Unexpected Journey #6 - All ways lead to Rome ! Remote Code Execution on MicroFocus Secure Messaging Gateway

JUN 22, 2018

It has been a quite while since I haven't released a new part of unexpected journey article serie. Particularly this small 0-day research project has been certainly didactic to me. Thus, I've decided to write down the process of achieving remote code execution on MicroFocus Secure Messaging Gateway product.

Me and my team detects lots of security products during the OSINT phase of engagement. For whom haven't heard of this article series before, I do pick a security products during OSINT, and then perform security research on them, finding 0-day so that we can

break into targeted infrastructure. It's all part of the penetration test process. *(Yeah, there is a huge difference between pentest and vulnerability assessment!)*

All Ways Lead to ROME

During this security research, I've came across with more than 30+ vulnerability *(Remember pure native PHP projects like an early 2000, you know what I mean!)*. As far as I can tell, combination of these vulnerabilities can lead to 5 different RCE scenario. But from the point of a ~~hacker~~ pentester perspective, I am not tasked to find out all vulnerabilities. I'm not tasked to uncover security issues of security products *(which I love to do this but our client's concern was not that)*. All I need is to find a way to break in.

#Vulnerability 1 - SQL Injection is Everywhere ! But Choose One Wisely

As I told before, there was a multiple SQL Injection flaws. Here is the one obvious SQLi flaw that I've spotted at publicly accessible endpoint.

```
1  // Lookup the database connection information for the QMS server
2  $QmsQuery = "SELECT DatabaseHost, DatabaseName, LoginName, LoginPassword FROM 
3              "JOIN ModuleName ON ModuleName.idModuleName=ServerModule.idModuleNa
4              "JOIN ServerModuleDatabaseConnection ON ServerModuleDatabaseConnec
5              "JOIN DatabaseConnection ON DatabaseConnection.idDatabaseConnectio
6              "WHERE APIName='qms' AND idServer=" . $_GET[ 'serverid' ] . " AND 
7  $Result = pg_query( $dbconn, $QmsQuery );
8  if ( pg_num_rows ( $Result ) > 0 )
9  {
10     $row = pg_fetch_row ( $Result );
11     $QmsDbConnectString = "host=" . $row[0] . " dbname=" . $row[1] . " user="
12     // ... OMITTED CODE ...
13 }
```

What's happening here is that application retrieves a data from database and then use it to dynamically generate Postgresql connection by using `$QmsDbConnectString` variable. But the returned data from database, which can be manipulated by exploiting SQLi flaw, haven't been used for any context. For this reason, we need to use Time-based SQLi exploitation techniques. Beside that, Further investigation showed us that **stacked-queries** are enabled.

Following items are important in terms of exploit development:

- Having a fully working exploit code (preferably msf module)
- Validity of actions in every step of exploitation

- Speed of exploitation process

Even if we're able to execute our own queries *(thanks to stacked-query)* Time-based SQL Injection is incapable of providing us the validity of taken actions reliably, such as user creation etc, as well as speed of exploitation and exploit code development *(Who wants to deal with a Time-based SQLi attacks during module development ?!! I'm certainly NOT)* .

I've kept reading source code while keeping these limitation in my mind and came across with following code block at /api/1/enginelist.php file.

```php
<?php
  include_once( "./api_support.php" );

  $AppKey = getAppKey();
  $EngineArray = getEngineArray( $dbconn, $AppKey );  // Get an ordered list o

  if ( sizeof( $EngineArray ) > 0 )
  {
    print( '<?xml version="1.0" encoding="UTF-8"?>' );
    print( '<response>' );
    print( '<engines>' );
    foreach( $EngineArray As $Engine )
    {
      print( '<engine>' );
      print( '<host>' . $Engine[0] . '</host>' );
      print( '<description>' . htmlspecialchars( $Engine[1], ENT_XML1 | ENT_CO
      print( '<priority>' . $Engine[2] . '</priority>' );
      print( '</engine>' );
    }
    print( '</engines>' );
    print( '</response>' );
  }
  else
  {
    // No scan engine found (Possibly invalid AppKey or No scan engine connect
    http_response_code( 401 );
  }

?>

```

Here is the definition of two important function.

```
function getAppKey()
{
  $appkey = '';
  if ( !empty( $_POST[ 'appkey' ] ) )
  {
    $appkey = $_POST[ 'appkey' ];
  }
  elseif ( !empty( $_GET[ 'appkey' ] ) )
  {
    $appkey = $_GET[ 'appkey' ];
  }
  return $appkey;
}


function getEngineArray( $dbconn, $AppKey )
{
  // Lookup all the assigned servers
  // connection address, engine name, priority, ssl
  $EngineListQuery = "SELECT DISTINCT ScanEngineProperty.ValueString || Comput
    "ScanEngine.Description AS Description, " .
    "coalesce(ScanEnginePropertyPriority.ValueInt,1)+coalesce(InterfaceEngineP
    "coalesce( ScanEngineEnableSsl.ValueInt, 0 ) AS SslConnection FROM ScanEng
    "JOIN ScanEngine ON ScanEngine.idScanEngine = ScanEngineProperty.idScanEng
    "JOIN ScanEngineOUSet ON ScanEngineOUSet.idScanEngine = ScanEngine.idScanE
    "JOIN ScanEngineKey ON ScanEngineProperty.idKey = ScanEngineKey.idScanEngi
    "LEFT JOIN ScanEngineProperty AS ScanEnginePropertyPriority ON ScanEngineP
    "LEFT JOIN InterfaceEnginePriorityInfluence ON InterfaceEnginePriorityInfl
    "LEFT JOIN ScanEngineProperty AS ScanEngineBindAddressPlain ON ScanEngineB
    "LEFT JOIN ScanEngineProperty AS ScanEngineBindAddressSsl ON ScanEngineBin
    "LEFT JOIN ScanEngineProperty AS ScanEngineEnableSsl ON ScanEngineEnableSs
    "WHERE ScanEngineOUSet.idOrganizationalUnit IN " .
    "( " .
      "SELECT idOrganizationalUnit FROM ScanEngineOUSet " .
      "UNION " .
      "SELECT idOrganizationalUnit FROM InterfaceOUSet WHERE InterfaceOUSet.En
    ") " .
    "AND ScanEngineKey.Value='ConnectionAddress';";
  $result = pg_query( $dbconn, $EngineListQuery );
  $EngineArray = array();
  if ( $result )
  {
    while ( $row = pg_fetch_row( $result ) )
    {
      array_push( $EngineArray, array( $row[0], $row[1], $row[2], $row[3] ) );
    }
    if ( sizeof( $EngineArray ) > 0 )
    {
```

```
49        usort( $EngineArray, 'CompareEngineRecords' );
50  //        error_log( 'After Engine Sort: ' . returnFormattedArray( $EngineArra
51      }
52    }
53    return ( $EngineArray );
54  }
```

`$AppKey` variable is being initiated with data taken by client. And the same variable is being passed to the `getEngineArray()` function which is using the same variable within SQL query. Yet another obvious SQL Injection case. But this one is providing what exactly we need. If you look at the for loop you will see that result of the SQL query is being returned back to the client in the form of XML. Whenever we need to validate a one action taken during the exploitation, we could simply send one HTTP request to this vulnerable end-point by providing SQLi payload and then receive a HTTP response which contains a data returned from query !

Stacked Query is enabled. So what we are waiting for ?!

Once you have SQLi vulnerability with a stacked-query ability, there is lots of way to reach ROME. Here is some of the different routes to the ROME and the reason why I didn't choose these path.

- What I want in the end is a fully automated single metasploit module that gives your remote shell. So dumping current users hash and then performing hash-cracking is off the table.
- This product is using Postgresql and we have stacked-query ability. So we could easily create a new table, and copy a content of a file into to that table as a record and read the data by select query. But Postgresql service is running with low-privileged user permission. All the configuration files and web folders are owned by root user. So we won't be able to read content of any file.
- We could also create a file that contains our payload instead of reading any file but same limitations goes for this as well. All folders belongs to the root user and none of them have write permission for non-root user.

For these reasons, I decided to go with a "create an administrator user" way. In order to do create user and then login with a newly created user, you need to fully understand login process. Following code snippet is responsible of login process.

```
1 $result = pg_query($dbconn, "SELECT Account.idAccount,PasswordStoreMethod.Stor
2     "FROM Account " .
3     "JOIN PasswordStoreMethod ON PasswordStoreMethod.idPasswordStoreMethod=Acc
4     "LEFT JOIN AccountProperty AS ManagementMethod ON ManagementMethod.idAccou
```

```
    "LEFT JOIN AccountProperty AS ProvisionRefreshNextTimestamp ON ProvisionRe
    "WHERE Account.LoginName ILIKE '" . pg_escape_string( $_POST[ 'username' ]

if ( !$result || pg_numrows( $result ) != 1 )
{
    $errcode = 49;
}
else
{
    $row = pg_fetch_row( $result );
    if ( $row[ 2 ] == 0 )
    {
        $errcode = 61;
    }
    else
    {
        $iManagementMethod = $row[ 3 ];
        // If the user is managed by provisioning, check whether the password
        if ( $iManagementMethod == 1 && $row[ 4 ] < 0 && !isset( $_POST[ 'reen
        {
            $bRequireRevalidation = true;
        }
        else
        {

            $UserID = $row[ 0 ];
            $CurrentPasswordHashMethod = $row[ 1 ];

            $HashedPassword = "0";
            if ( $CurrentPasswordHashMethod == "securemd5" )
            {
                $HashedPassword = CreateEncryptedPassword( $_POST[ 'password'
            }

            $LimitInterfaceClause = "";

            // Interface limit is passed in on second authentication attempt i
            if ( isset( $_POST[ 'LimitInterfaceId' ] ) && $_POST[ 'LimitInterfa
            {
                $LimitInterfaceClause = " AND UserInterface.idUserInterface="
            }

            $result = pg_query($dbconn, "SELECT DISTINCT UserInterface.idUserI
                "JOIN RoleType ON RoleType.idRoleType=OURoleSet.idRoleType " .
                "JOIN UserRole ON UserRole.idRoleType=RoleType.idRoleType AND
                "JOIN OuRoleAssignedUserInterface ON OuRoleAssignedUserInterfa
                "JOIN UserInterface ON UserInterface.idUserInterface=OuRoleAss
                "JOIN Account ON Account.idAccount=UserRole.idAccount " .
                "WHERE Account.Enabled=1 AND Account.Password='" . pg_escape_s
```

```
54                  "AND OURoleSet.idOU = Account.idOwnerOu " .
55                  $LimitInterfaceClause .
56                  "ORDER BY ButtonPresentationOrder;" );
57
58              if ( !$result || pg_numrows( $result ) == 0 ){
59                  // ... OMITTED CODE ...
60              }
61          }
62      }
63  }
```

First of all, look at to the first line. Usage `ILIKE` instead of `=` operand gives us ability to complete login process just by knowing password. Use `a%` as a username and supply password of user contains `a` within it simply complete login ! But this bug useless for us of course. I want to have fully automated exploitation.

What is very interesting in here is between lines 33-37. Here is what happens step by step.

1. Find a user just by using username taken from client.
2. Retrieve the data of that user from database.
3. Detect `$CurrentPasswordHashMethod` of given user at line 31. *(Hint: Having CurrentPasswordHashMethod field in database means that they changed hashing algorithm in the past. They need to support older hashing algorithm in terms of backward compatibility.)*
4. Set `$HashedPassword` to zero (This will have a major role later !)
5. If the user is being configuered to login by using `securemd5` , calculate the hash ! Otherwise `$HashedPassword` will remain zero.
6. Continue to login process by using given username and hashedpassword value.

Now we need to understand `CreateEncryptedPassword` method. Here is the implementation of it.

```
1  function CreateEncryptedPassword( $_Password, $_UserId )
2  {
3    global $g_PrivateKey;
4
5    // Take an MD5 hash from the original password
6    $HashedPassword = md5( $_Password );
7
8    // Append the private key and the login id to prevent duplicate hashes expos
9    $HashedPassword .= $g_PrivateKey;
10   $HashedPassword .= $_UserId;
```

```
11
12    // Rehash the joined hash
13    $HashedPassword = md5( $HashedPassword );
14
15    return ( $HashedPassword );
16  }
```

We have salting operating in here. But problem is `$g_privateKey` value is being generated during installation and saved into the source.xml files. Since whole files of project owned by a root user. We are not be able to get the value by exploiting SQL Injection value. There is two way to continue to exploration. Either we need to find a way to read content of source.xml file or we need to find a way to bypass login process.

Lets go back to login process and examine it closely. Here is the most interesting part !

```
$HashedPassword = "0";
if ( $CurrentPasswordHashMethod == "securemd5" )
{
  $HashedPassword = CreateEncryptedPassword( $_POST[ 'password' ], $UserID );
}
```

What would happen if we change `$CurrentPasswordHashMethod` to something else ? `$HashedPassword` will remain ZERO and then it will be used within another SQL query (line47) that validates password at database.

Let me sum up what we're going to do.

1. We will create an user by exploiting SQL Injection issue.
2. Since `$CurrentPasswordHashMethod` variable is populated by idpasswordstoremethod field of new user. We can set it to anything we want !
3. When the login process is being triggered for our user, `$CurrentPasswordHashMethod` value will be different then the securemd5. Which cause `$HashedPassword` remain as zero.
4. When the query defined at 47th line executed, it will try to find a user that have zero as a password.

Basically, we need to execute following series of query in order to complete login process with our newly created user without know what `$g_privateKey` value is.

```
1  INSERT INTO account VALUES (1337, 1, 'hacker', '0', '', 1,1337);
2  INSERT INTO UserRole VALUES (9999,1337,1),(9998,1337,2);
```

Isn't that awesome ?!

> Every software bug doesn't mean you have a vulnerability.
> But under the rare conditions every bug can be useful.

Vulnerability 2 - Authenticated Command Injection

Let me remind you one thing before continuing to the story. We were at the middle of the pentest. I don't have days or weeks to spend on this product but only hours ! So I tried to find a shortest route to Rome. Following code sections are only accessible for a user who have certain privileges located at manage_domains_dkim_keygen_request.php file.

```
<?php

include_once("../../../../http_cgi/security/clientsettings.php");

// Test rights to the provided record id
$PermissionsOk = false;
$result = pg_query( $dbconn, "SELECT idOwnerOu FROM Domain JOIN DkimSignature (
if ( $result && $row = pg_fetch_row( $result ) )
{
    $PermissionsOk = CheckRightsToOu( $row[ 0 ] );
}

if ( !$PermissionsOk )
{
    // ... OMITTED CODE ...
}
else
{

    // Retrieve the domain and selector for key generation
    $result = pg_query( $dbconn, "SELECT Domain,Selector FROM DkimSignature WH(
    if ( $result && $row = pg_fetch_row( $result ) )
    {
        $Domain = $row[ 0 ];
        $Selector = $row[ 1 ];
    }

    if ( !isset( $Domain ) || !isset( $Selector ) )
    {
            // ... OMITTED CODE ...
    }
    else
    {
```

```
34          // Generate directories for signature creation
35
36          $DkimSigningPath = realpath( $_SERVER['DOCUMENT_ROOT'] . "/../http_loc
37          @mkdir( $DkimSigningPath );
38          $DkimSigningPath .= $_POST[ "DkimRecordId" ] . "/";
39          @mkdir( $DkimSigningPath );
40
41          $SystemCommandResult = system( "opendkim-genkey -b 2048 -r -s " . $Sel
42
43          // ... OMITTED CODE ...;
44      }
45  }
46  ?>
47
```

We have another SQL Injection here but it's not what we're looking for. What particularly interesting for us is line 41. It does execute operating system command with a 2 dynamic variable `$Selector` and `$Domain`. Where are those variable are coming from ? yeah, they are coming from database. If we find a procedure that insert data into the DkimSignature table and the trigger this code section will give us a Command Injection ability !

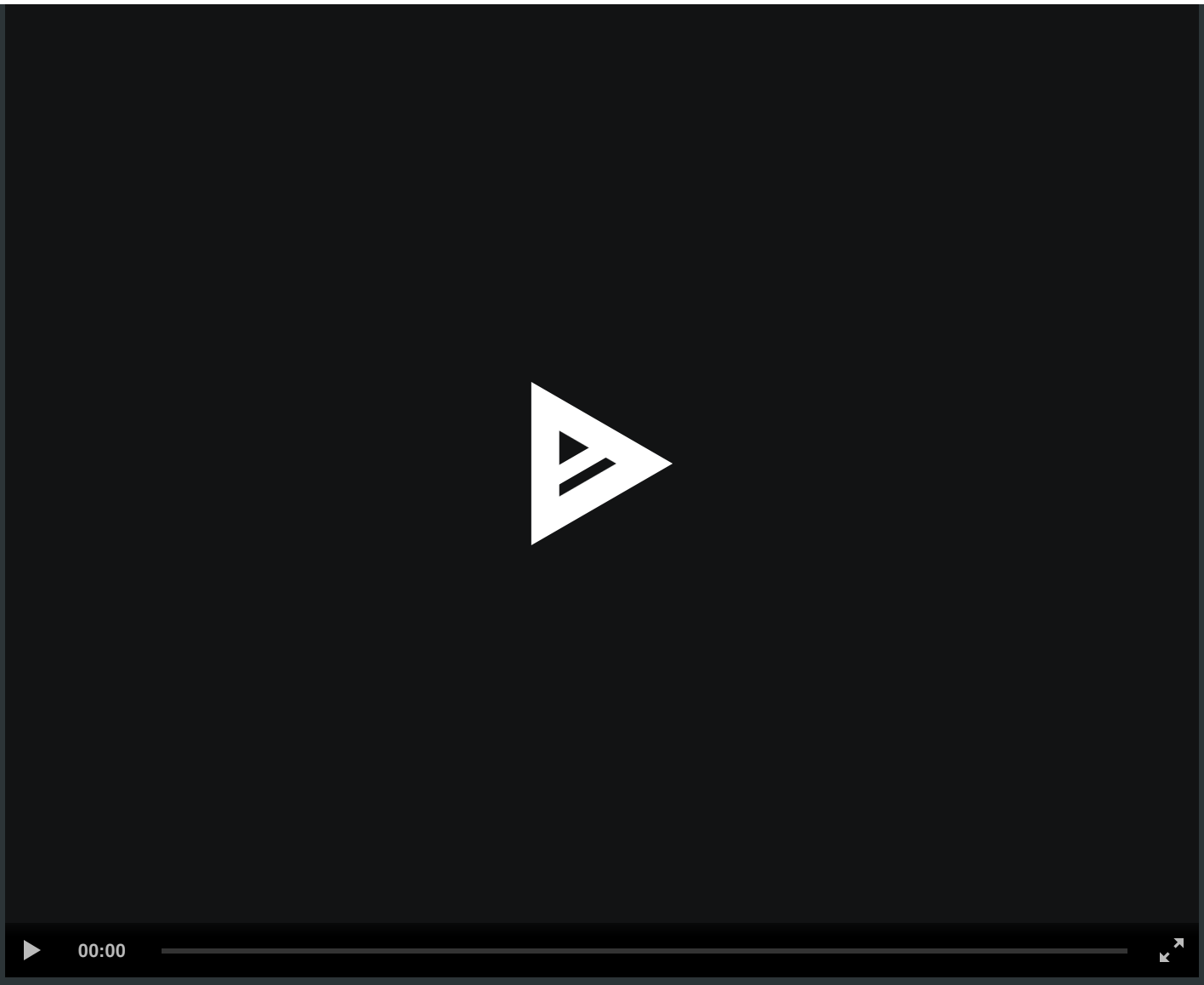
Putting All Things Together

Let me sum up for readers who may be lost or couldn't clearly understand what we've done so far.

1. We detected unauthenticated SQLi but it's useless in terms of certain rules.
2. We detected another unauth SQLi which gives us ability to retrieve back the result of SQL query.
3. We found a bug within login procedure which gives us ability to login without having private key that being used as a salt during login. (Set password field of any user to zero and change `$CurrentPasswordHashMethod` to something else)
4. We found authenticated command injection.

Metasploit Module

https://github.com/rapid7/metasploit-framework/pull/10255



Timeline

Congratulations to Micro Focus for their rapid response ! Here is the timeline of the process.

- 19 June 2018 22:31 GMT +1 - Finding 0day.
- 19 June 2018 22:50 GMT +1 - Implenetation of metasploit module.
- 19 June 2018 23:03 GMT +1 - Cyber intelligence sharing with GPACT customers.
- 21 June 2018 23:59 GMT +1 - First contact with vendor. We've told them that we are expecting to see hot-fix within 7 days.
- 22 June 2018 00:02 GMT +1 - Rapid response from vendor (not automatically generated:).
- 22 June 2018 01:13 GMT +1 - Vendor have confirmed of our finding.
- 27 June 2018 00:13 GMT +1 - Vendor issued CVE-2018-12464 and CVE-2018-12465.
- 28 June 2018 20:09 GMT +1 - Vendor released the fix.

- 28 June 2018 20:09 GMT +1 - We decided to withhold the publication of metasploit module for another 7 days. We want to give a enough time to companies who are currently using this product so they can update their systems.

Leave a Reply

Your email address will not be published.

Required fields are marked *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Verifying...

Privacy • Help

Post Comment

YOU MAY ALSO ENJOY…

Search

Search