---

title: Smart contract security guidelines description: A checklist of security guidelines to consider when building your dapp author: "Trailofbits" tags: ["solidity", "smart contracts", "security"] skill: intermediate lang: en published: 2020-09-06 source: Building secure contracts sourceUrl: https://github.com/crytic/building-secure-contracts/blob/master/development-guidelines/guidelines.md

---

Follow these high-level recommendations to build more secure smart contracts.

Design guidelines {#design-guidelines}

The design of the contract should be discussed ahead of time, prior to writing any line of code.

Documentation and specifications {#documentation-and-specifications}

Documentation can be written at different levels, and should be updated while implementing the contracts:

- **A plain English description of the system**, describing what the contracts do and any assumptions on the codebase.
- **Schema and architectural diagrams**, including the contract interactions and the state machine of the system.Slither printers can help to generate these schemas.
- **Thorough code documentation**, the Natspec format can be used for Solidity.

On-chain vs off-chain computation {#on-chain-vs-off-chain-computation}

- **Keep as much code as you can off-chain.** Keep the on-chain layer small. Pre-process data with code off-chain in such a way that verification on-chain is simple. Do you need an ordered list? Sort the list offchain, then only check its order onchain.

Upgradeability {#upgradeability}

We discussed the different upgradeability solutions in our blogpost. Make a deliberate choice to support upgradeability or not prior to writing any code. The decision will influence how you structure your code. In general, we recommend:

- **Favoring contract migration over upgradeability.** Migration systems have many of the same advantages as upgradeable ones, without their drawbacks.
- **Using the data separation pattern over the delegatecallproxy one.** If your project has a clear abstraction separation, upgradeability using data separation will necessitate only a few adjustments. The delegatecallproxy requires EVM expertise and is highly error-prone.
- **Document the migration/upgrade procedure before the deployment.** If you have to react under stress without any guidelines, you will make mistakes. Write the procedure to follow ahead of time. It should include:
- The calls that initiate the new contracts
- Where are stored the keys and how to access them
- How to check the deployment! Develop and test a post-deployment script.

Implementation guidelines {#implementation-guidelines}

Strive for simplicity. Always use the simplest solution that fits your purpose. Any member of your team should be able to understand your solution.

Function composition {#function-composition}

The architecture of your codebase should make your code easy to review. Avoid architectural choices that decrease the ability to reason about its correctness.

- **Split the logic of your system**, either through multiple contracts or by grouping similar functions together (for example, authentication, arithmetic, ...).
- **Write small functions, with a clear purpose.** This will facilitate easier review and allow the testing of individual

components.

Inheritance {#inheritance}

- **Keep the inheritance manageable.** Inheritance should be used to divide the logic, however, your project should aim to minimize the depth and width of the inheritance tree.
- **Use Slither's inheritance printer to check the contracts' hierarchy.** The inheritance printer will help you review the size of the hierarchy.

Events {#events}

- **Log all crucial operations.** Events will help to debug the contract during the development, and monitor it after deployment.

Avoid known pitfalls {#avoid-known-pitfalls}

- **Be aware of the most common security issues.** There are many online resources to learn about common issues, such as Ethernaut CTF, Capture the Ether, or Not so smart contracts.
- **Be aware of the warnings sections in the Solidity documentation.** The warnings sections will inform you about non-obvious behavior of the language.

Dependencies {#dependencies}

- **Use well-tested libraries.** Importing code from well-tested libraries will reduce the likelihood that you write buggy code. If you want to write an ERC20 contract, use OpenZeppelin.
- **Use a dependency manager; avoid copy-pasting code.** If you rely on an external source, then you must keep it up-to-date with the original source.

Testing and verification {#testing-and-verification}

- **Write thorough unit-tests.** An extensive test suite is crucial to build high-quality software.
- **Write Slither, Echidna and Manticore custom checks and properties.** Automated tools will help ensure your contract is secure. Review the rest of this guide to learn how to write efficient checks and properties.
- **Use crytic.io.** Crytic integrates with GitHub, provides access to private Slither detectors, and runs custom property checks from Echidna.

Solidity {#solidity}

- **Favor Solidity 0.5 over 0.4 and 0.6.** In our opinion, Solidity 0.5 is more secure and has better built-in practices than 0.4. Solidity 0.6 has proven too unstable for production and needs time to mature.
- **Use a stable release to compile; use the latest release to check for warnings.** Check that your code has no reported issues with the latest compiler version. However, Solidity has a fast release cycle and has a history of compiler bugs, so we do not recommend the latest version for deployment (see Slither's solc version recommendation).
- **Do not use inline assembly.** Assembly requires EVM expertise. Do not write EVM code if you have not *mastered* the yellow paper.

Deployment guidelines {#deployment-guidelines}

Once the contract has been developed and deployed:

- **Monitor your contracts.** Watch the logs, and be ready to react in case of contract or wallet compromise.
- **Add your contact info to blockchain-security-contacts.** This list helps third-parties contact you if a security flaw is discovered.
- **Secure the wallets of privileged users.** Follow our best practices if you store keys in hardware wallets.
- **Have a response to incident plan.** Consider that your smart contracts can be compromised. Even if your contracts

are free of bugs, an attacker may take control of the contract owner's keys.