Global Information Assurance Certification Paper

Finding Evil in the Whitelist

GIAC (GSEC) Gold Certification

Author: Josh Johnson, jcjohnson34@gmail.com

Advisor: Stephen Northcutt

Accepted: March 23, 2015

Abstract

Application whitelisting technologies are extremely effective at reducing the ability for malicious code to run in an environment. For organizations with limited security budgets, built-in Windows features, such as AppLocker and Software Restriction Policies, offer the ability to implement low-cost whitelisting solutions that can significantly reduce the attack surface on Windows endpoints. While lacking centralized management and reporting consoles, these tools can be tested and deployed with limited effort using scripts to collect and analyze logs and Group Policy to manage whitelists.

Even though whitelisting provides greater protection to endpoints, emerging research is highlighting innovative whitelisting bypass techniques, and attackers are adopting new styles to evade this type of control. However, through regular log review and anomaly detection, organizations can detect and respond to these types of sophisticated attacks that are bypassing application whitelisting utilities. When looking for attacks that are bypassing AppLocker specifically, organizations can lean heavily on the use of PowerShell for log collection and automated analysis.

1. Introduction

Traditional security controls, such as antivirus and intrusion detection systems, can be generally classified as blacklisting technologies. This type of approach to information security, also known as a negative security model, relies on knowing the threats or vulnerabilities from which an organization would want to defend (Jönsson & Iveson, 2014). Signature-based blacklisting technologies are valuable when defending against well-known attack vectors, but they typically provide little value when unknown malicious code is used in an attack. If an attacker develops custom malware that has not been previously observed by security vendors, it is unlikely that the file(s) will match a signature in a blacklisting security product. A positive security model, or whitelist of what is acceptable, has better potential to defend against unknown, targeted attacks than relying on signatures or blacklists to mitigate threats.

The Critical Security Controls (CSCs) outline the most effective and necessary steps an organization must take for proper detection and prevention of cyber-attacks (SANS Institute). Within the CSCs, five "quick win" sub-controls are identified as having the greatest preventive effect. The first of these quick wins is application whitelisting (CSC 2-1), which is included under the second Critical Control, "Inventory of Authorized and Unauthorized Software." Application whitelisting involves developing an inventory of all acceptable software that can run on a system. Once the whitelist is developed, any software not whitelisted will be denied from executing on the system. This technology is effective at limiting the ability for unknown malware to run on systems, as only approved software is whitelisted.

Several powerful commercial application whitelisting technologies are available for organizations to purchase. These solutions generally offer robust methods for building whitelists that allow for simplicity in the deployment of the technology, and some solutions include vendor-maintained whitelists to ease the burden on IT teams responsible for the products' implementation. While these commercial products are excellent tools, organizations may also have the option of using AppLocker, a powerful application whitelisting technology that is built into Windows operating systems.

Josh Johnson, jcjohnson34@gmail.com

1.1. Overview of AppLocker

Organizations running Enterprise or Ultimate versions of Windows 7/8 as well as Windows Server Operating systems are able to implement AppLocker at no additional licensing costs (Microsoft, 2012). This application whitelisting utility allows organizations to develop lists of acceptable applications for certain types of software, and then deny the execution of any software not explicitly whitelisted.

While AppLocker is a "free" tool included with the Windows operating system, deployment and maintenance can be more time consuming and difficult than other commercial whitelisting products if the appropriate planning steps are not taken. AppLocker does not have a central management console, and all configurations must be done through Group Policy. Furthermore, AppLocker logs events to the local Windows Event Log on individual endpoints and does not have built-in remote logging capabilities. Even though these limitations may seem like significant disadvantages when compared to commercial whitelisting tools, AppLocker can still be deployed with reasonable effort using a process similar to the Step-By-Step Guide published by Microsoft (2012). Section 2 of this paper describes the deployment process, and sample PowerShell scripts are provided as appendices that can be customized to help organizations deploy and maintain Applocker policies in their environments.

1.2. Importance of Detection Capabilities

AppLocker is a powerful technology that has the capability to prevent the execution of malware when robust whitelisting policies are in place. However, the technology has well-known weaknesses, described in more detail in section 3 of this paper, which can limit its effectiveness in certain situations. Even though AppLocker cannot be the solution to all security problems, it can be implemented as an effective layer of defense, augmenting the overall security posture an organization.

While bypassing AppLocker is possible in certain situations, its logging features combined with its ability to limit the attack surface on endpoints provides for greater detection capabilities. At the Mandiant-run security conference MIRcon 2014, security professionals, Aaron Beuhring and Kyle Salous, presented on the effectiveness of whitelisting and log monitoring in defending against sophisticated attackers. Their approach involves using application whitelisting to narrow the attack surface on systems,

limiting an attacker's options. Since whitelisting prevents an attacker from simply convincing users to run malicious executables, an attacker should be forced to use exploits to gain access to a system. Other patching and exploit mitigation techniques should be implemented as preventive controls against these threats, but since application whitelisting funnels down the available attack techniques, defenders can focus more effort on detection where successful attacks are viable (Beuhring & Salous, 2014).

Section 4 of this paper describes detection techniques that can be used for identifying situations in which an AppLocker-protected system has been compromised. Furthermore, a sample PowerShell script has been provided in Appendix B that attempts to detect bypass situations and provide additional information to teams responsible for reviewing AppLocker logs.

2. Deploying AppLocker

AppLocker works by whitelisting and blacklisting executable files on Windows systems. The specific file types and associated extensions compatible with AppLocker are listed in Table 1 (Microsoft, 2012).

File Type	File Extension
Executable	.exe
Windows Installer	.msi, .msp
Script	.bat, .cmd, .js, .ps1, .vbs
Packaged App	.appx
DLL	.dll, .ocx

Table 1. AppLocker supported file types

AppLocker permits three types of rules to be created – Publisher, Hash and Path rules. Publisher rules rely on the digital signature of a file and its associated signing entity. Hash rules are implemented by calculating the SHA-256 value of a file and then either permitting or denying the execution of a file based on the hash value. Lastly, path rules permit or deny files based on local or network location. Administrators may use a

combination of these three types of rules in order to develop a robust application whitelisting policy.

2.1. Planning for an AppLocker Deployment

The planning phase of an AppLocker rollout can have a significant impact on the project's overall success. Specific policy and infrastructure controls that are put in place by security-conscious organizations can allow for more rapid AppLocker deployments than environments without these controls. The following planning steps are not necessarily pre-requisites for a successful deployment, but they can limit the need for extensive manual policy development and allow for an easier and more robust rollout.

Since AppLocker is built into the Windows OS and managed via Group Policy, revoking administrator rights from end users has a significant impact on the effectiveness of this application whitelisting tool. First, user accounts with administrator privileges have the ability to modify the AppLocker configuration registry keys and disable it entirely (Microsoft, 2012). As a result, an attacker with an administrator account would have no trouble bypassing whitelisting controls on an AppLocker-protected computer. It is also important to consider administrator rights during whitelist rule development as one of the default AppLocker rules permits administrators to run any application regardless of the application's existence in the whitelist. Implications of this rule are discussed in more detail in section 3.2.1, but if end users are given admin rights, the overall effectiveness of AppLocker is diminished. Finally, restricting administrative privileges is considered one of the "first five" quick wins within the Critical Security Controls (SANS Institute), included as CSC 12-1 under "Controlled Use of Administrative Privileges." If using AppLocker specifically, it may be more beneficial for an organization to address this initiative and remove admin rights where possible before deploying application whitelisting.

A Public Key Infrastructure and strict code-signing requirements can also greatly aid in an AppLocker deployment. Since one of the rule creation options is based on publisher, organizations can leverage code signing of scripts and executables to easily whitelist applications. If written internally, developers can sign each version using a code-signing certificate from an internal Certificate Authority. As long as the CA that issued the code-signing certificate is trusted by AppLocker-protected computers, the

signing entity can be used in AppLocker rules. If the issuing CA is not trusted by the endpoint that is attempting to run an executable, the standard AppLocker prevention message is displayed to the user. This is true even if a rule has been created to permit the execution of any digitally signed executables. Figure 1 shows an executable signed with code-signing certificate issued by a malicious CA as well as the resulting AppLocker message on a system that permits executables signed by any publisher.

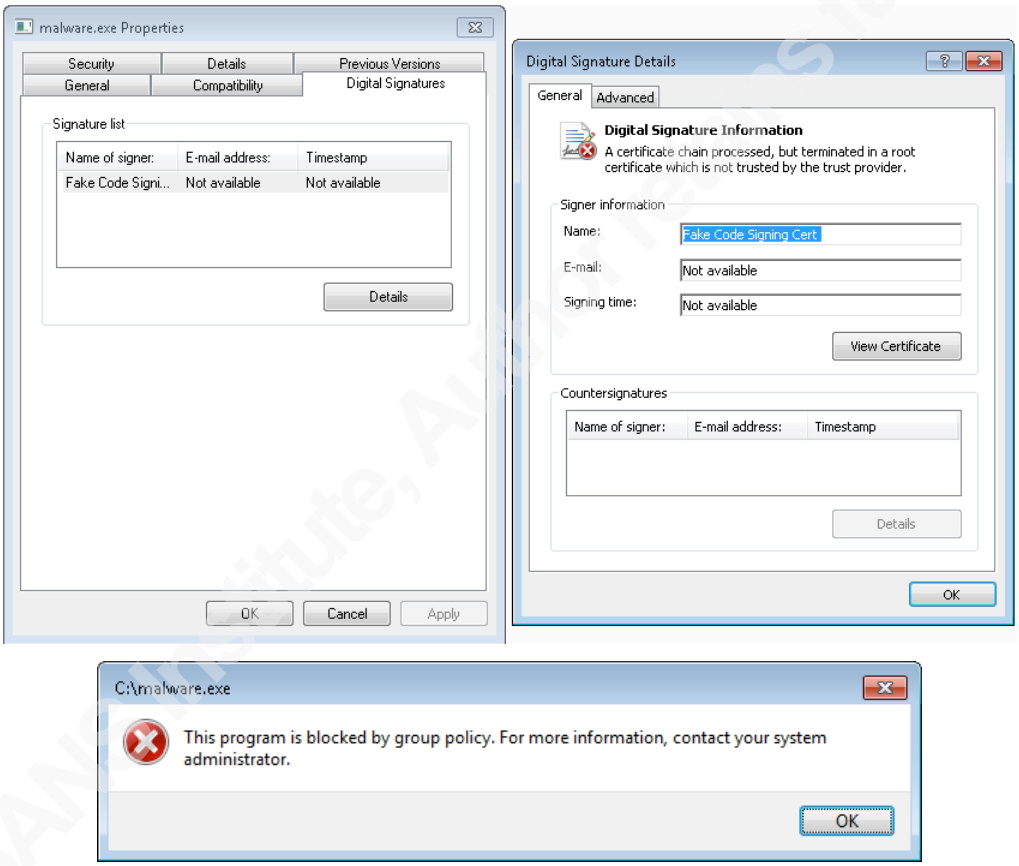

Figure 1. Example of AppLocker blocking an executable signed by an untrusted publisher

Standardized configurations for endpoints can also assist with an AppLocker deployment. Understanding what software will be installed with a baseline image is a prime starting point for developing AppLocker policies, especially if deployed systems do not vary much from the default image. With a baseline image available, several Windows PowerShell Cmdlets have been published by Microsoft and can be used to

easily identify and whitelist software on a known-clean system. Building and maintaining secure standard configurations (CSC 3-1) is also considered one of the "first five" quick wins in the Critical Security Controls (SANS Institute), under the "Secure Configurations for Hardware and Software on Mobile Devices, Laptops, Workstations, and Servers" control. If an organization has already developed secure baseline images, whitelisting with AppLocker is an easier process.

If business units require different application whitelisting policies, a well-designed Active Directory structure will aid in deployment. AppLocker policies can be linked to Organizational Units or AD Groups based on business role of employees or computers. As a result, business roles such as Research and Development can have separate AppLocker policies from HR or Accounting departments. With these key planning phases addressed, an AppLocker deployment can occur more easily.

2.1.1. Sample AppLocker Rollout

The AppLocker deployment process generally involves four phases per application whitelisting policy:

1. Create Baseline Rules
2. Enable AppLocker in Audit Mode
3. Review AppLocker logs and add to whitelist as needed
4. Enable AppLocker in Enforcement Mode

First, an organization must review its environment and determine whether to use the default AppLocker rules or develop individual rules for each whitelisted application. For example, the default Executable Rules allow all files located under the Program Files and Windows directories to run and also include a rule to permit administrators to run any application (Microsoft). As long as users do not have administrator privileges and file system permissions are configured to prevent write-access to these directories by standard user accounts, these default rules can provide significant protection as well as flexibility. However, if end users have write-permissions on any directories in these paths, AppLocker's effectiveness is limited. If this is the case, organizations should consider manually whitelisting individual applications using hash or publisher rules so that user-writeable paths are not inherently trusted. The AppLocker PowerShell Cmdlets

made available via Microsoft can make this process significantly easier by analyzing a known clean system and generating rules based on the systems' contents.

Once a set of baseline rules has been generated, the policy can be configured in an audit mode. In this mode, instead of preventing non-whitelisted files from running on endpoints, AppLocker simply logs a specific event to the local Windows Event log warning that these files would have been blocked if the policy were in enforcement mode. Once initially developed, the AppLocker policy and configuration can be imported into a Group Policy Object (GPO). The GPO can then be linked to a pilot group to analyze the coverage of the AppLocker policy. In order to ensure a cleaner user experience, organizations should build their AppLocker policies using audit mode on production systems to ensure there are no applications that are missed before moving to an enforcement mode.

One disadvantage of using AppLocker versus a commercial application whitelisting utility is the ability to review logs in a central location. Instead, logs are located locally on each machine under *Applications and Services Logs-Microsoft-Windows-AppLocker*, with individual Event IDs for audit mode and enforcement mode events. Using a log forwarding utility or perhaps simply a PowerShell script, relevant AppLocker logs can be gathered from remote endpoints and reviewed in a single location by an analyst. Appendix A includes a sample PowerShell script that gathers Audit mode logs from endpoints for easy review. If an analyst wants more information about an individual file, he or she may invoke additional PowerShell Cmdlets such as Get-AppLockerFileInformation to determine how to create an appropriate whitelist rule (Microsoft).

Once an organization feels comfortable with the results of audit mode logs, indicating an adequate whitelisting policy, they may transition into an enforcement mode. At this point, only whitelisted files will be permitted to execute on AppLocker-protected endpoints. If appropriate effort is not given to the planning, policy development, and audit log review phases of the project, moving to enforcement mode can be burdensome on IT teams supporting AppLocker, as business software may not have been included in the whitelist.

2.1.2. Maintaining AppLocker Policies

Application whitelisting utilities, such as AppLocker, cannot typically be left unmaintained after deployment since business environments are constantly changing. Along with new software deployments and changes to existing applications, the whitelist must evolve at a similar pace with the business. Teams taking on AppLocker deployment projects should understand these operational and support implications and ensure the capacity to continuously support AppLocker is available.

When modifying, removing or adding a new AppLocker rule, the type of rule can have significant security implications. For example, a path rule can permit whitelisting multiple files in a single directory quite easily. However, if the directory's permissions are not well designed, an attacker or nefarious user could move malicious executables into the directory for them to be automatically trusted. Hash rules provide a specific and uneasy to duplicate hash value that a file must match in order to run. While this provides assurance that an imposter cannot be easily forged, any legitimate updates or modifications to the whitelisted file will change the hash value and the AppLocker rule will require updating. Publisher rules allow for an appropriate balance between security and the ability for maintenance and updates to applications. However, not all organizations and software vendors digitally sign their applications. For example, even the most current version of the popular SSH client, Putty, is not digitally signed. As a result, organizations must whitelist Putty.exe using a hash or path rule instead of a publisher rule. Because each application is different, organizations must weigh the maintainability versus protection offered by their AppLocker whitelist when updating and adding new rules.

3. Understanding Potential Weaknesses in AppLocker

While AppLocker is extremely effective at limiting the attack surface on endpoints, no single application whitelisting utility can alone be considered a complete defense against advanced threats. Application whitelisting sets the bar higher than traditional blacklisting controls, such as antivirus, but those implementing AppLocker should understand that this tool cannot provide ubiquitous protection against all known attack vectors. Sophisticated attackers are able to identify and take advantage of inherent as

well as implementation flaws within application whitelisting suites to run and install malicious software.

3.1. Inherent Weaknesses

Similar to other application whitelisting products, AppLocker has fundamental weaknesses and coverage gaps that should be fully understood by organizations deploying this tool. A lack of understanding of these issues can leave a false sense of confidence regarding the extent of protection being provided by AppLocker. One type of weakness includes design and implementation flaws with how AppLocker is integrated into Windows operating systems. Although not a weakness in the software itself, it should be understood that AppLocker does not cover all possible attack vectors, including some well-known and common offensive techniques such as exploits and macro-based malicious code.

3.1.1. Windows Integration Issues

Since AppLocker's release, security researchers have been discovering vulnerabilities in its design and integration into the Windows operating system. Some of these flaws have been corrected through patches and hotfixes while others are accepted as potential bypass opportunities (Microsoft, 2012). For example, in 2011, researchers found that running processes could make Windows API calls with special flags that allowed non-whitelisted executables to run on an AppLocker protected system (Stevens). This meant that whitelisted applications controlled by an attacker could launch executables even if they were implicitly denied by the AppLocker policy. A Mount Knowledge blog post after Didier Stevens' research was published provided a proof of concept macro to demonstrate this flaw from within Word and Excel (2011). Since this flaw's discovery, Microsoft has issued a hotfix (KB2532455), correcting the issue.

With all software, inherent flaws can lead to significant vulnerabilities. As AppLocker continues to be more widely deployed in organizations, attackers will likely increase their research related to AppLocker vulnerabilities in order to develop new bypass techniques. Organizations should monitor security research around AppLocker as well as other application whitelisting utilities to understand and react to newly discovered vulnerabilities.

3.1.2. Scope of Protection

Aside from development weaknesses, it should be understood that AppLocker is not meant to mitigate all attacks against Windows systems. First, AppLocker cannot aid in the detection or prevention of exploits against whitelisted software. Since AppLocker simply permits applications to run based on the publisher, hash or location of the executables, it does not have the capability to prevent anomalous behavior including exploitation within whitelisted applications. As a result, shellcode embedded within a successful exploit can run without any interrogation by AppLocker. While the code embedded within an exploit will run successfully, an attacker would need to fully understand the AppLocker policy in order to download additional malware that would be permitted to run. For example, if shellcode simply makes Windows API calls to download and run another executable, the downloaded file would need to satisfy an AppLocker rule before it would be permitted to run on the system. Because of its ability to hamper this stage of an attack, AppLocker adds an effective layer of defense against exploits containing downloader or dropper payloads (Milunski, Dereszowski & Cox, 2012). However, if installing additional malware is not an attacker's goal, exploits can still be used to attack legitimate applications and run malicious code on AppLocker-protected systems.

Another well-known weakness in AppLocker is its inability to prevent macros within Microsoft Office documents from running. As there may be a legitimate business need for Office, specific applications such as Word and Excel would be included in the whitelist. Since AppLocker permits these applications, macros embedded in their respective documents are inherently trusted by AppLocker. As a result, VBA code in macros can be used to establish command and control channels and perform other malicious actions on behalf of an attacker, all running within a whitelisted executable. Organizations should implement other controls to limit the risk related to the execution of macros, but it should be understood that AppLocker cannot be used to mitigate this risk (Microsoft, 2012).

Lastly, similar to its inability to limit exploitation and macros, AppLocker cannot prevent the abuse of whitelisted executables. If an attacker gains access to a system running AppLocker, all whitelisted applications are available for use on the compromised

system. If post-exploitation tasks can be accomplished using built-in Windows tools and other whitelisted applications, an attacker would not need to download any additional malware in order to achieve his or her goals. Instead, permitted executables would be used for nefarious purpose, allowing an attacker to take action on objectives while working within the whitelist.

In 2014, FireEye published research indicating that attackers are increasingly using built-in Windows management tools like PowerShell and WinRM to traverse a network rather than installing custom malware (Kazanciyan & Hastings). As a result, forensic artifacts left behind after an intrusion are changing, forcing incident response teams to understand how to analyze these types of post-exploitation actions. When looking at this research from an application whitelisting perspective, it becomes apparent that organizations should fully understand the impact of whitelisting seemingly innocuous Windows applications if they can be used for harm by attackers who have already compromised a system.

3.2. Implementation Flaws

While AppLocker has inherent weaknesses that can be targeted by attackers, organizations can also significantly limit the effectiveness of an application whitelisting policy by creating poorly designed rules. With unlimited time and resources, an attacker could find holes in almost any whitelisting policy. However, a design goal when building the whitelist should be to make this process difficult and costly for an attacker.

3.2.1. Administrator Privileges

With administrator rights on a system, bypassing AppLocker becomes trivial. AppLocker relies on the Application Identity service to profile executable files and compare applications with the rules in the whitelist. An administrator can simply stop this service and render AppLocker useless. While AppLocker can still provide limited value if users have administrator privileges, it should be understood that bypass of the application whitelisting policy will be no trouble for an attacker with these rights.

3.2.2. Path Rule Considerations

Poorly designed path rules can also lead to trivial bypass conditions for attackers. When creating a rule based on a file's location, the permissions on the parent directory

should be verified before adding it to the whitelist. If the directory permits write access for standard users, an attacker using a compromised account could put malware into the directory and the malicious code would be automatically whitelisted. Path rules should be carefully planned before they are implemented in an AppLocker policy, as weak path rules can lead to simple bypass conditions that significantly limit the effectiveness of this application whitelisting tool.

3.2.3. Script Interpreters

Another serious weakness in AppLocker's capabilities involves scripting language interpreters such as Perl and Python. On Windows systems, these interpreters are installed as a series of executables and libraries. When running a script, they create a process, interpret and execute the code in the script. In order to integrate AppLocker to which control which scripts can run, the created process must make calls to AppLocker prior to running the script. At the time of this writing, only Microsoft's built-in script interpreters make these calls and interface with AppLocker while non-Microsoft interpreters, such as Perl, Python and Ruby, do not. If an organization supports business applications on one of these non-Microsoft platforms, related executables will need to be whitelisted for their respective scripts to run properly. This also means that any script supported by the interpreter will be permitted to run and cannot be controlled by AppLocker. For example, Python scripts are invoked via Python.exe, so once the interpreter is whitelisted, any Python script can be run on the system without verification by AppLocker.

As these scripting languages have extensive power to manipulate the underlying operating system, blindly whitelisting any script of a given language can have serious implications. Many offensive tools, such as Metasploit and the Social-Engineer Toolkit, are written in scripting languages, so these would be automatically whitelisted if their respective interpreters were permitted by the AppLocker policy. Furthermore, an attacker with the ability to run an interpreter like Python without restrictions would have no problem compromising a system, establishing command and control channels, performing post-exploitation tasks and exfiltrating data using only capabilities within the whitelisted interpreter.

Since Microsoft has enabled integration with its modern scripting language, Windows PowerShell, specific scripts written in this language can be controlled with AppLocker. However, PowerShell offers a command line interface that allows users to issue individual commands without ever invoking an actual script. As a result, if PowerShell's interpreter, powershell.exe, is whitelisted, attackers on the system can abuse it. Similar to Perl and Python, PowerShell is tightly integrated into the operating system and has useful built-in functionality to interact with the system as well as domain services such as Active Directory. With PowerShell available, an attacker has the tools necessary to accomplish post-exploitation tasks on a system and network. Researchers are increasingly publishing new attack techniques and toolsets to attack systems using only PowerShell (Kennedy & Kelley, 2010). Furthermore, incident response research shows evidence that PowerShell is being used in targeted attacks against organizations. As a result, even if specific PowerShell scripts have been whitelisted, organizations should understand the risk of whitelisting powershell.exe on standard user workstations.

3.2.4. Java

Perhaps the most significant implementation weakness in an AppLocker deployment is the whitelisting of Java. Similar to many script interpreters, Java has significant access to the underlying operating system but does not integrate with AppLocker to support whitelisting of individual Java-based applications. As a result, AppLocker cannot whitelist specific JAR files or applets; instead, Java executables must be explicitly trusted or blacklisted as part of the application whitelisting policy. Java is common in business applications for several reasons including its ability to run the same code across different operating systems and portability and its ease of development. Because Java is so common, excluding it entirely from a whitelist could prevent the execution of business applications. This inability to limit exactly which Java applications are permitted can undermine an application whitelisting effort if Java-based attacks are utilized.

For the same reasons that Java is popular among business application developers, it is also a commonly used platform among attackers who write malicious software. Java provides the ability for an attacker to create weaponized code that is operating system agnostic and does not need to be as customized for a specific target as native applications.

As a result, the exact same Java-based malware could potentially be used against a large number of targeted systems, even if their configurations vary significantly.

To further complicate matters, Java has been riddled with vulnerabilities throughout its history. Specific version dependencies for legacy business applications can increase the difficulty for organizations to maintain current versions of Java, leaving it open to exploitation. Traditional blacklisting technologies, like antivirus, are capable of identifying known exploits and previously identified Java-based malware. However, signature-based security solutions are typically not adequate for detecting or preventing unknown malicious code. Coupling this idea with the fact that AppLocker cannot control Java-based applications, organizations deploying AppLocker must be fully aware of this significant weakness and implement compensating controls if Java must be whitelisted.

4. Increasing Detection Capabilities

While AppLocker certainly has weaknesses that can leave application whitelisting bypass opportunities available to attackers, its benefits include the ability to greatly reduce the attack surface on systems. Through learning how systems operate under normal conditions, organizations can focus their attention on what is actually running on AppLocker protected systems and look for deviations from what is normal rather than relying on signature and blacklist-based technologies to adequately identify threats. With the ability to baseline the execution of applications on a system, organizations can monitor and alert on anomalous activity including identifying the execution of potentially dangerous but whitelisted executables. Furthermore, organizations can use built-in logging capabilities of applications, like Java, to further understand what non-native code was run on a system.

4.1. Monitoring the execution of applications

Understanding what files are blocked as well as permitted by AppLocker can provide actionable data for security teams to respond to threats. Each type of AppLocker event has a unique Event ID within the Windows event logs, and desired events can be aggregated to a central location for review. If an unknown executable is observed being blocked across multiple systems, it may be worth investigating to identify the source and

nature of the file. Additionally, when malicious executables are identified in AppLocker logs, security teams may trace the file to its originating location identifying which controls were bypassed for the file to be placed on the system and nearly executed. With this data, existing network and host-based controls can be better tuned or additional controls may be implemented.

The PowerShell script in Appendix A can easily be modified to collect events in which AppLocker is blocking the execution of files. In the configuration section at the beginning of the script, simply set the "$AuditMode" variable to "$false", and the script will collect files that have been blocked by AppLocker. Regularly reviewing these logs can reveal the value provided by AppLocker as well as possible situations in which business interruption is occurring due to misconfigured AppLocker policies.

4.1.1. Using PowerShell to Monitor Whitelisted Files

While AppLocker cannot stop all attacks and may be bypassed, its logs can also be monitored to help detect intrusions in which application whitelisting controls are subverted. Just as AppLocker logs individual events when it blocks applications, it also logs an event every time an application is permitted to run. These events can also be aggregated to a central location and reviewed to detect anomalies or other possibly malicious patterns. Appendix B includes a PowerShell script, AppLocker_Log_Analysis.ps1, which collects permitted events from the AppLocker logs of remote systems and analyzes these logs to identify potential threats that are permitted by the whitelist. This proof of concept script attempts to perform the following high-level actions on AppLocker-protected systems:

1. Collects AppLocker logs for permitted running of executables
2. Searches the logs for possible signs of attacker activity
3. Aggregates runs of individual executables into a summary format, showing how many times each executable has been run, the SHA-256 hash of each file, and the first and last observed run times
4. Monitors paths in cases where organizations must create dangerous AppLocker Path Rules, alerting if files in the paths change or new files are added

5. Searches logs for signs of Java execution and attempts to correlate the AppLocker events with Java Usage Tracker logs to show which Java files were run

The AppLocker_Log_Analysis.ps1 script requires PowerShell 4.0, and its options and examples can be shown by running the following commands at a PowerShell prompt:

```
>Get-Help AppLocker_Log_Analysis.ps1
>Get-Help AppLocker_Log_Analysis.ps1 -examples
```

These commands show how to run the script and enable the individual modules or full functionality based on the needs of the user. If no arguments are provided, the script will simply collect the AppLocker logs for permitted execution from all PCs defined in the configuration section of the script. This is accomplished by running the Get-WinEvent Cmdlet using the appropriate AppLocker Event IDs and subsequently storing the results in CSV files for review. Collection of events from multiple systems has been optimized using a function called Get-AsyncEvent, written by Jason Walker in 2013. This function uses runspaces and multiple threads to query several systems at the same time.

Additionally, the script will parse the results and look for evidence of powerful executables that standard users may not need to run. These are files such as cmd.exe and reg.exe, which are used often by IT users, but may not be normally run on standard user systems. If these files are found in the AppLocker logs, this could indicate an attacker using native tools within the operating system instead of downloading and running malware on the system. Additionally, this could indicate an attacker who has compromised credentials and has accessed a system using a legitimate account rather than using exploits or malware. The list of executables used in the search has been sourced from a 2014 blog post by Patrick Olsen on sysforensics.org; the URL for the post can be found in the references section of this paper. If any of these executables are found in the AppLocker logs, the script will write them to the PowerShell command line console as well as aggregate these events into a file called AppLocker_Alerts.csv.

If the *-getstats* switch is used, the script will parse the gathered logs and calculate statistics for each file that has been run. These statistics include the number of times the

file was executed, its SHA-256 hash value and the first/last run times observed in the logs. If users' actions on a particular systems are very consistent, this view of the data could potentially reveal outliers if an attacker gains access and breaks the pattern of consistency. Figure 2 shows the results of the *-getstats* switch opened in Microsoft Excel.

MachineName	FilePath	Hash	NumRuns	FirstSeen	LastSeen
gsec-win7	%OSDRIVE%\USERS\USER2\DESKTOP\ACRORD32.EXE	0x7DA5D295D945903F6AE5BEF2DF3CBDA482C0A	1	1/27/2015 18:46	1/27/2015 18:46
gsec-win7	%PROGRAMFILES%\INTERNET EXPLORER\IEXPLORE.EXE	0x86611B8CA28A53A7B8BC44979B0978D52B367A	4	1/25/2015 22:57	1/25/2015 22:57
gsec-win7	%PROGRAMFILES%\JAVA\JRE1.8.0_31\BIN\JAVA.EXE	0x8524E109C6BA97500154348B129A29FB7F92889	4	1/25/2015 23:00	1/26/2015 23:14
gsec-win7	%PROGRAMFILES%\JAVA\JRE1.8.0_31\BIN\JP2LAUNCHER.EXE	0xD2FD607D28995A44717A012CFF17444878C538	2	1/25/2015 23:00	1/25/2015 23:00
gsec-win7	%SYSTEM32%\CMD.EXE	0x9E131D3B1D6481B6027035128F290935DB3E418	4	1/26/2015 23:04	1/26/2015 23:10
gsec-win7	%SYSTEM32%\CONHOST.EXE	0x0EAD3F4E989F8756EF21DC19D12207DCC89792	8	1/25/2015 23:01	1/26/2015 23:14
gsec-win7	%SYSTEM32%\DLLHOST.EXE	0x9A00E2E4B3D514C7D29B66243F31F9DC9AB22B	13	1/27/2015 15:17	1/27/2015 17:08
gsec-win7	%SYSTEM32%\FTP.EXE	0x3F68101A7C2062BEFB91E4C2EBE7E8B862DB3A	3	1/25/2015 23:02	1/26/2015 23:31
gsec-win7	%SYSTEM32%\GPUPDATE.EXE	0x06C481816D7A99FEDA592E7172002F8754FC944	5	1/27/2015 16:24	1/27/2015 16:28
gsec-win7	%SYSTEM32%\NOTEPAD.EXE	0x15FB289DB42A1601933E4FAB114CE8FEA1F90D	1	1/26/2015 11:49	1/26/2015 11:49
gsec-win7	%SYSTEM32%\PING.EXE	0xA34D8A440AC42CE19BA1B55B25B5C07888BAF	3	1/25/2015 23:01	1/27/2015 16:25
gsec-win7	%SYSTEM32%\RUNDLL32.EXE	0x62FAE8127BC3FB336F4A0CDE1D6ED20B6F82E7	5	1/25/2015 22:57	1/25/2015 22:59
gsec-win7	%SYSTEM32%\SEARCHPROTOCOLHOST.EXE	0x6812993A59E64A55856274AE26BB7ED48EEB31	4	1/25/2015 22:57	1/25/2015 22:59
gsec-win7	%SYSTEM32%\SLUI.EXE	0x4AE50D4E0807C62CC9118284A39F87CF78B81FA	19	1/26/2015 23:44	1/27/2015 16:25
gsec-win7	%WINDIR%\EXPLORER.EXE	0xC10DC8CF897E25BA4BD708EBBE983921203525	1	1/26/2015 11:49	1/26/2015 11:49

Figure 2. Results of *-getstats* switch, opened in Microsoft Excel

When the *-checkpaths* switch is provided, the script will parse the directories defined in the configuration section of the script. It will then inventory the files in each directory, calculating the SHA-256 hash value of each file and logging the results to a file called AppLocker_Monitored_Paths.csv. On subsequent runs of the script, alerts will be generated if any files within the path change or if any new files are added to the monitored directories. This can be helpful if business requirements result in AppLocker path rules for executables in directories that may have weak write-access permissions such as file shares. In such cases, this script could be run regularly to ensure these paths are not modified. Optionally, the *–nologs* switch can be provided to skip the log collection from endpoints and only check paths. Figure 3 shows an example AppLocker_Monitored_Paths.csv file opened in Microsoft Excel.

Path	File	Hash
\\dc-01\netlogon	logonscript.cmd	0x497B22D4E86A3CAA9F5BAA24435A99AC1154094A0B9302B9BCD9D6544D6EFBE9
\\dc-01\netlogon	share_config.cmd	0x031CAE78A9B30C77DE9B71C0C02A7CBAD1FB302D7798946A0EDADA3F48D45D57

Figure 3. Results of *-checkpaths* switch, opened in Microsoft Excel

Josh Johnson, jcjohnson34@gmail.com

If any files change or a new file is added, users running the script will see output in the Windows PowerShell console similar to what is shown in figure 4.

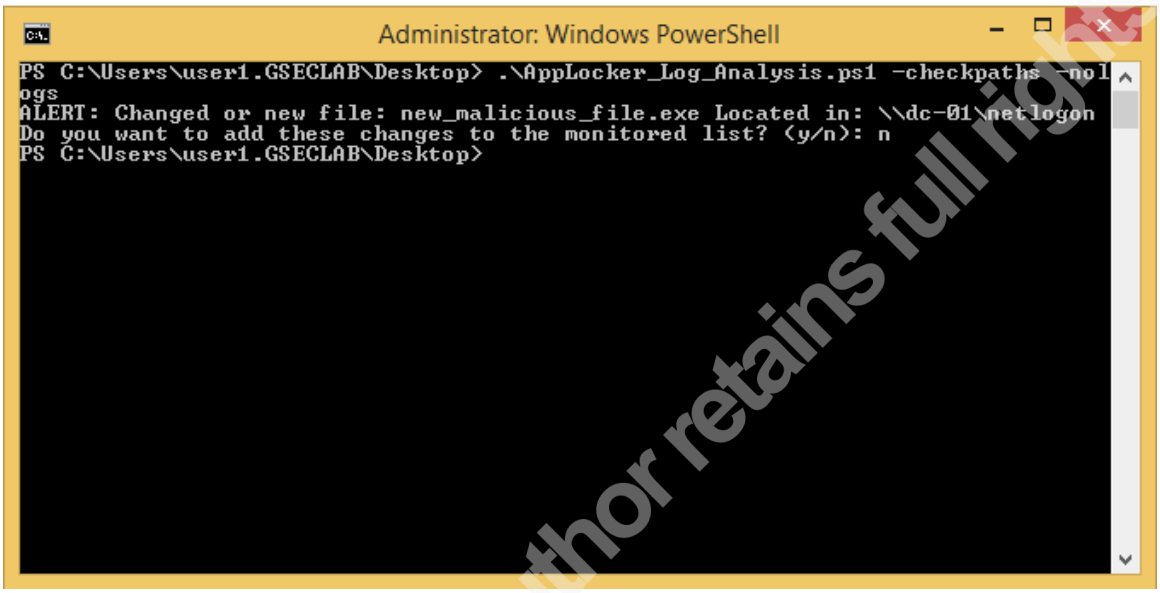

Figure 4. Results of *-checkpaths* switch when a new/modified file is discovered

4.2. Monitoring Java

If Java.exe is whitelisted for business applications, AppLocker cannot provide significant protection against Java-based threats. As a detective control, organizations can review AppLocker logs where Java is invoked to determine what Java executable code was run. The Java Runtime Environment provides a useful feature called Java Usage Tracker that can aid in this effort. When enabled, this feature logs the name and location of the executable as well as the arguments and other details surrounding its execution each time Java is invoked. When correlating AppLocker logs with Java Usage Tracker logs, it is possible to identify more details around what Java was doing each time it was run.

Java Usage Tracker can be enabled by creating a file called *usagetracker.properties* in <JRE Installation directory>/lib/management/ (e.g. C:\Program Files (x86)\jre7\lib\management\usagetracker.properties) containing the following line:

com.oracle.usagetracker.logToFile = ${user.home}/.java_usagetracker

Once this file is created, the JRE will log all execution to the initiating user's home directory in a file called *.java_usagetracker*. These logs can be correlated with AppLocker permitted execution logs to provide a clearer picture of what executables were run on a system when Java is involved.

The script in Appendix B, when invoked with the *-checkjava* switch, will attempt to automatically correlate AppLocker logs with Java Usage Tracker logs. Results of the script will be written to a file called AppLocker_Java_Logs.csv, containing the location of the file run as well as well as the arguments provided on startup. In the event of Java-based malware in the form of a JAR file in a user's AppData directory, this script would provide a security team with the location of the malicious file for further analysis.

4.3. Example Use Case

This section outlines a scenario in which a Java-based attack can be detected through the monitoring and correlation of AppLocker and Java Usage Tracker logs. In this example, the victim, 10.10.10.4, was running a Windows 7 Enterprise system with the default AppLocker rules configured in Enforcement mode. The attacker, 10.10.10.11, was using the Java_Signed_Applet Metasploit module in order to gain access to the victim's system. This Metasploit module creates a malicious signed Java applet and requires social engineering for a victim to visit the web page hosting the applet. If the user permits the applet to run, the attacker's payload is invoked.

In this example, the attacker chose to use a Java-based exploit as well as a Java-based payload providing a Metasploit Meterpreter shell. Since the exploit invoked the whitelisted "JP2Launcher.exe" Java executable, and since the payload ran via the whitelisted "Java.exe" file, AppLocker does not have the capability to prevent this attack. However, detecting this type of attack could be accomplished easily through reviewing logs, especially if the attacker uses native Windows tools to accomplish post-exploitation goals.

When the victim visits the attacker's malicious web page hosting a self-signed Java applet, modern versions of Java prompt the user with a security warning. However, assuming the user will not always show proper security awareness, this warning may be ignored and the malicious applet will be run. As a result, once the prompt is accepted, the attacker's Meterpreter session is established as shown in figure 5.

```
msf exploit(java_signed_applet) >
[*] 10.10.10.4       java_signed_applet - Handling request
[*] 10.10.10.4       java_signed_applet - Sending SiteLoader.jar. Waiting for user to click 'accept'...
[*] Sending stage (30355 bytes) to 10.10.10.4
[*] Meterpreter session 6 opened (10.10.10.11:4444 -> 10.10.10.4:53234) at 2015-01-25 14:56:49 -0500
sessions -i 6
[*] Starting interaction with 6...
```

Figure 5. Successful attack yields a Meterpreter session for attacker

At this point, the payload was running and Java.exe has been logged by AppLocker. Even though the payload is a Java-based Meterpreter session, other post-exploitation actions can still be observed in the AppLocker logs as well. For example, if the attacker runs the Meterpreter "shell" command, cmd.exe executes and is reflected in the AppLocker logs. From a Windows shell, subsequent commands are also logged by AppLocker. In the case of this attack, the attacker confirmed network connectivity by pinging the system he controls as shown in figure 6.

```
meterpreter > shell
Process 1 created.
Channel 1 created.
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\user2\Desktop>ping 10.10.10.11
ping 10.10.10.11

Pinging 10.10.10.11 with 32 bytes of data:
Reply from 10.10.10.11: bytes=32 time<1ms TTL=64
Reply from 10.10.10.11: bytes=32 time<1ms TTL=64
Reply from 10.10.10.11: bytes=32 time<1ms TTL=64
Reply from 10.10.10.11: bytes=32 time<1ms TTL=64

Ping statistics for 10.10.10.11:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 6. Attacker executing "shell" and "ping" commands on compromised system

Shown in figure 7, the attacker discovered a sensitive file called passwords.txt. Once the attacker found this file, it was transferred to the attacker-owned system using the built-in Windows FTP client. Figure 8 shows the attacker creating and running an FTP script that is able to exfiltrate the 'passwords.txt' file.

```
C:\Users\user2\Desktop>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 4C2A-1C1D

 Directory of C:\Users\user2\Desktop

01/25/2015  10:53 PM    <DIR>          .
01/25/2015  10:53 PM    <DIR>          ..
01/25/2015  10:11 PM               180 passwords.txt
               1 File(s)            180 bytes
               2 Dir(s)  20,750,376,960 bytes free
```

Figure 7. Attacker discovers "passwords.txt" file

```
C:\Users\user2\Desktop>echo open 10.10.10.11 21> ftp.txt
echo open 10.10.10.11 21> ftp.txt

C:\Users\user2\Desktop>echo ftp>> ftp.txt
echo ftp>> ftp.txt

C:\Users\user2\Desktop>echo ftp>> ftp.txt
echo ftp>> ftp.txt

C:\Users\user2\Desktop> echo bin>> ftp.txt
        echo bin>> ftp.txt

C:\Users\user2\Desktop>echo put passwords.txt /upload/pass.txt>> ftp.txt
echo put passwords.txt /upload/pass.txt>> ftp.txt

C:\Users\user2\Desktop>echo quit>> ftp.txt
echo quit>> ftp.txt

C:\Users\user2\Desktop>ftp -s:ftp.txt
ftp -s:ftp.txt
User (10.10.10.11:(none)): open 10.10.10.11 21


bin
put passwords.txt /upload/pass.txt
quit

C:\Users\user2\Desktop>
```

Figure 8. Attacker constructs FTP script and executes it to steal passwords.txt

AppLocker has not prevented any of the actions at this point since whitelisted, built-in Windows tools have been used by the attacker. However, logs have been created that can help defenders identify and respond to this issue. If the AppLocker_Log_Analysis.ps1 script is used with the *-checkjava* switch, it becomes apparent that this machine may have been compromised. The console output of the script is shown in Figure 9. Immediately, an analyst running the script should become aware that cmd.exe, ping.exe and ftp.exe were run on the compromised system. As long as

these are not normally run on the users' system, this should be a strong indicator that further analysis is required.

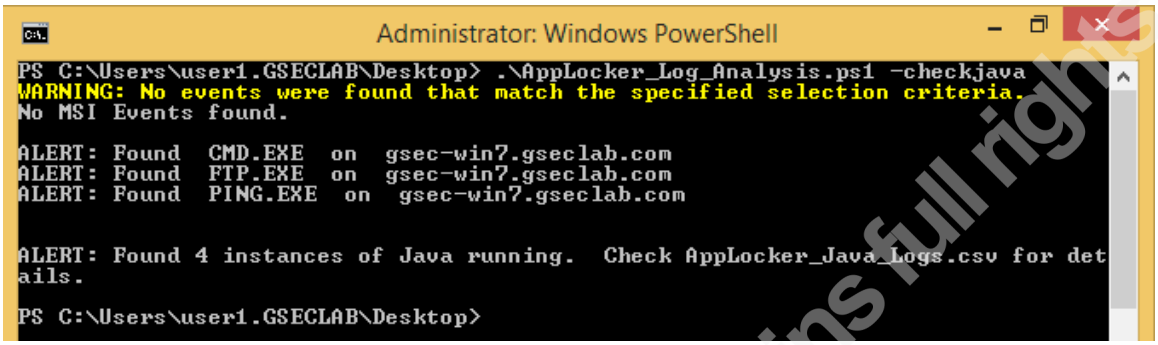

Figure 9. Results of *-checkjava* switch showing alerts for potentially dangerous files

Once the AppLocker_Java_logs.csv file is opened and examined, additional indicators will be uncovered. If Java-based business applications have been inventoried, reviewing these logs against the known applications should show anomalous Java usage. Figure 10 shows the .csv file opened in Excel.

MachineName	TimeRun	JavaApp	AppArgs	ClassPath
gsec-win7	Sun Jan 25 23:00:30 EST 2015	"sun.plugin2.main.client.PluginMain read_pipe_name=jpi2_pid2592_pipe2	"6.1"	"'-Xbootclasspath/a:C:\Program Files (x86)\Java\jre1.8.0_31\lib\deploy.jar;C:\Program Files (x86)\Java\jre1.8.0_31\lib\javaws.jar;C:\Program Files (x86)\Java\jre1.8.0_31\lib\plugin.jar' -Djava.security.manager -D__jvm_launched=370888627012 -D__applet_launched=370888513055 "
gsec-win7	Sun Jan 25 23:00:38 EST 2015	"http://10.10.10.11/sploit/:launchjnlp=code=SiteLoader codebase=http://10.10.10.11/sploit/ width=1 archive=/sploit/SiteLoader.jar height=1 "	"'-Xbootclasspath/a:C:\Program Files (x86)\Java\jre1.8.0_31\lib\deploy.jar;C:\Program Files (x86)\Java\jre1.8.0_31\lib\javaws.jar;C:\Program Files (x86)\Java\jre1.8.0_31\lib\plugin.jar' -Djava.security.manager -D__jvm_launched=370888627012 -D__applet_launched=370888513055 "	"C:\Program Files (x86)\Java\jre1.8.0_31\classes"
gsec-win7	Sun Jan 25 23:00:43 EST 2015	"GralpktGUU.Payload"	""	"C:\Users\user2\AppData\Local\Temp\~spawn9047651396822970636.tmp.dir"
gsec-win7	Sun Jan 25 23:00:44 EST 2015	"GralpktGUU.Payload"	""	"C:\Users\user2\AppData\Local\Temp\~spawn8338690638644601675.tmp.dir"

Josh Johnson, jcjohnson34@gmail.com

Figure 10. Results of -*checkjava* switch, opened in Microsoft Excel

In the CSV file, the logged JavaApp field actually includes the URL from which the malicious applet originated. If this URL is not recognized, it may be worth investigating to see if it is legitimately used for business purposes or if this is a malicious site. This could also be helpful if web server is still serving the file and the incident response team needs a copy of the "SiteLoader.jar" file for analysis. Furthermore, the randomly named ".payload" files and their location within the user's AppData directory could be another indicator that an attack has occurred against this system. With all of this data, an incident response team should be able to isolate the infected PC even though AppLocker does not have the capability to prevent this type of intrusion.

Future enhancements to the AppLocker_Log_Analysis.ps1 script could include the capability to automatically retrieve files of interest based on the Java Usage Tracker logs. Other script interpreters like Perl or Python may have or develop functionality similar to Java's Usage Tracker; if this becomes available, these logs could also be integrated with AppLocker logs to provide a clearer picture of the code executing on a PC. However, if non-Microsoft script interpreters must be whitelisted, it may be worthwhile to implement additional compensating controls such as virtualizing the systems that run these applications and consistently reverting to a known clean state. If non-Microsoft script interpreters can be excluded from the whitelist, this script can be a useful detective control that takes advantage of AppLocker's powerful logging capabilities to detect threats that have bypassed AppLocker.

5. Conclusion

AppLocker provides significant protection against malware on Windows systems. An effective AppLocker implementation has the ability to limit the execution of executables, certain types of scripts, installers and libraries to a list of known and approved files. While not as flexible as some commercial whitelisting products, AppLocker does provide options to create rules based on file location, hash value or publisher. These three options are accommodating to most types of Windows software and can, if created properly, provide significant assurance that only whitelisted files may

run on the system. A design goal when developing application whitelisting policies should be to force attackers to use exploits instead of allowing for users to fall victim to social engineering and run potentially malicious executables.

Although AppLocker has the ability to greatly reduce an attacker's options for gaining access to a system, it is not a foolproof technology. Attackers who use exploits to gain access to an AppLocker-protected system have full use of any whitelisted applications, and sophisticated attackers can accomplish a great deal of post-exploitation work using only tools native to the Windows operating system. Furthermore, inherent weaknesses such as the inability to prevent macros within Office documents leave distinct opportunities for attackers to run malicious code against systems running AppLocker. Implementation weaknesses like whitelisting script interpreters that are not compatible with AppLocker can be significant, self-inflicted holes in an otherwise robust whitelisting policy. Lastly, since so many business applications run on Java, whitelisting the Java Runtime Environment opens additional attack vectors and demands compensating controls to protect systems running AppLocker.

Organizations should regularly review AppLocker policies, looking for potentially weak rules and testing the protection on production systems. With an understanding of what holes may exist in an AppLocker policy, additional preventive and detective controls may be implemented. The regular review of AppLocker logs can be an effective detective control, showing which files have been blocked and possibly more importantly, which files have been permitted to run. Since attackers do not necessarily need anything other than a set of credentials and built-in Windows tools to traverse an internal network, AppLocker logs should be reviewed for signs of this activity. Additionally, if a weak rule exist such as whitelisting a path that does not have strict write-access permissions, additional monitoring should be implemented as a compensating control. AppLocker's built-in logging provides the possibility for an effective detective control on top of the tool's excellent preventive capabilities.

The sample PowerShell scripts in Appendix A and Appendix B serve as examples of the powerful detection capabilities that are associated with AppLocker log aggregation and review. These techniques could be further expanded to better automate the alerting process and improve the performance of the scripts. Additional statistical analysis could

also be implemented to automatically detect anomalous activity within AppLocker logs for an individual system or group of systems.

With a proper understanding of AppLocker's strengths and weaknesses, organizations can build low-cost, effective application whitelisting policies that provide significant protection on Windows systems. If additional detective and preventive measures are taken to cover weaknesses in the AppLocker policies on systems, an attacker's task of compromising a system and maintaining persistence becomes significantly more difficult. As a result, if no budget for commercial options is readily available, AppLocker can still be implemented with reasonable effort to implement the first of the quick wins in the Critical Security Controls.

References

Beuhring, A., & Salous, K. (2014, January 1). APT Detection with Whitelisting and Log Monitoring. Retrieved December 2, 2014, from https://dl.mandiant.com/EE/library/MIRcon2014/MIRcon_2014_IR_Track_APT_Detection.pdf

Critical Security Controls for Effective Cyber Defense. (n.d.). Retrieved December 1, 2014, from https://www.sans.org/critical-security-controls/

Jönsson, P., & Iveson, S. (2014). Security. In *F5 Networks Application Delivery Fundamentals Study Guide* (1st ed., pp. 168-170). Philip Jönsson & Steven Iveson.

Kazanciyan, R., & Hastings, M. (2014, August 7). Investigating PowerShell Attacks. Retrieved December 2, 2014, from https://www.fireeye.com/content/dam/legacy/resources/pdfs/fireeye-lazanciyan-investigating-powershell-attacks.pdf

Kennedy, D., & Kelley, J. (2010, January 1). PowerShell - It's Time to Own. Retrieved December 2, 2014, from http://media.blackhat.com/bh-us-10/presentations/Kennedy_Kelly/BlackHat-USA-2010-Kennedy-Kelly-PowerShell-Its-Time-To-Own-slides

Microsoft. (2012, June 21). AppLocker Settings. Retrieved January 29, 2015, from https://technet.microsoft.com/en-us/library/ee844171.aspx

Microsoft. (2012, June 27). AppLocker Step-by-Step Guide. Retrieved December 2, 2015, from https://technet.microsoft.com/en-us/library/dd723686(v=ws.10).aspx

Microsoft. (2012, October 18). AppLocker Overview. Retrieved December 2, 2014, from https://technet.microsoft.com/en-us/library/hh831440.aspx

Microsoft. (2012, October 18). Requirements to Use AppLocker. Retrieved January 29, 2015, from https://technet.microsoft.com/en-us/library/ee424382(v=ws.10).aspx

Microsoft. (2014, June 5). Plan security settings for VBA macros for Office 2013. Retrieved December 2, 2014, from https://technet.microsoft.com/en-us/library/ee857085(v=office.15).aspx

Microsoft. (n.d.). AppLocker Cmdlets in Windows PowerShell. Retrieved December 2, 2014, from https://technet.microsoft.com/en-us/library/ee460962.aspx

Microsoft. (n.d.). Understanding AppLocker Rules. Retrieved December 2, 2014, from https://technet.microsoft.com/en-us/library/dd759068.aspx

Microsoft. (n.d.). You can circumvent AppLocker rules by using an Office macro on a

computer that is running Windows 7 or Windows Server 2008 R2. Retrieved December 2, 2014, from http://support.microsoft.com/kb/2532445

Milunski, J., Dereszowski, A., & Cox, R. (2012, January 1). Reverse Engineering Malware and Mitigation Techniques. Retrieved December 2, 2014, from video.ch9.ms/teched/2012/eu/SIA404.pptx

Mount Knowledge. (2011, January 28). Bypassing Windows AppLocker using VB script in Word and Excel. Retrieved December 2, 2014, from http://www.mountknowledge.nl/2011/01/28/bypassing-windows-applocker-using-vb-script-in-word-and-excel/

Olsen, P. (2014, January 23). Do not fumble the lateral movement. Retrieved December 2, 2014, from http://sysforensics.org/2014/01/lateral-movement.html

Oracle. (2014, August 1). Java Platform, Standard Edition: Usage Tracker Overview. Retrieved December 2, 2014, from http://docs.oracle.com/javacomponents/usage-tracker/overview/

Stevens, D. (2011, January 23). Circumventing SRP and AppLocker, By Design. Retrieved December 2, 2014, from http://blog.didierstevens.com/2011/01/24/circumventing-srp-and-applocker-by-design/

Appendix A - PowerShell Script for Audit Mode Analysis (AppLocker_Audit_Collection.ps1)

```
import-module ActiveDirectory
########## Configuration ##########

# Change $AuditMode to "$false" if you want to search for Enforcement events
where AppLocker blocked files from running.
# Leaving it at $true will search for events in which AppLocker would have
blocked files if it were in enforcement mode.
$AuditMode = $true

# Change startDate and endDate to configure the timespan for log collection
$startDate = [datetime]"1/1/2015 8:00:00 AM"
$endDate = Get-Date

# Modify $pcList variable to adjust what systems are being queried.  The below
line searches for all of the Computer
# objects in the "gseclab.com" domain, so this should be modified to search
within your AD structure. It can also be
# populated with a comma separated list of PC names instead of searching AD.
$pcList += @((Get-ADComputer -Filter * -SearchBase "DC=gseclab,DC=com" -
Property *  | Select-Object Name).name)

###################################
Function Get-AsyncEvent{
#Get-AsyncEvent from Jason Walker at https://gallery.technet.microsoft.com/Get-
AsyncEvent-multi-Get-ed5c68c3
[CmdletBinding()]
 param(

[Parameter(Mandatory=$false,ValueFromPipeline=$true,ValueFromPipelineByProperty
Name=$true,Position=0)]
    [Alias("CN","Computer")]
    [String[]]$ComputerName = $env:COMPUTERNAME,

    [Parameter(Mandatory=$true,Position=1)]
    [HashTable]$FilterHashtable,

    [Parameter(Mandatory=$false)]
    [Int64]$MaxEvents = 25,

    [Parameter(Mandatory=$false)]
    [Int]$ThrottleLimit = 20,

    [Parameter(Mandatory=$false)]
    [System.Management.Automation.PSCredential]
    [System.Management.Automation.Credential()]$Credential =
[System.Management.Automation.PSCredential]::Empty
    )

Begin
{
    $Start = Get-Date
    $RunspaceCollection = @()
    $ProgressCounter = 0

    Write-Verbose "Removing ComputerName and ThrottleLimit from
PSBoundParameters because they will be splatted"
    $null = $PSBoundParameters.Remove('ComputerName')
    $null = $PSBoundParameters.Remove('ThrottleLimit')

    If(-not $PSBoundParameters.MaxEvents){
        #Write-Verbose "Maxevents not specified. Adding default value to
PSBoundParameters"
        #$PSBoundParameters.Add('MaxEvents',$MaxEvents)
    }
    $PSBoundParameters.GetEnumerator() | ForEach-Object
{"PSBoundParameters:$($_.Key) = $($_.Value)"} | Write-Verbose
```

Josh Johnson, jcjohnson34@gmail.com

```
    $RunspacePool =
[RunspaceFactory]::CreateRunspacePool(1,$ThrottleLimit,$host)
    $RunspacePool.Open()
    $Parameters = $PSBoundParameters

    $Scriptblock = {
        Param(
        $Computer,$Parameters
        )
        Try{
            Get-WinEvent -ComputerName $Computer @Parameters -ErrorAction Stop
        }
        Catch{
            Write-Warning $_
        }
     }
 }
 Process
{
    Foreach($Computer in $ComputerName)
    {
        #Create a PowerShell object to run and add the command.
        $Powershell =
[PowerShell]::Create().AddScript($ScriptBlock).AddArgument($Computer).AddArgume
nt($Parameters)
        #Specify runspace to use
        $Powershell.RunspacePool = $RunspacePool
        #Create Runspace collection
        [Collections.Arraylist]$RunspaceCollection += New-Object -TypeName
PSObject -Property @{
            Runspace   = $PowerShell.BeginInvoke()
            PowerShell = $PowerShell
            Computer   = $Computer
        }
    }
    While($RunspaceCollection){
        Foreach($Runspace in $RunspaceCollection.ToArray())
        {
            If($Runspace.Runspace.IsCompleted)
            {
                $ProgressCounter++
                Write-Progress -Activity "Collecting Event Logs" -Status "Last
completed: $($Runspace.Computer)" -PercentComplete
(($ProgressCounter/$ComputerName.Count)*100)
                #Display results
                $Runspace.PowerShell.EndInvoke($Runspace.Runspace)
                $Runspace.PowerShell.Dispose()
                Write-Verbose "Removing $($Runspace.Computer)"
                $RunspaceCollection.Remove($Runspace)
            }
        }
    }
}
End{
    $RunspacePool.Close()
    $TimeSpan = New-TimeSpan -Start $Start -End (Get-Date)
    Write-Verbose "Script elapsed time : $($TimeSpan.Hours) hours
$($TimeSpan.Minutes) minutes and $($TimeSpan.Seconds) seconds"
    }
}

if($AuditMode){
       $exeID = 8003
       $msiID = 8006
}
else{
       $exeID = 8004
       $msiID = 8007

}
$myEXEEvents = @()
```

Josh Johnson, jcjohnson34@gmail.com

```
$myEXEEvents += Get-AsyncEvent -ComputerName $pcList -FilterHashtable
@{logname="Microsoft-Windows-AppLocker/EXE and DLL"; ID=$exeID;
starttime=$startDate; endtime=$endDate} | select-Object MachineName,
TimeCreated, Message

if($myEXEEvents.count -gt 0){
        $myEXEEvents | export-csv "AppLocker_EXE_Events.csv"
             write-host "EXE events written to AppLocker_EXE_Events.csv`n"
}
else{
        write-host "No EXE Events found.`n"
}

$myMSIEvents = @()
$myMSIEvents += Get-AsyncEvent -ComputerName $pcList -FilterHashtable
@{logname="Microsoft-Windows-AppLocker/MSI and Script"; ID=$msiID;
starttime=$startDate; endtime=$endDate} | select-Object MachineName,
TimeCreated, Message
if($myMSIEvents.count -gt 0){
    $myMSIEvents | export-csv "AppLocker_ScriptInstaller_Events.csv"
      write-host "MSI Events written to AppLocker_ScriptInstaller_Events.csv`n"
}
else{
     Write-Host "No Script/MSI Events found.`n"
}
```

Josh Johnson, jcjohnson34@gmail.com

Appendix B - PowerShell Script to Monitor Whitelisted Files (AppLocker_Log_Analysis.ps1)

```
<#

.SYNOPSIS
This script is used to collect AppLocker logs from endpoints, storing them in
local CSV files and alerting on potentially dangerous execution of applications
that could indicate post-exploitation actions.  Optionally, additional switches
can be used to perform analysis on the logs.

.DESCRIPTION
Additional switches can be used to perform analysis on the logs.

-nologs                Does not collect logs from endpoints.  This is only
                          useful when -checkpaths is used as well.


-getstats           Summarizes the collected AppLocker logs, collecting the
                          hash as well as statistical information about each
                          executable that was run. NOTE: This can take quite
                          some time to run if a large number of machines are
                          being analyzed.

-checkpaths     Analyzes directories in potentially weak path rules to
                          ensure files are not added/modified.  A file
                          `"AppLocker_Monitored_Paths`"is created that lists
all
                          of the files in each path as well as their SHA256
Hash.

-checkjava          Correlates AppLocker events where Java.exe is run with
                          Java_usagetracker files to provide insight on which
                          Java files were run.  NOTE: Java Usage Tracker must
                          be enabled and logging to users' home directories
for
                          this switch to work.


.EXAMPLE
./AppLocker_Log_Analysis.ps1

Retreives logs for the endpoints defined in the "$pcList" variable. Alerts on
any potentially dangerous executables being run on these endpoints. Alerts are
written to AppLocker_Alerts.csv.  If there are alerts on a PC where users would
not normally run these executables, consider checking the machine for
additional signs of compromise.

.EXAMPLE
./AppLocker_Log_Analysis.ps1 -nologs -checkpaths

Retreives a list of files in the directories defined in the $pathsToMonitor
variable.  Alerts if new files are added to the monitored directories or if
existing files are modified.  This option does not pull logs from endpoints.

.EXAMPLE
./AppLocker_Log_Analysis.ps1 -getstats

Retreives logs from the endpoints defined in the "$pcList" variable.
Additionally calculates the SHA-256 hash of each executable, the number of
times each executable was run as well as the first seen and last seen run times
for each executable.

.EXAMPLE
./AppLocker_Log_Analysis.ps1 -checkjava

Retreives logs for the endpoints defined in the "$pcList" variable.
Additionally, if Java.exe is observed in the logs, the script will attempt to
scrape .java_usagetracker files to determine which Java executable was run and
with which arguments.  Results are written to AppLocker_Java_Logs.csv.
```

Josh Johnson, jcjohnson34@gmail.com

```
.LINK
http://www.sans.org/reading-room/

#>
param(
       [switch]$nologs,
       [switch]$getstats,
       [switch]$checkpaths,
       [switch]$checkjava
)

import-module ActiveDirectory
########## Configuration ##########

# Change startDate and endDate to configure the timespan for log collection
$startDate = [datetime]"1/1/2015 8:00:00 AM"
$endDate = Get-Date
# Modify $pcList variable to adjust what systems are being queried.  The below
line searches for all of the Computer
# objects in the "gseclab.com" domain, so this should be modified to search
within your AD structure. It can also be
# populated with a comma separated list of PC names instead of searching AD.
$pcList += @((Get-ADComputer -Filter * -SearchBase "DC=gseclab,DC=com" -
Property *  | Select-Object Name).name)

# Modify $pathsToMonitor to indicate the paths which should be inventoried and
then monitored for changes/additions.
# This can be a comma separated list of paths or a single directory.
$pathsToMonitor=@("\\dc-01\netlogon")

###################################
Function Get-AsyncEvent{
   #Get-AsyncEvent from Jason Walker at
https://gallery.technet.microsoft.com/Get-AsyncEvent-multi-Get-ed5c68c3
       [CmdletBinding()]

       param(

       [Parameter(Mandatory=$false,ValueFromPipeline=$true,ValueFromPipelineByPr
opertyName=$true,Position=0)]
              [Alias("CN","Computer")]
              [String[]]$ComputerName = $env:COMPUTERNAME,

              [Parameter(Mandatory=$true,Position=1)]
              [HashTable]$FilterHashtable,

              [Parameter(Mandatory=$false)]
              [Int64]$MaxEvents = 25,

              [Parameter(Mandatory=$false)]
              [Int]$ThrottleLimit = 20,

              [Parameter(Mandatory=$false)]
              [System.Management.Automation.PSCredential]
              [System.Management.Automation.Credential()]$Credential =
[System.Management.Automation.PSCredential]::Empty
              )

       Begin
       {
              $Start = Get-Date
              $RunspaceCollection = @()
              $ProgressCounter = 0

              Write-Verbose "Removing ComputerName and ThrottleLimit from
PSBoundParameters because they will be splatted"
              $null = $PSBoundParameters.Remove('ComputerName')
              $null = $PSBoundParameters.Remove('ThrottleLimit')

              If(-not $PSBoundParameters.MaxEvents){
```

Josh Johnson, jcjohnson34@gmail.com

```
                        #Write-Verbose "Maxevents not specified. Adding default
value to PSBoundParameters"
                        #$PSBoundParameters.Add('MaxEvents',$MaxEvents)
                }

                $PSBoundParameters.GetEnumerator() | ForEach-Object
{"PSBoundParameters:$($_.Key) = $($_.Value)"} | Write-Verbose

                $RunspacePool =
[RunspaceFactory]::CreateRunspacePool(1,$ThrottleLimit,$host)
                $RunspacePool.Open()
                $Parameters = $PSBoundParameters

                $Scriptblock = {
                        Param(
                        $Computer,$Parameters
                        )

                        Try{
                                Get-WinEvent -ComputerName $Computer @Parameters -
ErrorAction Stop
                        }
                        Catch{
                                Write-Warning $_
                        }
                }
        }
        Process
        {
                Foreach($Computer in $ComputerName)
                {
                        #Create a PowerShell object to run and add the command.
                        $Powershell =
[PowerShell]::Create().AddScript($ScriptBlock).AddArgument($Computer).AddArgume
nt($Parameters)
                        #Specify runspace to use
                        $Powershell.RunspacePool = $RunspacePool
                        #Create Runspace collection
                        [Collections.Arraylist]$RunspaceCollection += New-Object -
TypeName PSObject -Property @{
                                Runspace   = $PowerShell.BeginInvoke()
                                PowerShell = $PowerShell
                                Computer   = $Computer
                        }
                }

                While($RunspaceCollection){
                        Foreach($Runspace in $RunspaceCollection.ToArray())
                        {
                                If($Runspace.Runspace.IsCompleted)
                                {
                                        $ProgressCounter++
                                        Write-Progress -Activity "Collecting Event
Logs" -Status "Last completed: $($Runspace.Computer)" -PercentComplete
(($ProgressCounter/$ComputerName.Count)*100)
                                        #Display results

        $Runspace.PowerShell.EndInvoke($Runspace.Runspace)
                                        $Runspace.PowerShell.Dispose()
                                        Write-Verbose "Removing $($Runspace.Computer)"
                                        $RunspaceCollection.Remove($Runspace)
                                }
                        }
                }
        }
        End{
                $RunspacePool.Close()
                $TimeSpan = New-TimeSpan -Start $Start -End (Get-Date)
                Write-Verbose "Script elapsed time : $($TimeSpan.Hours) hours
$($TimeSpan.Minutes) minutes and $($TimeSpan.Seconds) seconds"
        }
}
```

Josh Johnson, jcjohnson34@gmail.com

```
Function checkPaths{
	$filename = "AppLocker_Monitored_Paths.csv"

	$hashArr=@()
	foreach($path in $pathsToMonitor){
		$filesToHash = get-childitem $path -file -force -name
		foreach($file in $filesToHash){
			$filepath = $path + "\" + $file
			$applockerInfo = get-applockerfileinformation -Path
$filepath
			$filehash = $applockerInfo.hash.hashdatastring
			$hashObj = [pscustomobject]@{'Path' = $path; 'File' =
$file; 'Hash' = $filehash; }
			$hashArr += $hashObj
		}
	}
	if(test-path $filename){
		$loadedFiles = import-csv $filename
		$foundNew = 0
		foreach($newHash in $hashArr){
			$newPath = [String]$newHash.path
			$newFile = [String]$newHash.file
			$newHashUnique = [String]$newHash.hash
			$found = 0
			$counter = 0
			foreach($loadedFile in $loadedFiles){
				$loadedPath = [String]$loadedFile.path
				$loadedFile = [String]$loadedFile.file
				$loadedHash = [String]$loadedFiles[$counter].Hash
				$counter++
				if($found -eq 0){
					if(($loadedPath -eq $newPath) -and
($loadedFile -eq $newFile) -and ($loadedHash -eq $newHashUnique)){
						$found = 1
					}
				}
			}
			if($found -eq 0){
				#we didn't find the newhash object
				write-host 'ALERT: Changed or new file:'$newHash.File
"Located in:"$newHash.path
				$foundNew =1
			}
		}
		if($foundNew -eq 1){
			$save = Read-Host 'Do you want to add these changes to the
monitored list? (y/n)'
			if($save -eq 'y'){
				$hashArr |export-csv $filename
				write-host "Wrote changes to "$filename
			}
		}
		else{
			write-host "No new files or changes in monitored paths."
		}
	}
	else{
		write-host "First time monitoring paths. Wrote results to
"$filename
		$hashArr | export-csv $filename
	}
}
Function getLogs{

	$myEXEEvents += Get-AsyncEvent -ComputerName $pcList -FilterHashtable
@{logname="Microsoft-Windows-AppLocker/EXE and DLL"; ID=8002;
starttime=$startDate; endtime=$endDate} | select-Object MachineName, @{Label =
"TimeCreated";Expression = {Get-Date $_.TimeCreated -Format "M/d/yyyy
HH:mm:ss"}}, Message, UserId
	if($myEXEEvents.count -gt 0){
	  $myEXEEvents | export-csv $exename
		write-host "EXE events written to $exename`n"
```

Josh Johnson, jcjohnson34@gmail.com

```
    }
    else{
        write-host "No EXE Events found.`n"
    }


        $myMSIEvents += Get-AsyncEvent -ComputerName $pcList -FilterHashtable
@{logname="Microsoft-Windows-AppLocker/MSI and Script"; ID=8005;
starttime=$startDate; endtime=$endDate} | select-Object MachineName, @{Label =
"TimeCreated";Expression = {Get-Date $_.TimeCreated -Format "M/d/yyyy
HH:mm:ss"}}, Message, UserId
        if($myMSIEvents.count -gt 0){
          $myMSIEvents | export-csv $msiname
               write-host "Script/MSI Events written to $msiname`n"
        }
    else{
         Write-Host "No Script/MSI Events found.`n"
        }

        $unique = import-csv $exename | Sort-Object Message -unique | select-
object MachineName,Message,TimeCreated

        <#Parse the csv for "blacklisted" apps that have been run.
        Sources for below data at: http://sysforensics.org/2014/01/lateral-
movement.html and
https://dl.mandiant.com/EE/library/MIRcon2014/MIRcon_2014_IR_Track_APT_Detectio
n.pdf

        - AT.EXE (scheduled jobs/tasks)
        - SCHTASKS.EXE (scheduled jobs/tasks)
        - CMD.EXE (Obviously common, but I included it anyway.)
        - NET.EXE (net view, etc.)
        - NET1.EXE (net use)
        - NETSTAT.EXE (netstat -ano)
        - REG.EXE (reg query and reg add)
        - SC.EXE (interact with services)
        - SYSTEMINFO.EXE (system profiling)
        - TASKKILL.EXE (kill running processes - taskkill /f /im <process_name>
or by PID.)
        - TASKLIST.EXE (tasklist /v)
        - POWERSHELL.EXE (interact with powershell)
        - NBTSTAT.EXE (profile)
        - XCOPY.EXE (copy files around)
        - NSLOOKUP.EXE (profile)
        - QUSER.EXE (profile)
        - PING.EXE (check connectivity)
        - FTP.EXE (download/upload)
        - BITSADMIN.EXE (download/upload)
        - ROUTE.EXE (adding persistent routes)
        - REGSVR32.EXE (services)
        - MAKECAB.EXE (compression before exfil)
        #>
        $blacklist =
"AT.EXE","SCHTASKS.EXE","CMD.EXE","NET.EXE","NET1.EXE","NETSTAT.EXE","REG.EXE",
"SC.EXE","SYSTEMINFO.EXE","TASKKILL.EXE","TASKLIST.EXE","POWERSHELL.EXE","NBSTA
T.EXE","XCOPY.EXE","NSLOOKUP.EXE","QUSER.EXE","PING.EXE","FTP.EXE","BITSADMIN.E
XE","ROUTE.EXE","REGSVR32.EXE","MAKECAB.EXE"

        $alertsCSV = @()
        foreach($file in $unique){
               foreach($badexe in $blacklist){
                      $exeWildcard = "*\"+$badexe+"*"
                      if($file.message -like $exeWildcard){
                             write-host "ALERT: Found " $badexe " on "
$file.machinename
                             $alertData = [pscustomobject]@{'MachineName' =
$file.machinename; 'FilePath' = $file.message -replace " was allowed to
run.","";     'TimeCreated' = $file.timecreated}
                             $alertsCSV += $alertData
                      }
               }
        }
```

Josh Johnson, jcjohnson34@gmail.com

```
        $alertsCSV | export-csv "AppLocker_Alerts.csv"
    write-host "`n"
}
Function checkJava{
        $fullCSV = import-csv $exename
        $javaInstances = @($fullCSV | where {$_.message -like "*JAVA.EXE*"})
    $javaInstances += @($fullCSV | where {$_.message -like
"*JP2LAUNCHER.EXE*"})
    $machineNames = $javaInstances | Group-Object machinename

        if($javaInstances.count -ne 0){
          $found = 0
                $AppLockerJavaCSV=@()
                foreach($machine in $machineNames){
              $users = $machineNames.Group | Group-Object UserID

                foreach($user in $users){
                    $sid = [System.Security.Principal.SecurityIdentifier]$user.name
                            $username =
$sid.Translate([System.Security.Principal.NTAccount]).value -creplace
'^[^\\]*\\',''
                            $UTPath = "\\" + $machine.Name +
"\c$\Users\"+$username+"\.java_usagetracker"
                if(test-path $UTPath){
                    $java_usagetracker = get-content $UTPath
                    $fDate = $user | ForEach-Object {$_.Group |Sort-Object
{$_.TimeCreated -as [datetime]} | Select -First 1}
                    $lDate = $user | ForEach-Object {$_.Group |Sort-Object
{$_.TimeCreated -as [datetime]} | Select -Last 1}

                    $possibleStartTime = [DateTime]$fDate.TimeCreated
                    $possibleEndTime = [DateTime] $lDate.TimeCreated

                    $tz =
[Regex]::Replace([System.TimeZoneInfo]::Local.StandardName, '([A-Z])\w+\s*',
'$1')#from   https://getatip.wordpress.com/category/powershell
                    $formattedStartTime = '{0:ddd MMM dd HH:mm:ss yyyy}' -f
[DateTime]$possibleStartTime.AddSeconds(-2)
                    $yr = $formattedStartTime -match "([0-9]{4})"
                    $DateMinusYear = $formattedStartTime -replace "\s([0-
9]{4})$", ""
                                $finalStartDate = $DateMinusYear +" " + $tz +
" " +$Matches[0]

                    $formattedEndTime = '{0:ddd MMM dd HH:mm:ss yyyy}' -f
[DateTime]$possibleEndTime.AddSeconds(2)
                    $yr = $formattedEndTime -match "([0-9]{4})"
                    $DateMinusYear = $formattedEndTime -replace "\s([0-
9]{4})$", ""
                    $finalEndDate = $DateMinusYear +" " + $tz + " "
+$Matches[0]

                    foreach($line in $java_usagetracker){
                        $splitLine = $line.Split(",")
                        $dateSection = $splitLine[1].replace("`"","")

                        if(($dateSection -le $finalEndDate) -and ($dateSection
-ge $finalStartDate)){
                            $appName = $splitLine[3]
                            $appArgs = $splitLine[12]
                            $classPath = $splitLine[13]
                            $javaData = [pscustomobject]@{'MachineName' =
$machine.Name; 'TimeRun' = $dateSection; 'JavaApp' = $appName; 'AppArgs' =
$appArgs; 'ClassPath' = $classPath}
                                        $AppLockerJavaCSV += $javaData
                                        $found++
                        }
                    }
                }
                else{
                    write-host "Couldn't find - "$UTPath". Unable to search for
Java app execution."
```

Josh Johnson, jcjohnson34@gmail.com

```
                        }
                }
                  }
                  $AppLockerJavaCSV | export-csv "AppLocker_Java_Logs.csv"
          if($found -gt 0){
              write-host "ALERT: Found $found instances of Java running.  Check
AppLocker_Java_Logs.csv for details.`n"
            }
    }
}
Function getStats{

       $fullCSV = import-csv $exename
       $unique = $fullCSV | Sort-Object Message -unique | select-object
MachineName,Message,TimeCreated
       if(Test-Path "Applocker_Statistics.csv"){
              $oldStats = import-csv "AppLocker_Statistics.csv"
              $oldStats_copy = $oldStats
              $newStatsArr = @()
              $foundObj
              foreach($currentApp in $unique){
                     $found = 0

                     foreach($oldFile in $oldStats_copy){
                            $currentMessage = $currentApp.Message -replace " was
allowed to run.",""
                            if($oldFile.filepath -eq $currentMessage){
                                   $found = 1
                                   $foundObj = $oldFile
                            }
                     }
                     if($found -eq 0){
                            $matches = 0
                            $newAppObj
                            $filepath = "\\" + $currentApp.machinename + "\"+
$currentApp.message -replace "%OSDRIVE%","c$" -replace "%WINDIR%","c$\Windows" -
replace "%SYSTEM32%","c$\Windows\System32" -replace
"%PROGRAMFILES%","c$\Program Files (x86)" -replace " was allowed to run.",""
                            $fullpath64 = "\\" + $file.machinename + "\"+
$file.message -replace "%OSDRIVE%","c$" -replace "%WINDIR%","c$\Windows" -
replace "%SYSTEM32%","c$\Windows\System32" -replace
"%PROGRAMFILES%","c$\Program Files" -replace " was allowed to run.",""
                            $matches = $fullCSV | group-object message | where
{$_.name -like $file.message}
                            $filehash = "null"
                            if(test-path $filepath){
                                   $applockerInfo = get-applockerfileinformation
-Path $filepath
                                   $filehash = $applockerInfo.hash.hashdatastring
                            }
                            elseif(test-path ($fullpath64)){
                            $applockerInfo = get-applockerfileinformation -Path
$fullpath64
                            $filehash = $applockerInfo.hash.hashdatastring
                            }
                            else{
                                   $filehash = "file not available"
                            }

                            $fDate = $fullCSV | group-object message | foreach-
object {$_.Group| sort-object {$_.TimeCreated -as [datetime]} | select -First
1}     | where {$_.message -like $currentApp.message}
                            $lDate = $fullCSV | group-object message | foreach-
object {$_.Group| sort-object {$_.TimeCreated -as [datetime]} | select -Last 1}
       |where {$_.message -like $currentApp.message}

                            $newData = [pscustomobject]@{'MachineName' =
$currentApp.machinename; 'FilePath' = $currentApp.message -replace " was
allowed to run.",""; 'Hash' = $filehash; 'NumRuns' = $matches.count;
'FirstSeen' = $fDate.TimeCreated; 'LastSeen' = $lDate.TimeCreated}
                            $newStatsArr += $newData
                     }
```

Josh Johnson, jcjohnson34@gmail.com

```
                        else{
                                $oldLastSeenTime = $foundObj.LastSeen
                                $matches = @($fullCSV | where {$_.message -like
$currentApp.message}| where {[datetime]$_.TimeCreated -gt
[datetime]$oldLastSeenTime})
                                $lDate = $fullCSV | group-object message | foreach-
object {$_.Group| sort-object {$_.TimeCreated -as [datetime]} | select -Last 1}
                        |where {$_.message -like $currentApp.message}
                                $newMatches = [int]$foundObj.numRuns + $matches.count
                                $foundObj.numRuns = $newMatches
                                $foundObj.LastSeen = $lDate.TimeCreated
                                $newStatsArr += $foundObj
                        }
                }
                $newStatsArr | export-csv "AppLocker_Statistics.csv"
                write-host "Statistics written to AppLocker_Statistics.csv"
        }
        else{
                $statsCSV= @()
                foreach($file in $unique){
                        $matches = 0
                        $fullpath = "\\" + $file.machinename + "\"+ $file.message -
replace "%OSDRIVE%","c$" -replace "%WINDIR%","c$\Windows" -replace
"%SYSTEM32%","c$\Windows\System32" -replace "%PROGRAMFILES%","c$\Program Files
(x86)" -replace " was allowed to run.",""
                        $fullpath64 = "\\" + $file.machinename + "\"+ $file.message
-replace "%OSDRIVE%","c$" -replace "%WINDIR%","c$\Windows" -replace
"%SYSTEM32%","c$\Windows\System32" -replace "%PROGRAMFILES%","c$\Program Files"
-replace " was allowed to run.",""
                        $matches = $fullCSV | group-object message | where {$_.name
-like $file.message}
                        $filehash = "null"
                        if(test-path $fullpath){
                                $applockerInfo = get-applockerfileinformation -Path
$fullpath
                                $filehash = $applockerInfo.hash.hashdatastring
                        }
                        elseif(test-path ($fullpath64)){
                                $applockerInfo = get-applockerfileinformation -Path
$fullpath64
                                $filehash = $applockerInfo.hash.hashdatastring
                        }
                        else{
                                $filehash = "file not available"
                        }
                        $fDate = $fullCSV | group-object message | foreach-object
{$_.Group| sort-object {$_.TimeCreated -as [datetime]} | select -First 1}|
where {$_.message -like $file.message}
                        $lDate = $fullCSV | group-object message | foreach-object
{$_.Group| sort-object {$_.TimeCreated -as [datetime]} | select -Last 1} |where
{$_.message -like $file.message}
                        $newData = [pscustomobject]@{'MachineName' =
$file.machinename; 'FilePath' = $file.message -replace " was allowed to
run.",""; 'Hash' = $filehash; 'NumRuns' = $matches.count; 'FirstSeen' =
$fDate.TimeCreated; 'LastSeen' = $lDate.TimeCreated}
                        $statsCSV += $newData
                }
                $statsCSV | export-csv "AppLocker_Statistics.csv"
                write-host "Statistics written to AppLocker_Statistics.csv"
        }
}
$today = get-date -format Mddyy
$date = get-date
$msiname = "AppLocker_"+$today+"_"+$date.Hour+"_msi_events.csv"
$exename = "AppLocker_"+$today+"_"+$date.Hour+"_exe_events.csv"

if(!$nologs){
        getLogs
}
if($getstats){
        getStats
}
```

Josh Johnson, jcjohnson34@gmail.com

```
if($checkpaths){
        checkPaths
}
if($checkjava){
        checkJava
}
```