Global Information Assurance Certification Paper

Supplementing Windows Audit, Alerting, and Remediation with PowerShell

GIAC (GCWN) Gold Certification

Author: Daniel Owen, ggold@danielowen.com
Advisor: Adam Kliarsky
Accepted: October 20, 2017

Abstract

This paper outlines the use of PowerShell to supplement audit, alerting, and remediation platform for Windows environments. This answers the question of why use PowerShell for these purposes. Several examples of using PowerShell are included to start the thought process on why PowerShell should be the security multi-tool of first resort. Coverage includes how to implement these checks in a secure, automatable way. To demonstrate the concepts discussed, small code segments are included. The intent of the included code segments is to inspire the reader's creativity and create a desire to use PowerShell to address challenges in their environment. Finally, a short section includes resources for code examples and learning tools. While some knowledge of PowerShell will aid the reader, the intended audience of this paper is the PowerShell novice.

1. Introduction

Understanding what exists in the protected environment is the beginning of any successful defensive security program, and internal auditing is a path toward gaining that understanding. Audits further allows the testing of assumptions about the existing security posture and comparison to the expected or documented standard (Christopher, 2010). Studies have shown that implementing the first five CIS Controls, from the Center for Internet Security, prevent ~85% of attacks seen in the wild. All five of these controls require an audit component to find success or prove their implementation. CIS further advises an audit as a foundational step toward developing a plan for implementing the CIS Critical Controls. (Center for Internet Security, n.d.) While the emphasis of this paper is practical security improvements, there is overlap with third-party audit controls such as PCI or HIPAA. As such, references that illuminate their relationship are also included.

As an extension of point-in-time auditing, it is critical to detect and quickly remediate changes to standard secure configurations. It is not realistic for an organization to expect to be able to do this manually. Through Continuous Risk Treatment (CRT), we can automate the process of detecting, altering, and in some cases remediating configuration skew (Steffan & Sandage, 2017). To this goal, the paper discusses the process of taking a script initially used for point in time audits and automating it to provide continual coverage.

A force multiplier allows an increased output from a given input (Kaufman, 2012, p. 158-159). A lever, such as a crowbar, is a simple physical world example of a force multiplier. Spending a relatively small amount of time using PowerShell as a force multiplier generates dividends many times over in time savings and better outcomes.

This paper explores methods of using PowerShell to supplement existing auditing tools and for using the data to automate alerting and remediation efforts. Through this process, system defenses are significantly improved.

One goal of this paper is to introduce the security practitioner to a sampling of ways to use PowerShell in a defensive manner; however, the larger goal is to inspire the

reader to expand upon the examples in this paper and use PowerShell to fill gaps in their own security infrastructure.

2. Why PowerShell?

2.1 Supplemental tool

PowerShell should not be the only tool used for auditing, alerting, or remediation but will add to the processional's toolbox.

PowerShell is one of the more versatile tools currently available while still retaining an approachable learning curve. In addition, a number of security professionals are already using PowerShell so it is easy to learn and borrow from the community.

In addition, many third-party tools can use PowerShell to extend their functionality. As an example, Nessus can use PowerShell for compliance auditing. Paul Asadoorian demonstrates this in a number of examples for a Tenable blogpost showing Nessus rules written using PowerShell (Asadoorian, 2012).

In the case of using PowerShell to extend the functionality of Nessus, Tenable has provided a set of PowerShell cmdlets to integrate directly with the Nessus API (Tenable, 2015). At a more basic level, PowerShell can be used as a data transformation tool using CSV exports from within Nessus as was demonstrated in a SANS Internet Storm Center Handler's Diary by Rob VandenBrink (VandenBrink, n.d.). By combining these two approaches, relatively complex automation, can be achieved.

PowerShell can be used both as a standalone tool or to fill in holes where existing tools are incomplete.

2.2 Flexibility

PowerShell is built on the .NET Framework (Microsoft, n.d.a). This presents great flexibility in that PowerShell scripts have access similar to any other .NET language. Cmdlets are the building blocks of PowerShell scripts. They use a basic verb-noun naming convention and accept parameters to control their usage. As an example, Get-ADUser is the cmdlet used to query Active Directory to retrieve user objects.

Additional cmdlets that expose more functionality in the underlying .NET Framework are included in each new version of PowerShell. It is also possible to call .NET Framework classes directly (Wilson, 2010).

There are 1285 cmdlets in PowerShell 5 (Wilson, 2015). With this wide array of cmdlets, a process that cannot be audited or automated exclusively with PowerShell cmdlets is a rare challenge.

2.3 Part of the Operating System

PowerShell was first released as an optional feature of Windows 2008 (Vanover, 2009). For later versions of Windows, it became a standard part of the install. PowerShell is now part of the operating system for all supported versions of Windows and does not require any additional software to be loaded.

This is an advantage in that other scripting languages require additional interpreters. Installing and maintaining interpreters outside the standard Microsoft patching cycle adds additional management overhead, complexity, and cost, while expanding the attack surface. Bruce Schneier summed this issue up succinctly in *Secrets and Lie* when he said "Simply put, complexity is the worst enemy of security. As systems get more complex, they necessarily get less secure" (Schneier, 2015, p.3).

2.4 The future of Windows administration

The future of Windows Server administration has less to do with the Graphical User Interface (GUI) today than it did prior to the release of PowerShell. This becomes more obvious with each new Windows release. Server Core for Windows 2008 is Microsoft's first attempt at a server operating system without a GUI. The stated goal for Server Core is that it is a lighter weight installation requiring less server resources, less management, and a smaller attack surface. The central concept behind Server Core management is that the system is primarily managed using PowerShell or remote administration tools (Microsoft, n.d.b).

With each subsequent Windows Server release, Microsoft has evolved the Server Core option. In Windows Server 2016, Microsoft has taken the minimalist operating system even further with Nano. For heavily virtualized and cloud environments Nano is

an even lighter operating system (Ferrill, 2016). Due to Nano Server's minimalist nature many tools that Windows administrators have become accustomed to do not work. This includes many Microsoft standbys such as Group Policy and System Center Configuration Manager. Even the version of PowerShell in Nano has limitations (Poggemeyer & Jaimeo, 2017). This is all to say that for Nano Server, custom scripting may be the only option for management and automation, at least in the short term.

Microsoft is trying to change the administration of Windows servers. Simply stated, PowerShell is the future of Windows administration and automation. Furthermore, the speed at which a competent scripter can complete and automate tasks relative to the repeated time cost of someone manually repeating tasks is significant. Between these two truths, PowerShell is the way of Windows administration going forward. The only remaining question is how long, not if, those who refuse to learn PowerShell can survive in their profession. While this may affect Windows administrators first, security professionals should not expect any less radical a change.

2.5 Other Scripting Languages

There are a number of other scripting languages that can be used for development on Windows. These include Visual Basic, batch scripts, Python, Perl, and Bash. For the reasons outlined above, this paper concentrates on PowerShell as the preferred language for scripting on Windows. Development in other scripting languages can use many of these same concepts, but other languages may be more limited in functionality. Carefully consider the significant advantages to using PowerShell for Windows automation before making a decision to use an alternative development language.

3. Uses of PowerShell

3.1 Administrative Group Members

There are a number of highly privileged groups in Active Directory that are critical to its operation. For this reason, they are tempting targets for attackers. For example, the Domain Admins group is described by Microsoft as having "complete control over all domain controllers and all directory content stored in the domain" and

"can modify the membership of all administrative accounts in the domain" (Microsoft, n.d.c). This makes the Domain Admins group a tempting target for attackers.

Following the concept of least privilege, which requires granting the user the minimum possible access so that they can still complete their tasks (Bishop, 2002), there should be as few people in privileged groups, such as the Domain Admins group, as possible. A Domain Admin's primary role is as a database administrator for Active Directory; therefore, it is not desirable to have users logged in as a Domain Admin for other tasks. For this reason, quick alerting for Domain Admins group changes is critical. This allows quick remediation when someone who is not authorized is added to the group, which, in turn, helps to protect the company from rogue or malicious acts as well as mistakenly overprovisioned users.

To accomplish this audit and remediation goal, users in the Domain Admins group are compared to a list of users who are authorized to be members of the group. To complete this in a script, there are two easy ways to define the authorized list. The list can be either a file that the script reads or a comparison group from Active Directory. Both approaches have advantages and disadvantages. The advantage of using an Active Directory "authorization" group is that it is easy to manage and document. This may be of limited use since the script protects against an adversary who already has rights in Active Directory. As an example, frequently, attackers clone an existing Domain Admin group member as a form of persistence. The authorizing group is also included in the copied account. For this reason and simplicity, this script uses a text file stored outside of Active Directory. Registry hives, SQL databases, or a number of other solutions would also be options for storing the authorized users.

```
$allowed = Import-Csv 'C:\scripts\get-admins\allowed.csv' #Path to file containing SIDs of users allowed to be a member of Domain Admins
$group = "Domain Admins" #Group being protected
$members = Get-ADGroupMember $group #Get members of the group we are protecting
if ($allowed.count -eq 0) {exit} #Failsafe to kill the script if the text file does not contain any data or is not read for some reason.
ForEach ($m in $members) { #Create a loop to look at each member of the $group
    $account = $m.sid.ToString() #Create a string variable that holds the SID for the member of $group to be compared. This simplifies code later.
    if ($allowed.sid -notcontains $account) { #Does the $allowed array contain the member of $group currently being compared? If not take action.
        Disable-ADAccount -Identity $account #Disable the AD account that was added to $group but is not in the $allowed file.
        Remove-ADGroupMember -Identity $group -Members $account -Confirm:$false #Remove $account from the $group.
        if ((Get-EventLog -LogName Application -Source "get-admins-script" -ErrorAction SilentlyContinue) -eq $null) {
            New-EventLog -LogName Application -Source 'get-admins-script' #Create a new event log type in the Application log if it does not exist.
        }
        Write-EventLog -LogName Application -Source "get-admins-script" -EntryType Warning -EventId 0
        -Message "$account was removed from $group as part of the get-admins security process." #Write an event to the local systems Security Event Log.
    }
}
```

Figure 1 - This sample script is included in Appendix A as get-admins.ps1

Daniel Owen, ggold@danielowen.com

The 16 lines of sample code above demonstrate a simple script for detecting unauthorized additions to the Domain Administrators group. Once the script detects an unauthorized user, it disables the Active Directory account and removes the user from the group. Finally, the script writes a log file to the system's Application log to allow for auditing. In order to keep the code as tight as possible, there is no white space, and there are additional steps that would typically be included for maintainability as well as additional remediation. A more complete version of this script is included in Appendix A as get-admins.ps1.

3.2 Identifying outdated local account password

Many security practitioners believe that regular password changes are part of good account hygiene. Further, some regulatory or best practices frameworks require changing of passwords on a regular basis. For example, PCI DSS Version 3.2 requirement 8.2.4 states that users must "change user passwords/passphrases at least once every 90 days" (PCI Security Standards Council, 2016). Item 1.1.2 of the CIS Microsoft Windows Server 2016 RTM (Release 1607) Benchmark requires "Ensure 'Maximum password age' is set to '60 or fewer days, but not 0'" (Center for Internet Security, 2017). A simple PowerShell script can test both these regulatory requirements as well as to assure that old passwords do not linger for longer than policy allows.

The script that follows looks at every enabled Active Directory account and creates a Comma Separated Values (CSV) file listing all accounts that are beyond the desired password age. The sample script is four lines with documentation and coding styled for ease of reading, but if one was to give up these niceties, this could become a one-line script. PowerShell makes this type of reporting simple.

```
1  $maxage = 90 #Maximum days since last password change
2  $date = (Get-Date).AddDays(-$maxage) #Date of oldest password that does not require a change
3  $users = Get-ADUser -Filter {PasswordLastSet -lt $date -and Enabled -eq $true} #Array that holds accounts that are past due for password change
4  $users | Export-Csv -NoTypeInformation c:\temp\oldpass.csv #Export all users who need to change their passwords to a CSV
```

Figure 2 – An extended version of this script is included in Appendix A as get-old password.ps1

This can further be expanded by adding a few more lines of code that create a Help Desk ticket or send an email to review accounts that are beyond their timeouts. Scheduling this to run as frequently as needed will close the loop on automation. This last

option is a good example of the flexibility of PowerShell which is to a great extent limited only by the user's imagination and skill unlike more traditional audit tools that are limited by their developer's imagination and desires. Appendix A houses this sample script as get-oldpassword.ps1.

3.3 Identify unauthorized email forwards

Email has made communication within and between companies easier, but with ease comes risk of unauthorized data leakage. One example of that unauthorized communication is the automated forwarding of company email to an account outside the company's control. This can happen when a user wants to use a non-company account and forwards email to their preferred platform or when an attacker does the same.

While the first may be a policy violation, the second is potentially more worrying as it allows an attacker to begin to profile a target user and organization. The attacker is able to see what typical organizational emails look like as well as collect insider communications about the company and their clients. Using the collected data, other attacks such as phishing, ransomware, or financial fraud can be launched (Cidon, 2017).

There are two approaches the defender can take when looking at this issue. Assuming forwarding is never allowed, fully automated attack detection and remediation can be achieved. This will follow a process similar to:

1) Detect creation of a forward.
2) Remove the email forward.
3) Disable the account or change the account password.
4) Contact the user to determine whether this was an external attack or a user action. Depending on the answer, other internal processes for security incidents or policy compliance remediation will follow.

Obviously, which, if any, of these steps receive automation in a given company are determined by different risk acceptance levels. In a company with a higher risk tolerance, a lower tolerance for impacting end users, or that allows users to auto-forward their email a more restrained approach must be taken. In this case, a regular list is created

so the user can be contacted outside of email to confirm they created the forward rule. The security incident process is initiated if the user did not create the rule.

Multiple single line scripts to identify accounts in Exchange or Office365 that have been configured to forward email can be found with a simple web search (Grogan, 2011). This code can be used as is, for a one-time test, or built upon as part of a larger script that generates automated alerting and remediation as previously described. Once again, only the imagination and needs of the script's author limit this script.

3.4 Identify Inactive Accounts

Much as it is important to disable unused services, it is also critical to disable or remove unneeded Active Directory accounts. This helps to identify users who may have left employment, service accounts for applications that are no longer in use, or other accounts that have become dormant.

This type of cleanup may also be a direct or indirect regulatory requirement. PCI DSS version 3.2 section 8.1.4 states unequivocally "remove/disable inactive user accounts within 90 days" (PCI Security Standards Council, 2016). HIPPA is somewhat less prescriptive with regulation 164.308(a)(3)(ii)(C) that states, "Implement procedures for terminating access to electronic protected health information when the employment of a workforce member ends" (Public Welfare, 2007).

Using a script that reviews Active Directory for last login date meets both the compliance and the regulatory requirements. While PCI DSS may be comfortable with an unused account being active for 90 days, a high security environment may require shorter timeouts. Canned tools may not provide the flexibility to change timeouts easily, but PowerShell allows the automation to meet the security needs of the company using the script rather than a third-party compliance standard. This could include, but are not limited to, such unique items as different policies based on Organizational Unit, group membership, manager, or time of year.

Since this is a common need for security professionals and systems administrators, searching popular code repositories identifies multiple scripts to automate this search. As an example, Microsoft partner TSO has created GetInactiveComputer.ps1

and made it available through TechNet (TSO, 2013). This simple script is both functional and a good starting point for a more complex script. On the other end of the complexity continuum, Luca Sturlese of 9to5IT has published PS-ManageInactiveAD to GitHub. Included in that package is Find-ADInactiveUsers.ps1 which has more included functionality and can be controlled using runtime variables (Sturlese, 2016). Scheduling this script can provide automated remediation efforts.

PowerShell skills are valuable, but this use case demonstrates leveraging PowerShell without writing the first line of code since suitable solutions were already available. Further, it is often far more efficient to use or modify existing free scripts rather than writing code.

3.5 Business Logic Errors

There are a number of business specific errors that can have a negative effect on security, but are unlikely to be addressed in a canned security audit solution. These are excellent opportunities for PowerShell to show its flexibility.

There are often Active Directory groups whose membership is based on some other attribute of the account. This could be as simple as all members of a group must be located in a specific geographic location or could be more complex and include all users who report directly or indirectly to a specific manager, have a specific title level, and are located in a specific location. As mentioned before, this is an esoteric set of requirements, but with PowerShell, it is trivial to produce a report of users who are out of compliance or even auto remediate the situation.

For the first example, a group made up of members of a specific geographic location, a sample script can easily be created. The first step to writing this script is to decide on the desired result. For this example, the script is looking for any Active Directory user in the "US_Associates" group whose account object does not show their location as United States. There might be other constraints such as excluding service accounts or users of a specific job title but for simplicity, this example is limited.

```powershell
$group = Get-ADGroupMember 'US_Associates' #Create a new variable with all members of the group we are interested in
$invalidmembers = @() #Initialize the array that will hold invalid users
ForEach ($g in $group) { #Check the country for every user who is in the group
    $tuser = Get-ADUser $g.sid -Properties country #Read the AD record for the current member being reviewed
    if ($tuser.country -notlike 'US') {$invalidmembers = $invalidmembers+$tuser} #If the user's country is not "US" add it to the variable
}
$invalidmembers | Export-Csv -NoTypeInformation c:\temp\invalidmembers.csv #Export the results to a CSV file
```

Figure 3 - This sample script is included in Appendix A as get-invalidgroupmembers.ps1

By adding the single line "Remove-ADGroupMember -Identity 'US_Associates' -Members $invalidmembers -Confirm:$false", this script can be taken a step further. This automates the removal of the users from the group.

It may be desirable when automating the removal of users from a group to send an audit log to a human for final review. This can be completed easily through email by using the Send-MailMessage cmdlet. For usability, this is wrapped in a conditional so an email is sent only if an object has been removed from the group.

```powershell
if ($invalidmembers.Count -gt 0){
    Remove-ADGroupMember -Identity 'US_Associates' -Members $invalidmembers -Confirm:$false
    Send-MailMessage -To "user1@example.com" -From "user2@example.com" -Subject "Users removed from US
}
```

Figure 4 – This is an extension to get-invalidgroupmembers.ps1

This demonstrates how a script can be extended after it is completed to add functionality or can be built in parts as new needs are discovered. The code for this script is reproduced in Appendix A as get-invalidgroupmembers.ps1.

Taking the second more esoteric example from before, this next example creates a list of users who report directly or indirectly to a specific manager, have a specific title level, and are located in a specific location.

PowerShell does not have a way to natively, recursively create a list of a manager's direct and indirect reports, nor is this a trivial scripting exercise. As has been mentioned previously, when a script is going to require significant code it is often best to leverage an existing script if possible. In this case, Microsoft MVP, François-Xavier Cat has already written the code to recurse management levels and posted it to TechNet as Get-ADDirectReports (Cat, 2015). As part of the custom script, it is imported. This is a good example of mixing existing code with new custom scripting in PowerShell. This script remains very small and simple by using existing code.

Daniel Owen, ggold@danielowen.com

```powershell
. "C:\scripts\Get-ADDirectReport.ps1" #Dot include script to do recursive lookup based on manager
$reports = Get-ADDirectReports -Identity user1 -Recurse #Find all direct and indirect reports for user1
$reportsfinal = @() #Initialize variable
foreach ($r in $reports) { #Create a list of users who will go on the report
    $user = Get-ADUser -Identity $r.SamAccountName -Properties country, title #Lookup user in AD
    #Final removal of users who do not meet additional requirements
    if (($user.country -like 'US')  -and ($user.title -like 'manager' )) {$reportsfinal = $reportsfinal+$user }
}
$reportsfinal | Export-Csv -NoTypeInformation C:\temp\selectreports.csv #Export to CSV
```

Figure 5 - This sample script is included in Appendix A as get-recursivemanager.ps1

There are hundreds of examples of very specific business logic issues for which an existing security application is unlikely to be found. As demonstrated, PowerShell can make quick work of those issues.

4. Automating Scripts

As has been alluded to earlier, running a PowerShell script on an as needed basis can be extremely useful but automating the script adds a new dimension to the force multiplier effect.

Automation opens up new opportunities such as regularly generated reports, generating automated tickets for remediation, and fully automated remediation efforts among other options. As has been said before, the developer's imagination and needs are the only limiting factor in the opportunities for automation.

There are a number of ways to automate scripts and this paper looks at two of the most common. Security concerns introduced by automation are also considered.

4.1 Scheduled Tasks

Using the Windows Task Scheduler to run a PowerShell script is likely the easiest and most common way to automate the running of a PowerShell script on a repeating basis.

Scheduled tasks can be created directly on the machine that runs the task or a Group Policy can be used to push the scheduled task to remote systems.

Creating tasks locally is useful for tasks that connect remotely to other systems to make changes or gather information. This can also be useful for scripts that only need to run on a single system.

Pushing a scheduled task to remote systems is beneficial for situations where a script needs to run on a regular basis without interaction with other systems. This can also be helpful for machines that are not always accessible across the network, such as laptops. Since Group Policy refreshes on a regulars basis, this has the added benefit of being self-correcting if the scheduled task is changed or removed.

Using third-party job schedulers can achieve the same goals. Other schedulers are out of scope for this paper but should be considered in environments that have standardized on alternate schedulers.

4.2 Automation Security

It is important to consider the security ramifications of running scheduled tasks using Task Scheduler. Many automated tasks require elevated rights so it is critical to consider the tradeoffs inherent in automating PowerShell scripts.

PowerShell code can be signed using a code-signing certificate issued by a trusted internal PKI server or a third-party certificate authority. By setting the Execution Policy to "AllSigned" PowerShell only executes signed code. Running "Set-ExecutionPolicy AllSigned –Force" from within PowerShell achieves this goal (Perez, 2013).

Setting the Execution Policy is not a guarantee that unsigned code will not execute. Scott Sutterland authored the article "15 Ways to Bypass the PowerShell Execution Policy" (Sutherland, 2014) in which he catalogs a number of ways achieve this. Even considering this, setting the execution policy does help reduce risk in reference to automation. Sutterland's work concentrates on a user who already has access to run their scripts as an elevated user. In the case of automation, the "AllSigned" setting is there to reduce the risk of a script on the server's hard drive being modified and then executed as a scheduled task.

Credentials may be stored on a system running scheduled tasks. These credentials can be recovered using tools such as mimikatz (Delpy, 2017). When an attacker gains admin level access to a system where scheduled tasks are running, the assumption should be that credentials have been be stolen. For this reason, it is best not to use a full user account. It is safer to run the scheduled task as Network Service or System since these services have no password to be stolen and are therefore more secure options. These

services, via their Computer object in Active Directory, can still be delegated access to remote resources. If a true user account must be used, Kerberos S4U (Services for User) can be used with constrained delegation, which limits the damage of lost credentials (Fossen, 2014, p. 56).

5. Additional Sources of PowerShell Scripts

There are a number of additional sources of complete code or scripts that can be built upon. As has been mentioned before, when a viable script exists it is a waste of resources to write that script again. Similarly, by looking at someone else's code and borrowing ideas even challenging scripts can be completed.

Microsoft encourages the use of GitHub repositories for shared coding projects (Harry, 2017) and many developers have followed this recommendation. This includes the PowerShell Team's own repository (Microsoft, n.d.d). Jason Fossen, the lead author for *SANS 505 - Securing Windows and PowerShell Automation*, also has a GitHub repository that included numerous security specific PowerShell scripts (Fossen, n.d.).

There are many PowerShell scripts as well as extensive guides and advice available as part of the Microsoft TechNet web pages. This includes the "Hey, Scripting Guy!" blog which covers topics in a start to finish learning style that can be helpful for both the novice and experienced scripter.

Numerous individuals and companies are publishing PowerShell scripts and advice as well as dozens of books on the topic. The challenge is easily too much information rather than a lack of coverage for the topic of PowerShell. Defensive PowerShell security is less well covered but the topic is gaining interest. These scripts often closely intersect with the more generalist systems administration topics found in non-specialist areas.

6. Conclusion

PowerShell works as a force multiplier to allow security professionals be more efficient in their efforts. For the IT audit professional, it allows for efficient collection of

data. For the analyst, it allows efficient review of data. For the defensive generalist it allows for automated alerting and remediation.

As has been said repeatedly, only the imagination and the needs of the user, limit the benefits of PowerShell for the security professional. This paper has covered a small number of use cases as examples, but only scratches the surface of what is possible.

The quickest way to prove the value of PowerShell in an environment is to pick a problem that current tools are not adequately identifying or that requires repeated manual intervention and spend a day with PowerShell. At the end of that time, the utility of PowerShell as leverage should be obvious. PowerShell quickly becomes the easy route to better Windows security efforts once a professional starts utilizing it and discovers its multiple uses.

Daniel Owen, ggold@danielowen.com

References

Asadoorian, P. (2012, April 26). Compliance Auditing with Microsoft PowerShell. Retrieved from http://www.tenable.com/blog/compliance-auditing-with-microsoft-powershell

Bishop, M. (2002). Design principles. In *Computer security: Art and science*. Retrieved from https://www.safaribooksonline.com/library/view/computer-security-art/0201440997/ch13.html#ch13lev2sec1

Cat, F. (2015, February 2). Get-ADDirectReports (Recursive DirectReports). Retrieved from https://gallery.technet.microsoft.com/scriptcenter/Get-ADDirectReport-962616c6

Center for Internet Security. (2017, March 31). *CIS Microsoft Windows Server benchmarks*. Retrieved from https://www.cisecurity.org/benchmark/microsoft_windows_server/

Center for Internet Security. (n.d.). *Guide to the first 5 CIS Controls (v6.1)*. Retrieved from https://learn.cisecurity.org/first-five-controls-download

Christopher, L. J. (2010). The principles of auditing. In *Network security auditing: The complete guide to auditing network security; measuring risk; and promoting compliance*. Retrieved from https://www.safaribooksonline.com/library/view/network-security-auditing/9781587059407/

Cidon, A. (2017, August 30). Threat spotlight: Office 365 account compromise — the new "insider threat". Retrieved September 2, 2017, from https://blog.barracuda.com/2017/08/30/threat-spotlight-office-365-account-compromise-the-new-insider-threat/

Delpy, B. (2017, January 3). Howto ~ scheduled tasks credentials. Retrieved September 3, 2017, from https://github.com/gentilkiwi/mimikatz/wiki/howto-~-scheduled-tasks-credentials

Ferrill, T. (2016, July 23). Windows Server 2016 feature highlight: Nano Server. Retrieved from http://www.tomsitpro.com/articles/windows-server-2016-nano-server,2-897.html

Daniel Owen, ggold@danielowen.com

Fossen, J. (2014). *Securing Windows and resisting malware - Server hardening & dynamic access control*. Bethesda, MD: The SANS Institute.

Fossen, J. (n.d.). Enclave Consulting LLC. Retrieved from https://github.com/EnclaveConsulting

Grogan, A. (2011, October 20). Get all users who have a forwarding address set. Retrieved from http://www.msexchange.org/kbase/ExchangeServerTips/ExchangeServer2010/ManagementAdministration/Getallusershohaveaforwardingaddressset.html

Harry, B. (2017, March 31). Shutting down CodePlex. Retrieved September 3, 2017, from https://blogs.msdn.microsoft.com/bharry/2017/03/31/shutting-down-codeplex/

Kaufman, J. (2012). Value delivery. In *The personal MBA: Master the art of business* (pp. 158-159). New York, NY: Penguin Group.

Microsoft. (n.d.a). *Windows PowerShell Overview*. Retrieved from https://technet.microsoft.com/en-us/library/cc732114(v=ws.10).aspx

Microsoft. (n.d.b). About server core. Retrieved from https://msdn.microsoft.com/en-us/library/ee391626(v=vs.85).aspx

Microsoft. (n.d.c). Securing Active Directory administrative groups and accounts. Retrieved from https://technet.microsoft.com/en-us/library/cc700835.aspx

Microsoft. (n.d.d). PowerShell team. Retrieved from https://github.com/powershell

PCI Security Standards Council. (2016, April). *Payment Card Industry (PCI) Data Security Standard requirements and security assessment procedures version 3.2*. Retrieved from https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2.pdf

Perez, C. (2013, March 5). PowerShell basics - Execution policy and code signing part 1. Retrieved from https://www.darkoperator.com/blog/2013/3/5/powershell-basics-execution-policy-part-1.html

Poggemeyer, L., & Jaimeo. (2017, February 28). Install Nano Server. Retrieved from https://docs.microsoft.com/en-us/windows-server/get-started/getting-started-with-nano-server

Public Welfare, 45 C.F.R. § 164.308 (2007).

Daniel Owen, ggold@danielowen.com

Schneier, B. (2015). Introduction. In *Secrets and lies: Digital security in a networked world; 15th anniversary edition* (p. 3). Retrieved from https://www.safaribooksonline.com/library/view/secrets-and-lies/9781119092438/

Steffan, T., & Sandage, T. (2017). Automating security operations. In *Automating security in the cloud: Modernizing governance through security design*. Retrieved from https://www.safaribooksonline.com/library/view/automating-security-in/9781491960745/

Sturlese, L. (2016, September 2). PS-ManageInactiveAD. Retrieved September 2, 2017, from https://github.com/9to5IT/PS-ManageInactiveAD

Sutherland, S. (2014, September 9). 15 ways to bypass the PowerShell execution policy. Retrieved from https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/

Tenable. (2015, January 21). Posh-Nessus: PowerShell module for automating Tenable Nessus vulnerability scanner. Retrieved from https://github.com/tenable/Posh-Nessus

TSO. (2013, August 27). Get inactive computer in domain based on last logon time stamp. Retrieved September 2, 2017, from https://gallery.technet.microsoft.com/scriptcenter/Get-Inactive-Computer-in-54feafde

Vanover, R. (2009, April 13). Enabling PowerShell on Windows Server 2008. Retrieved from http://www.techrepublic.com/blog/the-enterprise-cloud/enabling-powershell-on-windows-server-2008/

VandenBrink, R. (n.d.). Nessus and Powershell is like chocolate and peanut butter! Retrieved from https://isc.sans.edu/forums/diary/Nessus+and+Powershell+is+like+Chocolate+and+Peanut+Butter/20431/

Wilson, E. (2010, October 26). *Learn how to use .NET Framework commands inside Windows PowerShell*. Retrieved from https://blogs.technet.microsoft.com/heyscriptingguy/2010/10/26/learn-how-to-use-net-framework-commands-inside-windows-powershell/

Daniel Owen, ggold@danielowen.com

Wilson, E. (2015, August 2). Weekend scripter: Exploring Windows PowerShell 5.0. Retrieved from https://blogs.technet.microsoft.com/heyscriptingguy/2015/08/02/weekend-scripter-exploring-windows-powershell-5-0/

Appendix A

It should be noted, for readability of sample code, these scripts have been kept simple. Minor changes will make these more flexible at the expense of simplicity. As an example, looping through an array of groups rather than checking one hard coded group in get-admins.ps1 is more useful in production. Further, the scripts use a number of constants that are expedient for small sample scripts, but variables would better handle these settings as the complexity and re-usability of the scripts grow.

A.1 Sample code for get-admins.ps1

```
#Script to identify unauthorized accounts added to the Domain Admins group

#region variables
$allowed = Import-Csv 'C:\scripts\get-admins\allowed.csv' #Path to file
containing SIDs of users allowed to be a member of Domain Admins
$group = "Domain Admins" #Group being protected
$members = Get-ADGroupMember $group #Get members of the group we are protecting
#endregion

if ($allowed.count -eq 0) {exit} #Failsafe to kill the script if the text file
does not contain any data or is not read for some reason.

ForEach ($m in $members) { #Create a loop to look at each member of the $group
    $account = $m.sid.ToString() #Create a string variable that holds the SID
for the member of $group to be compared. This simplifies code later.
    if ($allowed.sid -notcontains $account) { #Does the $allowed array contain
the member of $group currently being compared? If not take action.
        Disable-ADAccount -Identity $account #Disable the AD account that was
added to $group but is not in the $allowed file.
        Remove-ADGroupMember -Identity $group -Members $account -Confirm:$false
#Remove $account from the $group.
        #This is a good place to send an email alert or write an alert to a
console.
        if ((Get-EventLog -LogName Application -Source "get-admins-script" -
ErrorAction SilentlyContinue) -eq $null) {
            New-EventLog -LogName Application -Source 'get-admins-script'
#Create a new event log type in the Application log if it does not exist.
        }
        Write-EventLog -LogName Application -Source 'get-admins-script' -
EntryType Warning -EventId 0
        -Message "$account was removed from $group as part of the get-admins
security process." #Write an event to the local systems Security Event Log.
    }
}
```

A.2 Sample code for get-oldpassword.ps1

```
#Script to find users who have not changed their password recently
$maxage = 90 #Maximum days since last password change
$date = (Get-Date).AddDays(-$maxage) #Date of oldest password that does not
require a change
$users = Get-ADUser -Filter {PasswordLastSet -lt $date -and Enabled -eq $true}
#Array that holds accounts that are past due for password change
$users | Export-Csv -NoTypeInformation c:\temp\oldpass.csv #Export all users
who need to change their passwords to a CSV
Send-MailMessage -To "user1@example.com" -From "user2@example.com" -Subject
"Users with old passwords" -Body "See the attached CSV file for a list of users
with passwords over $maxage days old" -Attachments 'c:\temp\oldpass.csv' -
SmtpServer smtp.example.com #Send an email
```

Daniel Owen, ggold@danielowen.com

A.3 Sample code for get-invalidgroupmembers.ps1

```
#Script to find users in the 'US_Associates' group that do not have a country
entry of 'US'
$group = Get-ADGroupMember 'US_Associates' #Create a new variable with all
members of the group we are interested in
$invalidmembers = @() #Initialize the array that will hold invalid users
foreach ($g in $group) { #Check the country for every user who is in the group
    $tuser = Get-ADUser $g.sid -Properties country #Read the AD record for the
current member being reviewed
    if ($tuser.country -notlike 'US') {$invalidmembers =
$invalidmembers+$tuser} #If the user's country is not "US" add it to the
variable invalidmembers
}
$invalidmembers | Export-Csv -NoTypeInformation c:\temp\invalidmembers.csv
#Export the results to a CSV file
if ($invalidmembers.Count -gt 0){ #If there are invalid members cleanup and
repot
    Remove-ADGroupMember -Identity 'US_Associates' -Members $invalidmembers -
Confirm:$false #Remove invalid members from the group
    Send-MailMessage -To "user1@example.com" -From "user2@example.com" -Subject
"Users removed from US_Associates AD group" -Body "See the attached CSV file
for a list of users removed from the US_Associates AD Group" -Attachments
'c:\temp\invalidmembers.csv' -SmtpServer smtp.example.com #Send an email
}
```

A.4 Sample code for get-recursivemanager.ps1

```
#Script to find a specific group of users based on manager, title, and
locations
#Get-ADDirectReport.ps1 can be downloaded from
https://gallery.technet.microsoft.com/scriptcenter/Get-ADDirectReport-962616c6
. "C:\scripts\Get-ADDirectReport.ps1" #Dot include script to do recursive
lookup based on manager
$reports = Get-ADDirectReports -Identity user1 -Recurse #Find all direct and
indirect reports for user1
$reportsfinal = @() #Initialize variable
foreach ($r in $reports) { #Create a list of users who will go on the report
    $user = Get-ADUser -Identity $r.SamAccountName -Properties country, title
#Lookup user in AD
    #Final removal of users who do not meet additional requirements
    if (($user.country -like 'US')  -and ($user.title -like 'manager' ))
{$reportsfinal = $reportsfinal+$user }
}
$reportsfinal | Export-Csv -NoTypeInformation C:\temp\selectreports.csv #Export
to CSV
```

Daniel Owen, ggold@danielowen.com