rConfig 3.9.4 multiple vulnerabilities

07 Sep 2020 exploit web

Overview

During my OSWE journey I used to try to re-discover known vulnerabilities by watching exploit-db stream to narrow the scope.

Later last year vikingfr worked on rConfig 3.9.x and had found a neat path from zero to root starting with a pre-auth sql injection.

To refresh my code audit skills, this march I decided to make some practice again: let's start the timer and see what I can find.

Flow

Without reading too much information to avoid spoilers (I already know there is at least one sql injection so I'm BIASed), I installed the app and started crawling the tree to get *confident* with the structure.

The source is quite clean and separated between exposed pages, classes included out from the document root, data, and so on. Of course what I'm most interested in, at first, is a pre-auth bug. I choosed to see how authentication works before to look for IDOR.

Login

Login process uses a class Process defined in file *www/lib/crud/userprocess.php*, which handles login requests by calling a login() function from class Session defined in file *classes/usersession.class.php* (line 137 circa), that invokes checkLogin().

```
function preLogin() {
    global $session, $form;
    /* Login attempt */
    $retval = $session->login($_POST['user'], $_POST['pass'], isset($_POST['remember']));
    /* Login successful */
    if ($retval) {
        header("Location: " . $session->referrer);
    }
    /* Login failed */ else {
        $_SESSION['value_array'] = $_POST;
        $_SESSION['error_array'] = $form->getErrorArray();
        header("Location: " . $session->referrer);
    }
}
NORMAL  userprocess.php
```

By looking at the file, I saw that it's not possible to invoke any of these functions by calling them from the file, so if I'll find something vulnerable I also have to find a sink.

As soon as the class is *constructed*, she runs startSession() that runs checkLogin(). The first thing checkLogin() does is to check for some cookies, I'll be back on this later.

She now checks if there is an LDAP server configured. Because I don't have it ready I'll skip (spoiler: there could be another vulnerability here because $subuser has never been sanitized before it's used at line 137, but timer said I must stop. if somebody has time to dig please let me know.) and go for the *else* branch at line 213.

The function confirmUserPass() is defined at *classes/userdatabase.class.php* and looks like safely queries the database by using a prepared statement.

```
function confirmUserPass($username, $password) {
    /* Verify that user is in database */
    $this->connection->query("SELECT password FROM " . TBL_USERS . " WHERE username = :username");
    $this->connection->bind(':username', $username);
    $result = $this->connection->resultset();
    $num_rows = $this->connection->rowCount();
    if (!$result || $num_rows < 1) {
        return 1; //Indicates username failure
    }
    /* Retrieve password from result, strip slashes */
    $dbarray = $result[0];
    /* Validate that password is correct */
    if ($password == $dbarray['password']) {
        return 0; //Success! Username and password confirmed
    } else {
        return 2; //Indicates password failure
    }
}
```

If you look closely, you'll spot a possible PHP type juggling vulnerability (I didnt validated/exploited this one, ping me if you do): we cannot control the second part of the check ($dbarray['password']) but we can partially control the first one.
I think it's not exploitable because user controlled value is hashed as md5, and as far as I remember it will be possible to have a 32byte string or an empty $password.

I also saw that confirmUserPass() receives the password as md5 string, and I saw that the comparison is made on the field retrieved from the database, therefore we know that passwords are stored as md5. This could be an issue because md5 is deprecated and not safe nowdays, but I'll notice later that rConfig requires strong passwords so it's not interesting now, but worth a fix in my opinion (I'm a fan of https://github.com/defuse/password-hashing, that has a huge pro: dev can change algorithm/round/salt size very easily).
Of course if it was a real review I've **strongly** suggested to get rid of md5 as soon as possible, but this is another sort of exercise.

login() function checks for return and exit if user or password isn't valid.
SQL queries are good, and given that I haven't found anything useful here I'll note md5 and type juggling and move forward.

```
} else {
    $result = $database->confirmUserPass($subuser, md5($subpass));

    /* Check error codes */
    if ($result == 1) {
        $field = "user";
        $form->setError($field, "* Username not found");
    } else if ($result == 2) {
        $field = "pass";
        $form->setError($field, "* Invalid password");
    }

    /* Return if form errors exist */
    if ($form->num_errors > 0) {
        return false;
    }

    /* Username and password correct, register session variables */
    $this->userinfo = $database->getUserInfo($subuser);
    $this->username = $_SESSION['username'] = $this->userinfo['username'];
    $this->userid = $_SESSION['userid'] = $this->generateRandID();
    $this->userlevel = $this->userinfo['userlevel'];
}
```

Remember me

As previously said, the function login() checks if *rememberme* flag has been checked: if it's true, she sets two cookies for later usage.

```
/**
 * This is the cool part: the user has requested that we remember that
 * he's logged in, so we set two cookies. One to hold his username,
 * and one to hold his random value userid. It expires by the time
 * specified in constants.php. Now, next time he comes to our site, we will
 * log him in automatically, but only if he didn't log out before he left.
 */
if ($subremember) {
    setcookie("cookname", $this->username, time() + COOKIE_EXPIRE, COOKIE_PATH);
    setcookie("cookid", $this->userid, time() + COOKIE_EXPIRE, COOKIE_PATH);
}
```

Before going back where these cookies are validated I'll check where userid value used at line 250 came from.
I saw that it comes from a generateRandID() function, which generates a new ID of 16chars for this purpose:

```
/* Username and password correct, register session variables */
$this->userinfo = $database->getUserInfo($subuser);
$this->username = $_SESSION['username'] = $this->userinfo['username'];
$this->userid = $_SESSION['userid'] = $this->generateRandID();
$this->userlevel = $this->userinfo['userlevel'];
```

During the login phase, that actually does some sort of random by using a known vulnerable mt_rand() function (see more here) and generates an md5 ouf of it.

```
/**
 * generateRandID - Generates a string made up of randomized
 * letters (lower and upper case) and digits and returns
 * the md5 hash of it to be used as a userid.
 */
function generateRandID() {
    return md5($this->generateRandStr(16));
}

/**
 * generateRandStr - Generates a string made up of randomized
 * letters (lower and upper case) and digits, the length
 * is a specified parameter.
 */
function generateRandStr($length) {
    $randstr = "";
    for ($i = 0; $i < $length; $i++) {
        $randnum = mt_rand(0, 61);
        if ($randnum < 10) {
            $randstr .= chr($randnum + 48);
        } else if ($randnum < 36) {
            $randstr .= chr($randnum + 55);
        } else {
            $randstr .= chr($randnum + 61);
        }
    }
    return $randstr;
}
}
```

The verification process takes place somewhere around checkLogin(), where userid taken from cookie cookid is checked with what's stored in the database, and if it's false passes over to standard auth.

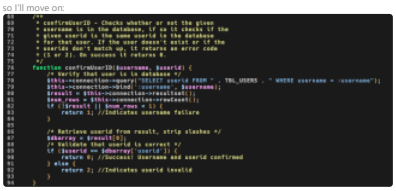
Recent Posts

- Symfony JMose CommandScheduler RCE
- rConfig 3.9.4 multiple vulnerabilities
- Achieve Pareto Principle in secure code review, or die trying
- Long the Ripper
- eLearnSecurity eXploit Development Student

Tags

assembly
certifications
courses
exploit
noise
red
tools
web

confirmUserID() function looks safe, still a type juggling one, but I think again not exploitable so I'll move on:

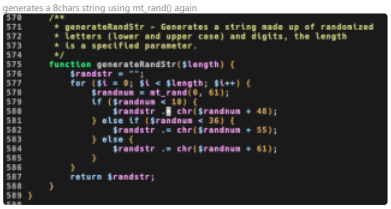


Anyway I think it could be possible to bypass login by abusing mt_rand() weakness, but we need some value to get the seed, therefore a valid account.
Plus, we don't know if somebody had logged in using rememberme function. Still a vulnerability I'd fix, but not useful to me right now.

Will add this mt_rand() and a new type juggling to my notes, just in case.

Password reset

Back at *www/lib/crud/userprocess.php* to review procForgotPass() function, I see that she generates a 8chars string using mt_rand() again



Then she loads user information and sends an email with the new password.

I don't see any other weakness but mt_rand() usage, that would still need some valid value as far as I know, and a weak password generation (new password length is 8 chars). An attacker could try to bruteforce the password online, but it would generate a lot of noise, and the user would receive a notification for the change as soon as she checks her email.
Again vulnerabilities I would fix, but again out of scope now.

Registration

The function in charge of registering new users is procRegister(), defined in file *www/lib/crud/userprocess.php*.
What's really interesting here is the call at line 96: function register() is called with usual value (user, password, password confirm, email) plus an interesting one: ulevelid.

If you open file *classes/usersession.class.php* you can see that, after doing some sanity checks for username, password, and email (the regex looks not valid to me, but we don't care about it right now) the function addNewUser() is called with the same argument.

By reading this function, you see that ulevelid is used to assign privilege to users: there is no validation against that field, but this isn't a big issue, because we will exploit it using a perfectly valid one: 9.
User level are defined at *www/install/config.inc.php.template* as:

- define("ADMIN_LEVEL", 9);
- define("USER_LEVEL", 1);
- define("GUEST_LEVEL", 0);

adding a new user with a ulevelid of 9 result in a new admin user created.

This vulnerability has been assigned CVE-2020-13638, and will probably be fixed in release 3.9.7 (please rease Update).

Privilege escalation

I don't really need to do any privilege escalation because I'm already admin on the application, but as exercise I tried to see if ulevelid could be abused to achieve LPE.

The file *www/lib/crud/userprocess.php* also defines the function procEditAccount(), which allows a user to modify details of her own account.
The function does a sane check against user and password match, therefore you cannot change password or email for another user.
What we see is that $subuserid is passed from a POST straight to the update query: we could achieve LPE by updating our own user setting ulevelid as 9.

Remote code execution

I achieved admin access on the webapp but no strict auth bypass so far, time to look for code execution.

Because it's PHP, the first thing I usually do is to grep for functions commonly used to execute code: shell_exec, proc_open, exec, system, backtick and so on.

By grepping *exec* I can found some interesting lines, unfortunately some of them uses *escapeshellarg* which prevents lot of attacks.
There are some known bypass, mostly by injecting arguments to the command run. For example kacperszurek shows some easy bypass.

Unfortunately, the command we should tamper is touch, which doesn't have useful flags:

- www/lib/ajaxHandlers/ajaxAddTemplate.php: exec("touch " . escapeshellarg($fullpath));
- www/lib/ajaxHandlers/ajaxPurgeConfigs.php: exec("rm -fr " . escapeshellarg($row));
- www/lib/ajaxHandlers/ajaxPurgeConfigs.php: exec('find /home/rconfig/data/. -type d -empty -delete');
- www/lib/ajaxHandlers/ajaxEditTemplate.php: exec("touch " . escapeshellarg($fullpath));

Grep shows three more interesting calls:

- www/lib/crud/search.crud.php: exec("find /home/rconfig/data" . $subDir . $nodeId . " -maxdepth 10 -type f | wc -l", $fileCountArr);
- www/lib/crud/search.crud.php: exec(escapeshellarg($command), $searchArr);
- www/lib/ajaxHandlers/ajaxArchiveFiles.php: exec($commandString);

Starting with *www/lib/crud/search.crud.php*, I saw that $subDir cannot be easily tampered but $nodeId comes straight from a GET

The second one looks weird, because there is a nested escapeshellarg, that is not exploitable like escapeshellcmd but I'll test anyway because my memory could be faulty.
What I tried is to inject an argument into find, that has a -exec arg, and *could* be exploitable. Unfortunately I've not been able to get anything out of this. Clock is ticking, I'll try to get back here later.

By looking at github commit I see that this escapeshellarg has been added recently with a commit message that says: Resolved - Mysql & LFI Injection risks.

OK, it's not a LFI nor a Mysql injection here, it's a **command** one, but it doesn't matter. What matters here is that the dev was aware of the vulnerability and tried to patch (note: this was part of what vikingfr discovered earlier, but I didn't knew this because I tried to avoid as much spoil as possible).
Because you should **never** trust patches, you should also review commit when you find messages like this.

And this is the case of a incomplete fix: we still have two unescaped variable taken from the GET and passed to the exec call, both $searchTerm and $grepNumLineStr are used with no sanitization.

This of course leads to an easy RCE: the query can be done only by authorized users, doesn't matter by the level, but we already achieved auth bypass so we now are zero->admin->RCE.

NOTE: this vulnerability has been fixed in 3.9.6, please read Update

Because of the architecture of the app itself, it will be very easy to escalate to root, but I won't disclose any details here.

Sql Injection

I started this journey knowing I'm looking for a sql injection, but the time I gave myself for this exercise is almost over.

Rushing, I'll grep for SELECT/UPDATE/INSERT with a match on $ to look for queries done with a variable. Of course it could be escaped before, but it's a good starting point (note: I have my own script that greps with a context, so I'm *sure* I won't miss multiline queries).

Suddenly four interesting files came out:

- www/compliancepolicies.inc.php
- www/compliancepolicyelements.inc.php
- www/devices.inc.php
- www/snippets.inc.php

Because the vulnerability is almost the same, I will discuss just the first one.

Vulnerable code looks like

As you can see, $searchColumn is not escaped nor used as param in a prepared statement nor the query uses parameters. This will be an easy sql injection.

Looking at the file itself, we know that it's reachable from the webserver. Reading it from the beginning also shows that it can be used without authentication, leading to at least four pre-auth sql injection.

During a real engagement I would have it exploited to download IP/user/password of controlled devices (see rConfig website to see what she does and how a dump of the database could be useful to an attacker).
This could have huge impact on the network, because devices' data are encrypted with an hardcoded key.

I'd suggest the dev to encrypt IP/user/password of managed devices with a unique key, generated during rConfig setup, splitted between filesystem and database like Filippo Valsorda explain in his blog for hashing. This would make more difficult for an attacket to read both the part of the key and decrypt data.

This vulnerability has been assigned four CVEs, one foreach vulnerable endpoint: CVE-2020-10546 CVE-2020-10547 CVE-2020-10548 CVE-2020-10549.

NOTE: this vulnerability has been fixed in 3.9.7, please read Update

This attack is not over yet: if you notices that *$db2* handler, it pointed me to also review how that object is built and if it's abusable, and I think this will lead to another blog post about a stacked sql injection.

Conclusion

This could've been a good playground for OSWE preparation, not so complex and with some pathes from zero to root by chaining multiple vulnerabilities.

I've not fully tested two RCE, and did not look for more "public" pages. There is still room for analysis, please ping me if you do some so we can share knowledge.

This journey also reminded me of a very very important thing: **never** trust a patch, always review because it's partial/incomplete or maybe introduced more bugs.

Updates

Stephen Stack, lead dev of rConfig, was kind enough to follow up with a fix for some of the reported vulnerabilities with version 3.9.6 and hoply in 3.9.7 later.