title: Smart contract security checklist description: A suggested workflow for writing secure smart contracts author: "Trailofbits" tags: ["smart contracts", "security", "solidity"] skill: intermediate lang: en published: 2020-09-07 source: Building secure contracts sourceUrl: https://github.com/crytic/building-secure-contracts/blob/master/development-guidelines/workflow.md

Smart contract development checklist {#smart-contract-development-checklist}

Here's a high-level process we recommend following while you write your smart contracts.

Check for known security issues:

- Review your contracts with Slither. It has more than 40 built-in detectors for common vulnerabilities. Run it on every check-in with new code and ensure it gets a clean report (or use triage mode to silence certain issues).
- Review your contracts with Crytic. It checks for 50 issues that Slither does not. Crytic can help your team stay on top of each other too, by easily surfacing security issues in Pull Requests on GitHub.

Consider special features of your contract:

- Are your contracts upgradeable? Review your upgradeability code for flaws with `slither-check-upgradeability` or Crytic. We've documented 17 ways upgrades can go sideways.
- Do your contracts purport to conform to ERCs? Check them with `slither-check-erc`. This tool instantly identifies deviations from six common specs.
- Do you have unit tests in Truffle? Enrich them with `slither-prop`. It automatically generates a robust suite of security properties for features of ERC20 based on your specific code.
- Do you integrate with 3rd party tokens? Review our token integration checklist before relying on external contracts.

Visually inspect critical security features of your code:

- Review Slither's inheritance-graph printer. Avoid inadvertent shadowing and C3 linearization issues.
- Review Slither's function-summary printer. It reports function visibility and access controls.
- Review Slither's vars-and-auth printer. It reports access controls on state variables.

Document critical security properties and use automated test generators to evaluate them:

- Learn to document security properties for your code. It's tough as first, but it's the single most important activity for achieving a good outcome. It's also a prerequisite for using any of the advanced techniques in this tutorial.
- Define security properties in Solidity, for use with Echidna and Manticore. Focus on your state machine, access controls, arithmetic operations, external interactions, and standards conformance.
- Define security properties with Slither's Python API. Focus on inheritance, variable dependencies, access controls, and other structural issues.
- Run your property tests on every commit with Crytic. Crytic can consume and evaluate security property tests so everyone on your team can easily see that they pass on GitHub. Failing tests can block commits.

Finally, be mindful of issues that automated tools cannot easily find:

- Lack of privacy: everyone else can see your transactions while they're queued in the pool
- Front running transactions
- Cryptographic operations
- Risky interactions with external DeFi components

Ask for help {#ask-for-help}

Ethereum office hours run every Tuesday afternoon. These 1-hour, 1-on-1 sessions are an opportunity to ask us any questions you have about security, troubleshoot using our tools, and get feedback from experts about your current approach. We will help you work through this guide.

Join our Slack: Empire Hacking. We're always available in the #crytic and #ethereum channels if you have any questions.