CHECKLISTE

Checkliste: Smart Contracts audit-ready machen

30 Prüfpunkte vor dem Smart-Contract-Audit: Scope und Doku, Tests, Foundry-Fuzz- und Invariantentests, Slither-Triage, Zugriffskontrolle und Deployment.

30 Prüfpunkte	Haken Sie jeden Punkt ab, sobald er umgesetzt ist. Die Online-Version enthält dieselben Erläuterungen mit Links.

Online-Version: sigmacode.io/de/insights/smart-contract-audit-readiness-checklist

Ein Smart-Contract-Audit ist teuer und strikt zeitlich begrenzt. Die Auditoren haben eine feste Anzahl an Tagen, und jede Stunde, die sie damit verbringen, die beabsichtigte Funktionsweise zu rekonstruieren, einem sich ändernden Commit hinterherzulaufen oder fehlende Tests nachzuholen, fehlt bei der Logik, an der tatsächlich Geld verloren gehen kann. Der Bericht füllt sich dann mit fehlender NatSpec, Floating Pragmas und ungetesteten Revert-Pfaden statt mit den Problemen, für deren Suche Sie bezahlen.

Eine gut vorbereitete Codebasis bekommt für dasselbe Budget ein tieferes Audit. Diese Checkliste fasst zusammen, was wir vorbereiten, bevor Code an externe Auditoren geht: 30 konkrete Prüfpunkte zu Dokumentation, Code-Hygiene, Tests, Zugriffskontrolle, Integrationen, Deployment und Organisation. Sie ersetzt das Audit nicht, sondern sorgt dafür, dass die Audit-Zeit dort eingesetzt wird, wo sie zählt.

1. Scope und Dokumentation

Auditoren können Verhalten nur gegen eine Absicht prüfen, die sie verstehen – also muss diese Absicht schriftlich vorliegen.

- [ ] 01 **Commit-Hash einfrieren.** Übergeben Sie den Auditoren einen getaggten Commit und ändern Sie den geprüften Code während des Audits nicht. Fixes entstehen auf einem separaten Branch und werden in der Fix-Review-Runde geprüft.
- [ ] 02 **Scope-Dateiliste mit nSLOC bereitstellen.** Listen Sie jeden Vertrag im Scope mit Pfad und normalisierten Codezeilen (nSLOC) auf und benennen Sie ausdrücklich, was nicht im Scope ist (Bibliotheken, Mocks, Skripte, bereits auditierte Dateien). Auditoren kalkulieren und planen auf Basis von nSLOC, eine genaue Zahl vermeidet Überraschungen auf beiden Seiten.
- [ ] 03 **Architekturüberblick schreiben.** Ein bis zwei Seiten, die die Verträge, ihre gegenseitigen Aufrufe, die Verträge mit Guthaben und die wichtigsten Nutzerabläufe beschreiben. Ein einfaches Diagramm der Aufrufe und Token-Flüsse erspart den Auditoren Stunden an Reverse Engineering.
- [ ] 04 **Verhalten und Invarianten in Prosa spezifizieren.** Beschreiben Sie, was jede externe Funktion tun muss und welche Eigenschaften immer gelten, etwa „die Summe der Nutzerguthaben entspricht `totalAssets`“ oder „nur der Timelock kann Gebühren ändern“. Diese Aussagen sind die Grundlage sowohl für das Review der Auditoren als auch für Ihre eigenen Invariantentests.

- [ ] 05 **Bekannte Probleme und akzeptierte Risiken dokumentieren.** Listen Sie bekannte Einschränkungen und bewusst getroffene Designentscheidungen auf, etwa Zentralisierungs-Trade-offs oder nicht unterstützte Token-Typen. So landen sie nicht im Bericht, und die Auditoren sehen, wo Sie bereits nachgedacht haben.

2. Code-Hygiene und statische Analyse

Rauschen im Code kostet Audit-Zeit und erzeugt Findings mit geringem Wert – entfernen Sie es, bevor die Auditoren es sehen.

- [ ] 06 **Compiler festlegen und null Warnungen erreichen.** Verwenden Sie ein exaktes `pragma solidity 0.8.x;` statt eines Floating `^0.8.0` und stimmen Sie Version, Optimizer-Runs, `via_ir` und `evm_version` in `foundry.toml` auf das spätere Deployment ab. Jede Compiler-Warnung wird behoben oder bewusst begründet.
- [ ] 07 **Abhängigkeiten festlegen und auflisten.** Fixieren Sie OpenZeppelin Contracts und jede andere Bibliothek auf ein exaktes Release, etwa ein getaggtes v5.x-Submodule oder eine exakte Version in `package.json`. Listen Sie alle Abhängigkeiten in der Audit-Dokumentation auf und kennzeichnen Sie kopierte oder angepasste Dateien.
- [ ] 08 **Toten Code, TODOs und Debug-Ausgaben entfernen.** Löschen Sie ungenutzte Funktionen, auskommentierten Code, `console.log`-Imports und übrig gebliebene Test-Helfer aus den Verträgen im Scope. Offene TODOs signalisieren unfertige Logik und werden auch so gemeldet.
- [ ] 09 **NatSpec für externe und öffentliche Funktionen vervollständigen.** Jede externe und öffentliche Funktion braucht `@notice`, `@param`, `@return` und, wo relevant, einen Hinweis zu Zugriffsrechten und Reverts. Verwenden Sie Custom Errors und emittieren Sie Events für jede Zustandsänderung, die off-chain relevant ist.
- [ ] 10 **Slither ausführen und jedes Finding triagieren.** Führen Sie `slither` auf dem eingefrorenen Commit aus und ordnen Sie jedes Finding als behoben, False Positive oder akzeptiert ein, jeweils mit einer kurzen Begründung. Ein zweites Tool wie Aderyn findet oft andere Muster, und die geteilte Triage zeigt den Auditoren, welche automatisierten Findings bereits erledigt sind.

3. Tests

Tests zeigen den Auditoren, wie der Code gedacht ist, und ermöglichen ihnen, schnell Proofs of Concept zu schreiben.

- [ ] 11 **Jede externe Funktion abdecken und den Report bereitstellen.** Jede externe und öffentliche Funktion braucht mindestens einen Test des Normalfalls. Erzeugen Sie mit `forge coverage` einen Line- und Branch-Coverage-Report und erklären Sie nicht abgedeckte Branches, statt sie zu verstecken.
- [ ] 12 **Revert-Pfade und Grenzfälle testen.** Prüfen Sie mit `vm.expectRevert`, dass unberechtigte Aufrufe, ungültige Parameter, Nullbeträge und Grenzwerte mit dem erwarteten Custom Error revertieren. In ungetesteten Revert-Pfaden verstecken sich Validierungs- und Zugriffsfehler.
- [ ] 13 **Fork-Tests gegen echte Integrationen ausführen.** Wenn das Protokoll mit externen Verträgen wie DEXes, Lending-Märkten oder Oracles interagiert, testen Sie gegen einen Mainnet- oder L2-Fork mit fixierter Blocknummer. Mocks beweisen nur, dass Ihr Code mit Ihren Annahmen über die Gegenseite funktioniert.

4. Fuzzing und Invariantentests

Fuzzing findet Eingabekombinationen, an die niemand gedacht hat, und Invariantentests prüfen, dass das System über ganze Aufrufsequenzen konsistent bleibt.

- [ ] 14 **Jede Funktion mit numerischen Eingaben fuzzen.** Schreiben Sie Foundry-Fuzz-Tests für Beträge, Zeitstempel, Share-Berechnungen und Gebührenlogik und begrenzen Sie Eingaben mit `bound()` statt mit übermäßigem `vm.assume`. Typische Ziele sind Rundungsrichtung und Überläufe bei Extremwerten.
- [ ] 15 **Stateful Invariantentests mit Handlern schreiben.** Nutzen Sie Foundry-Invariant-Testing mit Handler-Verträgen, die das System in realistischen Sequenzen aus Sicht mehrerer Akteure aufrufen. Prüfen Sie nach jeder Sequenz Solvenz, Werterhaltung und Zugriffsrechte.
- [ ] 16 **Schriftliche Invarianten und Testsuite synchron halten.** Jede Invariante aus Ihrer Spezifikation sollte einem Invariantentest entsprechen, und jeder Invariantentest sollte auf die Spezifikation zurückführbar sein. Führen Sie die Suite in der CI mit sinnvoller Anzahl an Runs und Tiefe aus, nicht nur lokal mit Standardwerten.

5. Zugriffskontrolle und Upgradefähigkeit

Privilegierte Funktionen und Upgrades können jeden Vermögenswert im System bewegen oder sperren – das Vertrauensmodell muss daher explizit sein.

- [ ] 17 **Jede privilegierte Rolle und Funktion auflisten.** Dokumentieren Sie jede Rolle, welche Funktionen sie aufrufen darf, wer sie hält und was im schlimmsten Fall passiert, wenn ihr Schlüssel kompromittiert wird. Nach diesem Abschnitt zu Vertrauensannahmen fragen Auditoren als Erstes.
- [ ] 18 **Admin-Rechte hinter Multisig und Timelock legen.** Kritische Parameter und Upgrades gehören unter die Kontrolle eines Multisigs wie Safe, mit einem `TimelockController` oder einer vergleichbaren Verzögerung für Änderungen, die Nutzergelder betreffen. Dokumentieren Sie Signer-Schwelle und Verzögerung.
- [ ] 19 **Storage-Layout upgradefähiger Verträge prüfen.** Verwenden Sie für Proxies die upgradefähigen Verträge von OpenZeppelin v5 mit ERC-7201-Namespaced-Storage und validieren Sie Upgrades mit dem OpenZeppelin-Upgrades-Tooling. Liegt ein Upgrade im Scope, legen Sie den Storage-Layout-Diff zwischen den Versionen bei.
- [ ] 20 **Initializer und Upgrade-Funktionen schützen.** Rufen Sie `_disableInitializers()` im Konstruktor der Implementierung auf, setzen Sie die Modifier `initializer` und `reinitializer` korrekt ein und beschränken Sie `_authorizeUpgrade` bei UUPS-Proxies (ERC-1822). Ein ungeschützter Implementierungsvertrag ist ein Finding, das Auditoren nie melden müssen sollten.

6. Externe Integrationen

Jeder externe Aufruf ist eine Annahme über fremden Code, und jede Annahme sollte dokumentiert und getestet sein.

- [ ] 21 **Veraltete und ausfallende Oracles behandeln.** Prüfen Sie `updatedAt` gegen den Heartbeat des Feeds, lehnen Sie Preise von null oder darunter ab und prüfen Sie auf L2s den Sequencer-Uptime-Feed. Legen Sie fest und dokumentieren Sie, was das Protokoll tut, wenn das Oracle nicht verfügbar ist.

- [ ] 22 **Token-Besonderheiten berücksichtigen.** Legen Sie fest, welche Token-Typen unterstützt werden, und behandeln oder schließen Sie Fee-on-Transfer-, Rebasing-, Nicht-18-Dezimalstellen- und nicht standardkonforme ERC-20-Token ausdrücklich aus. Nutzen Sie `SafeERC20` für Transfers und messen Sie Saldodifferenzen, wo der tatsächlich erhaltene Betrag zählt.
- [ ] 23 **Reentrancy-Angriffsflächen erfassen.** Listen Sie jeden externen Aufruf und jeden Token-Transfer auf, folgen Sie Checks-Effects-Interactions und nutzen Sie `ReentrancyGuard` (oder `ReentrancyGuardTransient` mit EIP-1153), wo Zustand geteilt wird. Berücksichtigen Sie funktionsübergreifende und Read-only-Reentrancy sowie Callbacks von ERC-721-, ERC-1155- und ERC-777-Token.
- [ ] 24 **MEV und Front-Running berücksichtigen.** Versehen Sie Swaps und Einzahlungen mit Slippage-Limits und Deadlines, schützen Sie ERC-4626-Vaults gegen den Inflationsangriff des ersten Einzahlers und prüfen Sie jede Logik, die von der Transaktionsreihenfolge abhängt. Dokumentieren Sie, welche Reihenfolgerisiken Sie akzeptieren.

7. Deployment und Betrieb

Auditierter Code ist nur so sicher wie die Art, wie er deployt und betrieben wird.

- [ ] 25 **Deploy-Skripte reproduzierbar machen und Parameter dokumentieren.** Deployen Sie mit Foundry-Skripten vom eingefrorenen Commit, halten Sie jeden Konstruktor- und Initializer-Parameter in versionierter Konfiguration und nehmen Sie die Skripte in den Scope oder zumindest in das Übergabepaket auf. Ein korrekter Vertrag mit falschen Parametern ist trotzdem fehlerhaft.
- [ ] 26 **Auf einem Testnet mit verifizierten Quellen proben.** Führen Sie genau dasselbe Deployment-Skript auf einem Testnet und auf einem Mainnet-Fork aus, verifizieren Sie die Quellen im Block-Explorer und prüfen Sie, dass die Rollen bei den vorgesehenen Adressen gelandet sind. Geben Sie den Auditoren die Testnet-Adressen, damit sie mit einem echten Deployment arbeiten können.
- [ ] 27 **Pause, Runbook und Monitoring vorbereiten.** Legen Sie fest, wer was pausieren darf, schreiben Sie ein kurzes Incident-Runbook und richten Sie Alarme für Rollenänderungen, Upgrades, große Abhebungen und Pausenzustände ein. Auditoren prüfen die Notfallpfade, also müssen diese vor dem Audit existieren.

8. Audit-Organisation

Gute Organisation hält das Audit in Bewegung und sorgt dafür, dass die Fix-Runde tatsächlich genutzt wird.

- [ ] 28 **Technischen Ansprechpartner und Kanal benennen.** Bestimmen Sie einen Entwickler, der die Codebasis kennt und Fragen innerhalb von Stunden beantworten kann, und vereinbaren Sie einen gemeinsamen Kanal für die Dauer des Audits. Jede unbeantwortete Frage bremst das Review.
- [ ] 29 **Zeit für Fixes und eine Fix-Review-Runde einplanen.** Planen Sie Entwicklerkapazität direkt nach Eingang des Berichts ein und vereinbaren Sie vorab, dass die Auditoren die Fixes prüfen. Ungeprüfte Fixes können nach Abschluss des Audits neue Fehler einführen.
- [ ] 30 **Frühere Audits und Reviews offenlegen.** Teilen Sie frühere Audit-Berichte, deren Fix-Status und interne Reviews. So können sich die Auditoren auf die Änderungen konzentrieren und prüfen, dass frühere Findings nicht zurückgekehrt sind.

Wie wir helfen können

Wir bieten einen Audit-Readiness-Sprint mit festem Umfang über ein bis zwei Wochen an: Threat Model, Gap-Analyse der Testsuite, Foundry-Fuzz- und Invariantentests, Slither-Triage, eine priorisierte Liste von Fixes und ein Audit-Übergabepaket mit Scope, Dokumentation und bekannten Problemen. Wir sind keine Audit-Firma, und der Sprint ersetzt kein externes Audit; er bereitet Ihren Code so vor, dass die Auditoren ihre Zeit in die Logik investieren können. Details finden Sie auf unserer Preisseite und bei unseren Blockchain- und Web3-Leistungen.

Für eine kostenlose Einschätzung Ihrer Codebasis erzählen Sie uns von Ihrem Projekt. Gerne unterzeichnen wir vorab ein gegenseitiges NDA, bevor Sie Code teilen.