EXTROPY.IO

Report for:

0xPunchline Protocol

August 2021

Version: 1.0

Prepared By: Extropy.IO
Email: info@extropy.io
Telephone: +44 1865261424

Table of Contents

1 Using This Report

To facilitate the dissemination of the information within this report throughout your organisation, this document has been divided into the following clearly marked and separable sections.

Document Breakdown		
0	Executive Summary	Management level, strategic overview of the assessment and the risks posed to the business
1	Technical Summary	An overview of the assessment from a more technical perspective, including a defined scope and any caveats which may apply
2	Technical Findings	Detailed discussion (including evidence and recommendations) for each individual security issue which was identified
3	Methodologies	Audit process and tools used

Disclaimer

The audit makes no statements or warranty about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the code to purpose, or their bug free status. The audit documentation is for discussion purposes only.`

Document Control

Client Confidentiality

This document contains Client Confidential information and may not be copied without written permission.

Proprietary Information

The content of this document should be considered proprietary information and should not be disclosed outside of 0xPunchline protocol.
Extropy gives permission to copy this report for the purposes of disseminating information within your organisation or any regulatory agency.

Document Version Control			
Data Classification	Client Confidential		
Client Name	0xPunchline protocol		
Document Title	0xPunchline Smart Contracts Audit		
Author	Extropy Audit Team		

Document History			
Issue No.	**Issue Date**	**Issued By**	**Change Description**
1.0	13/8/2021	Laurence Kirk	Released to client

Document Distribution List	
Senior Developer	0xPunchline Protocol
Laurence Kirk	CEO, Extropy

2 Executive Summary

Extropy was contracted by 0xPunchline protocol to conduct a code review and smart contracts vulnerability assessment in order to identify security issues that could negatively affect 0xPunchline's business or reputation if they led to the compromise or abuse of systems.
This report presents the findings of the smart contract security assessment conducted on behalf of 0xPunchline protocol The assessment was conducted between 05/08/21 and 13/08/2021 and was authorised by 0xPunchline protocol.

2.1 Assessment Summary

The contracts are of an acceptable standard, they are derived from known contracts, namely the ERC20 standard and the Masterchef contract from Sushiswap.
All of the issues found are of low or informational severity, nevertheless we recommend that the issues be addressed.

The results of the MythX analysis are attached, all the issues it highlights are false positives, or are included in the following list.

The following table breaks down the issues which were identified by phase and severity of risk.

Phase	Description	Critical	High	Medium	Low	Info	Total
1	Initial Audit	0	0	0	4	6	10

3 Technical Summary

3.1 Scope

File Name	SHA-1 Hash
src/contracts/Punch.sol	27bd7c9153781233ebdf40ec4431cee3bfbf3b59
src/contracts/MasterPunch.sol	252723515a28c945a593c3e9f03ebb3e61d850b9

4 Technical Findings – Code Audit

The remainder of this document is technical in nature and provides additional detail about the items already discussed, for the purposes of remediation and risk assessment.

4.1 Rounding Error

Risk Rating	Low

On line 197:

uint sacrifice = _amount.div(1000);

pool.lpToken.safeTransferFrom(address(msg.sender), address(BURN), sacrifice);

If the amount was smaller than 1000, safeMath will round the result to 0

An attacker could deposit amounts smaller than 1000 and keep repeating that indefinitely, not burning tokens which could have potential side effects.

Affects contract:
MasterPunch

4.2 Use a fixed and consistent pragma version

Risk Rating	Low

A floating pragma version is used

pragma solidity ^0.8.0;

Recommendation

Use a fixed and consistent solidity version in contracts and libraries.

Affects contract:
MasterPunch
Punch

References
SWC-103

4.3 Unused Constant

Risk Rating	Low

This public constant variable was hardcoded but never used, on line 55:

```
// BONUS MULIPLIER FOR EARLY PUNCH MAKERS.
UINT256 PUBLIC CONSTANT BONUS_MULTIPLIER = 10;
```

It could be that the developers meant to add it on line 185, but instead they hardcoded the number 10:

```
punch.mint(devaddr, punchReward.div(10));
```

However, it wouldn't make much sense to call it BONUS_MULTIPLIER since it is being used in a division, therefore either the division shown above should really be a multiplication, or the BONUS_MULTIPLIER variable was simply left in the code by mistake, or it was not implemented.

Recommendation
Remove the constant if not needed.

Affects contract:
MasterPunch

4.4 Forcibly sending LP tokens

Risk Rating	Low

It could be possible for a user to forcibly transfer LP tokens to the MasterPunch's on one or more pools, which could affect the number of Punch tokens minted to users.

On line 175 UpdatePool() reads the MasterPunch's balance from the pool selected:

```
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
```

The `lpSupply` variable is then used to calculate the pool's `accPunchPerShare`:

```
pool.accPunchPerShare = pool.accPunchPerShare.add(
    punchReward.mul(1e12).div(lpSupply)
);
```

which is then used to calculate the amount of Punch to tranfer users when they call wither `deposit()` on line 205 or in `withdraw()` on line 227:

```
uint256 pending =
    user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
        user.rewardDebt
    );
safePunchTransfer(msg.sender, pending);
```

4.5 Missing Use of NatSpec Format

Risk Rating	Info

Poor commenting practices, example on line 92:

```
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
```

Recommendation

The above example could be more formally rewritten by using the '@dev' tag instead of 'XXX'

```
// @dev DO NOT add the same LP token more than once. Rewards will be messed up if you do.
```

Affects contract:
MasterPunch

References
https://docs.soliditylang.org/en/v0.5.8/natspec-format.html

4.6 Unneeded library use

Risk Rating	Info

Solidity version 8 provides the overflow / underflow functionality provided by SafeMath

Recommendation

Remove the SafeMath library.

Affects contract:
MasterPunch

4.7 Restrict Visibility (gas saving)

Risk Rating	Info

function getMultiplier's visibility on line 129 could be restricted to internal.

Recommendation

Check that `getMultiplier()` does not require public or external visibility as per protocol design, if it doesn't then limit its visibility to `internal`

Affects contract:
MasterPunch

4.8 Use of comparison operators

Risk Rating	Info

On line 172:

```
if (block.number <= pool.lastRewardBlock) {
    return;
}
```

pool.lastRewardBlock` can only be updated inside the same `updatePool` function on line 190:
pool.lastRewardBlock = block.number;

Recommendation

Change the evaluation to use strictly equal `==` operator rather than the `<=` less than or equal to operator. This can improve code readability and perhaps gas.

Affects contract:
MasterPunch

4.9 State variable visibility is not set

Risk Rating	Info

It is best practice to set the visibility of state variables explicitly. The default visibility for "masterPunch" is internal, make sure that internal visibility is expected.
Recommendation
Explicitly set the visibility of the address variable masterPunch.

Affects contract:
Punch

References
SWC-108

4.10 Burn() function without access control

Risk Rating	Info

There is no modifier for the external burn function.

```
function burn(uint256 _amount) external {
```

```
    _burn(msg.sender, _amount);
}
```

Recommendation
Verify that burning tokens on purpose couldn't influence reward rations in the MasterPunch contract and that it is a design decision to expose the burn() function to external calls.

Affects contract:
Punch

5 Tool List

The following tools were used during the assessment:

Tools Used	Description	Resources
Solidity Metrics	Static analysis	https://github.com/ConsenSys/solidity-metrics
SWC Registry	Vulnerability database	https://swcregistry.io/
Mythx	Static Analysis	https://mythx.io/

5.1 Tailored Methodologies

5.1.1 Audit Goals

1. We will audit the code in accordance with the following criteria:

- **Sound Architecture**

  This audit includes assessments of the overall architecture and design choices. Given the subjective nature of these assessments, it will be up to the development team to determine whether any changes should be made.

- **Smart Contract Best Practices**

  This audit will evaluate whether the codebase follows the current established best practices for smart contract development.

- **Code Correctness**

  This audit will evaluate whether the code does what it is intended to do.

- **Code Quality**

This audit will evaluate whether the code has been written in a way that ensures readability and maintainability.

- **Security**

  This audit will look for any exploitable security vulnerabilities, or other potential threats to the users.

- **Testing and testability**

- This audit will examine how easily tested the code is, and review how thoroughly tested the code is.

Although we have commented on the application design, issues of crypto-economics, game theory and suitability for business purposes as they relate to this project are beyond the scope of this audit.

5.2 Test Methodology

The security audit is performed in two phases:

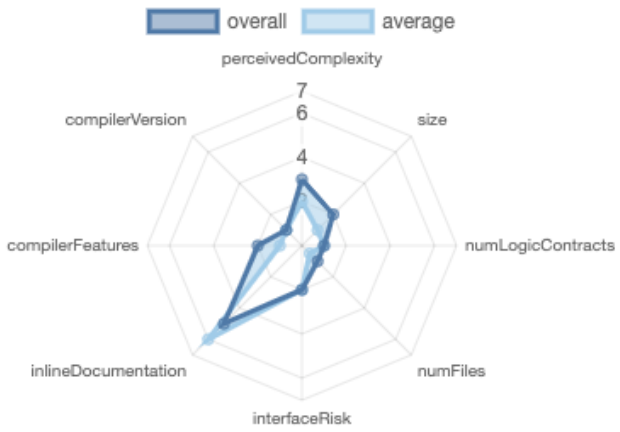
a. **Independent Code Review**

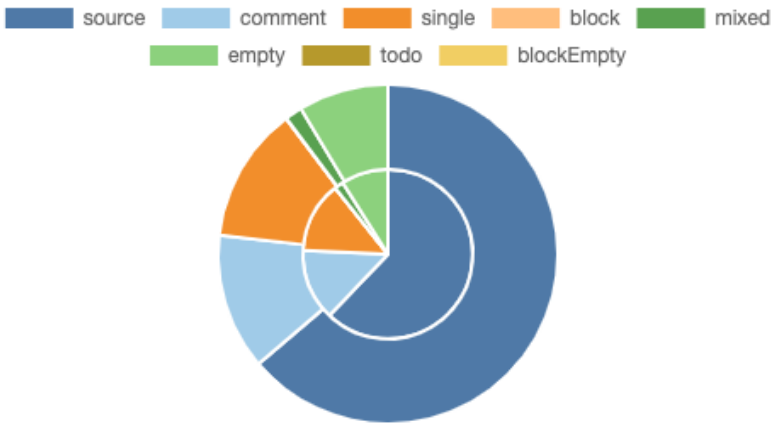
b. The code is inspected separately by four team members checking for software errors and known vulnerabilities.

c. **Static Analysis**

The code is subject to static analysis using Solidity Metrics and Mythx

5.3 Solidity Code Metrics





5.4 Mythx Findings

The MythX findings are appended to the end of this report. The issues marked as medium in that report are false positives, all other issues from the MythX where appropriate are included in the issues reported above.

MythX

Analysis 3b5a0e94-18b2-4c3c-9ffc-77c69c7f4f97 MythX

Started	Thu Aug 12 2021 15:01:14 GMT+0000 (Coordinated Universal Time)
Finished	Thu Aug 12 2021 15:47:20 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	Contracts/MasterPunch.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	2	38

ISSUES

MEDIUM

SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source file

contracts/MasterPunch.sol

Locations

```
730 require(isContract(target), "Address: call to non-contract");
731
732 (bool success, bytes memory returndata) = target.call{value: value}(data);
733 return _verifyCallResult(success, returndata, errorMessage);
734 }
```

MEDIUM

SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source file

contracts/MasterPunch.sol

Locations

```
1677 return;
1678 }
1679 uint256 lpSupply = pool.lpToken.balanceOf{address(this)};
1680 if (lpSupply == 0) {
1681 pool.lastRewardBlock = block.number;
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
6   ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
7
8   pragma solidity ^0.8.0;
9
10  /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
95  ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
96
97  pragma solidity ^0.8.0;
98
99  /*
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
126 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
127
128 pragma solidity ^0.8.0;
129
130 ////import "../IERC20.sol";
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
161 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
162
163 pragma solidity ^0.8.0;
164
165 ////import "./IERC20.sol";
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
523 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
524
525 pragma solidity ^0.8.0;
526
527 ////import "../utils/Context.sol";
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
602 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
603
604 pragma solidity ^0.8.0;
605
606 /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
820 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: Copyright
821
822 pragma solidity ^0.8.0;
823
824 ////import '@openzeppelin/contracts/access/Ownable.sol';
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
865 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
866
867 pragma solidity ^0.8.0;
868
869 // CAUTION
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
1099 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
1100
1101 pragma solidity ^0.8.0;
1102
1103 /**
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
1401 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
1402
1403 pragma solidity ^0.8.0;
1404
1405 ////import "../IERC20.sol";
```

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

contracts/MasterPunch.sol

Locations

```
1505 ////// SPDX-License-Identifier-FLATTEN-SUPPRESS-WARNING: MIT
1506
1507 pragma solidity ^0.8.0;
1508
1509 ////import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1673 // Update reward variables of the given pool to be up-to-date.
1674 function updatePool(uint256 _pid) public {
1675 PoolInfo storage pool = poolInfo[_pid];
1676 if (block.number <= pool.lastRewardBlock) {
1677 return;
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1674 function updatePool(uint256 _pid) public {
1675 PoolInfo storage pool = poolInfo[_pid];
1676 if (block.number <= pool.lastRewardBlock) {
1677 return;
1678 }
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1705 UserInfo storage user = userInfo[_pid][msg.sender];
1706 updatePool(_pid);
1707 if (user.amount > 0) {
1708 uint256 pending =
1709 user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1712 safePunchTransfer(msg.sender, pending);
1713 }
1714 pool.lpToken.safeTransferFrom(
1715 address(msg.sender),
1716 address(this),
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
727 string memory errorMessage
728 ) internal returns (bytes memory) {
729 require(address(this).balance >= value, "Address: insufficient balance for call");
730 require(isContract(target), "Address: call to non-contract");
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1717 _amount
1718 );
1719 user.amount = user.amount.add(_amount);
1720 user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);
1721 emit Deposit(msg.sender, _pid, _amount);
```

LOW

SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1717 _amount
1718 );
1719 user.amount = user.amount.add(_amount);
1720 user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);
1721 emit Deposit(msg.sender, _pid, _amount);
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1718 );
1719 user.amount = user.amount.add(_amount);
1720 user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);
1721 emit Deposit(msg.sender, _pid, _amount);
1722 }
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1718 );
1719 user.amount = user.amount.add(_amount);
1720 user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);
1721 emit Deposit(msg.sender, _pid, _amount);
1722 }
```

LOW

SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1718 );
1719 user.amount = user.amount.add(_amount);
1720 user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);
1721 emit Deposit(msg.sender, _pid, _amount);
1722 }
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1748 UserInfo storage user = userInfo[_pid][msg.sender];
1749 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1750 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1751 user.amount = 0;
1752 user.rewardDebt = 0;
```

LOW

SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1749 pool.lpToken.safeTransfer(address(msg.sender), user.amount);
1750 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1751 user.amount = 0;
1752 user.rewardDebt = 0;
1753 }
```

LOW

SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1750 emit EmergencyWithdraw(msg.sender, _pid, user.amount);
1751 user.amount = 0;
1752 user.rewardDebt = 0;
1753 }
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1677 return;
1678 }
1679 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1680 if (lpSupply == 0) {
1681 pool.lastRewardBlock = block.number;
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1708 uint256 pending =
1709 user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
1710 user.rewardDebt
1711 );
1712 safePunchTransfer(msg.sender, pending);
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1707 if (user.amount > 0) {
1708 uint256 pending =
1709 user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
1710 user.rewardDebt
1711 );
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1707 if (user.amount > 0) {
1708 uint256 pending =
1709 user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
1710 user.rewardDebt
1711 );
```

LOW

SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source file

contracts/MasterPunch.sol

Locations

```
1755 // Safe punch transfer function, just in case if rounding error causes pool to not have enough PUNCHs.
1756 function safePunchTransfer(address _to, uint256 _amount) internal {
1757 uint256 punchBal = punch.balanceOf(address(this));
1758 if (_amount > punchBal) {
1759 punch.transfer(_to, punchBal);
```

LOW

SWC-108

State variable visibility is not set.

It is best practice to set the visibility of state variables explicitly. The default visibility for "masterPunch" is internal. Other possible visibility settings are public and private.

Source file

contracts/MasterPunch.sol

Locations

```
829 address constant DEV2 = 0x4a286ce263A907d75d69A13F0cf0Eb205479D3C4;
830 address constant DUCK = 0x1c6a596F46dEA8A46A385d186C203259CF495dC5;
831 address masterPunch;
832
833 modifier onlyMasterPunch() {
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

contracts/MasterPunch.sol

Locations

```
1603  massUpdatePools();
1604  }
1605  uint256 lastRewardBlock = block.number;
1606  totalAllocPoint = totalAllocPoint.add(_allocPoint);
1607  poolInfo.push(
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

contracts/MasterPunch.sol

Locations

```
1649  uint256 accPunchPerShare = pool.accPunchPerShare;
1650  uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1651  if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1652  uint256 multiplier =
1653  getMultiplier(pool.lastRewardBlock, block.number);
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

contracts/MasterPunch.sol

Locations

```
1651  if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1652  uint256 multiplier =
1653  getMultiplier(pool.lastRewardBlock, block.number);
1654  uint256 punchReward =
1655  multiplier.mul(punchPerBlock).mul(pool.allocPoint).div(
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

contracts/MasterPunch.sol

Locations

```
1674 function updatePool(uint256 _pid) public {
1675 PoolInfo storage pool = poolInfo[_pid];
1676 if (block.number <= pool.lastRewardBlock) {
1677 return;
1678 }
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

contracts/MasterPunch.sol

Locations

```
1679 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1680 if (lpSupply == 0) {
1681 pool.lastRewardBlock = block.number;
1682 return;
1683 }
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

contracts/MasterPunch.sol

Locations

```
1682 return;
1683 }
1684 uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1685 uint256 punchReward =
1686 multiplier.mul(punchPerBlock).mul(pool.allocPoint).div(
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. 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.

Source file

`contracts/MasterPunch.sol`

Locations

```
1692  punchReward.mul(1e12).div(lpSupply)
1693  );
1694  pool.lastRewardBlock = block.number;
1695  }
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

contracts/MasterPunch.sol

Locations

```
1648 UserInfo storage user = userInfo[_pid][_user];
1649 uint256 accPunchPerShare = pool.accPunchPerShare;
1650 uint256 lpSupply = pool.lpToken.balanceOf(address(this));
1651 if (block.number > pool.lastRewardBlock && lpSupply != 0) {
1652 uint256 multiplier =
```

Source file

contracts/MasterPunch.sol

Locations

```
1521 //
1522 // Have fun reading it. Hopefully it's bug-free. God bless.
1523 contract MasterPunch is Ownable {
1524 using SafeMath for uint256;
1525 using SafeERC20 for IERC20;
1526
1527 address constant public BURN = 0x000000000000000000000000000000000000dEaD;
1528
1529 // Info of each user.
1530 struct UserInfo {
1531 uint256 amount; // How many LP tokens the user has provided.
1532 uint256 rewardDebt; // Reward debt. See explanation below.
1533 //
1534 // We do some fancy math here. Basically, any point in time, the amount of PUNCHs
1535 // entitled to a user but is pending to be distributed is:
1536 //
1537 // pending reward = (user.amount * pool.accPunchPerShare) - user.rewardDebt
1538 //
1539 // Whenever a user deposits or withdraws LP tokens to a pool. Here's what happens:
1540 // 1. The pool's `accPunchPerShare` (and `lastRewardBlock`) gets updated.
1541 // 2. User receives the pending reward sent to his/her address.
1542 // 3. User's `amount` gets updated.
1543 // 4. User's `rewardDebt` gets updated.
1544 }
1545 // Info of each pool.
1546 struct PoolInfo {
1547 IERC20 lpToken; // Address of LP token contract.
1548 uint256 allocPoint; // How many allocation points assigned to this pool. PUNCHs to distribute per block.
1549 uint256 lastRewardBlock; // Last block number that PUNCHs distribution occurs.
1550 uint256 accPunchPerShare; // Accumulated PUNCHs per share, times 1e12. See below.
1551 }
1552 // The PUNCH TOKEN!
1553 Punch public punch;
1554 // Dev address.
1555 address public devaddr;
1556 // PUNCH tokens created per block.
1557 uint256 public punchPerBlock;
1558 // Bonus muliplier for early punch makers.
1559 uint256 public constant BONUS_MULTIPLIER = 10;
1560 // Info of each pool.
1561 PoolInfo[] public poolInfo;
1562 // Info of each user that stakes LP tokens.
1563 mapping(uint256 => mapping(address => UserInfo)) public userInfo;
1564
```

```
// Total allocation poitns. Must be the sum of all allocation points in all pools.
uint256 public totalAllocPoint = 0;
event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
event EmergencyWithdraw(
address indexed user,
uint256 indexed pid,
uint256 amount
);

constructor(
Punch _punch,
address _devaddr,
uint256 _punchPerBlock
) {
punch = _punch;
devaddr = _devaddr;
punchPerBlock = _punchPerBlock;
}

function setPunchPerBlock(uint _punchPerBlock, bool _withUpdate) external onlyOwner {
if (_withUpdate) {
massUpdatePools();
}
punchPerBlock = _punchPerBlock;
}

function poolLength() external view returns (uint256) {
return poolInfo.length;
}

// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(
uint256 _allocPoint,
IERC20 _lpToken,
bool _withUpdate
) public onlyOwner {
if (_withUpdate) {
massUpdatePools();
}
uint256 lastRewardBlock = block.number;
totalAllocPoint = totalAllocPoint.add(_allocPoint);
poolInfo.push(
PoolInfo({
lpToken: _lpToken,
allocPoint: _allocPoint,
lastRewardBlock: lastRewardBlock,
accPunchPerShare: 0
})
);
}

// Update the given pool's PUNCH allocation point. Can only be called by the owner.
function set(
uint256 _pid,
uint256 _allocPoint,
bool _withUpdate
) public onlyOwner {
if (_withUpdate) {
massUpdatePools();
```

```
}
totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
_allocPoint
);
poolInfo[_pid].allocPoint = _allocPoint;
}

// Return reward multiplier over the given _from to _to block.
function getMultiplier(uint256 _from, uint256 _to)
public
pure
returns (uint256)
{
return _to.sub(_from);
}

// View function to see pending PUNCHs on frontend.
function pendingPunch(uint256 _pid, address _user)
external
view
returns (uint256)
{
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][_user];
uint256 accPunchPerShare = pool.accPunchPerShare;
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (block.number > pool.lastRewardBlock && lpSupply != 0) {
uint256 multiplier =
getMultiplier(pool.lastRewardBlock, block.number);
uint256 punchReward =
multiplier.mul(punchPerBlock).mul(pool.allocPoint).div(
totalAllocPoint
);
accPunchPerShare = accPunchPerShare.add(
punchReward.mul(1e12).div(lpSupply)
);
}
return user.amount.mul(accPunchPerShare).div(1e12).sub(user.rewardDebt);
}

// Update reward vairables for all pools. Be careful of gas spending!
function massUpdatePools() public {
uint256 length = poolInfo.length;
for (uint256 pid = 0; pid < length; ++pid) {
updatePool(pid);
}
}

// Update reward variables of the given pool to be up-to-date.
function updatePool(uint256 _pid) public {
PoolInfo storage pool = poolInfo[_pid];
if (block.number <= pool.lastRewardBlock) {
return;
}
uint256 lpSupply = pool.lpToken.balanceOf(address(this));
if (lpSupply == 0) {
pool.lastRewardBlock = block.number;
return;
}
uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
uint256 punchReward =

```

```
multiplier.mul(punchPerBlock).mul(pool.allocPoint).div(
totalAllocPoint
);
punch.mint(devaddr, punchReward.div(10));
punch.mint(address(this), punchReward);
pool.accPunchPerShare = pool.accPunchPerShare.add(
punchReward.mul(1e12).div(lpSupply)
);
pool.lastRewardBlock = block.number;
}

// Deposit LP tokens to MasterPunch for PUNCH allocation.
function deposit(uint256 _pid, uint256 _amount) public {
PoolInfo storage pool = poolInfo[_pid];

uint sacrifice = _amount.div(1000);
pool.lpToken.safeTransferFrom(address(msg.sender), address(BURN), sacrifice);
_amount = _amount.sub(sacrifice);

UserInfo storage user = userInfo[_pid][msg.sender];
updatePool(_pid);
if (user.amount > 0) {
uint256 pending =
user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
user.rewardDebt
);
safePunchTransfer(msg.sender, pending);
}
pool.lpToken.safeTransferFrom(
address(msg.sender),
address(this),
_amount
);
user.amount = user.amount.add(_amount);
user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);
emit Deposit(msg.sender, _pid, _amount);
}

// Withdraw LP tokens from MasterPunch.
function withdraw(uint256 _pid, uint256 _amount) public {
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
require(user.amount >= _amount, "withdraw: not good");
updatePool(_pid);
uint256 pending =
user.amount.mul(pool.accPunchPerShare).div(1e12).sub(
user.rewardDebt
);
safePunchTransfer(msg.sender, pending);
user.amount = user.amount.sub(_amount);
user.rewardDebt = user.amount.mul(pool.accPunchPerShare).div(1e12);

uint sacrifice = _amount.div(1000);
pool.lpToken.safeTransfer(address(BURN), sacrifice);
pool.lpToken.safeTransfer(address(msg.sender), _amount.sub(sacrifice));

emit Withdraw(msg.sender, _pid, _amount.sub(sacrifice));
}

// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public {

```

```
PoolInfo storage pool = poolInfo[_pid];
UserInfo storage user = userInfo[_pid][msg.sender];
pool.lpToken.safeTransfer(address(msg.sender), user.amount);
emit EmergencyWithdraw(msg.sender, _pid, user.amount);
user.amount = 0;
user.rewardDebt = 0;
}

// Safe punch transfer function, just in case if rounding error causes pool to not have enough PUNCHs.
function safePunchTransfer(address _to, uint256 _amount) internal {
uint256 punchBal = punch.balanceOf(address(this));
if (_amount > punchBal) {
punch.transfer(_to, punchBal);
} else {
punch.transfer(_to, _amount);
}
}

// Update dev address by the previous dev.
function dev(address _devaddr) public {
require(msg.sender == devaddr, "dev: wut?");
devaddr = _devaddr;
}
}
```