ERC-6492 Security Audit

Report Version 1.0

February 18, 2024

Conducted by the **Hunter Security** team:

George Hunter, Lead Security Researcher
deadrosesxyz, Senior Security Researcher

Table of Contents

1 About Hunter Security

Hunter Security is a duo team of independent smart contract security researchers. Having conducted over 50 security reviews and reported tens of live smart contract security vulnerabilities, our team always strives to deliver top-quality security services to DeFi protocols. For security review inquiries, you can reach out to us on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

The Hunter Security team was engaged by Ambire to review EIP-6492 and the ERC-6492 smart contract implementation, UniversalSigValidator.

Overview

Project Name	ERC-6492: Signature Validation for Predeploy Contracts
Repository	https://github.com/ethereum/ercs
Commit hash	8dd085d159cb123f545c272c0d871a5339550e79
Resolution	aa3a0880f807951572fee9524d77f919999ca5a0 (in AmbireTech/ERCs)
Methods	Manual review, BTT

Timeline

-	February 13, 2024	Audit kick-off
v0.1	February 17, 2024	Preliminary report
v1.0	February 18, 2024	Mitigation review

Scope

ercs/ERCS/erc-6492.md

Issues Found

High risk	1
Medium risk	2
Low risk	1
Informational	12

5 Consultants

George Hunter - a proficient and reputable independent smart contract security researcher with over 50 solo and team security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, Maverick, Ambire, and other leading protocols. George's extensive experience in traditional audits and meticulous attention to detail contribute to Hunter Security's reviews, ensuring comprehensive coverage and preventing vulnerabilities from slipping through.

deadrosesxyz - a proficient and reputable bug bounty hunter with over 10 live smart contract vulnerability reports, confirmed and effectively mitigated through Immunefi. He has made significant contributions to securing protocols such as Yearn, Velodrome, Euler, SPool, and other leading DeFi protocols. His creativity and experience in hunting vulnerabilities in live protocols provide unmatched value to Hunter Security's reviews, uncovering unique vulnerabilities and edge cases that most auditors overlook.

6 System overview

ERC-6492 consists of one main and one helper smart contract, called respectively `UniversalSigValidator` and `ValidateSigOffchain`. The `UniversalSigValidator` offers signature verification methods that wrap the well-known ERC-1271 `isValidSignature(hash, signature)`.

The way it works is that it accepts the signer, hash, and signature to validate, and checks whether the signature ends in `magicBytes`, called `ERC6492_DETECTION_SUFFIX`, which indicates that the signer is a counterfactual wallet. In that case, it would attempt to deploy or "prepare" the wallet by decoding the passed counterfactual signature into three parameters: `create2Factory`, `factoryCalldata`, and `sigToValidate`. Then, it tries to invoke the ERC-1271 `isValidSignature(hash, signature)` method on the passed signer address. If the signer is an EOA and the signature is not counterfactual, a standard `ecrecover` verification will be executed.

The helper contract `ValidateSigOffchain` is meant to be used off-chain via `eth_call` so that developers are able to use the standard without the pre-deployed the singleton.

6.1 Codebase maturity

Code complexity - *Good*
While the `UniversalSigValidator` is minimalistic, it accepts multiple parameters with a wide range of possible values creating numerous combinations of possible paths, which adds decent complexity.

Security - *Good*
The code is part of an Ethereum Improvement Proposal and has been reviewed by numerous developers and researchers. However, this is the first formal security audit it has undergone, and it is recommended to be reviewed by more profesional parties in the future.

Decentralization - *Excellent*
The contract is an immutable singleton, and there are no privileged addresses within the system.

Testing suite - *N/A*
Not provided.

Documentation - *Excellent*
The requirements and expected behavior of the implementation have been thoroughly described in the Ethereum Improvement Proposal document.

Best practices - *Excellent*
The contract adheres to all best practices.

6.2 Privileged actors

- None

6.3 Threat model

What in ERC-6492 has value in the market?

The `UniversalSigValidator` contract is not expected to hold or manage any user funds. The most important aspect of this protocol is the accuracy of the returned value from signature verification.

What are the worst-case scenarios for the UniversalSigValidator or users interacting with it?

- The signature verification forwards an incorrect result to the caller (i.e. returns `true` for invalid signatures).
- The signature verification process induces side effects through the factory call, but it fails to revert when the `allowSideEffects` flag is set to `false`.
- Denial-of-Service of the contract's functionality, such as infinite recursion by self-calling `isValidSigImpl` endlessly.
- Signature format collision.

What are the main entry points and non-permissioned functions?

- `UniversalSigValidator.isValidSigImpl()` - contains the main logic of the standard including recognition and deployment of counterfactual wallets, ERC-1271 and ecrecover signature validation, handling side effects, etc.
- `UniversalSigValidator.isValidSigWithSideEffects()` - same as `isValidSigImpl`, but `allowSideEffects` is hard-coded as `true` and `tryPrepare` as `false`.
- `UniversalSigValidator.isValidSig()` - same as `isValidSigImpl`, but returns `true`/`false` instead of reverting if the call reverted due to side effects from the counterfactual deployment.
- `ValidateSigOffchain.constructor()` - deploys an instance of the `UniversalSigValidator` contract and returns the result of `isValidSigWithSideEffects` via revert. Meant to be used off-chain via `eth_call`.

6.4 Observations

Several interesting design decisions were observed during the review of ERC-6492, some of which are listed below:

- The so-called `ERC6492_DETECTION_SUFFIX` constant represents a magic value appended to a normal (ERC-1271) signature. It ends in a `0x92` byte, which helps avoid collision with ecrecover signatures that should end with `0x27` or `0x28`.
- Instead of passing the type of signature verification scheme the caller would like to use, the `UniversalSigValidator` determines the verification scheme based on whether the passed signer address is an EOA or a deployed contract, as well as whether the signature was wrapped with the `ERC6492_DETECTION_SUFFIX`.
- Instead of passing `salt` and `bytecode`, a `factoryCalldata` parameter of type bytes is used to make verification compliant with any factory interface.
- There are two modes for setting up a counterfactual wallet:

  1) deploy it via a `create2` factory
  2) "prepare" it by passing the necessary contract address and calldata along with the wrapped counterfactual signature in place of the `create2Factory` and `factoryCalldata`, and set the `tryPrepare` flag to true.

- If ERC-1271 signature verification fails either because the call failed or the returned value was not `0x1626ba7e`, retry the validation, but this time assuming the prefix is a prepare call.
- The caller can specify an `allowSideEffects` boolean flag which, if set to true, will revert the verification, returning the result (signature validity status) as a revert message instead of a return value. This is done to protect from re-entrancy issues potentially caused by the external call to the `create2Factory`.
- The `ValidateSigOffchain` contract is a helper that allows developers to perform validation in a single `eth_call` without pre-deploying the `UniversalSigValidator` singleton.

6.5 Useful resources

The following resources were used to research more about the problem addressed by EIP-6492, its origin, and the evolution of its implementation over time.

- *Counterfactual ERC-1271 Signature Validation Problem*
- *Ethereum Magicians Forum EIP-6492 Discussion*
- *Ethereum Engineering Group: Counterfactual Wallets & ERC-6492*

7 Findings

7.1 High

7.1.1 The side effects invariant can be broken

Severity: *High*

Context: ERC6492.sol#L53

Description: The core problem ERC-6492 aims to solve is signature validation for pre-deploy (counterfactual) smart contract wallets. It achieves this by introducing a wrapped format for smart contract signatures:

- If the contract is not deployed yet, wrap the signature as follows: `concat(abi.encode((create2Factory, factoryCalldata, originalERC1271Signature), (address, bytes, bytes)), magicBytes)`
- If the contract is deployed but not ready to verify using ERC-1271, wrap the signature as follows: `concat(abi.encode((prepareTo, prepareData, originalERC1271Signature), (address, bytes, bytes)), magicBytes)`; `prepareTo` and `prepareData` must contain the necessary transaction that will make the contract ready to verify using ERC-1271 (e.g. a call to migrate or update)

The additional parameters provided as part of the signature are used to deploy or "prepare" the counterfactual smart wallet for ERC-1271 signature verification. The following code executes these actions:

```
if (isCounterfactual) {
    address create2Factory;
    bytes memory factoryCalldata;
    (create2Factory, factoryCalldata, sigToValidate) =
        abi.decode(_signature[0:_signature.length - 32], (address, bytes, bytes));

    if (contractCodeLen == 0 || tryPrepare) {
        (bool success, bytes memory err) = create2Factory.call(factoryCalldata);
        if (!success) revert ERC6492DeployFailed(err);
    }
}
```

This introduces a risk of reentrancy (and untrusted calls in general) that may affect the caller if it is a smart contract. The risk is documented in the `Security Considerations` section of the EIP:

"Deploying a contract requires a `CALL` rather than a `STATICCALL`, which introduces reentrancy concerns. This is mitigated in the reference implementation by having the validation method always revert if there are side-effects, and capturing its actual result from the revert data."

Therefore, we define the invariant that the changes caused by the above call to `create2Factory` should always be reverted at the end of the signature verification if the `allowSideEffects` boolean flag has been set to `false`.

How do we break this invariant?

We notice the validation logic for this is handled in the following lines of code:

```
if (contractCodeLen == 0 && isCounterfactual && !allowSideEffects) {
    // if the call had side effects we need to return the
    // result using a 'revert' (to undo the state changes)
    assembly {
        mstore(0, isValid)
        revert(31, 1)
    }
}
```

We see 3 conditions used to determine whether the above call has been executed and therefore potentially causes side effects that may affect the caller:

1. `isCounterfactual` - as seen in the first code snippet, we only execute the call if the signature is counterfactual.
2. `!allowSideEffects` - this input parameter is used to determine whether the caller wants to revert or keep any side effects.
3. `contractCodeLen == 0` - this is used to determine whether we entered the if-check body where the call is actually executed.

The problem here comes in the 3rd condition - the untrusted external call to the `create2Factory` is executed if `contractCodeLen == 0 || tryPrepare`, not just `contractCodeLen == 0`.

Therefore, two potential attack vectors are opened from this flaw:

1. A caller who wants to verify a signature for an already deployed counterfactual smart wallet passes `allowSideEffects=false` and `tryPrepare=true`. The `UniversalSigValidator` performs the call to "prepare" the smart wallet, calls `isValidSignature` on the signer, and receives a `magicValue` equal to `ERC1271_SUCCESS`. However, when we reach the above if-check, `contractCodeLen == 0` is `false`, and despite the fact that we performed a call with side effects and `allowSideEffects` is `false`, we do not revert the call.
2. A caller who wants to verify a signature for an already deployed counterfactual smart wallet passes `allowSideEffects=false` and `tryPrepare=false` (the difference with the first scenario is that `tryPrepare` is now `false`). The `UniversalSigValidator` directly attempts to verify the ERC-1271 signature, calls `isValidSignature` on the signer, and the call either reverts or does not return `ERC1271_SUCCESS`. Then `isValidSigImpl` is called internally, but this time with the `tryPrepare` flag set to `true`. This makes the contract to call the `create2Factory` with the `factoryCalldata` read from the signature which is a call with side effects. Then, regardless of whether the signer returns `ERC1271_SUCCESS`, the flawed check will again pass and not revert the call.

We see how 2 of the main and most common flows are actually not re-entrancy safe despite the `allowSideEffects` flag.

Recommendation: Consider implementing the following change:

```
if ((contractCodeLen == 0 || tryPrepare) && isCounterfactual && !allowSideEffects) {
```

Resolution: Resolved. The recommended fix was implemented.

7.2 Medium

7.2.1 Incorrect revert message encoding

Severity: *Medium*

Context: ERC6492.sol#L103

Description: The `isValidSig` method serves as a wrapper, performing an external call to the contract's `isValidSigImpl` function to enable a `try {} catch {}` block.

The purpose of this wrapper is to capture errors that occur when side effects have been triggered and return them as return data rather than error messages. If the error isn't caused by side effects, it simply forwards the revert message as follows:

```
} catch (bytes memory error) {
    ...
    assembly {
        revert(error, len)
    }
}
```

The problem lies in how the first argument of the revert operation in assembly is passed. Currently, passing the `error` variable points to the start of where the variable's length is stored, rather than the actual start of the error message. Therefore, the forwarded error message ends up excluding the last 32 bytes of the true revert reason, with the length being incorrectly prepended as the first 32 bytes.

Recommendation: Consider implementing the following change:

```
assembly {
    revert(add(error, 0x20), len)
}
```

Resolution: Resolved. The recommended fix was implemented.

7.2.2 ERC-1271 signatures shorter than 32 bytes will not work

Severity: *Medium*

Context: ERC6492.sol#L28

Description: The last 32 bytes of a counterfactual signature are occupied by the `ERC6492_DETECTION_SUFFIX` constant, which serves the purpose of recognizing whether the signature is passed for a counterfactual wallet or not.

The check for this is implemented in the `UniversalSigValidator` contract as follows:

```
bool isCounterfactual =
    bytes32(_signature[_signature.length - 32:_signature.length]) ==
        ERC6492_DETECTION_SUFFIX;
```

The standard and contract aim to support ERC-1271 smart wallet signatures as well, which can have any value. However, the above line will revert the call with an EVM error caused by underflow if the `_signature` is less than 32 bytes long.

Recommendation: Consider adding a length check before performing the slicing operation to prevent arithmetic underflow errors:

```
bool isCounterfactual = _signature.length >= 32 &&
    bytes32(_signature[_signature.length - 32:_signature.length]) ==
        ERC6492_DETECTION_SUFFIX;
```

Resolution: Resolved. The recommended fix was implemented.

7.3 Low

7.3.1 A user can front-run a counterfactual signature verification and force it to revert

Severity: *Low*

Context: ERC6492.sol#L29-L38

Description: ERC-6492 counterfactual signature consists of `create2Factory`, `factoryCalldata`, and `sigToValidate` parameters (as well as the `ERC6492_DETECTION_SUFFIX` constant). If a counterfactual signature is passed to the `isValidSigImpl` method and the signer address has no existing code or the `tryPrepare` flag is set to `true`, an external call will be executed:

```
if (isCounterfactual) {
    address create2Factory;
    bytes memory factoryCalldata;
    (create2Factory, factoryCalldata, sigToValidate) =
        abi.decode(_signature[0:_signature.length - 32], (address, bytes, bytes));

    if (contractCodeLen == 0 || tryPrepare) {
        (bool success, bytes memory err) = create2Factory.call(factoryCalldata);
        if (!success) revert ERC6492DeployFailed(err);
    }
```

The problem is that if the signature verification happens on-chain, an adversary can simply front-run the victim's transaction and deploy or "prepare" the contract before that, effectively forcing the victim's transaction to revert.

Recommendation: Consider whether this is the intended behavior or if it would be better to ignore the `success` status of the above call.

Resolution: Resolved. The return value from the call to `create2Factory` is no longer used to determine whether the deployment failed. Instead, the code size of the signer after the attempted deployment is checked. In the case of a `tryPrepare` call, the status and error message are used only after and if the ERC-1271 signature verification failed in order to still perform the check if the call failed.

7.4 Informational

7.4.1 Signature verification will revert with an EVM error if the wallet still has no code

Severity: *Informational*

Context: ERC6492.sol#L45

Description: If the passed signer has no code at the beginning of the function and the signature is counterfactual, a deployment will be attempted using the `create2Factory` and `factoryCalldata` parameters. It is assumed that if the call to the `create2Factory` succeeds, then the counterfactual wallet at the `_signer` address has been successfully deployed and now has code.

However, the `create2Factory` may fail silently (i.e., return `false` instead of reverting) or simply execute some other logic that does not deploy code at the passed `_signer` address.

In that case, the `try IERC1271Wallet(_signer).isValidSignature(_hash, sigToValidate)` line will directly revert due to the `extcodesize` check performed before executing the call. The returned error will be an EVM error instead of a custom error, and a retry with the `tryPrepare` flag set to `true` will not be performed.

Recommendation: Consider whether this behavior is expected, or if this case should be handled with a custom check.

Resolution: Resolved. An explicit signer's code size check was implemented right after the call to `create2Factory`.

7.4.2 Missing parameter in the retry checks

Severity: *Informational*

Context: ERC6492.sol#L49-L51, ERC6492.sol#L65-L67

Description: If the ERC-1271 signature verification fails either due to a revert in the `isValidSignature` function call or because of a negative returned value, the `isValidSigImpl` is executed again, but with the `tryPrepare` flag set to true:

```
try IERC1271Wallet(_signer).isValidSignature(_hash, sigToValidate) returns (bytes4
    magicValue) {
    bool isValid = magicValue == ERC1271_SUCCESS;

    // retry, but this time assume the prefix is a prepare call
    if (!isValid && !tryPrepare && contractCodeLen > 0) {
        return isValidSigImpl(_signer, _hash, _signature, allowSideEffects, true);
    }
    ...
} catch (bytes memory err) {
    // retry, but this time assume the prefix is a prepare call
    if (!tryPrepare && contractCodeLen > 0) {
        return isValidSigImpl(_signer, _hash, _signature, allowSideEffects, true);
    }
    revert ERC1271Revert(err);
}
```

One parameter is missing in the if-checks above - `isCounterfactual`. If the signature is not a counterfactual one, then there is no point in executing the verification again with a `tryPrepare` flag set to `true`, because it's only used when `isCounterfactual` is `true` and therefore won't change anything if `isCounterfactual` is `false`.

Recommendation: Consider adding `&& isCounterfactual` to the above if-checks.

Resolution: Resolved. The recommended fix was implemented.

7.4.3 A sanity check can be added to reduce attack surface

Severity: *Informational*

Context: ERC6492.sol#L35

Description: The contract accepts multiple input parameters that determine the execution path of the function. Two of these parameters are `contractCodeLen` (based on the passed `_signer`) and the `tryPrepare` boolean flag.

There is no valid scenario where both `contractCodeLen == 0` and `tryPrepare` are `true`.

Recommendation: Consider adding a check to reduce the attack surface by limiting the possible combinations of input parameters and therefore execution paths.

Resolution: Acknowledged.

7.4.4 Any tokens held by or approved to the contract can be stolen

Severity: *Informational*

Context: ERC6492.sol#L36

Description: The `UniversalSigValidator` can function as a multi-call contract because of the external call made to the `create2Factory` parameter, which provides the `factoryCalldata`. The `factoryCalldata` is also an input parameter controlled by the user.

If the contract holds any funds, has tokens approved to it, or is set as an owner/admin/manager of another contract, anyone will be able to steal those funds or privileges.

Recommendation: Consider documenting this behavior.

Resolution: Acknowledged.

7.4.5 Missing validation for malleable signature's S value

Severity: *Informational*

Context: ERC6492.sol#L74-L81

Description: The `isValidSigImpl` method performs ecrecover signature verification if the passed signer address lacks existing code and the signature is not counterfactual (i.e., does not end with `ERC6492_DETECTION_SUFFIX`):

```
require(_signature.length == 65, "SignatureValidator#recoverSigner: invalid
    signature length");
bytes32 r = bytes32(_signature[0:32]);
bytes32 s = bytes32(_signature[32:64]);
uint8 v = uint8(_signature[64]);
if (v != 27 && v != 28) {
    revert("SignatureValidator: invalid signature v value");
}
return ecrecover(_hash, v, r, s) == _signer;
```

Only the V value is validated to be in the correct range, but the S value is not. This means that malleable signatures with flipped S and V values can be used and passed as valid for a signer.

Recommendation: Consider checking for malleable S values as well.

Resolution: Acknowledged.

7.4.6 Check for invalid signature V is unnecessary

Severity: *Informational*

Context: ERC6492.sol#L78-L80

Description: The `isValidSigImpl` method conducts ecrecover signature verification if the passed signer address lacks existing code and the signature is not counterfactual (i.e., does not end with `ERC6492_DETECTION_SUFFIX`).

It includes the following check on the V value:

```
if (v != 27 && v != 28) {
    revert("SignatureValidator: invalid signature v value");
}
```

This check is redundant because the ecrecover precompile already performs this verification and returns address(0) if `v` is invalid.

Recommendation: Consider removing the redundant check.

Resolution: Acknowledged.

7.4.7 Counterfactual verification will not work for privileged deployers

Severity: *Informational*

Context: ERC6492.sol#L36

Description: This is a general informational issue regarding the protocol's concept. If a user's wallet can only be deployed by a permissioned entity, such as a `factoryDeployer` in some protocols, or by the signer itself, then counterfactual signature verification using ERC-6492 cannot be performed.

Recommendation: Consider documenting this behavior and potentially implementing an alternative solution.

Resolution: Acknowledged.

7.4.8 Misleading variables names and comments

Severity: *Informational*

Context: ERC6492.sol#L7, ERC6492.sol#L25, ERC6492.sol#L30-L31

Description: Some comments, variables, and custom error names in the contract might be misunderstood by developers interacting with it:

The name of this error suggests that it indicates a failed deployment of a counterfactual wallet, but it is also used for the `tryPrepare` type of calls.

```
error ERC6492DeployFailed(bytes error);
```

This comment implies that ERC-1271 verification occurs only when the signer has code. However, if the counterfactual deployment fails silently (i.e., returns `false` instead of reverting), the signer address may still have no code, and ERC-1271 verification will be attempted.

```
// - ERC-1271 verification if there's contract code
```

These parameters, `create2Factory` and `factoryCalldata`, are not only used for their suggested purpose by their names, but also if the `tryPrepare` flag is set to true.

```
address create2Factory;
bytes memory factoryCalldata;
```

Recommendation: Consider correcting the above notes to improve clarity.

Resolution: Resolved. `ERC6492DeployFailed` was renamed to `ERC6492CallFailed` and ERC-1271 is no longer attempted if the signer has no code after the deploy call.

7.4.9 Missing Solidity version pragma line

Severity: *Informational*

Context: ERC6492.sol#L1

Description: The contract does not specify a Solidity version for compilation, which is dangerous for singleton contracts like this one because it can lead to varied behavior across different compiler and EVM versions.

Recommendation: It's advisable to specify the compiler and EVM version that the contract should be compiled with.

Resolution: Acknowledged.

7.4.10 No null address validation for signer and result from ecrecover

Severity: *Informational*

Context: ERC6492.sol#L15, ERC6492.sol#L81

Description: The `isValidSigImpl` method accepts a parameter `_signer` of type address for which the signature verification is executed.

If the null address is passed as the `_signer`, the function will either revert if the passed signature is counterfactual or succeed if the signature is invalid and `ecrecover` returns address(0).

Recommendation: Consider the intended behavior when address(0) is passed as the `_signer`. A safer approach might be to return false or revert at the beginning of the function body.

Resolution: Acknowledged.

7.4.11 isValidSigImpl can be called internally to avoid increasing the call stack depth

Severity: *Informational*

Context: ERC6492.sol#L88

Description: The `isValidSigWithSideEffects` method wraps `isValidSigImpl` and simply forwards the passed parameters, but hardcodes `allowSideEffects` and `tryPrepare` to `true` and `false` respectively:

```
function isValidSigWithSideEffects(address _signer, bytes32 _hash, bytes calldata
    _signature)
    external
    returns (bool)
{
    return this.isValidSigImpl(_signer, _hash, _signature, true, false);
}
```

The call to the `isValidSigImpl` method is done using the `this` keyword, which means it's executed as an external call rather than being called internally.

Recommendation: Consider making the function cleaner by removing `this.` and simply calling the `isValidSigImpl` method internally.

Resolution: Acknowledged.

7.4.12 Redundant address type casting

Severity: *Informational*

Context: ERC6492.sol#L21

Description: The first line of the `isValidSigImpl` body obtains the existing code size of the passed `_signer` address:

```
uint256 contractCodeLen = address(_signer).code.length;
```

The cast to type address is unnecessary as the variable `_signer` is already of type address.

Recommendation: Consider removing the type cast shown above.

Resolution: Resolved.