	State variable visibility is not set.	dex.sol	L: 6 C: 20
SWC-108	Low	State variable visibility is not set.	dex.sol	L: 7 C: 12
SWC-120	Low	Potential use of "block.number" as source of randomness.	dex.sol	L: 12 C: 29
SWC-120	Low	Potential use of "block.number" as source of randomness.	dex.sol	L: 23 C: 16

All relevant findings were founded in the manual code review.

THANK YOU FOR CHOOSING

// HALBORN