

INSTRUCTOR GUIDE AND NOTES

Anatomy of a Solana program

Reading a smart contract the way an attacker does. A 50-minute lesson on the tip vault, the missing check, and the two-line fix.

Series	Hack Your Future, PrivacySafe Foundation's series for building what's next
Audience	Learners with basic programming literacy. No Rust or Solana experience needed. EVM familiarity helps.
Length	50 minutes, with a 30-minute cut
Format	4 slides, live demo, 4 slides, discussion
You will need	Slides (light for print, dark for projecting), the interactive demo page, flash cards, optional Tip Vault source
Facts checked	October 27, 2026. Re-check the links in the reference list before you teach.

PrivacySafe Foundation, Inc. | Hack Your Future | These materials are dedicated to the public domain under CC0 1.0 and shared for the public good. The PrivacySafe and PrivacySafe Foundation names and logos are trademarks and are not included in that dedication.

The lesson in five sentences

1. Programs hold code and accounts hold everything else. The runtime lets only an account's owner program take lamports out of it, while any program may add lamports^[1].
2. The caller writes the account list, so every account a program receives is untrusted input^{[10][6]}.
3. Ask three questions of every account: did it sign, is it ours and the right type, and do these accounts belong together^{[2][3]}?
4. A missing check is a logic bug that compiles cleanly, so tests need hostile account lists, not only valid ones^[5].
5. The fix is a relationship check: `has_one` , `seeds` , or both for defense in depth^[2].

LEARNING OUTCOMES

By the end, students can:

1. Explain the account model: programs are stateless, accounts hold state and lamports^[1].
2. Explain why the caller, not the program, chooses which accounts to pass in^[10].
3. Name the three questions to ask about every account.
4. Find a missing relationship check in a short Anchor struct and fix it with `has_one` or `seeds`^[2].
5. Apply the eight-question checklist to an unfamiliar program.

Session plan

SEGMENT	MINUTES	NOTES
Before the demo, slides 1 to 4	12	Includes the prediction. Collect answers silently.
Live code walkthrough	8	Optional. Show the Withdraw struct if you have the Tip Vault source.
Demo	10	Four runs on the demo page, then the prediction answer.
After the demo, slides 1 to 4	12	Slide 3 (Solana versus EVM) is the longest.
Discussion	8	Two questions. Use pairs for the first three minutes.

30-MINUTE CUT

Skip the live code walkthrough. On the EVM slide, cover only the first three rows. Run the attack and the fix in the demo, and skip the other failures. Use one discussion question.

Before class

- ☐ Open both decks. Use the light version for printing handouts and the dark version for projecting.

- ☐ Open the demo page in a browser tab and set zoom to about 125% for the back row. Run "Mallory's attack" once so you know what appears.
- ☐ Optional: open the Tip Vault source in an editor with the `Withdraw` struct visible.
- ☐ Print the flash cards, double-sided, flipped on the long edge. Cut along the dashed lines.
- ☐ Decide how you will collect predictions: paper, a show of hands, or a chat poll.
- ☐ Optional: the lesson page has a walkthrough video of about five minutes, in a light version for the page and a dark version for projecting, each with a subtitles file. Play it as a recap, or share it with learners who missed the session. Check that it plays on the room computer first.
- ☐ Check access needs. The demo works with the keyboard, and status is shown with a symbol and text as well as color.

WHAT THE DEMO IS, AND IS NOT

The demo is a teaching simulation that runs in the browser. Addresses are shortened and made up, and the rent floor is a round illustrative number. It does not talk to a Solana cluster. The error names and numbers it shows are the real Anchor ones^[4], and the instruction data uses the discriminator method Anchor documents.

Slide notes: before the demo

Slide 1: Title and setup

about 2 min

SAY

This is a tip vault: about ninety lines of Rust written with Anchor. Anyone can tip into a vault, and only its owner can withdraw. Our job is to find the lines that make that promise true, then break one.

DO

Frame the session as attacker-style reading. We read for what is missing, not only for what is there.

CHECK FOR UNDERSTANDING

Ask: "If you were attacking this, what would you try first?" Good answers: pass someone else's vault, or sign as someone else.

WATCH FOR

Learners may expect an exotic exploit. It will be one absent line.

Slide 2: Programs hold code, accounts hold everything else

about 4 min

SAY

- The Tip Vault program is executable and stateless. It owns the vaults. The System Program owns the wallets.
- The runtime's one hard rule: only an account's owner program can take lamports out of it. Any program can add lamports^[1].
- Everything else is up to the program's own checks.

DO

Point at the arrows labeled "owns". Ask learners to say who can debit Alice's vault at the runtime level.

CHECK FOR UNDERSTANDING

"Who can take lamports out of Alice's vault at the runtime level?" Expected: only the Tip Vault program. Follow up: "So who decides whether Alice or Mallory gets them?" Expected: the program's own checks.

WATCH FOR

The word *owner* means two different things. The **owner program** is a runtime property of every account^[1]. The vault's **authority** is a field the program stores in the vault's data. Say the difference out loud.

Slide 3: The caller chooses which accounts to pass in

about 4 min

SAY

- Every instruction lists every account it will touch before it runs^[10].
- That list lets non-overlapping transactions run in parallel.

- It also means the attacker writes the list. A program that trusts an account because it showed up has already lost^[6].

Three questions for every account: did it sign (`Signer`), is it ours and the right type (`Account<Vault>`), and do these accounts belong together (`has_one` , `seeds`)^{[2][3]}?

DO

Open the demo page and point at "The transaction on the wire". Do not run anything yet.

CHECK FOR UNDERSTANDING

Ask learners to predict which of the three questions the vulnerable program skips.

WATCH FOR

Learners often assume the runtime stops a wrong account. It stops lamport and data changes the runtime does not allow, not business rules.

Slide 4: What to watch for, and the prediction

about 2 min

SAY

Two lists: how Solana works differently (program ID and upgrade authority, no storage variables, accounts declared up front, PDAs, rent, calling the System Program) and the security checks (owner and type, `Signer`, `has_one` and `seeds` , checked math and the rent floor).

DO

Read the prediction aloud: "If we delete the constraints from Withdraw, who can take the money?" Give 60 seconds of silence. Take a show of hands for four buckets without revealing the answer: only Alice, anyone with a wallet, nobody, only the deployer.

WATCH FOR

Students sometimes answer "nobody, the program will fail". The vulnerable version compiles cleanly.

Demo script (about 10 minutes)

Start on the **Vulnerable** version. The page checks, in order: signer, owner, type tag, relationship, rent floor, then the transfer. In real Anchor code the checks come from the account struct and its types^{[2][3]}; the demo shows them as steps so learners can see where each one sits.

Run 1: the legitimate path

1 min

SAY

Alice withdraws 2 SOL from her own vault. The signer, owner, type and rent checks pass and the money moves. But look at step 4: it is dashed. It never ran.

DO

Click "Alice withdraws her own tips", then Send transaction.

CHECK FOR UNDERSTANDING

Ask: "If this is all my test suite did, would I ever see the bug?" No. The happy path passes with the bug present.

Run 2: Mallory's attack

3 min

SAY

Mallory signs with her own key and passes in Alice's vault. The signer check passes because she signed. The owner check passes because it is a real vault. The type tag matches. Nothing compares `vault.authority` to the signer. Five SOL move to Mallory's wallet.

DO

Click "Mallory's attack" (reset balances first if you ran Run 1), then Send. Walk the steps slowly. Open "The transaction on the wire" and show that the only attacker-controlled input is which vault she put in slot 0.

CHECK FOR UNDERSTANDING

Ask: "Which check failed?" None. The bug is the check that never ran. Every check that ran was correct.

WATCH FOR

Some learners will say Mallory "hacked" the signer check. She did not. She used it correctly. It answers a different question.

Run 3: the fix

2 min

SAY

Switch to Fixed and send the same transaction. It fails with `ConstraintSeeds`, error 2006^[4]. No lamports move.

DO

Point at the highlighted lines in the code panel. `seeds` requires the vault's address to be derived from the signer's key. `has_one = authority` requires the stored authority to equal the signer^[2]. Either one closes this hole. Using both is defense in depth.

CHECK FOR UNDERSTANDING

Optional extra: pick Mallory's vault on Fixed. It succeeds. The fix blocks theft, not withdrawals.

WATCH FOR

In real Anchor, the constraints run in the order the macro generates them. The demo reports the first failing relationship check and does not try to reproduce that order.

Run 4: other failures

2 min

SAY

Pass Alice's wallet as the vault: the owner check rejects it with `AccountOwnedByWrongProgram`, error 3007^[4]. Withdraw too much: the rent floor stops it. A vault cannot drop below its rent-exempt minimum^[1].

DO

Click "Pass a wallet as the vault", then "Withdraw too much".

WATCH FOR

The rent floor in the demo is a round number for teaching. The real minimum depends on the account's size^[1].

Close the prediction

1 min

SAY

With the relationship constraints removed, anyone who can sign a transaction can take the money out of any genuine vault. The signer check and the type check both still pass.

DO

Return to slide 4. Ask who guessed correctly.

Slide notes: after the demo

Slide 1: What Mallory did

about 3 min

SAY

Four steps: she signs, she passes Alice's vault, the type tag matches, and nothing compares the authority to the signer. Every check that ran was correct. The bug is the check that never ran.

DO

Replay the demo run in your own words.

CHECK FOR UNDERSTANDING

Ask learners to say the thesis in one sentence.

Slide 2: The fix is two constraints

about 3 min

SAY

The vulnerable struct has a `Signer` and an `Account<Vault>`. The fixed struct adds `seeds`, `bump = vault.bump` and `has_one = authority` ^[2]. The vulnerable version compiles cleanly. This is a logic bug, not a syntax error.

CHECK FOR UNDERSTANDING

Ask: "Why does the compiler not catch this?" The types are all valid. The compiler cannot know which accounts must match.

WATCH FOR

The slide says the vault's stored authority. If an older copy says "owner", correct it to avoid the owner and authority mix-up.

Slide 3: Why Solana is different

about 5 min

SAY

TOPIC	TEACHING POINT
Where state lives	EVM contracts keep state inside the contract ^[11] . Solana keeps it in separate accounts the program owns ^[1] . This is why an attacker can swap accounts.
Who is calling	The EVM has one implicit <code>msg.sender</code> ^[12] . Solana has explicit Signer accounts, possibly several, and you must check them.
What a transaction touches	Declared up front, so transactions run in parallel ^[10] . Speed and attack surface come from the same design.
Paying for storage	A refundable rent-exempt deposit sized to the account ^[1] , not gas per storage write.

Upgrades	Solana programs are upgradeable by default through an upgrade authority ^[8] . EVM contracts are immutable unless you add a proxy ^[11] .
Reentrancy	A classic EVM attack class ^[13] . On Solana the runtime allows only direct self-recursion and caps call depth ^{[14][7]} . Say "much narrower", not "impossible".
Typical bug	EVM: reentrancy and unchecked external calls. Solana: missing signer, owner or relationship checks ^{[6][5]} .

CHECK FOR UNDERSTANDING

For mixed audiences, ask EVM learners to translate "missing relationship check" into their world. A common answer is a missing `require(msg.sender == owner)`. Note the difference: on the EVM you check one implicit value, on Solana you also validate every account the caller hands you.

Slide 4: Reading any Solana program

about 4 min

SAY

1. Is every authority account a `Signer`? Otherwise anyone can pass a key without its owner's consent^[5].
2. Is every data account checked for owner and type?
3. Is each account tied to the others with `has_one` or `seeds`?
4. Are PDAs derived with the canonical bump^{[2][9]}?
5. Is the target of every cross-program call pinned^[7]?
6. Is all arithmetic checked?
7. Can a closed account be revived or reinitialized?
8. Who holds the upgrade authority^[8]?

DO

Optional activity (5 minutes): give pairs a short Anchor program and ask them to answer the eight questions. Each pair reports one "no" or "I cannot tell".

Discussion (about 8 minutes)

QUESTION 1: WHO HOLDS THE UPGRADE AUTHORITY?

"Pick a Solana program you have used. Who holds its upgrade authority, and what could they change tomorrow?"

- Programs are upgradeable by default. The authority can replace the code, which can change every check we studied^[8].
- Authority options: a single key, a multisig, a governed process, or none at all, which makes the program immutable.
- Practical task: look up one program's upgrade authority in a block explorer.
- Balance: upgradeability allows bug fixes. An immutable program with today's bug would be stuck with it.

QUESTION 2: HOW DO WE TEST A CRAFTED ACCOUNT LIST?

"Mallory needed no exploit code, only a crafted account list. What does that mean for how we test these programs?"

- Valid-input tests pass on the vulnerable program, as Run 1 showed.
- Write negative tests: wrong signer, wrong owner, wrong type, mismatched relationship, duplicate accounts, closed accounts^[5].
- Ask: "How would you prove `has_one` is doing its job?" Pass a vault whose authority differs from the signer and assert the specific error^[4].

Answer key and assessment

QUESTION	ANSWER
Who can debit a vault's lamports at the runtime level?	Only its owner program. Any program can credit ^[1] .
Why did the signer check pass for Mallory?	She signed with her own key. Signer proves a key signed, not that it is the vault's authority.
What was missing?	A comparison of <code>vault.authority</code> to the signer.
Two ways to close the hole?	<code>has_one = authority</code> and <code>seeds</code> derived from the signer's key, or both ^[2] .
Which error means the seeds were wrong?	<code>ConstraintSeeds</code> , 2006 ^[4] .
Which error means the wrong program owns the account?	<code>AccountOwnedByWrongProgram</code> , 3007 ^[4] .

EXIT TICKET (3 MINUTES)

1. Name the three questions to ask about every account.
2. In the demo, why did the signer check pass for Mallory?
3. Write the one constraint that would have stopped her.

HOMEWORK

- Give a second vulnerable struct, such as a `close` instruction with a missing `has_one`, and ask for the missing check and the fix.
- Ask for a one-paragraph explanation, aimed at an EVM developer, of why `msg.sender` has no direct equivalent on Solana.
- Stretch: write three negative tests for the fixed Tip Vault: wrong signer, wrong account type, and a withdrawal that breaks the rent floor.

Common questions and misconceptions

QUESTION OR BELIEF	RESPONSE
"Why not just check the signer?"	Someone signed is not the same as the right person signed. Signer proves possession of a key, not authority over the vault.
"Anchor makes programs secure."	Anchor provides types and constraints. It will not add a constraint you forgot ^[3] .
"Reentrancy cannot happen on Solana."	It is far narrower, not absent. Direct self-recursion exists and depth is capped ^[14] ^[7] .
"Upgradeable means insecure."	It means the trust model includes whoever holds the authority ^[8] .
"PDAs are secret accounts."	They are public. They have no private key ^[1] ^[9] .
"Is this a real bug class?"	Yes. Missing signer, owner and relationship checks are standard entries in Solana security references ^[5] ^[6] .

Room and delivery tips

- Project the dark deck in a dim room and use the light deck for handouts.
- If the network is blocked, nothing breaks: the demo and slides work offline.
- Pause after each demo run and ask learners to say what they expect before you click.
- Use the flash cards as a warm-up the next day: pairs quiz each other for five minutes.

References and further reading

Citations in this guide use bracketed numbers. Links were checked on October 27, 2026. Documentation changes over time, so confirm details such as error numbers against the version you teach.

#	SOURCE	WHAT IT SUPPORTS
[1]	Solana documentation, Account Structure https://solana.com/docs/core/accounts/account-structure	Account fields; only the owner can debit lamports, any program can credit; rent-exempt balance as a refundable deposit; PDAs have no private key.
[2]	Anchor documentation, Account constraints https://anchor-lang.com/docs/account-constraints	The signer, mut, seeds, bump and has_one constraints; the canonical bump is used when none is given.
[3]	Anchor documentation, The program module (account validation) https://www.anchor-lang.com/docs/the-program-module	Accounts are validated by constraints and by account types such as Signer and Account.
[4]	anchor-lang crate, ErrorCode enum https://docs.rs/anchor-lang/latest/anchor_lang/error/enum.ErrorCode.html	Error numbers used in the demo: ConstraintHasOne 2001, ConstraintSeeds 2006, AccountOwnedByWrongProgram 3007.
[5]	Coral, Sealevel Attacks (GitHub) https://github.com/coral-xyz/sealevel-attacks	Worked examples of missing signer, owner, type and related account-validation bugs, with fixes.
[6]	Cantina, Securing Solana: A Developer's Guide https://cantina.xyz/blog/securing-solana-a-developers-guide	Solana's attacker-controlled account model; missing ownership, signer and type checks.
[7]	Solana documentation, Cross Program Invocation https://solana.com/docs/core/cpi	How programs call other programs, including the System Program.
[8]	Solana documentation, Deploying programs https://solana.com/docs/programs/deploying	Program deployment and the upgrade authority.
[9]	Solana documentation, Program Derived Address https://solana.com/docs/core/pda	Seeds, bumps and how PDAs are derived.
[10]	Solana documentation, Transactions https://solana.com/docs/core/transactions	Transactions list their accounts up front.

[11]	Ethereum.org, Introduction to smart contracts https://ethereum.org/developers/docs/smart-contracts/	Background on EVM contracts for the comparison slide.
[12]	Solidity documentation, Units and globally available variables https://docs.soliditylang.org/en/latest/units-and-global-variables.html	msg.sender, the implicit caller in the EVM.
[13]	SWC Registry, SWC-107 Reentrancy https://swcregistry.io/docs/SWC-107/	Reentrancy as a classic EVM attack class.
[14]	solana-program crate, InstructionError enum https://docs.rs/solana-program/latest/solana_program/instruction/enum.InstructionError.html	The runtime errors for call depth and reentrancy (CallDepth, ReentrancyNotAllowed).
[15]	Creative Commons, CC0 1.0 Universal https://creativecommons.org/publicdomain/zero/1.0/	The public-domain dedication that covers these lesson materials.

ABOUT THESE MATERIALS

The slides, demo, flash cards and this guide were written for the PrivacySafe Foundation Hack Your Future series. They are dedicated to the public domain under CC0 1.0^[15]. The Foundation names and logos are trademarks and are not included in that dedication.