

# SMART CONTRACT

---

## Security Audit Report

Project: PayPal USD (PYUSD)

Website: [www.paypal.com](http://www.paypal.com)

Platform: Ethereum

Language: Solidity

Date: May 11th, 2026

# Table of contents

Introduction .....	4
Project Background .....	4
Audit Scope .....	5
Claimed Smart Contract Features .....	6
Audit Summary .....	7
Technical Quick Stats .....	8
Code Quality .....	9
Documentation .....	9
Use of Dependencies .....	9
AS-IS overview .....	10
Severity Definitions .....	11
Audit Findings .....	12
Conclusion .....	14
Our Methodology .....	15
Disclaimers .....	17
Appendix	
• Code Flow Diagram .....	18
• Slither Results Log .....	19
• Solidity static analysis .....	20
• Solhint Linter .....	21

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

# Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the smart contracts of PayPal USD (PYUSD) from paypal.com were audited. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on May 11th, 2026.

## The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

# Project Background

The contracts implement **upgradeable proxy patterns** with administrative control for smart contract systems. This allows changing the logic (implementation) of a deployed contract while preserving the same storage and address. The solution ensures modularity, maintainability, and secure admin-controlled upgrades.

## Main Contracts:

1. Proxy – Base contract for delegating calls.
2. UpgradeabilityProxy – Allows changing the implementation address.
3. AdminUpgradeabilityProxy – Adds admin-only access for upgrades and management.
4. AddressUtils – Utility to check if an address is a contract.

## Audit scope

<b>Name</b>	<b>Code Review and Security Analysis Report for PayPal USD (PYUSD) Smart Contract</b>
<b>Platform</b>	<b>Ethereum</b>
<b>Language</b>	<b>Solidity</b>
<b>File</b>	AdminUpgradeabilityProxy.sol
<b>Smart Contract Code</b>	<a href="#">0x6c3ea9036406852006290770bedfcaba0e23a0e8</a>
<b>Audit Date</b>	May 11th, 2026

## Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
<p><b><u>Key Features of the AdminUpgradeabilityProxy contract:</u></b></p> <ul style="list-style-type: none"><li>● <b>Delegation &amp; Proxy:</b> Forward all calls and return data to implementation contract (Proxy).</li><li>● <b>Upgradeable Logic:</b> Swap implementation contract without changing storage (UpgradeabilityProxy).</li><li>● <b>Admin Control:</b> Only admin can upgrade or change proxy settings (AdminUpgradeabilityProxy).</li><li>● <b>Secure Fallback:</b> Non-admin calls are routed to implementation automatically.</li><li>● <b>Contract Verification:</b> Ensures only contract addresses are set as implementations (AddressUtils.isContract).</li></ul>	<p><b>YES, This is valid.</b></p>

# Audit Summary

According to the standard audit assessment, the Customer`s solidity smart contracts are **“Secured”**. Also, these contracts contain owner control, which does not make them fully decentralized.



We used various tools like Slither, Solhint and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all identified issues can be found in the Audit overview section.

**We found 0 critical, 0 high, 0 medium, 0 low, and 1 very low-level issue.**

**Investors Advice:** Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

## Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	"Out of Gas" Issue	Passed
	High consumption 'for/while' loop	Passed
	High consumption 'storage' storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

**Overall Audit Result: PASSED**

## Code Quality

This audit scope has 1 smart contract. Smart contracts contain Libraries, Smart contracts, inherits, and Interfaces. This is a compact and well-written smart contract.

The libraries in PayPal USD are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties/methods can be reused many times by other contracts in the PayPal USD.

The EtherAuthority team has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

## Documentation

We were given a PayPal USD smart contract code in the form of an [Etherscan](#) web link.

As mentioned above, code parts are well commented on. And the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

## Use of Dependencies

As per our observation, the libraries used in this smart contract infrastructure are based on well-known industry standard open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

# AS-IS overview

## Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	ifAdmin	modifier	Passed	No Issue
3	admin	external	ifAdmin	No Issue
4	implementation	external	ifAdmin	No Issue
5	changeAdmin	external	ifAdmin	No Issue
6	upgradeTo	external	ifAdmin	No Issue
7	upgradeToAndCall	external	ifAdmin	No Issue
8	admin	internal	Passed	No Issue
9	setAdmin	internal	Passed	No Issue
10	_willFallback	internal	Passed	No Issue
11	implementation	internal	Passed	No Issue
12	upgradeTo	internal	Passed	No Issue
13	setImplementation	write	Passed	No Issue

## Severity Definitions

Risk Level	Description
<b>Critical</b>	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
<b>High</b>	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial
<b>Medium</b>	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
<b>Low</b>	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, that can't have a significant impact on execution
<b>Lowest / Code Style / Best Practice</b>	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

# Audit Findings

## Critical Severity

No Critical severity vulnerabilities were found.

## High Severity

No High severity vulnerabilities were found.

## Medium

No Medium-severity vulnerabilities were found.

## Low

No Low severity vulnerabilities were found.

## Very Low / Informational / Best practices:

(1) Use the latest solidity version:

```
pragma solidity ^0.4.24;
```

Use the latest solidity version while contract deployment to prevent any compiler version-level bugs.

**Resolution:** Please use the latest solidity versions.(0.8.34)

## Centralization Risk

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet's private key would be compromised, then it would create trouble.

The following are Admin functions:

- `admin`:Returns current proxy admin address.
- `implementation`:Returns current implementation contract address.
- `changeAdmin`:Transfers proxy admin rights to a new address.
- `upgradeTo`:Updates proxy to use a new implementation contract.
- `upgradeToAndCall`:Upgrades implementation and immediately executes a function call on new implementation.

## Conclusion

We were given a contract code in the form of [Etherscan](#) web links. And we have used all possible tests based on given objects as files. We observed 1 informational issue in the smart contracts. And this issue is not critical. So, **it's good to go for the production.**

Since possible test cases can be unlimited for such smart contracts protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on standard audit procedure scope, is **"Secured"**.

# Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

## **Manual Code Review:**

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

## **Vulnerability Analysis:**

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

## **Documenting Results:**

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

## **Suggested Solutions:**

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

# Disclaimers

## EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

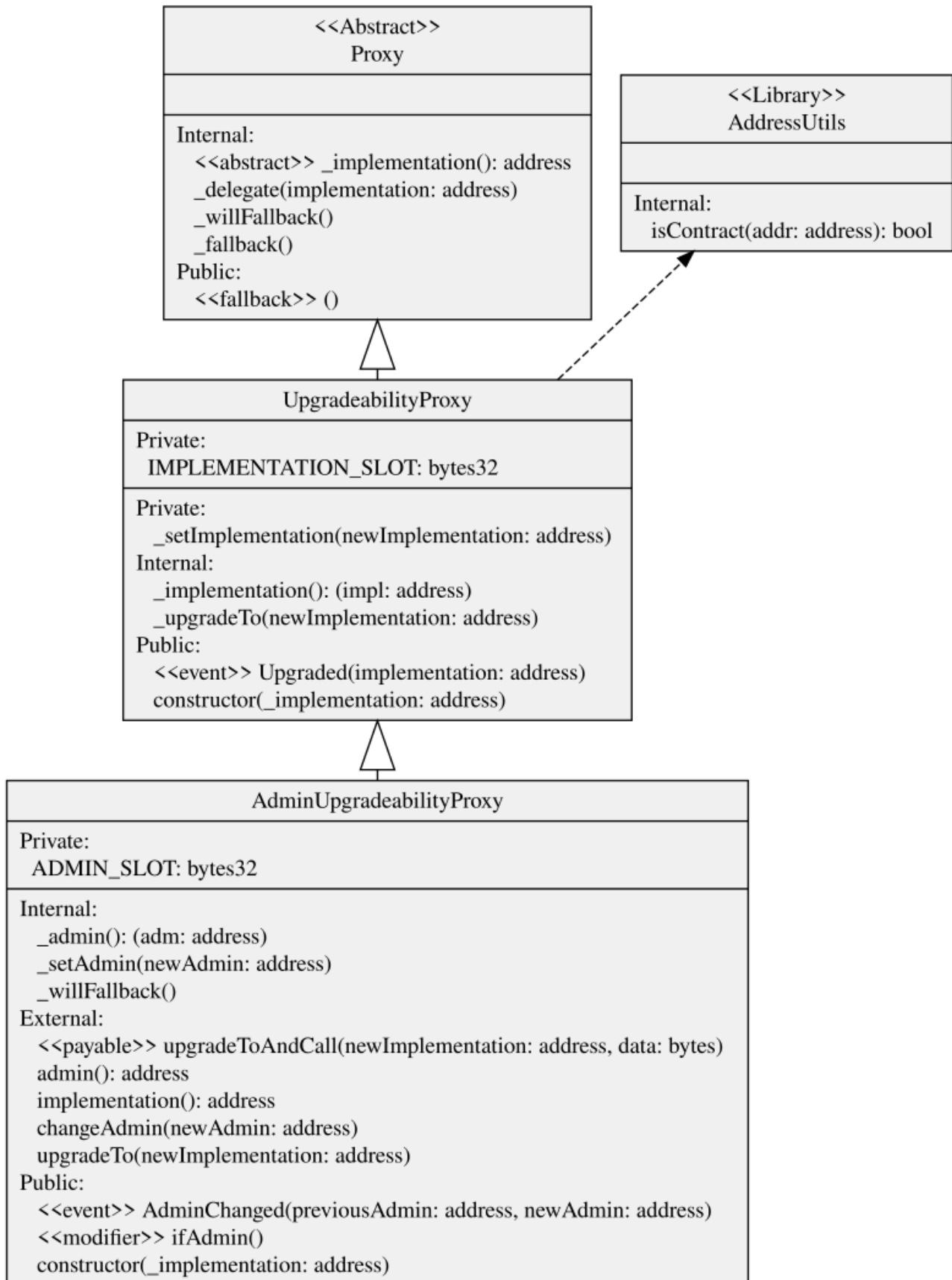
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

## Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

# Appendix

## Code Flow Diagram - PayPal USD (PYUSD)



## Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

### Slither Log >> AdminUpgradeabilityProxy.sol

```
INFO:Detectors:
UpgradeabilityProxy._setImplementation(address).newImplementation is
not checked for zero address
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation

INFO:Detectors:
AdminUpgradeabilityProxy.upgradeToAndCall(address,bytes) uses low-level
call:
- address(this).call.value(msg.value)(data)

INFO:Detectors:
Proxy._delegate(address) uses delegatecall:
- delegatecall(gas, implementation, 0, calldatasize, 0, 0)

INFO:Detectors:
Assembly usage found:
- Proxy._delegate(address)
- AddressUtils.isContract(address)
- UpgradeabilityProxy._implementation()
- UpgradeabilityProxy._setImplementation(address)
- AdminUpgradeabilityProxy._admin()
- AdminUpgradeabilityProxy._setAdmin(address)

INFO:Detectors:
Solidity version ^0.4.24 is old and may contain known compiler issues.
```

# Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

## AdminUpgradeabilityProxy.sol

### 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.

Pos: 292:8:

### Low level calls:

Use of "call": should be avoided whenever possible. It can lead to unexpected behavior if the return value is not handled properly. Please use Direct Calls via specifying the called contract's interface.

Pos: 272:16:

### Gas costs:

The gas requirement of the function

AdminUpgradeabilityProxy.upgradeToAndCall is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)

Pos: 270:4:

### Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

Pos: 164:8:

## Solhint Linter

Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

### AdminUpgradeabilityProxy.sol

```
Found more than One contract per file. 4 contracts found!
Pos: 1:2
Compiler version ^0.4.24 does not satisfy the ^0.8.0 semver
requirement
Pos: 1:2
Visibility modifier must be first in list of modifiers
Pos: 25:16
Error message for require is too long: 59 counted / 32 allowed
Pos: 9:163
Use Custom Errors instead of require statements
Pos: 9:163
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:167
Error message for require is too long: 54 counted / 32 allowed
Pos: 9:245
Use Custom Errors instead of require statements
Pos: 9:245
Visibility modifier must be first in list of modifiers
Pos: 78:269
Provide an error message for require
Pos: 9:271
Use Custom Errors instead of require statements
Pos: 9:271
Avoid using low level calls.
Pos: 17:271
Avoid to use ".call.value()()"
Pos: 17:271
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:279
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:291
Error message for require is too long: 50 counted / 32 allowed
Pos: 9:300
Use Custom Errors instead of require statements
Pos: 9:300
```



This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of EtherAuthority.

**Email: [audit@EtherAuthority.io](mailto:audit@EtherAuthority.io)**