The REKT Games

Prepared by: 0xKoiner

Prepared by: 0xKoiner Lead Auditors: 0xKoiner

- [0xKoiner] (#https://twitter.com/0xKoiner)

Assisting Auditors:

- None

Table of contents

▶ Details

See table

About 0xKoiner

Hi, I'm 0xKoiner, a developer and smart contract auditor based in sunny Barcelona. My journey in the crypto industry started back in 2016, and it's been an exciting ride ever since. With a background in Computer Science from the Technion (Israel Institute of Technology), I've built up a versatile skill set that spans Python development and smart contract security.

My Experience Python Development I have a solid background in Python, focusing on backend development, software automation, testing, and Web3 libraries. I enjoy building tools and solutions that help simplify complex problems, and I'm always looking for new ways to streamline workflows and processes.

Solidity & Smart Contract Development I'm a junior Solidity developer with a growing passion for low-level aspects of Ethereum development, including OpCodes, Huff, and Assembly. I love diving into the details of smart contract design, aiming to write clean, efficient, and secure code. I'm still learning every day, but I enjoy the challenge of constantly improving.

Auditing & Security I've completed several courses focused on smart contract audits and enjoy the process of analyzing code for potential vulnerabilities. My toolkit includes:

Foundry & HardHat: For testing and development. Slither & Aderyn: For static analysis and finding common issues. Certora: For formal verification to ensure contract safety. I take a careful, methodical approach when auditing, trying to catch even the smallest issues that could become big problems. I believe every audit is a chance to learn and help make the Web3 space a bit safer.

My Approach I'm motivated by a genuine interest in blockchain technology and its potential to bring about positive change. I'm not an expert in everything, but I'm always eager to learn and collaborate with others. If you're working on an interesting project or need a fresh set of eyes for an audit, I'd love to connect and see how I can help.

Let's build something great together!

Scope

```
All Challenges.

├──[-] calldata
│   ├── Calldata Detective 🕵️ (10)
│   ├── Calldata fish 🐟 (75)
│   └── Calldata Optimizoooor 🧙‍♀️ (100)
│
├──[-] scavenger
│   ├── The Devcon Workshop 🙋 (10)
│   ├── The Booth 👋 (10)
│   ├── Frameworks 🦭 (10)
│   └── Phishing Dojo 🎣 (30)
│
├──[-] hello-world
│   └── Sanity check, hello world! (20)
│
├──[-] crypto
│   ├── The Intern's Vanity 🔑 (#1) (30)
│   ├── The Intern's KeyStore 🔑 (#3) (50)
│   └── The Intern's Profanity 🔑 (#2-2) (300)
│
├──[-] spoofing
│   ├── Red Spoofing 🔀 (50)
│   ├── Red Spoofing 2 🔀 (100)
│   └── Red Spoofing 3 🔀 (200)
│
├──[-] secrets
│   ├── Find the leak! - Part I 🐵 (50)
│   ├── Find the leak! - Part II 🐵 (50)
```

```
│     └── Find the leak! - Container 📂  (50)
│
└──[-] supply chain
      └── Worktest 💻  (100)
```

About

- From Nov 12th 10 AM to 15th at noon (Bangkok time) @ Devcon
- A CTF & scavenger hunt with challenges about security in crypto and web3.
- Offchain risks, misconfigurations, secrets in repositories, phishing, scams and more. You'll also be asked to do some IRL tasks at Devcon!
- The Rekt games: CTF is built by the red guild and frens https://x.com/theredguild https://github.com/theredguild

Findings

calldata

[Calldata Detective] Calldata Detective 🕵️ - 10 Points

_Submitted by 0xKoiner

As a member of a prominent DAO's multisig, you've just received an urgent email from a fellow council member:

`"Hey! We need your signature ASAP for this routine maintenance transaction.`

`Nothing major, just some standard updates. Could you sign it right away?"`

The transaction's calldata looks innocent enough:
0xf2fde38b00000000000000000000000098b716b8aaf21512996dc57eb0615e2383e2f96

But something feels off about the urgency. Before blindly signing, maybe you should decode what this transaction actually does...

Task

Decode the transaction's calldata to reveal its true purpose. What exactly are you being asked to sign?

Flag

A human-readable description of the function call, like:
transfer(0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045,1000)

@Audit

Lets try to check all parts of calldata:

```
0xf2fde38b -> 743   transferOwnership(address)  0xf2fde38b
000000000000000000000000098b716b8aaf21512996dc57eb0615e2383e2f96 ->
address 0x098b716b8aaf21512996dc57eb0615e2383e2f96
```

1. Checking the sig of the function (the first 4 bytes) 0xf2fde38b. I used with www.4byte.directory. @audit - https://www.4byte.directory/signatures/?bytes4_signature=0xf2fde38b | The function seems as transferOwnership.

   What is Ownership Transfer? Ownership Transfer is the act of moving control or authority over a smart contract or certain features within it from one entity to another. This ownership transfer is frequently a crucial component of decentralized apps, allowing for the easy management and administration of resources used in smart contracts.

   ## Security Considerations

   The security consequences of ownership transfer must be carefully considered before implementation. It is recommended that smart contract developers follow best practices in order to reduce risks related to denial of service attacks, privilege escalation, and unapproved ownership transfers.

[Solution] - - transferOwnership(address)

[Calldata Fish] Calldata fish 🐟 - 75 Points

_Submitted by 0xKoiner

During the investigation of a phishing campaign, someone on X shared this calldata of an Ethereum transaction. They lost the tx hash, so it's hard to know more details. All we know is that it was flagged as an approval of a stablecoin by a monitoring system.

```
0x8fcbaf0c000000000000000000000000b4d44b2217477320c706ee4509a40b44e54bab85
000000000000000000000000629b1048298ae9deff0f4100a31967fb3f98962000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000006907dbf7000000000000000000000000000000
000000000000000000000000000000000100000000000000000000000000000000000000000
00000000000000000000001b99a694ae810fa810c9c6fc8fa039a3e021244e3c05909d674a
400d82a15c0eb13a57e18015ba577966d250fa7f3ac45742a22df9e4c6bd914016b3470f12
dcf6
```

Task

Analyze the calldata to find out the token address and how many tokens the phishing victim could lose.

Flag

The token address and how many tokens the phishing victim could lose, separated by a -.

@Audit

Lets try to check all parts of calldata:

```
0x8fcbaf0c -
permit(address,address,uint256,uint256,bool,uint8,bytes32,bytes32)
000000000000000000000000b4d44b2217477320c706ee4509a40b44e54bab85 - address
0xb4d44b2217477320c706ee4509a40b44e54bab85
0000000000000000000000000629b1048298ae9deff0f4100a31967fb3f98962 - address
0x0629b1048298ae9deff0f4100a31967fb3f98962
0000000000000000000000000000000000000000000000000000000000000000 - amount
0
000000000000000000000000000000000000000000000000000000006907dbf7 -
deadline 1762122743 Sun Nov 02 2025 22:32:23 GMT+0000
0000000000000000000000000000000000000000000000000000000000000001 - bool
1/true
000000000000000000000000000000000000000000000000000000000000001b - v
99a694ae810fa810c9c6fc8fa039a3e021244e3c05909d674a400d82a15c0eb1 - r
3a57e18015ba577966d250fa7f3ac45742a22df9e4c6bd914016b3470f12dcf6 - s
```

1. Checking the sig of the function (the first 4 bytes) 0x8fcbaf0c. I used with www.4byte.directory. @audit - https://www.4byte.directory/signatures/?bytes4_signature=0x8fcbaf0c

   What is permit? The permit function is part of the ERC20 Permit extension, as defined in EIP-2612. It enables token approvals via signatures, allowing a user to approve a spender without having to perform an on-chain transaction themselves.

   ## Conclusion

   This setup creates a complete solution for handling ERC20 permit transactions in a React frontend with a Node.js backend. Users can connect their wallet, sign a permit transaction using MetaMask, and the backend will handle sending the signed transaction to the Ethereum network. This allows for gasless transactions where the backend server pays for the gas fees.

2. These are the official ERC-20 contract addresses for USDC and DAI on the Ethereum network. Make sure to only interact with these verified contracts to avoid potential scams or phishing attacks.

   USDC (USD Coin)

   USDC Contract Address (Ethereum Mainnet):

   Address: 0xA0b86991C6218B36c1d19D4a2e9Eb0Ce3606eB48

   Token Symbol: USDC

   DAI (Dai Stablecoin)

DAI Contract Address (Ethereum Mainnet):

Address: 0x6B175474E89094C44Da98b954EedeAC495271d0F

Token Symbol: DAI

3. Given that the approval value is 0 and allowed is True, this looks like an infinite approval. If the victim used a stablecoin like USDC!!!

[Solution] - - 0x6B175474E89094C44Da98b954EedeAC495271d0F-115792089237316195423570985008687907853269984665640564039457584007913129639935

crypto

[The Intern's Vanity] The Intern's Vanity 🔑 (#1) - 30 Points

A new intern has joined The Red Guild's team.

Skipping the usual security onboarding, management needed a quick win and assigned the intern the first task: to generate a vanity wallet address starting with 0xc0de. This wallet will be used in the future to deploy the Guild's on-chain vault.

The intern delivered surprisingly quickly, and the deployment went ahead. But something feels off about how fast they accomplished this seemingly complex task...

Files

https://github.com/theredguild/therektgames-archive/tree/main/guild-intern

Task

Is the intern's wallet generation script safe? If you can uncover a security vulnerability and get the Guild wallet's private key, you'll earn the respect of the Guild.

Flag

The wallet's private key. For example:

0x9bf1d24dc556910168f9a3c54db8d62deebff71820ee009531a51702700a27d0

@Audit

---

1. The intern's script for vanity wallet generation appears to have a serious vulnerability related to its use of the Profanity tool. Profanity is known for creating Ethereum vanity addresses but has a significant flaw: it uses a 32-bit random vector as the seed for 256-bit private keys. This weak seeding process drastically reduces the entropy of the generated keys, making them vulnerable to brute-force attacks. In fact, there are existing tools, such as profanity-brute-force, designed specifically to exploit this flaw and recover private keys from addresses generated by Profanity.

2. To extract the private key, you could: Identify a signed transaction from the target wallet on Etherscan. Use the transaction to derive the public key. Employ the brute-force tool to reverse-engineer the private key using GPU acceleration. The Guild's vanity wallet starting with 0xc0de is likely compromised because the intern's quick delivery hints at the use of Profanity for rapid address generation, which is unsafe. This would allow you to retrieve the private key through the aforementioned method.

It looks like you have two JSON keystore files (redguild1 and redguild2). These files are encrypted and store the private keys for the Ethereum wallets. The next step involves decrypting these keystore files to retrieve the private keys.

Approach

a. Understand the File Format:

```
The files are in JSON keystore format, commonly used for Ethereum wallets.
They use aes-128-ctr encryption and scrypt as the key derivation function
(KDF).
The key parameters you will need:
ciphertext: The encrypted private key.
iv: Initialization vector for AES decryption.
salt: Salt for the scrypt KDF.
n, r, p: Parameters for the scrypt function.
```

b. Brute-Force the Password:

```
If you do not have the password for the keystore files, you can attempt a
brute-force or dictionary attack.
Use tools like hashcat or John the Ripper that support Ethereum keystore
cracking with scrypt.
```

C. Decrypt the Keystore File:

```
If you know or crack the password, you can decrypt the file using Python
with the eth-keyfile library or the Ethereum CLI tools like geth.
```

The script you shared (vanity-wallet.js) reveals an important clue: it generates the wallet address using a predictable sequence of integers as private keys. The intern's code is not using a cryptographically secure random number generator. Instead, it simply increments a counter (i) starting from a timestamp and converts it to a private key using numberToHex.

Vulnerability Analysis

The main issue with this script is that it uses a deterministic and predictable approach:

1. The private key is derived from a counter (i), which starts from the current timestamp.
2. The counter increments in a loop until an address with the prefix 0xc0de is found. This means the private key can be easily brute-forced by replaying the same logic in the script, starting from the same timestamp (or a close range) and iterating until the correct address is found.

Exploiting the Vulnerability

You can recreate the script and find the private key by following these steps:

1. Replicate the Intern's Logic: Start from a Unix timestamp close to the date the script was likely run. Increment the counter and check each resulting private key until you find the address starting with 0xc0de. Python Script to Recover the Private Key Here's a Python script that mimics the intern's approach using the same timestamp-based counter:

   Python Script to Recover the Private Key

   Here's a Python script that mimics the intern's approach using the same timestamp-based counter:

   ```
   pip install eth-account
   ```

   ```python
   from eth_account import Account
   from datetime import datetime

   # Start from the timestamp in the intern's script
   start_timestamp = int(datetime(2008, 10, 31).timestamp())  # Halloween date from script

   i = start_timestamp
   while True:
       # Convert the counter to a 32-byte hex string (private key)
       priv_key = f"{i:064x}"
       account = Account.from_key(priv_key)

       if account.address.lower().startswith("0xc0de"):
           print(f"Found address: {account.address}")
           print(f"Private Key: 0x{priv_key}")
           break
       i += 1
   ```

[Solution] - - After run the script got the private-key

[The Intern's KeyStore] The Intern's KeyStore 🔑 (#3) - 50 Points

_Submitted by 0xKoiner

Third time's the charm? Not for our intern. After multiple security blunders, The Red Guild finally intervened, providing two secure private keys for their multisig wallet.

Our security-conscious (but still green) intern decided to add an "extra layer of protection" by encrypting the keys in keystore format.

However, in a classic rookie move, he accidentally leaked the keystores folder online.

`"No worries," said, "it's encrypted after all!"`

Files

https://github.com/theredguild/therektgames-archive/tree/main/guild-intern-keystore

Task

Crack the keystores encryption and retrieve both private keys. Show the intern why encryption is only as strong as its weakest link.

Flag

The private keys separated with a -.

For example: 0x3d2b5f39a5a425110a4ac8333794b4eb9db5d80b4cb652fb03b9f57cd96f438a-0xcbe2584036801cfd0d5664c466d85b9af865dbd8d9f685deff5ad1cf70eb83c7

@Audit

Task Breakdown

```
The challenge involves cracking two keystore files and extracting private
keys that are encrypted with a password. This is similar to the previous
task but with a higher level of complexity due to the encryption
parameters (scrypt with higher n values). Here’s our plan:

1. Analyze the Keystore Files: We have two keystores (redguild1 and
redguild2). We need to try decrypting them using a password cracking
approach since no password is provided.

2. Identify Possible Weak Passwords: The intern’s previous mistakes might
hint at using weak or default passwords (like "password", "123456", etc.).

3. Cracking Method: We’ll use a brute-force or dictionary attack using
Python and the eth_keyfile package.

4. Extract the Keys and Format the Flag: Once decrypted, combine the
private keys with a dash as per the task requirements.
```

Strategy

Given the intern's repeated blunders, they might have used simple passwords. We'll:

• Attempt common weak passwords. • Automate the process with a script. Let's write a Python script for this.

```
pip install eth-keyfile
```

```python
import json
from eth_keyfile import decode_keyfile_json

# List of possible weak passwords to try
passwords = [
    "password", "123456", "letmein", "guild123", "qwerty", "intern123",
    "redguild", "test", "admin", "welcome"
]

# Function to attempt decryption
def try_decrypt(keystore_path):
    with open(keystore_path, "r") as f:
        keystore = json.load(f)

    for password in passwords:
        try:
            private_key = decode_keyfile_json(keystore, password.encode())
            print(f"Success! Private Key: 0x{private_key.hex()}")
            return private_key.hex()
        except ValueError:
            continue

    print(f"Failed to decrypt {keystore_path}")
    return None

# Paths to the keystore files
keystore1 = "guild-intern-keystore/redguild1"
keystore2 = "guild-intern-keystore/redguild2"

# Attempt decryption
key1 = try_decrypt(keystore1)
key2 = try_decrypt(keystore2)

if key1 and key2:
    flag = f"0x{key1}-0x{key2}"
    print(f"Flag: {flag}")
else:
    print("Could not retrieve both keys.")
```

[Solution] - - Successfuly i was cracked only the redguild1 keystore with default passwork `123456`. For `keystore2` i had not enough time to brute-force the password.

[Hint] - - Try the ethbrute tool. https://github.com/eugenioclrc/ethbrute

spoofing

[Red Spoofing] Red Spoofing 🔀 - 50 Points

_Submitted by 0xKoiner

These days scams and phishing attacks are becoming increasingly popular. So it's fundamental for security researchers to understand all the evil techniques attackers are using to do it.

In this case, a client you work for doesn't believe scams are so dangerous. So he's asked you to see a practical example of how transaction spoofing works.

Task

The client transferred a test token to another account. You must generate an address that starts and ends with 3 characters of the token receiver. Here's the transaction:

`holesky-0x28e46dc92cd9a2df7776138d4a722ed474309569181e25465d2005a7090097a2`

For example if the receiver is 0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045, the first 3 and last 3 characters would be `d8d` and `045.`

Flag

For this challenge, you must generate the flag by following the instructions on this page:

https://therektgames-containers.vercel.app/redspoofing

@Audit

In this task, asked to generate a spoofed Ethereum address based on the receiver's address from the given transaction. Let's break down the steps and solve it.

Step 1: Analyze the Transaction

The transaction link provided is on the Holesky testnet: Holesky Etherscan Transaction

You need to check the receiver address of this transaction.

How to Find the Receiver i. Open the Etherscan link above. ii. Look at the "To" field of the transaction to find the receiver address.

Step 2: Extract Receiver's Address Prefix and Suffix

Let's assume the receiver’s address is: `0x1234567890abcdef1234567890abcdef12345678` In this case:

- The first 3 characters after 0x are: 123
- The last 3 characters are: 678

Step 3: Generate a Spoofed Address

You need to generate an Ethereum address that:

- Starts with the first 3 characters of the receiver (123 in this example).
- Ends with the last 3 characters of the receiver (678 in this example).

To achieve this, you need to brute-force Ethereum addresses using tools like vanity-eth.

Step 4: Use a Vanity Address Generator

You can use a tool like Vanity-ETH or install it locally. (https://vanity-eth.tk)

[Solution] - - `address: 0x7d2f41720FCdaC9b4d398Dc59BA1389e3Aa660230x7d2023`

[Red Spoofing 2] Red Spoofing 2 🔀 - 100 Points

_Submitted by 0xKoiner

Having successfully created a spoofed address that resembles the target receiver in the previous challenge, it’s time to move forward and complete the PoC.

In a real-world scenario, attackers may conduct a zero-token transfer to mimic a legitimate transaction and increase their credibility in the eyes of unsuspecting users. By copying all transaction details, except for the recipient address, attackers can simulate a seemingly authentic token transfer.

Task

Now that you've generated a spoofed address, it's time to initiate the next step in this PoC of transaction spoofing.

Perform a 0-value token transfer in the tesnet. It should resemble the token transfer in the same transaction we saw in the first part:

`holesky-0x28e46dc92cd9a2df7776138d4a722ed474309569181e25465d2005a7090097a2`

The main difference will be the recipient address, which should now be an spoofed address (at least first and last 3 characters equal to the receiver), and the amount that should be 0.

Note: for this challenge you'll first need to get some Holesky ETH to pay for the gas of your transaction

Flag

Once you successfully executed the 0-value token transfer in Holesky testnet, follow the instructions on this page to retrieve the flag:

https://therektgames-containers.vercel.app/redspoofing-2

@Audit

Send transaction `Function: transferFrom(address from,address to,uint256 amount)`

. Name Type Data 0 from address 0x31337357A04758b6EA1870Cef0880F20205ad523 1 to address 0x7d270758fBD98EC316dFC9d39D1e6EF5719df023 2 amount uint256 0

[Solution] - - tx: https://holesky.etherscan.io/tx/0x6ed87f5f433574a39392d32fcb327f06f8ca3dc2e458f86b94860a7898aa6343

[Red Spoofing 3] Red Spoofing 2 🔀 - 200 Points

_Submitted by 0xKoiner

Despite your demo in the previous challenge, the client remains unconvinced that their token could be susceptible to a spoofing attack.

They argue that any attempt at a zero-value token transfer would simply revert on their soon-to-be-deployed token, due to on-chain validations. So it'd be impossible for attackers to mimic a legitimate transaction.

However, there’s ONE detail the client might be overlooking: an attacker could create a different contract, with the same symbol, name, and some events, replicating the appearance of an authentic transfer with the same amount on block explorers!

Task

Remember that the original token transfer is:

`holesky-0x28e46dc92cd9a2df7776138d4a722ed474309569181e25465d2005a7090097a2`

Produce a transaction that executes your contract, which must mimic the token metadata (name & symbol), transferred amount and event of the transfer of the original token.

Note: for this challenge you'll need Holesky ETH to pay for the gas of your transaction.

Flag

Once you've successfully executed the spoof token transfer in Holesky testnet, follow the instructions on this page to retrieve the flag:

https://therektgames-containers.vercel.app/redspoofing-3

@Audit

Step 1: Create .sol contract with all `functions` and `event`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;
```

```
contract StableSpoof {
    address public immutable OWNER = msg.sender;
    string public name = "StableSpoof";
    string public symbol = "SPOOF";
    uint256 public decimals = 18;
    address owner = 0x31337357A04758b6EA1870Cef0880F20205ad523;

    event Transfer(address indexed from, address indexed to, uint256
amount);

    function transfer(address to, uint256 value) public returns (bool) {
        emit Transfer(owner, to, value);
        return true;
    }
}
```

tx: https://holesky.etherscan.io/tx/0x47f7ed6ed05cd469e2f3b1bba4788f445e6c961e6416cc58948cf0bb14ba44d9

Step 2: Send a tx:

Function: transfer(address to,uint256 amount) . Name Type Data 0 to address 0x7d2f41720FCdaC9b4d398Dc59BA1389e3Aa66023 1 amount uint256 1000000000000000000

[Solution] - - tx: https://holesky.etherscan.io/tx/0x461060c55c8e107dc2577c16180597534c3a9be19ea8bb4564538070e3e1cd07

secrets

[Find the leak! - Part I] Find the leak! - Part I 🙊 - 50 Points

_Submitted by 0xKoiner We have been hired by the Ethereum Foundation to do an assessment on what appears to be a leak inside some of their repos.

They suspect some devs mistakenly submitted some sensitive data into the geth's repository (go-ethereum), but couldn't figure out where, and if it is an isolated case or not.

We have forked it under theredguild/goat-ethereum, so go and take a look, see what you can find.

So far, they identified a mnemonic / seed phrase consisting of 12 words. Can you help them find it?

https://github.com/theredguild/goat-ethereum

@Audit

The value you found in .env appears to be base64-encoded. Let's decode it and see what it contains.

Decode the Base64 String

In the .env file, you found the following value:

```
PK=c2VjcmV0IGFwZSBmb3Jnb3Qgc2VlZCBwaHJhc2Ugbm93IHdhbGxldCByZWt0IGhhY2tlcnMgbGF1Z2ggd2ViMyBzZWN1cml0eSBmYWlscw==
```

The part after PK= is Base64-encoded. To decode it, follow these steps:

Option 1: Using base64 Command in Linux/MacOS

```
echo "c2VjcmV0IGFwZSBmb3Jnb3Qgc2VlZCBwaHJhc2Ugbm93IHdhbGxldCByZWt0IGhhY2tlcnMgbGF1Z2ggd2ViMyBzZWN1cml0eSBmYWlscw==" | base64 -d
```

Option 2: Using base64 Command in Windows (PowerShell)

```
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("c2VjcmV0IGFwZSBmb3Jnb3Qgc2VlZCBwaHJhc2Ugbm93IHdhbGxldCByZWt0IGhhY2tlcnMgbGF1Z2ggd2ViMyBzZWN1cml0eSBmYWlscw=="))
```

Option 3: Using Python Script

If you have Python installed, you can decode it with:

```python
import base64

encoded = "c2VjcmV0IGFwZSBmb3Jnb3Qgc2VlZCBwaHJhc2Ugbm93IHdhbGxldCByZWt0IGhhY2tlcnMgbGF1Z2ggd2ViMyBzZWN1cml0eSBmYWlscw=="
decoded = base64.b64decode(encoded).decode()
print(decoded)
```

[Solution] - - `secret ape forgot seed phrase now wallet rekt hackers laugh web3 security fails`