PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For WINR Protocol

20 February 2023

paladinsec.co info@paladinsec.co

Table of Contents

Disclaimer

1 Overview

This report has been prepared for WINR Protocol on the Arbitrum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	WINR Protocol
URL	https://winr.games/
Platform	Arbitrum
Language	Solidity
Preliminary Contracts	https://github.com/JustbSerbia/winr-protocol/tree/a202240a729538f7681b9cf5c707717fc3aaa1cf
Final Contracts	https://github.com/WINRLabs/winr-protocol/tree/985b2b5a37eb41c5538dacf1950810c375e28166

1.2 Contracts Assessed

Name	Contract	Live Code Match
Vesting		
Winr		
VestedWinr		
DateTime		
RoleBasedAccessControl		
Governable		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
🔴 High	0	-	-	-
🟠 Medium	1	1	-	-
🟡 Low	10	8	1	1
🟣 Informational	15	14	1	-
Total	**26**	**23**	**2**	**1**

Classification of Issues

Severity	Description
🔴 **High**	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
🟠 **Medium**	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
🟡 **Low**	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
🟣 **Informational**	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 Global Issues

ID	Severity	Summary	Status
01	LOW	Governance risk: The tokens can be minted by governance wallets up to the configured maximum caps	PARTIAL

1.3.2 Vesting

ID	Severity	Summary	Status
02	MEDIUM	The same investor can be added twice by governance which will cause supply to be locked away	✓ RESOLVED
03	LOW	Investor start and end times do not align with the actual vest start and end times	✓ RESOLVED
04	LOW	Lack of explicit maximum supply handling can cause certain vesters to never receive their tokens	✓ RESOLVED
05	LOW	Recurring rewards are sometimes slightly reduced due to unnecessary rounding	✓ RESOLVED
06	INFO	Typographical errors	✓ RESOLVED
07	INFO	Gas optimizations	PARTIAL
08	INFO	Lack of validation	✓ RESOLVED

1.3.3 Winr

ID	Severity	Summary	Status
09	LOW	`BURNER_ROLE` seems to impose excessive privileges as users should not need a privilege to burn their own tokens	✓ RESOLVED
10	INFO	Typographical errors	✓ RESOLVED
11	INFO	`mint` will implicitly not mint any tokens once the maximum supply has been reached, which could lead to bugs within contracts which do not handle this case	✓ RESOLVED
12	INFO	The amount of tokens that can be minted can exceed `MAX_SUPPLY` as the cap does not take into consideration that burned tokens are removed from the supply	✓ RESOLVED
13	INFO	Initialization of `DEFAULT_ADMIN_ROLE` is redundant with Governable	✓ RESOLVED

1.3.4 VestedWinr

ID	Severity	Summary	Status
14	LOW	`BURNER_ROLE` seems to impose excessive privileges as users should not need a privilege to burn their own tokens	✓ RESOLVED
15	INFO	Typographical errors	✓ RESOLVED
16	INFO	`mint` will implicitly not mint any tokens once the maximum supply has been reached, which could lead to bugs within contracts which do not handle this case	✓ RESOLVED
17	INFO	The amount of tokens that can be minted can exceed `MAX_SUPPLY` as the cap does not take into consideration that burned tokens are removed from the supply	✓ RESOLVED
18	INFO	Initialization of `DEFAULT_ADMIN_ROLE` is redundant with Governable	✓ RESOLVED

1.3.5 DateTime

ID	Severity	Summary	Status
19	INFO	`DateTime` days will eventually start shifting compared to calendar days	✓ RESOLVED
20	INFO	Typographical errors	✓ RESOLVED

1.3.6 RoleBasedAccessControl

ID	Severity	Summary	Status
21	LOW	Lack of enumerability on role holders makes it difficult for people to inspect which accounts have privileges	✓ RESOLVED

1.3.7 Governable

ID	Severity	Summary	Status
22	LOW	Using the `DEFAULT_ADMIN_ROLE` for governance is a role management risk as this role wields much more power than a custom role	ACKNOWLEDGED
23	LOW	Governance can accidentally renounce their role by calling setGov to their own address	✓ RESOLVED
24	LOW	Inconsistent usage of `msgSender()` and `msg.sender`	✓ RESOLVED
25	INFO	OpenZeppelin's `_setupRole` is deprecated in favour of `_grantRole`	✓ RESOLVED
26	INFO	Gas optimization	✓ RESOLVED

2 Findings

2.1 Global Issues

The issues listed in this section apply to the protocol as a whole.

2.1.1 Issues & Recommendations

Issue #01	Governance risk: The tokens can be minted by governance wallets up to the configured maximum caps
Severity	● LOW SEVERITY
Description	Both the `Winr` and `VestedWinr` tokens can be minted by any governance wallet which has the `MINTER_ROLE`. This means that if the governance team decides to give this role to EOAs (normal wallets), there is a risk that these wallets end up in the wrong hands who can start over-minting and dumping the two tokens. It is crucial for the team to be extremely prudent with who has access to the minter role. Additionally, both of these tokens define an emergency feature to burn tokens on any address. This feature can solely be called by wallets with the `BURNER_ROLE`. Although this seems less risky, it in fact has the same impact as the `MINTER_ROLE` as the burner can burn all tokens from the liquidity pool and then drain the pool with a small swap.
Recommendation	Consider being extremely prudent with the roles: the `DEFAULT_ADMIN_ROLE` should be strictly wielded by a single well-defined and multi-party multi-signature wallet (ideally at least a 3 independent party minimum quorum). The `MINTER_ROLE` should ideally be solely wielded by clearly defined smart contracts with strict emission rates, or not at all. The `BURNER_ROLE` should remain undefined until needed, or be solely in the hands of the multisig and smart contracts. The tokens should finally also use `AccessControlEnumerable` instead of `AccessControl` to allow for users to more easily inspect who has these roles.

Resolution	● PARTIALLY RESOLVED The team understands this concern. This issue will be updated after deployment once the team moves to an adequate governance setup. The team has resolved some portions of this issue directly: - `BURNER_ROLE` has been removed - `AccessControlEnumerable` has been added

2.2 Vesting

`Vesting` defines all of the initial vesting schedules for the Winr and VestedWinr tokens. The contract defines various schedules:

- **Winr Labs**
  - Start: After 180 days
  - Duration: 3 years
  - Allocation: 150m Winr
- **Marketing**
  - Start: Instant
  - Duration: 2 years
  - Allocation: 70m Winr
- **Advisors**
  - Start: Instant
  - Duration: 3 years
  - Allocation: 15m Winr
- **Previous Holders**
  - Start: Instant
  - Duration: 2 years
  - Allocation: 10m Winr
- **Core Contributors**
  - Start: Instant
  - Duration: 2 years
  - Allocation: 175m Winr

The contract governance can at any point in time add in new vesting wallets using the `addInvestor` and `addInvestorBatch` functions. However, they cannot allocate more tokens than the aforementioned allocations to these investors, as the Vesting contract keeps track of the total allocated amounts and reverts any investor additions which would cause the allocation to exceed the cap.

Once wallets have been added to the different categories, these wallets can call `withdrawTokens` periodically to claim their vested tokens. With any claim, the tokens will get minted as either `Winr` or `VestedWinr` depending on the investor's configuration by the governance, which means that the Vesting contract has minting privileges. Tokens unlock on a day-by-day basis (every 24 hours) at the exact same time for everyone. Tokens also start vesting at the exact same time for everyone, starting from the (one-time) governance-configurable initial timestamp.

There is also a `recover` function which allows the contract owner to recover any tokens that may have been sent to the contract by mistake. Note that Winr and VestedWinr tokens are minted directly to investors and are therefore not at risk.

2.2.1 Privileged Functions

- `setInitialTimestamp [ DEFAULT_ADMIN_ROLE / Configurable once ]`
- `addInvestorBatch [ DEFAULT_ADMIN_ROLE ]`
- `addInvestor [ DEFAULT_ADMIN_ROLE ]`
- `removeInvestor [ DEFAULT_ADMIN_ROLE ]`
- `withdrawTokens [ vesters ]`
- `recoverToken [ DEFAULT_ADMIN_ROLE ]`
- `setGov [ DEFAULT_ADMIN_ROLE ]`
- `grantRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `renounceRole [ anyone with a role ]`

2.2.2 Issues & Recommendations

Issue #02	The same investor can be added twice by governance which will cause supply to be locked away
Severity	● MEDIUM SEVERITY
Description	The Vesting contract defines a number of categories, where each category has a limited allocation for investors. Any time an investor is added, a part of this supply is marked as allocated and cannot be re-allocated. However, the vesting contract has an error as it allows the same investor to be registered multiple times but with each registration, this investor's vest resets to the new configuration. Another side effect of this, which does not necessarily need to be fixed, is that each investor address can only subscribe to one category. If an investor is a part of multiple categories, they will need to use multiple addresses.
Recommendation	Consider checking that the investor does not exist yet within `addInvestor`. The secondary side-effect does not need to be resolved if it is acceptable.
Resolution	✓ RESOLVED The recommended check has been added.

Issue #03	Investor start and end times do not align with the actual vest start and end times
Severity	● LOW SEVERITY
Description	The contract exposes `view` functions which allow the frontend to display when an investor's vest starts and ends. However, this does not align with the actual vesting business logic as the actual business logic uses the initial timestamp instead of the vest creation timestamp as the "start time".
Recommendation	Consider removing these two variables as they might be confusing.
Resolution	✓ RESOLVED The variables have been removed.

Issue #04	Lack of explicit maximum supply handling can cause certain vesters to never receive their tokens
Severity	● LOW SEVERITY
Description	Both tokens have a maximum supply. If this supply is reached while vesters do not have their tokens yet, these vesters could never receive their tokens.
Recommendation	Consider carefully taking this into account with setting the emissions of other system components. The contract could also pre-mint its whole supply but this might be seen as a tokenomical downside as certain applications would show this value as the circulating supply.
Resolution	✓ RESOLVED The client has indicated they will take this into account with a management contract in the future.

Issue #05	Recurring rewards are sometimes slightly reduced due to unnecessary rounding
Severity	● LOW SEVERITY

Location	Line 311-316
	`uint256 everyRecurrenceReleaseAmount = vestingDistroAmount /`
	`    investor.categoryDetail.recurrence;`
	
	`[...]`
	
	`uint256 vestingUnlockedAmount = occurence *`
	`everyRecurrenceReleaseAmount;`

Description	The reward logic does a division before multiplication at some point, which causes a bit of precision to be lost due to Solidity strictly using integers.

Recommendation	Consider using the mul-div pattern:
	
	`uint256 vestingUnlockedAmount = occurence *`
	`vestingDistroAmount /`
	`    investor.categoryDetail.recurrence;`

Resolution	✓ RESOLVED

Issue #06	**Typographical errors**
Severity	● INFORMATIONAL

Description	We have consolidated the typographical issues into a single issue to keep the report brief and readable.
	
	Line 4
	`import “@openzeppelin/contracts/governance/TimelockController.sol";`
	
	Line 10
	`import "hardhat/console.sol";`
	
	The imports are unused and can be removed.
	
	Line 7
	`import "../../interfaces/tokens/IWINR.sol";`
	
	An `IMintable` interface would be cleaner here.
	
	Line 56
	`uint256 private _initialTimestamp;`
	
	This variable should be marked as `public` to improve readability. Both `_initialTimestamp` and `_totalAllocatedAmount` can then be renamed without an underscore as is best practice for public variables.
	
	Lines 57-58
	`IWINR public WINR;`
	`IWINR public vWINR;`
	
	Line 59
	`investors`
	
	An `investorsLength()` function should be added.

Line 95

```
/// @dev Checks that the contract is initialized.
```

This function in fact checks that the contract has not yet been initialized.

Line 106

```
constructor(address _Winr, address _vWinr, address _gov)
Governable(_gov) {
```

The two first arguments can be marked as IWINR to avoid casting them later on.

Line 176

```
/// @param _timestamp The initial timestamp, this timestap
should be used for vesting
```

This should say “timestamp”.

Line 211

```
function addInvestor(
```

The comments above this function appear to be outdated.

Line 230

```
(_tokensAllotment *
(categoryDetails[_category]._initialUnlockPercentage)) /
```

This line contains unnecessary sets of brackets. Especially the inner one seems rather silly, the outer one might help with readability.

Line 241

```
_totalAllocatedAmount = _totalAllocatedAmount +
_tokensAllotment;
```

It might be cleaner to use +=.

Line 253

```
investor.tokensAllotment >= tokensAvailable,
```

This should likely be an assertion as it seems to be unfailable within the current contract.

Line 254

```
"You can't take withdraw more than allocation"
```

Consider rewording this sentence to "You can't withdraw more than your allocation".

Line 289

```
investor.withdrawnTokens < investor.tokensAllotment
```

This requirement should likely be an early return which simply returns the `tokensAllotment` early.

Line 327

```
function recoverToken(address _token, uint256 amount)
external onlyRole(DEFAULT_ADMIN_ROLE) {
```

`_token` can be provided as `IERC20` to avoid having to cast it explicitly later on.

Finally, many variables are accessed multiple times from storage within the withdrawal logic — these variables could be cached to save gas.

We do understand that the client wants to keep their contract as is with minimal changes. Acknowledging these gas optimizations is fine as the old contract has apparently run in production in the past and optimizing gas is therefore not necessarily more beneficial than having a production tested contract.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

Issue #07	Gas optimizations
Severity	● INFORMATIONAL

Description	The whole contract is extremely gas inefficient with regards to storage usage. Several immutables are unnecessarily stored within storage, and are often redundantly stored. Ideally, not a single immutable variable needs to be stored in storage. Instead, pure function can be used to store the vesting type and category information. A simple approach could be a `function categoryInformation(Category category)` which returns the `CategoryDetails` struct. In our opinion, `VestingType` is redundant and can simply be an integer representing the frequency although it is always set to daily within this contract.
	
	<u>`Line 53`</u>
	`event RecoverToken(address indexed token, uint256 indexed amount);`
	
	Indexing amounts has no benefit. Consider removing the secondary index to save gas.
	
	<u>`Lines 73-75`</u>
	`uint256 afterCliffUnlockAmount;`
	`uint256 startTime;`
	`uint256 endTime;`
	
	These variables are all no longer in use. `afterCliffUnlockAmount` is never set, while the `startTime` and `endTime` do not actually relate to actionable business logic as the `_initialTimestamp` variable is used consistently as the actual `startTime`. All three should therefore in our opinion probably be removed to save gas.
	
	<u>`Line 77`</u>
	`CategoryDetail categoryDetail;`
	
	This is already stored within a mapping and gas is wasted to store it redundantly as it can be easily fetched.

	Lines 189-191
	`address[] memory _investors,`
	`uint256[] memory _tokensAllotments,`
	`Category[] memory _categories`
	
	All arrays can be marked as `calldata` to save gas.
	
	Line 215
	`) public onlyRole(DEFAULT_ADMIN_ROLE) {`
	
	This role check is redundant for the batch function. Consider using an internal function which is called by both external functions instead. The role check is then only done on the external functions.

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	● PARTIALLY RESOLVED Some of these recommendations have been implemented.

Issue #08	**Lack of validation**
Severity	● INFORMATIONAL
Location	Line 177 `function setInitialTimestamp(`
Description	Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase. It would be wise to validate that the initial timestamp is in the future to avoid an accidental unlocking of all tokens.
Recommendation	Consider validating the function parameters mentioned above.
Resolution	✓ RESOLVED The timestamp must now be in the future.

2.3 Winr

`Winr` is the main ERC20 token within the WINR Protocol. It extends the Governable dependency (see its section lower within this report) for governance functionality. Tokens can be minted by any governance address with the `MINTER_ROLE`.

The maximum supply of the token is determined during deployment and is stored as a constant state variable called `MAX_SUPPLY` which can be inspected by users. The amount of tokens in circulation cannot exceed `MAX_SUPPLY`.

Finally, the contract allows governance addresses with the `BURNER_ROLE` to also burn Winr tokens from any address. We have spoken about this to the team and they explained that they gave themselves this priviledge in case of emergencies, though they plan to have various tokenomical reasons to also burn tokens like "buyback and burn" and certain types of sales with the requirement that the purchasor burn their Winr.

2.3.1 Privileged Functions

- `mint [ MINTER_ROLE ]`
- `burn [ BURNER_ROLE ]`
- `setGov [ DEFAULT_ADMIN_ROLE ]`
- `grantRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `renounceRole [ anyone with a role ]`

2.3.2 Issues & Recommendations

Issue #09	**BURNER_ROLE seems to impose excessive privileges as users should not need a privilege to burn their own tokens**
Severity	● LOW SEVERITY
Description	The only way to burn one's Winr tokens is by acquiring the governance `BURNER_ROLE`. This means that contracts whose only purpose is to burn their own Winr balance will have this role. However, it would be much cleaner if there was an alternative function `burn(uint256 amount)` which allows a user to burn their own balance without the required priviledged role. This would reduce the number of wallets which can potentially maliciously burn tokens from users without their consent, as the BURNER_ROLE is exceptionally powerful in that regard and could even be used to drain the LP pair (by burning most of the Winr within this pair and then swapping out all the actually valuable tokens inside of the pair.
Recommendation	Consider being very strict with who receives the burner (and minter) roles as they are both equally risky from a governance risk perspective. Consider tightening the privileges as follows: - `burn(uint256 amount)` function to burn your own tokens without any role. - `BURNER_ROLE` is not granted on deployment if `_admin` is an EOA. Burner and minter roles should strictly never be granted to EOAs. - `burnFrom` can work without a role if `msg.sender` has received allowance from the from wallet — it might make sense to incorporate such behavior into `burnFrom` to avoid needing the `BURNER_ROLE` even further outside of emergencies. By using allowances, that role would be almost never needed. This is by far the recommended pattern.
Resolution	✓ RESOLVED The `BURNER_ROLE` has been removed in favour of the recommended changes. Governance can no longer burn any tokens from wallets other than their own.

Issue #10	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated the typographical issues into a single issue to keep the report brief and readable.
	
	Line 6
	`import "@openzeppelin/contracts/token/ERC20/IERC20.sol";`
	
	This import appears unused and redundant. It can therefore be removed.
	
	Lines 11-12
	`event Mint(address to, uint256 amount);`
	`event Burn(address from, uint256 amount);`
	
	The addresses within these events can be indexed to allow for more easy off-chain lookups.
	
	Lines 30 and 38
	`function mint(address account, uint256 amount) external virtual onlyRole(MINTER_ROLE) {`
	`function burn(address from, uint256 amount) external virtual onlyRole(BURNER_ROLE) {`
	
	The virtual keyword seems to be unnecessary within these function definitions.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

Issue #11	**`mint` will implicitly not mint any tokens once the maximum supply has been reached, which could lead to bugs within contracts which do not handle this case**
Severity	● INFORMATIONAL
Description	Once the maximum supply has been reached, `mint` will continue functioning. This has its advantages as any location calling `mint` will not start reverting suddenly. We have seen many bugs in contracts by our other clients where `mint` suddenly reverts and the reward contract's withdrawal method breaks. The Winr team envisioned this by not having mint revert, preventing such cases. We commend them for this. However, by not returning any information about whether the mint succeeded or failed, the rewarder contracts would have trouble figuring out what actually happened and whether they received any tokens. Such a function is therefore extremely prone to causing its calling contracts to forget to handle the case where the maximum supply is reached.
Recommendation	Consider instead either returning a boolean about whether any mint occurred or not, or returning the actually minted amount. The latter case could allow you to mint and return the "remaining" supply if not enough supply remains. Either works equally well from our perspective and is a subjective choice.
Resolution	✓ RESOLVED The data is now returned.

Issue #12	**The amount of tokens that can be minted can exceed `MAX_SUPPLY` as the cap does not take into consideration that burned tokens are removed from the supply**
Severity	● INFORMATIONAL
Description	The Winr token defines a maximum supply. However, in theory, more tokens than this cap can be minted over time as tokens will be taken out of circulation again through burning. This might be all right depending on the tokenomics of the project, therefore this issue has been rated as informational. Specifically, the following could occur: 1. The maximum supply is minted 2. 100 tokens are burned by the project 3. 100 tokens can be minted again This would result in a total minted supply of maximum supply + 100, as the cap only checks the circulating supply.
Recommendation	Consider whether this is desired. If not, consider keeping track of the total number of minted tokens and basing the cap on this number instead of the `totalSupply()`.
Resolution	✓ RESOLVED The `MAX_SUPPLY` now decreases on burns.

Issue #13	Initialization of `DEFAULT_ADMIN_ROLE` is redundant with Governable
Severity	● INFORMATIONAL
Description	During initialization of the WINR contract, `DEFAULT_ADMIN_ROLE` is granted to the `_admin`. This however already occurs during the initialization of the Governable library which this contract extends. The admin role is therefore unnecessarily granted twice which wastes gas and makes the contract more verbose.
Recommendation	Consider removing the `_setupRole` call within this contract while leaving it in the Governable contract.
Resolution	✓ RESOLVED

2.4 VestedWinr

`VestedWinr` is the secondary token within the Winr ecosystem. The team has explained that `VestedWinr` will be granted as rewards to ecosystem participants as rewards on bets, emissions to WLP holders and token stakers.

Though the vesting mechanism has not been included in this initial audit, the team has explained that `VestedWinr` will be vestable back to Winr through a linear vesting mechanism of 180 days. People can opt to vest for a shorter period but must then give up a part of their tokens. The minimum will be a 15 day vesting period by forfeiting half of the tokens.

A subsequent Paladin audit will dive more deeply into this mechanism, and we recommend carefully reading through that audit to fully understand the `Winr/ VestedWinr` dynamic.

`VestedWinr` is not transferable. The contract solely allows it to be transferred by whitelisted addresses, eg. the staking contract.

2.4.1 Privileged Functions

- `mint [ MINTER_ROLE ]`
- `burn [ BURNER_ROLE ]`
- `setGov [ DEFAULT_ADMIN_ROLE ]`
- `grantRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `renounceRole [ anyone with a role ]`

2.4.2 Issues & Recommendations

Issue #14	**BURNER_ROLE seems to impose excessive privileges as people should not need a privilege to burn their own tokens**
Severity	● LOW SEVERITY
Description	The only way to burn one's `VestedWinr` tokens is by acquiring the governance `BURNER_ROLE`. This means that many contracts whose only purpose is to burn their own `VestedWinr` balance will have this role. However, it would be much cleaner if there was an alternative function `burn(uint256 amount)` which allows a user to burn their own balance without the required priviledged role. This would reduce the number of wallets which can potentially `BURNER_ROLE` burn tokens from users without their consent, as the `BURNER_ROLE` is exceptionally powerful in that regard and could even be used to drain the LP pair (by burning most of the `VestedWinr` within this pair and then swapping out all the actually valuable tokens inside of the pair.
Recommendation	Consider being very strict with who receives the burner (and minter) roles as they are both equally dangerous from a governance risk perspective. Consider tightening the privileges as follows: - `burn(uint256 amount)` function to burn your own tokens without any role. - `BURNER_ROLE` is not granted on deployment if `_admin` is an EOA. Burner and minter roles should strictly never be granted to EOAs. - `burnFrom` can work without a role if the msg.sender has received allowance from the from wallet, it might make sense to incorporate such behavior into `burnFrom` to avoid needing the `BURNER_ROLE` even further outside of emergencies. By using allowances, that role would be almost never needed. This is by far the recommended pattern.

Resolution	✓ RESOLVED The `BURNER_ROLE` has been removed in favour of the recommended changes. Governance can no longer burn any tokens from wallets other than their own.

Issue #15	**Typographical errors**
Severity	● INFORMATIONAL

Description	We have consolidated the typographical issues into a single issue to keep the report brief and readable.
	
	<u>Line 10-11</u>
	`event Mint(address to, uint256 amount);`
	`event Burn(address from, uint256 amount);`
	
	The addresses within these events can be indexed to allow for more easy off-chain lookups.
	
	<u>Line 14</u>
	`mapping(address => bool) public wlAddresses;`
	
	This mapping could be replaced with a private `EnumerableSet` with appropriate getter methods (`length`, `index` and potentially a paginated/multi-fetch getter) to allow for users and partners to more easily inspect which contracts have been whitelisted. We strongly recommend such enumerability, similar to our `AccessControl` concerns.

Line 37

```
function setWlAccount(address _account) external
onlyGovernance {
```

This function lacks an event, consider emitting one. We also wonder whether there should be functionality to remove accounts from the whitelist again, though we do see the decentralization benefit of not being allowed to do this. A final perfection of this method could be a requirement to prevent the status from being set to the already configured status.

Lines 45 and 56

```
* @dev this function restricted about whitelisted accounts
```

Consider rewriting it to "transfers can only be made by whitelisted accounts".

Line 71

```
* @dev min function restricted about MAX_SUPPLY
```

This should say "mint function will not mint if it causes the total supply to exceed MAX_SUPPLY".

Lines 73 and 86

```
function mint(address account, uint256 amount) external
virtual onlyRole(MINTER_ROLE) {

function burn(address from, uint256 amount) external virtual
onlyRole(BURNER_ROLE) {
```

The virtual keyword seems to be unnecessary within these function definitions.

We recommend removing the `transfer` and `transferFrom` overrides and instead adding the override to `_transfer`, which overrides both functions at the same time.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

Issue #16	**mint will implicitly not mint any tokens once the maximum supply has been reached, which could lead to bugs within contracts which do not handle this case**
Severity	● INFORMATIONAL
Description	Once the maximum supply has been reached, `mint` will continue functioning. This has its advantages as any location calling `mint` will not start reverting suddenly. We have seen many bugs in contracts by our other clients where `mint` suddenly reverts and the reward contract's withdrawal method breaks. The Winr team envisioned this by not having mint revert, preventing such cases. We commend them for this. However, by not returning any information about whether the mint succeeded or failed, the rewarder contracts would have trouble figuring out what actually happened and whether they received any tokens. Such a function is therefore extremely prone to causing its calling contracts to forget to handle the case where the maximum supply is reached.
Recommendation	Consider instead either returning a boolean about whether any mint occurred or not, or returning the actually minted amount. The latter case could allow you to mint and return the "remaining" supply if not enough supply remains. Either works equally well from our perspective and is a subjective choice.
Resolution	✓ RESOLVED The data is now returned.

Issue #17	**The amount of tokens that can be minted can exceed `MAX_SUPPLY` as the cap does not take into consideration that burned tokens are removed from the supply**
Severity	● INFORMATIONAL
Description	`VestedWinr` defines a maximum supply. However, in theory, more tokens than this cap can be minted over time as tokens will be taken out of circulation again through burning. This might be all right depending on the tokenomics of the project, hence why this has been rated as informational. Specifically, the following could occur: 1. The maximum supply is minted 2. 100 tokens are burned by the project 3. 100 tokens can be minted again This would result in a total minted supply of maximum supply + 100, as the cap only checks the circulating supply.
Recommendation	Consider whether this is desired. If not, consider keeping track of the total number of minted tokens and basing the cap on this number instead of the `totalSupply()`.
Resolution	✓ RESOLVED The team has indicated that this is desired. No changes were made.

Issue #18	**Initialization of `DEFAULT_ADMIN_ROLE` is redundant with Governable**
Severity	● INFORMATIONAL
Description	During initialization of the `VestedWINR` contract, `DEFAULT_ADMIN_ROLE` is granted to the `_admin`. This however already occurs during the initialization of the Governable library which this contract extends. The admin role is therefore unnecessarily granted twice which wastes gas and makes the contract more verbose.
Recommendation	Consider removing the `_setupRole` call within this contract while leaving it in the Governable contract.
Resolution	✓ RESOLVED

2.5 DateTime

The `DateTime` library provides a single utility function, `diffDays`. This function calculates the difference in days between two timestamps represented by `fromTimestamp` and `toTimestamp`.

The function requires `fromTimestamp` to be less than or equal to `toTimestamp` and returns the difference in days as a number. This is calculated by having a constant `SECONDS_PER_DAY` that represents the number of seconds in a day. The number of days passed is therefore simply the number of times the `SECONDS_PER_DAY` number fits within the seconds elapsed. This constant is set to 24 * 60 * 60 for now which means that slowly the `DateTime` days will shift from calendar days through things like leap-seconds.

2.5.1 Issues & Recommendations

Issue #19	**`DateTime` days will eventually start shifting compared to calendar days**
Severity	INFORMATIONAL
Description	Days within the `DateTime` library are defined as 86,400 seconds. Due to the existence of things like leap-seconds, this will eventually cause these days to shift relative to a timezone. Eg. if this year a day started exactly at 00:00 UTC, it might start at a slightly different time in 100 years.
Recommendation	Consider whether this is a concern. This issue can simply be resolved on the note that it is of no concern as we agree that this logic is by far desired compared to more advanced `DateTime` logic. If calendar consistent `DateTime` logic is desired, we can help explore options as well, but the function will be significantly more complicated.
Resolution	✓ RESOLVED The team has stated that this is acceptable. No changes were made. We agree that this is fine and keeps the business logic simple.

Issue #20	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated the typographical issues into a single issue to keep the report brief and readable.
	
	`Line 6`
	`uint256 constant SECONDS_PER_DAY = 24 * 60 * 60;`
	
	The visibility of this line is implicit. Consider explicitly marking it as private or internal to better communicate the visibility to readers.
	
	`Line 13`
	`require(fromTimestamp <= toTimestamp);`
	
	This line lacks an explicit reversion message. it might be valuable to add such a message to better communicate the failure reason to whoever uses the library.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

2.6 RoleBasedAccessControl

`RoleBasedAccessControl` implements a role-based access control system using OpenZeppelin's `AccessControl` contract. This extension defines three additional roles with the `MINTER_ROLE`, `BURNER_ROLE`, and `BRIDGE_ROLE` constants, which are represented as the keccak256 hashes of strings. These roles can be used to enforce access restrictions on certain functions within the contract.

The following additional roles have been added in this extension:

- `MINTER_ROLE`
- `BURNER_ROLE`
- `BRIDGE_ROLE`

2.6.1 Privileged Functions

- `grantRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `revokeRole [ DEFAULT_ADMIN_ROLE / role's admin ]`
- `renounceRole [ anyone with a role ]`

2.6.2 Issues & Recommendations

Issue #21	Lack of enumerability on role holders makes it difficult for people to inspect which accounts have privileges
Severity	LOW SEVERITY
Description	It is often useful for users and partners to be able to inspect which accounts have certain roles. This way users can confirm that for example the `MINTER_ROLE` is solely limited to the multi-signature address. However, users can only check address by address whether an address has a role or not. To figure out the total list of addresses which have been assigned a role, there is no other way to go over all past events/historical transactions.
Recommendation	Consider moving to the OpenZeppelin extension `AccessControlEnumerable`. This is as simple as replacing the AccessControl dependency with the enumerable alternative.
Resolution	✓ RESOLVED The client has replaced the `RoleBasedAccessControl` and Governable dependencies with an Access dependency. This new dependency inherits the enumerable alternative.

2.7 Governable

Governable is a governance dependency which extends RoleBasedAccessControl with logic to setup a single account which bears the most powerful DEFAULT_ADMIN_ROLE.

The contract defines the onlyGovernance modifier, which can be used to restrict access to a function to solely the current governance address. The function setGov is defined to allow the governance address to be changed by the current governance address, but only if they have the DEFAULT_ADMIN_ROLE. After the change, the old governance address loses the DEFAULT_ADMIN_ROLE.

All issues from RoleBasedAccessControl apply to this dependency as well, since this contract inherits the former.

2.7.1 Privileged Functions

- setGov [DEFAULT_ADMIN_ROLE]
- grantRole [DEFAULT_ADMIN_ROLE / role's admin]
- revokeRole [DEFAULT_ADMIN_ROLE / role's admin]
- renounceRole [anyone with a role]

2.7.2 Issues & Recommendations

Issue #22	**Using the `DEFAULT_ADMIN_ROLE` for governance is a role management risk as this role wields much more power than a custom role**
Severity	● LOW SEVERITY
Description	Right now the contract seems to attempt to define the following requirement: Define a single role bearer, "governance", which is the sole account that can access functions protected by the `onlyGovernance` modifier. When this role bearer account wants to make another account "governance", it must pass on the role via `setGov` and remove itself from the role. However, due to the `DEFAULT_ADMIN_ROLE` being used for the "governance" role, the governance can also call `grantRole` and `revokeRole` to fully bypass the `setGov` logic. We also think that it is a dubious use of RBAC to have a holistic governance role which is incidentally also the default admin. Instead it might make more sense to have specific roles for specific tasks, similar to minting and burning.
Recommendation	Consider, if desired, for now instead having a `GOVERNANCE` role. This role can then be granted to addresses without them having to become default admins and have extreme amounts of privilege over the role management. This issue can also be resolved on the note that it is in fact desired to have this governance role be the `DEFAULT_ADMIN_ROLE` and that bypassing `setGov` is not a breakage of the requirements (as this might be fine potentially depending on the requirements).
Resolution	● ACKNOWLEDGED The client has replaced the `RoleBasedAccessControl` and `Governable` dependencies with an `Access` dependency. The team has indicated they understand this issue and will be careful with the role.

Issue #23	**Governance can accidentally renounce their role by calling `setGov` to their own address**
Severity	● LOW SEVERITY
Description	Governance could potentially accidentally renounce their privileges by calling `setGov` with their own address. This would cause the `_setupRole` call to do nothing while the `revokeRole` call still effectively renounces the role. It is best to block this behavior to protect the governance against accidental misuse, and instead require them to explicitly use the already existing renounceRole function in case they wish to renounce their role.
Recommendation	Consider adding the following requirement: `require(_gov != msg.sender, “VM: Already set”);` Consider also whether it might make sense to work with an acceptance-based (`proposeGov`, `acceptGov`) function set as this function still bears the risk that `setGov` was accidentally called to a non-existent address, which would effectively renounce the governance as well by accident.
Resolution	✓ RESOLVED The client has replaced the `RoleBasedAccessControl` and `Governable` dependencies with an Access dependency. `setGov` has been removed.

Issue #24	Inconsistent usage of `msgSender()` and `msg.sender`
Severity	● LOW SEVERITY
Description	The contract inconsistently uses both `msgSender()` and `msg.sender`. This could be a serious risk if ever dependencies decide to enable meta transaction usage. In that case, the meta transaction logic would malfunction and potentially call `revokeRole` on the wrong wallet. It should be noted that within the current codebase we audited, this has no side-effects or negative consequences.
Recommendation	Consider being consistent with the use of `msgSender()` — it is typically acceptable to not use it at all if there is no plan to ever support meta transactions. If used, however, it should be used consistently throughout the codebase to allow for integration of eventual meta-transactions. Consider replacing all occurrences of `msg.sender` with `msgSender()` throughout the codebase.
Resolution	✓ RESOLVED The client has replaced the `RoleBasedAccessControl` and `Governable` dependencies with an Access dependency. `msg.sender` is now used exclusively.

Issue #25	OpenZeppelin's `_setupRole` is deprecated in favour of `_grantRole`
Severity	INFORMATIONAL
Description	Although identical in functionality, OpenZeppelin has deprecated `_setupRole` since a few releases in favor of `_grantRole`. It is therefore best practice to no longer use the deprecated function and instead move to the recommended function. There is no functional difference between the two for now.
Recommendation	Consider using `_grantRole` instead.
Resolution	✓ RESOLVED The client has replaced the `RoleBasedAccessControl` and `Governable` dependencies with an Access dependency. `_grantRole` is now used.

Issue #26	Gas optimization
Severity	INFORMATIONAL

Location	`Line 19`
	`revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);`

Description	This function is an external function which means it wastes additional gas on doing a role check on `msg.sender`. Consider using `revokeRole` instead.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	✓ RESOLVED The client has replaced the `RoleBasedAccessControl` and `Governable` dependencies with an Access dependency. This logic has been removed.

PALADIN
BLOCKCHAIN SECURITY