JAVA DISCOVERY PROJECT

ABSTRACT

Discover running Java processes and Java applications on 4 different operating systems (Windows, Linux, AIX, Solaris)

Matthew Tang (Author & Developer)
Rui Dai (Analyst)
Naijing Yu (Analyst)

19/10/18

Contents

1. Problem Statement

Java is a computer programming language that is found everywhere. It is primarily used to develop applications such as web applications, standalone applications, and even applets. Web browsers such as Chrome and even Internet Explorer are Java-Compatible and require Java to work properly. In short, Java is simple and powerful. It is secure and portable and is almost always required for any operating system available today.

Java is often downloaded multiple times on any given operating system. There could be 10, 15, even 100+ Java applications installed. To add onto this, there are also numerous Java versions that have been released and can be found on a system (Most commonly Java SE 7 and Java SE 8). Java processes also exist and run in the background. Depending on which applications are launched, there can be various Java processes running simultaneously.

2. Analysis

2.1 Objective

There are two primary goals for this project:

1) Find all running Java processes on a server. (Should run at the start of every hour)

   There are 4 different operating systems that we are working with, Microsoft Windows, Linux, IBM AIX, and Oracle Solaris. The goal is to identify all running Java process and output the results into a CSV with detailed information about each process.

2) Find all installed Java applications on a server (Run daily or weekly depending on demand)

   With the 4 different operating systems, the goal is to search a server's file system for files and folders that relate to Java. All these files and folders along with detailed information will be outputted into a CSV.

With these two CSV's, a team will analyze these results and perform a *Java Subscription Analysis.*

2.2 Methodology

To solve these two goals, two separate scripts were developed. The 1st script was developed to solve goal #1: *Find all running Java processes on a server* and the 2nd script was developed for goal #2: *Find all installed Java applications on a server*.

Script 1 (Running processes):

- Identify all running java processes
- Find detailed information about each process
- Export results into a CSV

Script 2 (Installed applications):

- Search the server's file system for paths that contain specific "keywords"
- Keep files that end in 'java' and trim paths until we hit a folder containing a specific "keyword"
- Remove any duplicate trimmed paths
- Find detailed information about the path's
- Export results into a CSV

2.3 CSV Format (Running processes)

For each running process, the following information is required in the CSV:

1. Current Server
2. Process Name
3. User that Initiated the Process
4. Path of the Executable
5. Java Version (if available)
6. A Counter That Tracks How Many Times the Process has Been Found
7. Command Line for the Process
8. First Discovered Date (When the process was first found)
9. Last Discovered Date (When the process was last found)
10. **Company/Vendor (Only available on Windows Operating System)

Note: Field #9 will be updated constantly if the process was found at runtime and was found in the existing file since the script will be ran hourly.

2.4 CSV Format (Installed Applications)

For each application found, the following information is required in the CSV:

1. Current Server
2. Application Full Path
3. Path Type (Folder, Java Executable, or Normal Executable)
4. Java Version (if available)
5. File's/Folder's Last Modified Date
6. First Discovered Date (when the application was first found)
7. Last Discovered Date (when the application was last found)
8. **Company/Vendor (Only available on Windows Operating System)

Note: Field #7 will be updated constantly if the application was scanned at runtime and was found in the existing file since the script will be ran daily/weekly.

2.5 Report Storage

The location of where the report file is stored is very important. Many teams might need to have access to the CSV report. The report is stored in a shared folder that is open and assessable to everyone. It is stored at: \\CL11180.sunlifecorp.com\JavaDiscovery. The name of the report files is SERVERNAME_process.txt or SERVERNAME_application.txt. Although a share folder that is assessable to everyone is not secure, it is a quick and efficient way for multiple members to gather the information quickly without requiring access to a server.

2.6 File output

In UNIX related systems there is a small problem where the command line for script #1 may have commas inside them. This is problem affects the final output because the script will break up the command line into separate columns when a user opens the CSV file. Excel is not smart enough to distinguish between multiple delimiters as there are various delimiters embedded within the information such as: “,” “\” “/” “.” To solve this problem there is a small work around.

Only for Linux, AIX, and Solaris, will script #1 output a text file. Windows Script #1 and #2 and UNIX related systems script #2 will output a proper CSV report. It will contain all the same information but, the text file has the delimiter set to “;” instead of “,” to separate the headers. When a user needs to check this information, they need to just open Excel, import the data inside the text file, and then set the delimiter to “;”. This will fix the problem and organize the data very cleanly even if there are commas within a cell. Although, this is not truly a CSV (comma separated value file) it is an alternate solution due to the commas in the command line.

1) Open a new workbook
2) Click *Data* in the top bar
3) Click *From Text*
4) Browse to the text file
5) Click *Delimited* & *My Data Has Headers*
6) Press next and then change delimiter to semicolon
7) Press finish and data should be organized

3. Implementation

3.1 Script 1 – Find All Running Java Processes on a Server

- *Identify Running Java Processes*

  The first step is to find all running java processes. This is done by searching through each running process and filtering them by their command line. There will be a file containing *keywords* such as "java" or "jre" and if the processes' command line contains these words, it is considered a match. An example is shown below:

  Keyword: java. The process java.exe was found based on its command line.

  Command Line (a portion of it): "...\...\...\java\jre\bin\java" -Dlog4j.debug -Dlog4j.configuration=file:///C:/Program%20Files/Evolven%20Enlight/Agent/conf/log4j.

- *Remove Duplicates*

  In the above example a search for *java* and then a search for *jre* would produce the above process twice because it contains *java* and *jre* in its command line. Therefore, to avoid this, a check is required. Each process has a unique PID and if the PID has already been scanned, then that process can be skipped and duplicates matches can be avoided. If it has not, it is added to the CSV file.

- *Gather information about each process*

  Now that the unique running java processes are identified, the next step is to find more information about the processes. Each process has meta information associated with it and that information can be extracted and found. Click here for CSV file format

- *Report*

  With all the information, the final report is generated and exported into a CSV.

  This CSV is the passed onto another team for analysis.

A complete algorithm can be found below for a more complete understanding of: ***Finding all running Java processes on a server...***

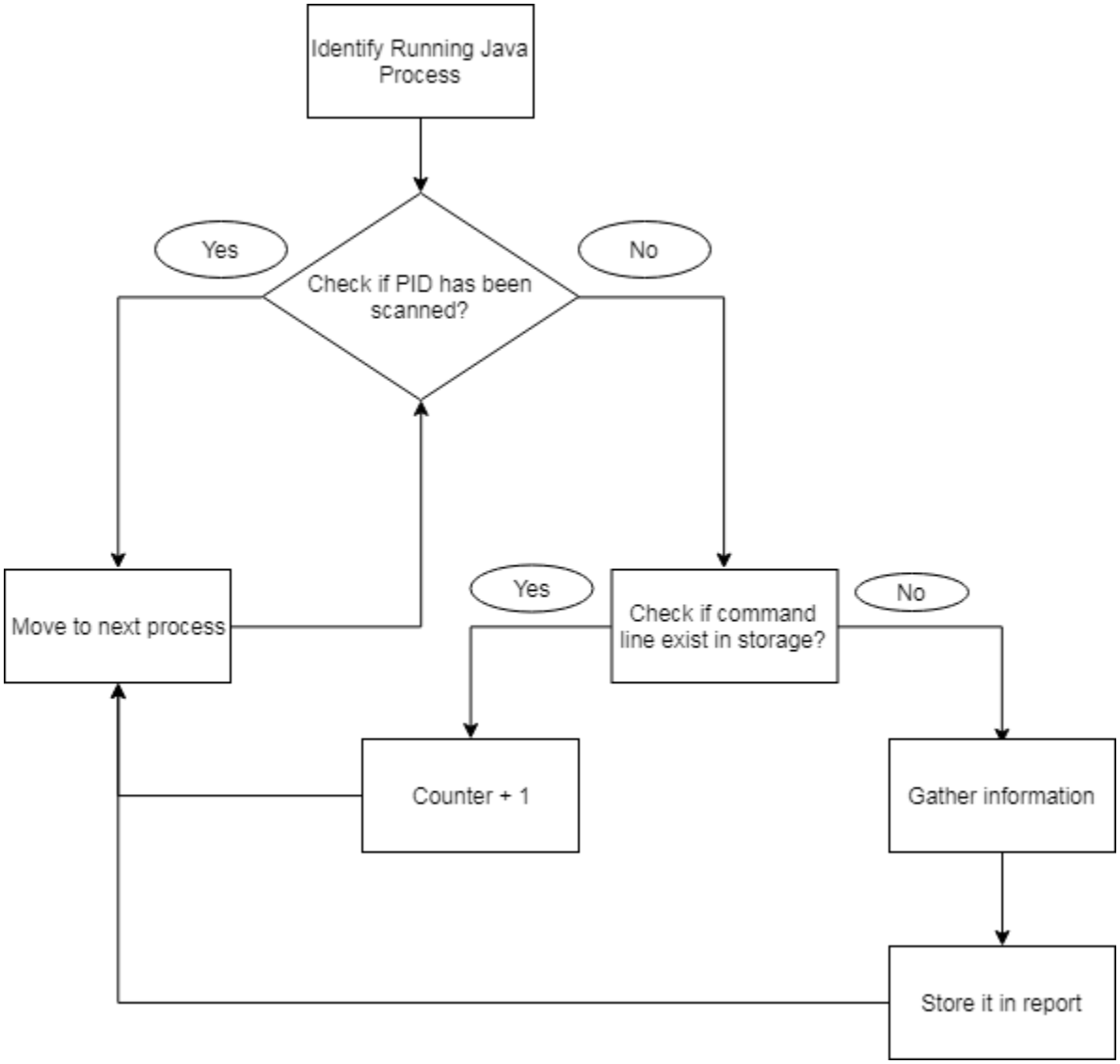

Figure1. Algorithm for Script 1 – Find All Running Java Processes

3.2 Script 2 – Find All Installed Java Applications on a Server

- *Recursively Search the File System*

  Initially, the entire file system is scanned and if the paths contain any of the keywords such as 'java' it is stored into an array. For example, scanning on the Windows Server would scan all the drives present (C:\. E:\, F:\, and etc.) except for the network drives. For UNIX related systems it would scan from root '/'. It does not matter where in the path the keyword is located, as long as it is present, it will be scanned and stored.

  Keywords: java, jre, jdk, jvm

  1) E:\Program Files\Java\jdk1.8.0_121\jre\bin\javaw.exe (Windows)

  2) C:\Users\xr35su\AppData\Roaming\Sun\Java (Windows)

  3) /apps/java/jdk1.5.0_08/bin/java (Linux)

  ...

- *Sort, Trim, and Remove Paths*

  Once we have all the paths containing the keywords, the goal is to remove paths and trim them to reduce the amount found in the CSV report. Each path can be classified into one of the two categories listed below:

  1. The path is a file...

  Ideally if the path is a file, the only ones that should be kept are ***executable*** *files.* Only two possibilities can occur if it is a file. (Note that these two options only indicate the positioning of (1) keyword. There could more than one that shows up in a path.)

❖ /.../.../.../keyword *Keyword is found at the end of the path*

Windows**:** For windows, all .exe files are kept. It could be something like /…/java1234.exe

Linux, AIX, and Solaris: In these operating systems, files do not contain *.exe* ending. Therefore, the files that should be kept are those whose lowest level working directory satisfy the following conditions: contains the keyword "java" and does not have a "."

❖ /.../keyword/.../... *There are no keywords at the end of the path*

In this situation, only the lowest level folder that contains one of the keywords should be kept (Known as "trimming the path"). Some examples of files are given below:

A) E:\Program Files\Java\jre1.8.0_144\bin\java.exe (Good, keep path)

B) /usr/lib/jvm-exports/jre-openjdk/security.log (Bad, need to trim)

→ /usr/lib/jvm-exports/jre-openjdk

C) /opt/bmc/Patrol3/BMCBPAInstall/jreinstaller (Bad, need to trim)

→ /

→ Path deleted, none of the upper level directories contain any of the keywords

Due to the way the recursion search works (it will be explained later), the above method can be simplified and no trimming actually needs to happen. In fact, if java is not found in the end (UNIX) or file does not have an .exe ending (Windows) the file path can actually be ignored and skipped! Below is a summary of what actually happens:

Windows:

If it is an executable keep the path, otherwise throw the path away

UNIX systems:

If it is a file and it contains the keyword "java" and it does not contain a "."

in the lowest level directory, keep the path, otherwise throw the path away

2. The path is a folder

If it is a folder, there are two possibilities that can happen. Note that these two options only indicate the positioning of (1) keyword. There could more than one that shows up in a path:

- /…/…/…/keyword *Keyword is found at the end of the path*

  In this situation, the entire path is kept. There is no need to modify and trim the path

- /.../keyword/.../...

  Similar to above, trimming must occur here. The trim happens until a keyword folder is hit. If the trim deletes everything, the path is skipped and not included in the CSV.

- *Find detailed information about the path's*

  Each path now requires detailed information to be found so that it can be stored in the CSV.

- *Report*

  With all the information, the final report is generated and exported into a CSV.

  This CSV is the passed onto another team for analysis.

A complete algorithm is shown below for a more complete understanding of: ***Finding all installed Java applications on a server***

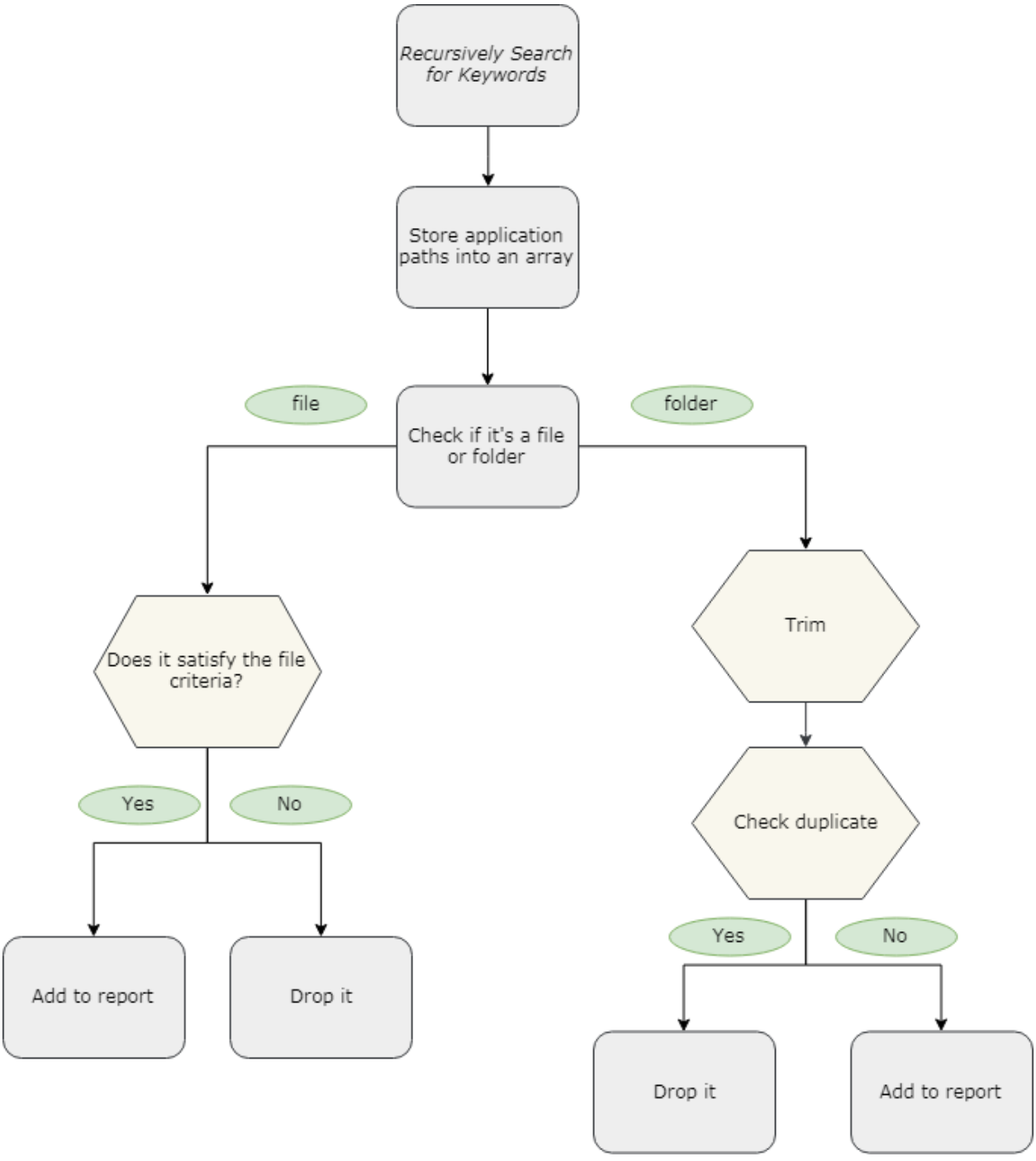

Figure 2. Algorithm for Script 2 – Find All Installed Java Applications

4. Deployment

4.1 Window Server Deployment

On a window's server, the CA tool is used to deploy scripts and files onto servers. With a package created, the tool can deliver this package to 100's of servers at the same time. The CA tool is very important and is used to deploy our scripts via a package.

Inside the package are the main scripts, Script 1 and Script 2, but there are two other ones as well. The first script is a setup script and the other is a task scheduler script. These two scripts are helper scripts and are used to create the right environment to run our main scripts.

4.1.1 Setup script (setup.ps1)

This script is the main setup script. When the CA tool deploys a package onto a server it creates a temp file and can run/execute a script. When it is done, the temp folder is deleted along with its contents. It is clear that this behaviour is not ideal, but, with the setup scripts it can help avoid this problem.

Within the setup script the first task that is done is to check if the server has access to the share folder. If it does not have access that means the report file can not be stored properly. In this situation, the CA tool will not do anything but output a message to tell the user to provide access to the share folder. If the server does have access, the next step is to check if *C:/JavaDiscovery and C:/JavaProcess* is created or not. This is where the scripts are stored so when the temp folders are deleted, the scripts are already stored locally. If the folder already exists, the folder is deleted to have a more recent version of the script in place. If it does not exist, the folder is created and

the main scripts are moved over. After the scripts are moved, the task scheduler script will run and then the main scripts will run. At this point, the CA tool has done its job and is no longer used.

4.1.2 Task Scheduler script (taskScheduler.ps1)

The task scheduler script is executed before the execution of Script #1 and Script #2. This script is essentially used to "create" a task on task scheduler. It creates a task and allows the main scripts to be ran automatically without the need of a user to execute the scripts. For example, it can tell the computer to run Script #2 and scan for java applications once every week on Monday until the end of time.

The primary benefit of this script is that it automatically creates a new task on *Windows Task Scheduler* when the package is deployed onto a new server. The task can then be configured and changed quite easily depending on future needs. Although you can create a new task manually, it would mean that a person must have access to 100+ servers and create 100+ tasks. This is not efficient at all. On the next page a flow chart helps describe the process described in this section. It only describes what happens for the application scripts but the same logic applies for the process scripts.

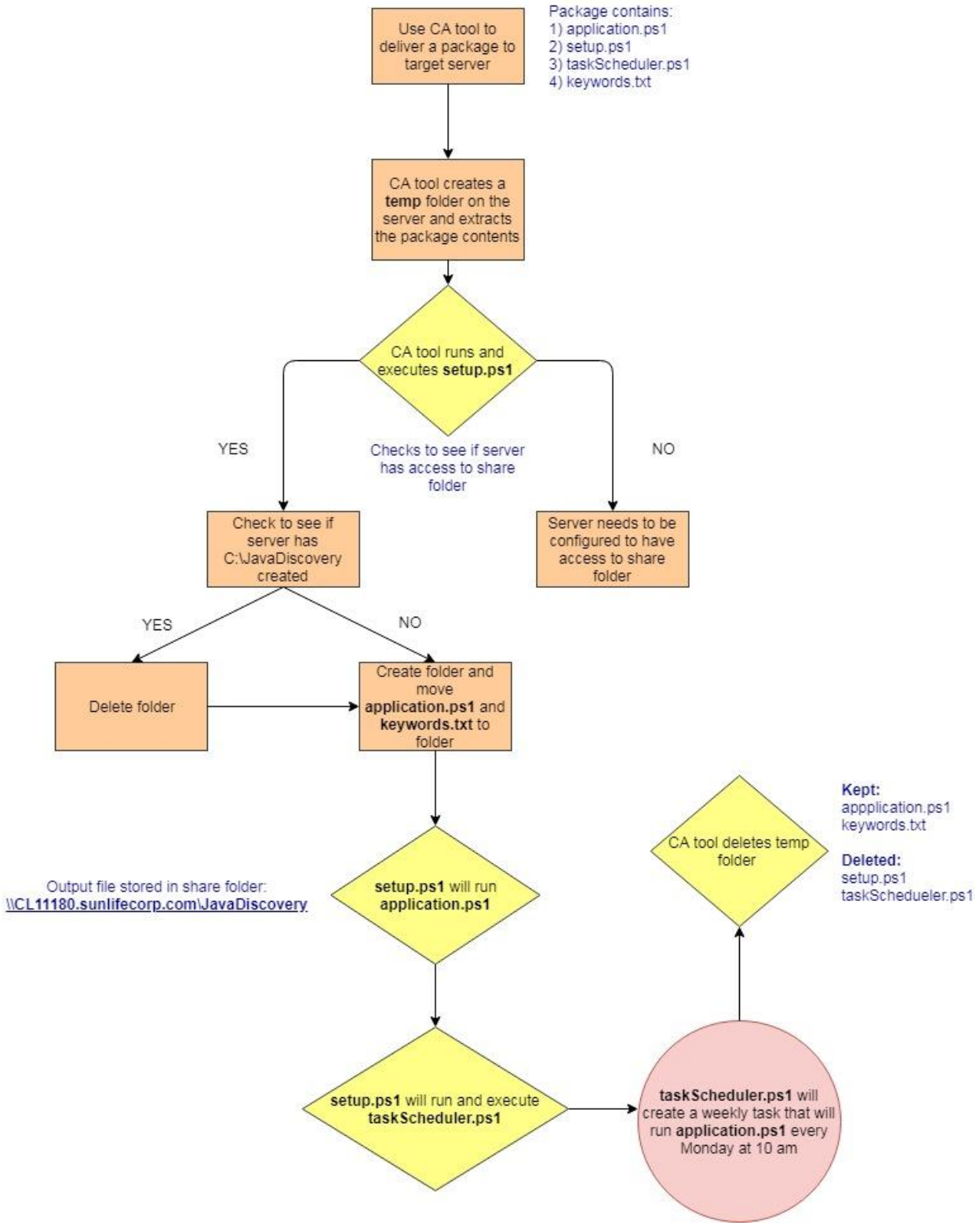

Figure 3. Algorithm for Deployment for the Application Scripts

5. Additional Information

5.1 What is considered a Java File?

A Java file differs from operating system to operating system. Sometimes, a Java file that is of interest can take the form of “java.exe”, for other times, just the word “java” is enough. Luckily, it is possible to narrow down the java files to two distinct cases.

1) Windows Operating Systems

   The main interest for java files are paths that contain: “java.exe” and “javaw.exe”

   Here are some examples…

   A) E:\Program Files\Java\jre1.8.0_144\bin\java.exe

   B) F:\PDConnectivityTest\jdk-11\bin\javaw.exe

2) UNIX Related Operating Systems (Linux, Solaris, AIX)

   On these operating systems, “java.exe” and “javaw.exe” do not exist. Instead, the java file should ideally contain only java. Here are some examples…

   A) /etc/alternatives/java

   B) /opt/ibm/db2/V10.5/java/jdk64/bin/java

These java files are extremely useful because these files allow us to find the installed java versions on the server. Any other file/folder will not yield this information.

5.2 How does path trimming work?

Path trimming was a concept that was introduced by Jennifer Pahati (Security Analyst) and developed by Matthew Tang. The idea was introduced to remove paths that are false positives and only keep the core paths that can be used to provide useful information in the analysis. The trimming method works based on the principle of a "keyword". Trimming can be summarized by the steps below:

1. Identify the last directory in a path and determine if it contains one of the keywords and does not contain a "." (used to get rid of files such as java.html or hello.java)
2. If yes: store the path, and move on.
3. If no: "Trim/Delete" that directory and move to the directory above it. Go back to step one.

*** If the only thing that is left is the root directory (for Unix: '/' and for Windows: C:/ , E:/ , …) the path is deleted. ***

Examples, Keywords (java, jre, jdk, jvm):

C:\hello\java1234\jdk12 (Keep!)

C:\hello\java\world\example.html (Trim!)

→ C:\hello\java\world (Trim!)

→ C:\hello\java (Keep!)

C:\hello\world\ java.txt (Trim!)

→ C:\hello\world (Trim!)

→ C:\hello (Trim!)

→ C:\ (Throw away)

5.3 Why do the paths need to be trimmed?

The way the recursive search is that it finds all paths that contain one or more of the keywords. This is the primary way to guarantee that all java installations on the system can be identified. However, there is a downfall to this. By using this method, it creates a lot of false positives. Below are some examples:

```
/home/ansible/two_liberty_core/roles/java_install
/home/ansible/two_liberty_core/roles/java_install/defaults
/home/ansible/two_liberty_core/roles/java_install/files
/home/ansible/two_liberty_core/roles/java_install/handlers
/home/ansible/two_liberty_core/roles/java_install/tasks
/home/ansible/two_liberty_core/roles/java_install/templates
/home/ansible/two_liberty_core/roles/java_install/vars
/home/ansible/two_liberty_core/roles/java_install/files/java_install.properties
/home/ansible/two_liberty_core/roles/java_install/tasks/java_install.yml
/home/ansible/two_liberty_core/roles/java_install/tasks/main.yml
```

Figure 4. Examples – Why paths need to be trimmed

Only the **first path** can be considered useful. There are a lot of files/folders that will be found that satisfy having a keyword in their path but do not satisfy the main goal of this project: *Find all installed Java applications on a server*. Majority of the paths are not needed and do not provide any useful information at all. Trimming these paths is necessary and what is interesting is that after you trim the rest of the paths (based on the trimming principle) all of them become the first path: /home/ansible/two_liberty_core/roles/java_install

This interesting fact will be discussed next and is vital in improving the efficiency of the program.

5.4 How does the recursive search work?

The command searches from the highest level directory and list all folder and files within it. If it is a folder, it then opens that folder and list all files and folders within that folder. This keeps happening until every single folder and file has been listed, hence the name ***recursive*** search.

```
/var/opt/BESClient/LMT/VMMAN/java/jre
/var/opt/BESClient/LMT/VMMAN/java/jre/bin
/var/opt/BESClient/LMT/VMMAN/java/jre/lib
/var/opt/BESClient/LMT/VMMAN/java/jre/plugin
/var/opt/BESClient/LMT/VMMAN/java/jre/bin/ControlPanel
/var/opt/BESClient/LMT/VMMAN/java/jre/bin/classic
/var/opt/BESClient/LMT/VMMAN/java/jre/bin/ikeycmd
/var/opt/BESClient/LMT/VMMAN/java/jre/bin/ikeyman
```

Figure 5. Examples – How the recursive search works

In the above example, the first path ends with *jre* as the folder name. The search will list all the items in the *jre* folder which are (*bin, lib,* and *plugin*). Once this is done and there are no more items, the search then opens the *bin* folder and lists all the items in *bin*. After this, it will then open any folders in *bin* and repeat the same process. If there are no more folders left in any subdirectory of *bin*, the command will start to list the items in *lib* and the process is repeated. Again, once there are no more items and folders left to open in *lib*, the command will then search for items in *plugin* until there is nothing left.

With how the recursive search works this allows for a very clean and efficient algorithm for trimming and storing useful paths. The highest level directories are always outputted first and then its items (lower directories) are listed below it. This means that for all files (if trimming needs to be done) the trimmed path would be found first before the file itself. With this in mind, as long as it is a folder it gets trimmed and stored (checking for duplicates before storage of course). If it is a

file, all that's needed is for the file criteria to be fulfilled. If it doesn't fulfill the conditions that means that the path needs to be trimmed. (In reality, it doesn't need to be trimmed since it is a known fact that the trimmed file path already exists from how the search path works). This concept allows the script to **SKIPS FILES IF IT DOESN'T SATISFY THE FILE CONDITIONS.**

5.5 How does the running counter work for identifying processes?

The running counter is a column header in the Java processes CSV file. It is primarily used for seeing how many times a certain java process is found. This information is useful because it can help determine which java process is found the most.

Suppose the scan runs once every hour for an entire week. This would mean that the scan happens 168 times. If a process is only found 5 times, it means that it is rarely being used. However, if a process is found 50 or even 75 times, that means this java process runs quite often. This information is really useful to the team that will perform a *Java Subscription Analysis.*

The second part of the running counter is during runtime. For example, if there are two instances of a file named hello.txt opened and the scanner searches for running notepad processes, it will see these two instances. Whether the scanner should count these two as one instance for updating the CSV or combine them and count them as two will be decided by the future teams that are working on it. In both situations, both running processes of hello.txt will have the same command line, path, and etc., but only their process ID differs. *Currently, the script will count multiple instances of the same process separately. So if there are two of the same java processes running, it will count that as 2 and will add 2 to the counter instead of 1.*

5.6 How do you catch errors?

Errors are quite often. Some common errors can include not having enough RAM in the server to run a JVM or even trying to find the date a path was created but having the path be deleted during runtime. All in all, errors are often and there are two main errors that are taken care of.

The first error is trying to find a version of a java file but running into an error. There are a lot of ways for errors to occur when finding a java version. If this happens and the version was not successfully found, the version of the path is just set to NONE and the next path is analyzed.

The second error is missing information. Sometimes whether it's a process or a path, the information is not found. If any of the headers are left blank, that specific path and some error handling information is added to an ERROR LOG and not placed in the CSV. This is specific to UNIX related systems.

5.7 How do we check for multiple instances of a process?

The answer is through the use of the command line. As long as two processes have the same command line we can consider them the same process. In fact when it comes time to update the CSV file, the check is performed using the command line. If the command line is found that means the process has already been scanned before and all that is needed is to increment that process' counter.

6. Differences in Operating Systems

Different operating systems have different specifications. This means that a general solution can be applied to each operating system, however, each differs slightly.

6.1 Windows

- Applications: On a windows environment, .exe files are available. This means that when the file search happens we can gather all .exe files without worrying about missing any of the key "java.exe" files that are required. However, the downside about this is the fact that the results would be broader (more paths inside). There will .exe files that may not be helpful but are only kept because its path has one or more keywords. An example:

  E:\Program Files\Java\jre1.8.0_144\bin\klist.exe

  It is not wise to just limit the search to /java.exe as this would throwaway /javaw.exe, /javac.exe, and etc. Obviously this function can be tuned later on if needed, however, it is best to leave it as the current implementation can provide useful information.
- Currently, on the windows servers can support company/vendor information for running processes and applications

6.2 IBM AIX

- AIX can only support 400 character match and as a result when checking for duplicates only the last 400 characters are checked in the command line. This is done because quite

often the last 400 characters are the arguments and that is often where the difference is between two processes or two applications.

6.3 Solaris and Linux

- These two operating systems work in a similar fashion. There are only minor programming syntax differences.

7. Limitations

7.1 Accuracy (Recursive File Search):

As mentioned before, the search works based on the presence of a keyword inside the path. But the reality of this is that it is not 100% complete and accurate. There is a possibility for a java installation to not have a keyword in its path. Java installations like this would not be found using the current search process and would require another method to find these exceptions.

7.2 CSV Details (Company)

As of right now, only a Windows server can provide the company information of a running java process and java executables. AIX, Solaris, and Linux have the capabilities of finding the company/vendor by navigating through packages, but that function **is not implemented yet**.

7.2 Potential Missing Processes (Windows Environment)

On the Windows server the current method to find a java process is based on its command line. Each running process has a command line and as long as the command line contains one of the following keywords, it will be found. However, there is a problem with that. There is a possibility that a java process will not have any of the keywords found in its command line. In a situation like that, the java process would be not discovered.

8. Development Evolution

This section is a short recap of what was done in order to develop a final solution. It will document what was done and what how it was done.

Initially, the idea was to create one script that could solve both problems. This was not efficient at all as the goals were completely different. Solving two problems with one script was difficult to debug, read, and keep track of what was going on. Eventually, two scripts were created to solve the independent problems.

Finding the running processes first was the first challenge. A java process can be determined by a number of different factors and there were a number of options to choose from: command line, path, process name, and etc. The preliminary idea was to search via process name. Any process whose name had java inside would be considered a java process. However, this approach did not work as there could be applications running java that might not be named java. For example, Eclipse as a platform uses java but when running task manager, eclipse could be running but not "java". At this point, instead of choosing the process name as a match, the team decided to chose the full command line. As long as the path process had java as one of the directories, then that would be considered a java process. Once again, another problem had shown up. There could be a possibility that a java process might not have the keyword java inside its path. It could have other words such as JRE (Java runtime environment). This is when the idea of the ***keywords*** came to place. A document contains a list of keywords and depending on the search, it can be broaden or refined by adding more keywords or removing them.

This idea propelled the development forward. With all the processes found, all that was left was to find details about each process which was accomplished with some detailed research. When everything seemed like it was finished, a new problem occurred. Each process has a unique command line and their arguments. What would happen if there are two of the same processes running at the same time? (Open a text file twice, as an example) Would this need to be considered one or two processes according to the counter? With some discussion, the team decided that they would consider the counter of this process as 2.

Now that the first problem was solved, it was time to move onto the next problem. Using the same idea with keywords, the script began searching through all currently installed/uninstalled programs. Paul Dally (Director of Infrastructure Architecture – Middleware) told myself that it would not be wise to search this way as there are times when java applications are not installed via the "normal" method. Meaning, installed in the usual way via a windows installer. In that case, those applications who were not installed in the normal way would not be found at all. Instead, he proposed a new way, searching through the file system. Using the keywords idea above, a file system search happened matching all paths that contained any of the keywords. Although this method worked, it was quite slow. This is again when Paul helped. As an alternative to doing string matches, he introduced the concept of regular expressions (regex). This method was infinitely faster as it matches based on search patterns.

With a collection of paths, the next step was to process the paths and find additional information about each path. In reality doing this task was tremendously slow. Each path had additional information and a series of commands needed to be run on each path in order to get this information. This is when Jennifer Pahati (Security Analyst) gave myself the idea to trim each path

if necessary until one of the keywords was hit. This idea helped to eliminate majority of the unnecessary paths in the CSV such as paths that were .html files, .java files, .log files, and etc.

Removing the duplicates was the next challenge. There were many trimmed paths that would all be the exact same. The first iteration to solve this problem was to first trim a path, check if the trimmed path was in the storage. If yes, then throw path away and move on, if no, keep path and add to storage. This was done for each path one by one. The above method was quite slow as that meant that you could trim a path and after trimming the path if it was already in storage, it would need to be discarded. This was quite often the case, you could trim 100+ paths and only 1 of them would actually be added,

This is when the second iteration came through. The idea was to reduce the amount of duplicate checking. To do this, all the paths were trimmed if needed (regardless of duplicates) and stored. Then the duplicates were removed. Again this was a little bit faster, but not fast enough. To improve on speed and efficiency, that would mean that the number of trims needed to be decreased or decrease the number of duplicate checks or both.

Due to how the recursive search works and some analysis on the results from method 1 and method 2, a third method was created. This incorporated the idea file checking and combined the idea of trimming and duplicate checking in one step. This idea took some time to develop, however, it is significantly faster than the other two methods and is the one that is currently used.

9. Improvements

This section is meant for those who want to improve the current solution. These are just some general suggestions that may or may not work:

9.1 Improve accuracy for finding java applications:

The file search **alone** may not be the best method to identify java applications. There are some other ways to identify whether it is a java application or not. This could involve looking for a "special file" that is unique to java installations, tokens, file size, and etc. These can be combined with the existing file search to help improve the accuracy and find java installations that could have been missed.

9.2 Improve duplicate check for AIX Servers:

As said above, there could be a problem where the command line is > 400 char and the difference is found within < 400 chars. This is not likely, but completely possible. If the team wants a 100 % duplicate check to work it would be wise to segment the command line into sections of 400 char's and check each section. This was not done in the current solution to save time but it is not a hard implementation to do.

9.3 Tune the results

The next team working on this project can implement an EXCLUDE file where they store words they do not want to find. These words can come from the results of the java subscription analysis. This could tune the results to be more of what they want and they can filter things out that they deem as unnecessary.