Code vulnerabilities put health records at risk

BY DENNIS BRINKROLF | OCTOBER 28, 2020

Security

OpenEMR is the most popular open source software for electronic health record and medical practice management. It is used world-wide to manage sensitive patient data, including information about medications, laboratory values, and diseases. Patients use OpenEMR to schedule appointments, communicate with physicians, and pay online invoices. Specifically in these tumultuous times of an ongoing pandemic, this is highly sensitive data and protecting it is a concern for everyone, and particularly in the U.S.. Companies in America are required to protect individually identifiable and electronic health information by the Health Insurance Portability and Accountability Act (HIPAA).

During our security research of popular web applications, we discovered several code vulnerabilities in OpenEMR 5.0.2.1. A combination of these vulnerabilities allowed remote attackers to execute arbitrary system commands on any OpenEMR server that uses the Patient Portal component. This can lead to the compromise of sensitive patient data, or worse, to a compromise of critical infrastructure.

In this blog post we analyze the technical root cause of three vulnerabilities and demonstrate how attackers could have built a chain for exploitation. We reported all issues responsibly to the affected vendor who rated the fixes as critical and released a security patch in August immediately to protect all users.

Impact

During the analysis of OpenEMR 5.0.2.1 we found the following code vulnerabilities:

1. Command Injection (admin privileges) (CVE-2020-36243)
2. Persistent XSS (admin privileges) (CVE-2021-32103)
3. Insecure API permissions (unauthenticated) (CVE-2021-32101)
4. SQL Injection (user privileges) (CVE-2021-32102, CVE-2021-32104)

The vulnerabilities impact OpenEMR's Patient Portal that needs to be active and accessible for online patients. A remote attacker can then insert a malicious JavaScript payload (XSS) into any user account. This works even when the portal's registration feature for new users is disabled.

Depending on the privilege role of the victim, further vulnerabilities in the backend can be exploited when a victim’s browser executes the XSS payload unconsciously. For example, if the victim is an administrator, the attacker can take over the entire server via a Command Injection vulnerability that allows to execute OS system commands. Other, lower privileged user sessions can be misused to exploit SQL injection vulnerabilities that enable to steal patient data from the database.

For demonstration purposes we’ve created a short video that shows how quick and easy a server is compromised.

Technical Analysis

In the following section, we dive into three of the code vulnerabilities we found in OpenEMR. These can be combined by an attacker to gain pre-auth command execution in the Patient Portal of OpenEMR 5.0.2.1 when targeting an administrator user.

1. Command Injection Vulnerability (CVE-2020-36243)

The most critical vulnerability hides in the backend of OpenEMR. Here, administrators can use a feature to create data backups. For this purpose, different SQL queries are constructed dynamically that are later executed **as system commands** when creating the backup file. The following (simplified) code shows the critical code where these system commands are created depending on the operating system (OS).

interface/main/backup.php

```
5   foreach ($_POST['form_sel_layouts'] as $layoutid) {
6      if (IS_WINDOWS) {
8         $cmd .= " echo DELETE FROM layout_options WHERE form_id = '" .
                     add_escape_custom($layoutid) . "'; >> " .
                     escapeshellarg($EXPORT_FILE) . " & ";
9      }
10     else {
11        $cmd .= "echo \"DELETE FROM layout_options WHERE form_id = '" .
                     add_escape_custom($layoutid) . "';\" >> " .
                     escapeshellarg($EXPORT_FILE) . ";";
12     }
13  }
14
15  exec($cmd);
```

In line 5, the values in the HTTP POST parameter `form_sel_layouts` are received and used as `$layoutid` variables. Then these values are concatenated into an OS command `$cmd` in line 11.

As we can see here, the user-controlled input `$layoutid` is sanitized with the help of the function `add_escape_custom()`. This custom function is defined in the OpenEMR code base and makes use of the PHP built-in function `mysqli_real_escape_string()` that is known to protect against SQL injection vulnerabilities. Finally, the concatenated OS command string is executed in line 19. At first sight, it looks like the developers carefully sanitized all user inputs.

To understand why the sanitization is not sufficient in this code we need to understand how commands are executed. When we look at the final value of the variable `$cmd` in line 15, the shell command looks like the following at runtime:

```
echo "DELETE FROM layout_options WHERE form_id = '$layoutid';"  >> /tmp/export;
```

Here, the variable `$layoutid` contains a user-controlled value. But why does the `add_escape_custom()` function not fully protect against a Command Injection vulnerability?

Let's assume the attacker sends the following payload:

```
?form_sel_layouts[]='sonar";source
```

This would result in the following shell command:

```
echo "DELETE FROM layout_options WHERE form_id = ' \'sonar\";source';"  >> /tmp/export;
```

As we can see, the quotes are escaped and an attacker cannot break out of the single quotes `'`. A SQL injection is successfully prevented. Double quotes `"` are also escaped by `mysqli_real_escape_string()` and we are not breaking out of the `echo` command either. However, there is another way to exploit a Command Injection vulnerability.

```
echo "DELETE FROM layout_options WHERE form_id = "touch sonarsource.txt;" ;" >> /tmp/export;
```

The problem here is that the `echo` shell command uses double quotes and thus allows to execute sub commands in Linux by using characters like backticks `` `` or `$()`. Once our backticks are found within the system command, our new, injected command is executed and the output result is inserted into the initial command. From here, an attacker can fully compromise the system and read sensitive data.

Patch

It is tempting to use the PHP built-in function `escapeshellarg()` as a patch since it is designed to escape all malicious characters needed for a Command Injection attack. However, in this case this function would introduce a SQL injection vulnerability instead because `escapeshellarg()` introduces new single quotes. These single quotes would break the SQL query and probably that is the reason why it was not used here in the first place.

As a solution to protect against both vulnerability types, it is enough to simply swap the single and double quotes.

```
echo 'DELETE FROM layout_options WHERE form_id = "$layoutid";' >> /tmp/export;
```

A SQL injection vulnerability is then still prevented because the double quote `"` characters are escaped. More importantly, *command substitution* can no longer be opened because now the argument of `echo` is in single quotes `'`, which don't allow sub commands.

2. Persistent Cross-Site Scripting Vulnerability (CVE-2021-32103)

So far an attacker can only trigger the Command Injection vulnerability manually if he or she logs in as an admin. With the help of another code vulnerability, the attack can be carried out with the help of a valid administrator that triggers the exploitation unknowingly. We discovered a Persistent Cross-Site Scripting vulnerability that enables this kind of attack.

The attacker's payload is hidden within the last name of a user account. This last name can be changed in line 4 of the following code. Note that this action can only be performed by an administrator (we will come back to this in the next section).

interface/usergroup/usergroup_admin.php

```
1  if (isset($_POST["privatemode"]) && $_POST["privatemode"] =="user_admin") {
2     if ($_POST["mode"] == "update") {
3        if ($_POST["lname"]) {
4           sqlStatement("update users set lname=? where id= ? ", array($_POST["lname"], $_POST["id"]));
5        }
6     }
7  }
```

The new last name is stored permanently in the database table *users*. At a different code location, this name is read from the database again to present it in the frontend. This happens, for example, when an administrator changes the password of the renamed user.

interface/usergroup/user_info.php

```
2  $userid = $_SESSION['authId'];
3  $user_name = getUserIDInfo($userid);
4  $user_full_name = $user_name['fname'] . " " . $user_name['lname'];
5  ?>
6  <legend><?php echo xlt('Change Password for') . " " . $user_full_name; ?></legend>
```

Here, in line 6, the user name is embedded into the HTML output without any sanitization.

This allows injection of malicious HTML code into the response page that will be rendered by the administrator's browser. When `<script>` tags are injected into the last name, then malicious JavaScript code can be executed that will be able to control the victim's browser and its further activities. For example, it can be used to trigger the previously introduced Command Injection vulnerability that only an administrator can execute (Cross-Site Scripting).

Patch

This vulnerability can be easily prevented by using the popular PHP function `htmlspecialchars()` in line 6. It encodes special HTML characters into HTML entities (e.g. `<` into `&lt;`) and thus prevents that malicious JavaScript code can be embedded into the name.

```
echo htmlspecialchars($user_full_name, ENT_QUOTES);
```

3. Insecure API Permissions (CVE-2021-32101)

gered by an administrator. And we've learned about a persistent XSS vulnerability that nt the XSS payload we again need administrator privileges. As long as the uld be no risk - right? This is true, as long as the permission system is secure.

portal/patient/_machine_config.php

```
3   OpenEMR\Common\Session\SessionUtil::portalSessionStart();
4   if (isset($_SESSION['pid'])
    && (isset($_SESSION['patient_portal_onsite_two'])
    || $_SESSION['register'] === true)) {
5       $pid = $_SESSION['pid'];
6       $ignoreAuth = true;
7   } else {
8       $ignoreAuth = false;
9       }
10  }
```

It creates a new session and checks in line 4 if the user is already on the portal page or whether she is currently trying to register. In this case, the authentication check is deactivated in line 6 (`$ignoreAuth = true`). Otherwise, the authentication check is active and the user has to authenticate.

Let's have a look at how the registration works in the Patient Portal. In the following code you can see the simplified *register.php*. In lines 4-6, the interesting session variables are set that indicate that we are within a new registration process.

portal/account/register.php

```
4   OpenEMR\Common\Session\SessionUtil::portalSessionStart();
5   $_SESSION['authUser'] = 'portal-user';
6   $_SESSION['pid'] = true;
7   $_SESSION['register'] = true;
```

No further checks are made and the session variable is not destroyed at the end of the file. An attacker could therefore make the first HTTP request to *register.php* which creates a session and sets the session variable `$_SESSION['register']` to *true*. Then, without completing the registration, the attacker can access the dispatcher and bypass the authentication because `$ignoreAuth` is set to *true*.

Once the authentication is bypassed, it is possible to use all features of the API as a *registered* Patient Portal user. This means an attacker can access all patient data or change the email address and passwords of the patients even if registration to the Patient Portal is closed.

Of special interest is the user controller of the API which can be used to change information of any backend user like the administrator. The attacker can now take advantage of the previously introduced Persistent XSS vulnerability by adding an XSS payload to the last name of the admin user. This XSS payload can then execute JavaScript code that exploits the Command Injection vulnerability. Ultimately, all three vulnerabilities are combined and lead to a pre-auth Command Execution in OpenEMR 5.0.2.1.

Timeline

Date	What
24.02.2020	We reported the vulnerabilities to the OpenEMR team
29.04.2020	OpenEMR team addresses the first vulnerabilities with a patch
11.08.2020	OpenEMR team releases another security patch

Summary

In this blog post we analyzed three code vulnerabilities found in OpenEMR, a widely adopted open source solution for electronic health records. The combination of these vulnerabilities can lead to a complete takeover of the OpenEMR application and put patient data as well as the infrastructure at risk. We've evaluated the root causes in the PHP code base and described how to fix them. Due to the severity of the issues, we postponed the release of these details for several months. If you are hosting an OpenEMR instance and have not yet updated your installation, we highly recommend that you do so now. Last but not least, we would like to thank the OpenEMR team who quickly released a patch version 5.0.2.2 after our reports.

Please stay healthy and secure!

You can join the discussion about this vulnerability in our community forum.

DENNIS BRINKROLF
Security Researcher
in

IDE extension that lets you fix coding issues before they exist!

Discover SonarLint →

Setup is effortless and analysis is automatic for most languages

Discover SonarCloud →

Fast, accurate Code Quality and Code Security analysis for most languages

Discover SonarQube →