Phage
SECURITY

Before-audit checklist

1. Have complete code (skip if you are interested only in the security part)

2. Writing tests
- How much coverage is enough?
- Unit tests vs integration tests?

3. Internal reviews
- Code reviews before audit, is it worth it?
- How and what to do

4. Clear documentation
- What should/shouldn't it have?

5. Scope definition
- What should be in scope?

6. Choose the right audit firm
- How to pick the right guys, the first time

Have complete code

Before you consider any sort of security, you must first have a MVP/POC. That means an actual piece of code that should be working on its own. All user paths from first interaction (deposit, stake, lock, etc.) to last (withdraw, redeem...) should be working with the intended variable inputs without reverting or leading to any errors.

If you have any integrations, make sure you follow their best practices (thoroughly read their docs, or at least make Claude read them and correct any code that goes against them). If you are unsure about some functions, most (if not all) projects have supportive developers that you can reach out to and who would be happy to look at your code and help with anything you are missing.

If you are doing heavy integrations, I would even suggest reaching out to them, as they know their code best and have seen many projects integrate beforehand and know what works and what doesn't.

Writing tests

The most dreaded part.

Most people would look at what coverage you have to determine your test suite. However, coverage is an inaccurate stat. It simply determines if your tests go through a given path, skipping the fact that you might have tested it wrongly, or with the best possible conditions.

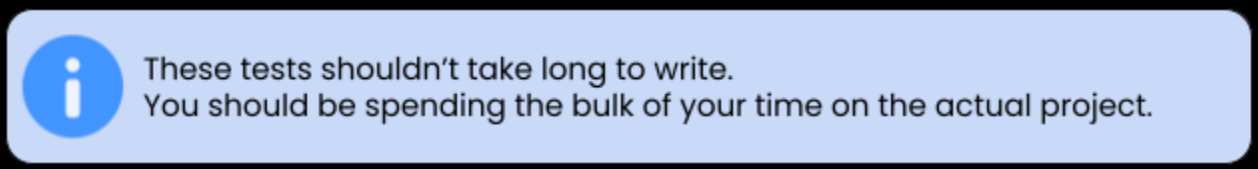
The amount of tests you want to write depends on how fast you want to launch (you can write tests during an audit, although I suggest having everything beforehand).

If you are in a rush, you would need the most basic tests:

1. Simple function tests without malicious intent (i.e., just regular user interactions) -> deposit, claim, withdraw
2. Simple user paths without malicious intent -> `deposit -> withdraw` and `stake -> claim -> withdraw`
3. Access control -> if users can call `onlyOwner` functions or if users can `withdraw` from other user accounts
4. Simple protocol integrations -> mock calls for the protocol you are integrating and distribute yield, then try claiming it

However, if you have a spare day or two, you should consider testing more thoroughly (all of the above and):

1. Users with malicious intent calling functions/flows -> try simulating first depositor bug if you are using 4626 or deposit/withdraw 1 wei
2. Advanced function flows -> `user1 deposit -> user2 withdraw -> user1 claim`
3. Integration tests to see if your integrations actually work



Notice that I said "making" as for tests you don't need to write them on your own. You can (and I suggest you) use AI. I prefer Cursor. The way to make it write the correct tests the right way is simple:

1. Make it create a testing diagram for every function and user flow (make sure it's correct)
2. Prompt it to make a complete setup (initialize and configure every contract) with a few user accounts
3. Start writing tests one by one
4. Correct it where necessary

You won't write them yourself, but you'll be supervising their creation. This would speed it up by 2-3 times.

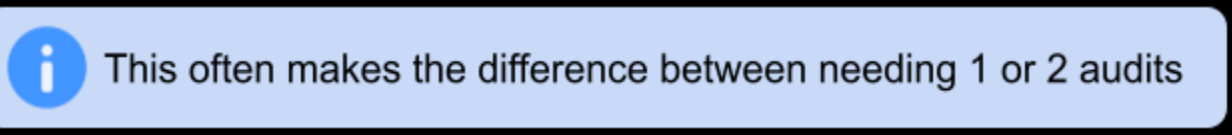
Internal reviews

You've written a lot of tests, but often they miss unique paths and "strange" conditions.

This is why you should audit your own contracts. Paying your developers (or doing it yourself) would cost you 2-3 days of "*no progress*" on your code. However, that can save you an extra audit.

What would you rather do

a) pay your devs to work a few more days ($300-500)
b) get an extra audit ($12k+)?



Before you start, I would recommend setting a time window that would force you to do about ~400 nSLOC per day (i.e., 1000 nSLOC -> 2-3 days). Since this is your own code and you know it well, you won't need extra time to understand it.

During that time, your only goal is to break everything you can. Don't write new code and don't re-edit old code, but just go around and try to exploit your own project.

Make GitHub issues with the vulnerabilities (or note them somewhere) and continue breaking your code. After the time is up, start fixing them as best as you can.

After you are done, make sure all of the tests run.

Clear documentation

This should not be complex. Don't overengineer it. No one needs GitBooks for every smart contract with every function and every variable and what they do. If they want that, they can open your GitHub and read the contracts (or interfaces) and get the info they need.

Your docs should be simple to understand, concise, and to the point. Again, use Claude to write them. It takes only a few minutes (with a good prompt), maybe an hour if you want them polished. Also, if you happen to have diagrams and schemes, feel free to show them in the docs. They provide a good visual reference.

Make sure you have some minimal nat-spec. You don't need to explain every single line, but have 1 or 2 sentences above functions that give a brief explanation why they are needed. Again Cursor can do that for you.

Scope definition

The simplest part. Get only logic contracts in scope. That means any contracts that are interfaces can be removed from the scope to reduce the nSLOC (and possibly the price). Finding bugs in interfaces is rare, and most are due to wrong interface implementation, which is implemented in the logic (i.e., already covered by the scope).

If that's your 2nd or 3rd audit, you can exclude simple helper functions (or implement OZ/Solady ones) from the scope. Auditors will still look at out-of-scope contracts that integrate with in-scope ones and try to find bugs there. However, note that if the helper functions are too complex, they may also need to be included in scope, just in case.

Choose the right audit firm

Choosing the right firm is not easy. You must consider:
- what your users think (most don't know which ones are good)
- what your VCs think or push you to choose (they still don't know which ones are good, but rely on their brand, which is very dangerous)
- what you and your developers think (these guys you can trust the most).

My first suggestion would be to not let yourself be pushed around by VCs telling you which firm you should pick. Resist them as much as you can and provide logical feedback on which ones you think are better and why. But how do you know which ones are better and why?

To know that you must first find a number of auditing firms and research them. Here is what you should be looking for:

1. **The firm doesn't do the audit, the auditors do**

Brands are a big thing. But in cybersecurity, the brand is 80% client communication and 20% delivering on results.

On top of that, most firms work with contractors, some have employees, but I've seen employees of big firms take things "on the side." So consider that most firms have access **to the same pool of auditors**,, where the same team of 3 can be one price at firmX and 2x more expensive at firmY **for the exact same guys**.

2. **Some auditors are better than others, and some are specialized**

Most (if not all) auditors have a GitHub portfolio/CV with their past experience. You will need it to determine how good they are. Look for contest wins, for which firms they have worked with or overall quantity of audits.

Some auditors are mediocre in general but are highly specialized in X niche (lending, bridges, Hyperliquid, or any interesting topic). It's generally good to have one of these as they would most certainly catch the rare bugs (missed by most).

3. **Some auditors are booked on 2 projects at the same time, reducing their quality of work**

When making deals, make sure to ask if all of the guys would be doing only this audit during the whole timeframe. Again, most big firms usually send a team of 2 juniors or intermediates for the whole audit and 1 lead auditor just to check at the end if they have missed anything.

Steps to take before you choose an auditing firm:
- Get a quote from a few firms
- Ask each about the team they plan on putting and their CVs
- Ask for people familiar with the project type you are building
- Look at each team and consider the price

Short summary of next steps:

1. Make sure your code is finished
2. Spend a few days with Claude writing tests
3. Do an internal review
4. Write short and clear docs
5. Scope your project
6. Research a few auditing firms