SANS Institute
Information Security Reading Room

Tracking Penetration Test Activities

Joshua Arey

Tracking Penetration Test Activities

GIAC (GCIH) Gold Certification

Author: Josh Arey, joshuaarey@gmail.com
Advisor: David Hoelzer

Accepted: 24 Feb 2020

Abstract

Most penetration testers ("pentesters") are required to track their actions during a penetration test event but rarely do so in enough detail to recreate all of their activities accurately. Instead, pentesters often only track activities that lead to findings disclosed in the final penetration testing ("pentest") report. Tracking testing activities can be challenging and often gets disregarded when it slows down a pentest engagement. Fortunately, there are automatic logging mechanisms on most pentest systems available for leveraging to help automatically track pentest activities. However, many logging capabilities do not sufficiently record the generated network traffic from the attacking system, and network monitoring tools do not record what actions triggered the sending of packets. Customizing system logging configurations and incorporating system monitoring tools such as auditd can help automatically track testing activities on Linux-based pentest systems. This additional logging allows for accurate tracking in enough detail for an auditor to accurately determine what actions a pentester took against the pentest targets.

1. Pentest Tracking Requirements

Many industries have governance in place that requires companies to regularly conduct penetration testing ("pentest") to meet regulatory requirements. The Payment Card Industry Data Security Standard (PCI-DSS) and National Institute of Standards and Technology (NIST) guidance directly state that pentesting is required. Other governance such as the Health Insurance Portability and Accountability Act (HIPAA) and the General Data Protection Regulation (GDPR) are generally interpreted to imply that the only way to comply is to conduct a pentest. PCI-DSS and NIST further specify the expected output from pentests. Included in these requirements are details about what data should be created and retained from a pentest. For PCI-DSS compliance, the "Penetration Testing Guidance Information Supplement" states that evidence must be retained, including "anything that may support the conclusion of the penetration test report" (Penetration Test Guidance Special Interest Group, 2015, p. 21). NIST guidance says that "assessors should keep a log that includes assessment system information and a step-by-step record of their activities" (Scarfone et al., 2008, p. 7-5). Despite these requirements, it is not uncommon for the only output of a pentest to be the final report detailing the discovered vulnerabilities. A pentest report alone does not fulfill the requirements of the regulations and standards mentioned above.

Even if regulations did not specify anything about tracking pentest activities, there are other benefits for pentesters to keep track of all their test activities. Fletcher suggests that pentesters "should try to write the methodology section of their report as you go" (2016). This constant tracking can help a pentest team know what they have or have not accomplished at any point during a test. Continually collecting data about test activities can prevent duplication of effort due to a lack of evidence that proves what activities have been accomplished. Comparing activity logs to test objectives can readily show how much of the test is complete at any given time during the pentest. At the end of a test, logs should be able to prove how the pentest met the agreed-upon test objectives.

When tracking is successful, a useful result is the ability to show that pentesters did not take specific actions against the target network. This information is vital in the

event of a real attack or other system anomalies that happen during the same timeframe as the pentest. If none of the pentesters' logs indicate they were interacting with the real-world victim machines, a customer can have confidence that the pentest team was not responsible for any actual damages because the logs would indicate what the testers were actually targeting during specific periods. However, if tester's logs are incomplete, a customer might not have confidence that the logs truly exonerate the pentester of any malicious or negligent activity.

While pentesters might accept the idea of logging, it can be a burdensome task that even the most diligent tester might not always complete. Manual logging can be seen as a hindrance to testing if it means regularly stopping in the middle of test activities to write down what actions were just taken, potentially interrupting test progress. Fortunately, there are tools to help keep track of activities, and many are installed by default on pentester's systems. Pentesters can leverage these tools to automatically gather all of the information they need to meet regulatory requirements, show proof of work, or potentially exonerate testers of malicious activities.

1.1. Challenges with Current Logging

This paper is limited to tracking pentest activities on Linux operating systems, though similar functionality and tools do exist for Windows operating systems. It is essential to understand which logging mechanism is used on the pentester's system to know how logs are stored and how to review them. Older Linux distributions used Syslog as the primary source of system logging, while newer systems use journald instead (Dean, 2018, p. 404). The `journalctl` command can be used to review `journald` output, while syslogs are usually analyzed with third-party tools or using the analysis tools that come with auditd. While the Syslog message format is an accepted IETF standard, many applications use their own log format (Gerhards, 2009). Instead of sending logs to the Syslog utility, applications frequently store logs in several well-known locations such as the `/var/log` directory, the `/tmp` directory, or hidden directories in the current user's home directory. Some applications also split logs into separate files based on the log type, such as "success" or "error" messages. These

disparities in how applications create and store log files can make it difficult to correlate logs that are created by a default Linux configuration.

Only a small number of standard applications enable logging by default, and some are not capable of logging at all. Even fewer custom scripts build in logging functionality during their development. Instead, it is common to rely on Bash history to track activities, but frequently these logs are incomplete due to some quirks in how Bash history gets logged by default. Because of these issues, alternate means are required to capture system activities.

Some pentesters choose to track activity using network monitoring tools. Van Doorn & Spithoff suggested using port mirroring network traffic to accomplish tracking (Van Doorn & Spithoff, 2018, p. 5). While network monitoring can go a long way in tracking activities, there are some deficiencies in this method as well. One problem is the technical requirements to capture network traffic. Pentest activities can generate a lot of network traffic, which can require a great deal of storage, and the storage device must also be able to write fast enough not to lose data (Koch, 2016, p. 6). Another disadvantage is the parsing of the captured network traffic. It often requires an expert to be able to interpret network traffic well enough to determine what the actions are taken on a system to create specific network traffic. Even the most advanced analyst might not be able to identify all of the commands that resulted in specific traffic, and the process is time-intensive regardless of the reviewer's skill level.

As an alternative or complement to using default system logging or network monitoring tools, auditd can be configured to capture all commands executed on a system during testing as well as all network activities with no additional effort from the system user. Audit logs can be correlated to identify what specific command generated specific network traffic while providing timestamps of when activities happened. This allows auditd to provide much of the missing details from the default logging of most Linux distributions.

Josh Arey, joshuaarey@gmail.com

2. Research Method

To test the difference between standard system logging and auditd logging, a virtual environment was created to allow the execution of multiple attack paths against known vulnerable targets to simulate activities that might be seen during a pentest. The environment consisted of a single pentest virtual machine (VM) running Kali Linux 2019.4. The target network was a small domain with standard services running with known vulnerabilities. The purposely vulnerable domain consisted of five systems, including three Windows Server 2008 R2 servers, an Ubuntu 16.04 host, and an Ubuntu 14.04 host. One of the Windows Servers was configured as a Domain Controller, and the other two Windows servers were joined to that domain, but the Ubuntu systems were not joined to the domain. Each target system was configured to allow a different vulnerable service as an entry point.

Five different attack paths were carried out from the attacker machine. Each attack path consisted of techniques from the MITRE adversarial tactics, techniques, and common knowledge (ATT&CK™) Matrix ("Enterprise techniques," n.d.). Each path consisted of an exploitable application for initial access, privilege escalation, persistence, lateral movement, and data exfiltration. Each attack path required the use of different tools, though some tools were used in more than one attack path. A variety of tools were chosen to include not only popular tools run from a Bash shell but also graphical tools to determine what could and could not be tracked when Graphical User Interfaces (GUIs). Several tools were chosen because of the large amount of network traffic generated during scanning. Table 1 lists the attack paths with their matching ATT&CK Matrix Technique ID. Each row in Table 1 represents a different attack path. Tools used to accomplish these techniques were:

- Burp Suite
- dirb
- Firefox
- FTP
- Hydra
- Metasploit
- Netcat
- Nmap
- sqlmap
- ssh
- VNC Viewer
- WPScan

Josh Arey, joshuaarey@gmail.com

Initial Access	Privilege Escalation	Persistence	Lateral Movement	Exfiltration
anonymous FTP (T1190)	sudo (T1169)	create account (T1136)	remote service (ssh authorized keys) (T1021)	FTP (T1048)
weak SSH password (T1190)	.bashrc commands (as root) (T1156)		third party software (VNC Server) (T1072)	SCP (T1048)
Wordpress SQLi (T1190)	DLL search order (T1038)	new service (T1050)	pass-the-hash (T1075)	WebDAV (T1048)
CMS exploit (T1190)	service permissions (T1058)	scheduled task (T1053)	admin shares (T1077)	C2 channel (T1041)
SMB (as SYSTEM) (T1190)		Winlogon helper (T1004)	logon scripts (T1037)	HTTP (T1048)

Table 1 – Attack Path Techniques Used

All five attack paths were run two times. The first set of attacks was run with the attack machine configured with necessary tools for the attacks, but no logging functions were changed from the default Kali 2019.4 configuration. After the first set of attack scenarios was run, files modified during the attacks were identified using the `find` command. Before the second set of attacks, all systems were reverted to the initial vulnerable configuration using VM snapshots. After the systems were reverted, auditd was installed on the Kali VM with basic network and command execution rules configured. After the second set of attacks was complete, an analysis of the Syslog data was performed.

3. Findings

The logged data varied in volume and completeness based on the different levels of logging, as expected. Because of the limited number of tools and techniques used for this experiment, it should be noted that the number and quality of logs created by different tools other than those used in this experiment could vary greatly. Additionally,

the community version of Burp Suite was used in this experiment and did not allow the saving of session data while the professional version does.

The ideal level of logging would capture what specific actions were taken on the attacking system resulting in the generation of network traffic directed at the target systems. Timestamps and identification of the user who took those actions against the target are also be needed to form a complete picture of exactly what happened during the test. It is also preferable to have the fewest number of log files as needed to provide accurate logging and to have all log files in the same location.

3.1. Built-in Logging without Modification

There were many shortfalls in tracking pentest activities using only the logs created by the default logging of Kali Linux. However, several tools did create higher quality logs than expected. Before analysis, the location of all the logs was identified because it was unknown where logs might have been written. Before attacks started, a file named `starttime` was created as a reference for the `find / -newer starttime > newer` command, which was run after the attacks to identify all of the modified files during the attack. This command was used to find all modifications between the `starttime` file creation time and completion of the attack scenarios. The results identified over one hundred thousand files that were modified during the attacks. However, more than 99% of those files belonged to the procfs and sysfs file systems, which record processes and system hardware information, respectively. Excluding files modified by these processes resulted in more than one thousand remaining files that were modified during the attack scenarios. Few of these files were useful in determining what actions had been taken on the attacking system. Instead, they mostly contained information about the current state of different applications being run without details that were necessary to correlate pentester actions to network traffic.

The only useful log files created during the attacks were stored in hidden directories in the user's home directory. Figure 1 shows the files and folders modified during the attacks. While it might be expected that some standard system logs were generated during the scenarios, the only meaningful log in the `/var/log` directory that

was modified during the attack scenarios was the Apache `access.log`, which showed the attacker downloading a file to the victim machine. This log did not provide any meaningful information about what the pentester did to trigger the access of the attacker's web page.

```
-rw-r--r--  1 root root 5047999 Jan  8 20:36 newer
drwxr-xr-x  9 root root    4096 Jan  8 19:37 .msf4
drwxr-xr-x  4 root root    4096 Jan  8 19:21 .sqlmap
drwx------  5 root root    4096 Jan  8 19:11 .mozilla
drwx------  7 root root    4096 Jan  8 19:11 .cache
drwx------  4 root root    4096 Jan  8 19:10 .BurpSuite
drwxr-xr-x  4 root root    4096 Jan  8 19:09 .java
drwx------  2 root root    4096 Jan  8 18:47 .ssh
-rw-r--r--  1 root root      32 Jan  8 18:38 starttime
```

Figure 1 – Files modified during the attack with default logging

3.1.1. Metasploit Framework Logs

During the testing, Metasploit Framework created a history file that was useful in recreating pentest activities. Similar to Bash history, Metasploit records all the commands executed within a `msfconsole` session. Metasploit logs were useful because Bash history did not record commands executing while in a Metasploit console. The Metasploit history file can be used to see all of the modules that were used and how they were configured before execution. However, version five of the Metasploit Framework allows pentesters to use shortcuts to select modules returned in search results. This means that instead of seeing the full name of a module in the history, the `use` command shows only a number as its argument instead of the module name. The example usage shown in Figure 2 would result in "`use 2`" being seen in the Metasploit history instead of "`use windows/smb/ms17_010_eternalblue`" as would have been seen in previous versions of Metasploit. The only way for an auditor to know what module was selected in such a scenario would be to execute the same search to identify which module corresponded to a specific number, which is not a practical or efficient activity to expect from an auditor.

Josh Arey, joshuaarey@gmail.com

```
msf5 > search ms17_010

Matching Modules
================

   #  Name                                           Disclosure Date
   -  ----                                           ---------------
   0  auxiliary/admin/smb/ms17_010_command           2017-03-14
   1  auxiliary/scanner/smb/smb_ms17_010
   2  exploit/windows/smb/ms17_010_eternalblue       2017-03-14
   3  exploit/windows/smb/ms17_010_eternalblue_win8  2017-03-14
   4  exploit/windows/smb/ms17_010_psexec            2017-03-14


msf5 > use 2
msf5 exploit(windows/smb/ms17_010_eternalblue) > 
```

Figure 2 – Metasploit Framework Version 5 search shortcut

Other similar shortcuts in Metasploit, such as the `services` command, can set attack targets without the history recording what IP address or hostname was set as the `RHOSTS` variable. Figure 3 shows the result of `RHOSTS` being set by the `services` command. This again would result in a Metasploit history file that does not contain information about what target hosts were chosen at the attack targets for an exploit before it was run. While these shortcuts did make it more difficult to track the pentester actions, the Metasploit history still provided useful information to recreate actions that the tester took.

```
msf5 exploit(windows/smb/ms17_010_eternalblue) > services -p 445 -R
Services
========

host            port  proto  name  state  info
----            ----  -----  ----  -----  ----
192.168.38.112  445   tcp    smb   open
192.168.38.114  445   tcp

RHOSTS => 192.168.38.112 192.168.38.114

msf5 exploit(windows/smb/ms17_010_eternalblue) > options

Module options (exploit/windows/smb/ms17_010_eternalblue):

   Name           Current Setting                Required  Description
   ----           ---------------                --------  -----------
   RHOSTS         192.168.38.112 192.168.38.114  yes       The target host(
   RPORT          445                            yes       The target port
```

Figure 3 – RHOSTS options set using the services command

3.1.2. Firefox Logs

Mozilla Firefox also created log files in the form of SQLite databases ("Places database," 2019). The `places.sqlite` database contained a table called `moz_places`, which captured information about which sites the attacker had visited during the test. This database included the last time each site was visited but not an individual entry for each time a site was visited. This proved to be less useful when the pentester connected to the same site many times, as is typical during pentesting. The fact that this information was in an SQLite database also made it less convenient to analyze. Figure 4 shows an example of an entry in the `moz_places` table. Another limitation of this output was that it did not contain details about the HTTP methods used or any HTML form data, which was stored in a separate `formhistory` database, making correlation even more difficult.

```
sqlite> select * from moz_places where url like 'http://192.168.38.112/cms/';
id|url|title|rev_host|visit_count|hidden|typed|frecency|last_visit_date|guid|
foreign_count|url_hash|description|preview_image_url|origin_id
22|http://192.168.38.112/cms/|Home - victim cms|211.83.861.291.|2|0|0|2048|15
78533753463243|Vw_CYhlDnVOR|0|125510488531529|||10
```

Figure 4 – Example SQLite output of Firefox places database

3.1.3. SQLmap Logs

SQLmap-stored data about executed commands and its results in an SQLite database as well. SQLmap organized output into folders named for the target of each command, as shown in Figure 5. The `target.txt` file contained the commands that were run, and the `log` file contained results of successful exploitation. While the timestamps on these files could be used to estimate when these activities occurred roughly, they were not definitive. For example, if SQLmap is attempting long-running time-based injections, the file timestamp would not indicate when the scan was started. If the pentester chooses to use a time-based attack to dump an entire database, which could take a very long time to complete, the duration would not be accurately reflected by the single timestamp and would be very difficult to correlate to network activity.

Josh Arey, joshuaarey@gmail.com

```
root@kali:~/.sqlmap/output# ls -alR
.:
total 12
drwxr-xr-x 3 root root 4096 Jan  8 19:25 .
drwxr-xr-x 4 root root 4096 Jan  8 19:21 ..
drwxr-xr-x 2 root root 4096 Jan  8 19:25 192.168.38.113

./192.168.38.113:
total 24
drwxr-xr-x 2 root root 4096 Jan  8 19:25 .
drwxr-xr-x 3 root root 4096 Jan  8 19:25 ..
-rw-r--r-- 1 root root 2268 Jan  8 19:25 log
-rw-r--r-- 1 root root 8192 Jan  8 19:25 session.sqlite
-rw-r--r-- 1 root root  258 Jan  8 19:25 target.txt
```

Figure 5 – SQLmap automatically generated logs

3.1.4. Bash History

Bash history is another standard log file used for tracking activities and is saved in a `.bash_history` file for each user. During the attack scenarios, some well-known limitations in how the `.bash_history` file is created were experienced, which reduced the log's usefulness. Figure 1 from above shows a list of all the files written to the user's home directory during the pentest activities. The `.bash_history` file was notably unmodified during the attack timeframe. This happened because the pentester did not close the terminal during the scenarios. Bash history is only written to file when a terminal gracefully exits. When a terminal crashed or was forcibly killed, including using the close button on a terminal, nothing was added to Bash history while using the default history configuration. Each separate terminal also maintained its own history until it was closed, meaning that reviewing the history file while activities were still happening resulted in an incomplete list of commands that had been executed. Additionally, Bash history did not store timestamps of when commands were executed.

3.1.5. Default Logging Capabilities Results

Of the tools used while performing the attack scenarios, only Bash, Firefox, Metasploit Framework, and SQLmap created any logs that could be used to recreate what happened on the attacker system. All output had limited value in trying to recreate what actions the pentester had taken. Minimal timestamp information made it impossible to know when most activities took place and lack of detail about whether commands completed successfully meant that there was no way to know what commands connected

to the remote machines. Additionally, there were few details about what traffic was created by automated scanners like Nmap, DIRB, SQLmap, and WPScan, resulting in almost no record of attacks that were attempted, yet unsuccessful. At best, the standard Linux logging mechanisms could identify a small portion of the pentest activities that created successful connections and almost no information about the pentest as a whole.

3.2. Auditd Configuration

After reverting all of the VMs to their initial configurations from before the first set of attacks, auditd was installed on the Kali VM. Installation of auditd can be done manually by downloading directly from the developer or by adding a repository and using the `aptitude` command to install auditd and all of its dependencies. The Kali Linux rolling repository does not have auditd available at the time of writing. As part of the installation, auditd adds examples of rules to the `/usr/share/doc/auditd/examples/rules` directory. The `71-networking.rules` example was copied to the `/etc/audit/rules.d/audit.rules` file, which is loaded by the auditd service when auditd is enabled to start with the system. This rule logged any "accept" or "connect" system calls and tagged them with the key "external-access." The example network rule is shown in Figure 6.

```
-a always,exit -F arch=b64 -S accept,connect -F key=external-access
```

Figure 6 – Auditd example network rule

Unfortunately, auditd does not provide an example rule to log all command executions. While all of the syscalls that can be monitored have manual page entries, a more readable list of available syscalls can be found at https://filippo.io/linux-syscall-table (Valsorda, n.d.). This site also contains links to Linux source code for each of the syscalls, which is necessary to understand the command arguments that auditd records. The syscall required to monitor program execution is called "execve," and the auditd rule to log program execution is shown in Figure 7.

```
-a always,exit -F arch=b64 -S execve -F key=exec
```

Figure 7 – Auditd program execution rule

3.2.1. Auditd Limitations

With auditd rules in place to track “execve," “connect," and “accept” system calls, the attack paths were all run again. Because auditd writes logs to a specific directory and file, the filesystem was not searched to identify files altered during the attack process. Instead, only the results in the `/var/log/auditd/audit.log` file were reviewed for their ability to recreate the actions that were taken during each attack path. While it is possible to view and search `audit.log` directly, the output is much more easily reviewed using the `ausearch` and `aureport` tools which are installed with auditd.

Initial analysis of auditd logs used `ausearch` to look for any program executions. This returned just over nine hundred events. While still a large amount of data to parse, it is significantly less than the total number of files modified on the system, as seen in the first attack scenario. Similar to the large number of files modified by the system during the first logging configuration, there were a large number of program execution syscalls that were likewise created by the system and, therefore, could be ignored for the purposes of tracking tester activities. Still, parsing this subset of data was difficult due to some limitations of the `ausearch` command. While `ausearch` can output as raw, CSV, text, or interpreted, each format provided different data fields, which meant that not all fields were shown in every format. Figures 8, 9, 10, and 11 are examples of the same search output in different formats.

```
root@kali:/var/log/audit# ausearch --format raw -a 30203
type=SYSCALL msg=audit(1578624066.628:30203): arch=c000003e syscall=42 success=yes exit=
0 a0=3 a1=5602e37b8c40 a2=10 a3=5602e3797f49 items=0 ppid=1403 pid=1745 auid=0 uid=0 gid
=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=2 comm="ftp" exe="/usr/bin/n
etkit-ftp" subj==unconfined key="network-access"
type=SOCKADDR msg=audit(1578624066.628:30203): saddr=02000015C0A8266C0000000000000000
type=PROCTITLE msg=audit(1578624066.628:30203): proctitle=667470003139322E3136382E33382E
313038
```

Figure 8 – Ausearch output as raw format

```
root@kali:/var/log/audit# ausearch --format csv -a 30203
NODE,EVENT,DATE,TIME,SERIAL_NUM,EVENT_KIND,SESSION,SUBJ_PRIME,SUBJ_SEC,SUBJ_KIND,ACTION,
RESULT,OBJ_PRIME,OBJ_SEC,OBJ_KIND,HOW
,SYSCALL,01/09/2020,20:41:06,30203,audit-rule,2,root,root,priviliged-acct,connected-to,s
uccess,{ fam=inet laddr=192.168.38.108 lport=21 },,socket,/usr/bin/netkit-ftp
```

Figure 9 – Ausearch output as CSV format

```
root@kali:/var/log/audit# ausearch --format text -a 30203
At 20:41:06 01/09/2020 root successfully connected-to 192.168.38.108 using /usr/bin/netk
it-ftp
```

Figure 10 – Ausearch output as text format

```
root@kali:/var/log/audit# ausearch --format interpret -a 30203
----
type=PROCTITLE msg=audit(01/09/2020 20:41:06.628:30203) : proctitle=ftp 192.168.38.108
type=SOCKADDR msg=audit(01/09/2020 20:41:06.628:30203) : saddr={ fam=inet laddr=192.168.
38.108 lport=21 }
type=SYSCALL msg=audit(01/09/2020 20:41:06.628:30203) : arch=x86_64 syscall=connect succ
ess=yes exit=0 a0=0x3 a1=0x5602e37b8c40 a2=0x10 a3=0x5602e3797f49 items=0 ppid=1403 pid=
1745 auid=root uid=root gid=root euid=root suid=root fsuid=root egid=root sgid=root fsgi
d=root tty=pts0 ses=2 comm=ftp exe=/usr/bin/netkit-ftp subj==unconfined key=network-acce
ss
```

Figure 11 – Ausearch output as interpreted format

3.2.2. Auditd Rule Modifications

The pentester activity that triggered the creation of logs in figures 8, 9, 10, and 11, was the `ftp 192.168.38.108` command shown in the "proctitle" field in the interpreted version of the `ausearch` output. Each output format appears to have enough information to identify which commands created network traffic from the attacker machine while annotating what time the actions took place. This information is necessary, but not sufficient for fully recreating what actions were taken if an auditor only looks at logs. The information that is still missing is which ports were used on each system for a specific command to connect to the target system. Also, the attacker's IP address may be inferred, but if this data is viewed out of context, it might not be enough information to identify specific connections.

Solving this problem required identifying what additional syscalls require monitoring to create a complete picture. Fortunately, only one additional syscall was required in this case: "getsockname." The resulting rule incorporating this syscall is shown in Figure 12.

Josh Arey, joshuaarey@gmail.com

```
root@kali:/var/log/audit# auditctl -l
-a always,exit -F arch=b64 -S connect,accept,getsockname -F key=network
-a always,exit -F arch=b64 -S execve -F key=exec
```

Figure 12 – Updated audit rules

The resulting logs included the target IP address as well as the target port. Logs also included the time and process ID that caused the socket name lookup. The `ausearch` for getsockname is shown in Figure 13.

```
root@kali:/var/log/audit# ausearch -i -x ftp -k getsockname | cut -d" " -f5-8,15,16
----
proctitle=ftp 192.168.38.108
saddr={ fam=netlink nlnk-fam=16 nlnk-pid=2856
arch=x86_64 syscall=getsockname success=yes exit=0 pid=2856 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=47070
arch=x86_64 syscall=getsockname success=yes exit=0 pid=2856 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=55579
arch=x86_64 syscall=getsockname success=yes exit=0 pid=2856 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=35975
arch=x86_64 syscall=getsockname success=yes exit=0 pid=2856 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=42295
arch=x86_64 syscall=getsockname success=yes exit=0 pid=2856 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=60595
arch=x86_64 syscall=getsockname success=yes exit=0 pid=2856 auid=root
```

Figure 13 – getsockname syscall log results

For validation, these search results can be compared to network packet captures to ensure that all unique connections were accounted for in the audit logs, as shown in Figure 14.

Wireshark · Conversations · any

Ethernet | IPv4 · 1 | IPv6 | TCP · 5 | UDP

Address A	Port A	Address B	Port B	Packets
192.168.38.100	47070	192.168.38.108	21	35
192.168.38.100	55579	192.168.38.108	20	1
192.168.38.100	35975	192.168.38.108	20	1
192.168.38.100	42295	192.168.38.108	20	1
192.168.38.100	60595	192.168.38.108	20	1

Figure 14 – Packet capture confirming log results

The results of having these three network syscalls monitored to provide information about the user who created the process, process ID, source IP and port, and destination IP and port. While Figure 15 truncates the time fields for viewability, time was also recorded for each log.

```
root@kali:/var/log/audit# ausearch -i -x ftp -k network -sv yes | cut -d" " -f5-8,15,16
----
proctitle=ftp 192.168.38.108
saddr={ fam=netlink nlnk-fam=16 nlnk-pid=3174
arch=x86_64 syscall=getsockname success=yes exit=0 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.108 lport=21
arch=x86_64 syscall=connect success=yes exit=0 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=47072
arch=x86_64 syscall=getsockname success=yes exit=0 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.108 lport=20
arch=x86_64 syscall=accept success=yes exit=7 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=32887
arch=x86_64 syscall=getsockname success=yes exit=0 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.108 lport=20
arch=x86_64 syscall=accept success=yes exit=7 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=46857
arch=x86_64 syscall=getsockname success=yes exit=0 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.108 lport=20
arch=x86_64 syscall=accept success=yes exit=7 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=40721
arch=x86_64 syscall=getsockname success=yes exit=0 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.108 lport=20
arch=x86_64 syscall=accept success=yes exit=7 pid=3174 auid=root
----
proctitle=ftp 192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=34705
arch=x86_64 syscall=getsockname success=yes exit=0 pid=3174 auid=root
```

Figure 15 – Network syscall logs

3.2.3. Modified Auditd Results

There are further considerations to take when adding auditd rules. The primary concern is the logging configuration, which by default, only keeps five logs, each log

only being eight megabytes. When trying to log all of the activities of a pentester, this is not close to enough logging capability to capture more than several minutes of scanning activities, which invariably creates thousands of log entries if all of the network activity is being tracked. Execution of all five pre-defined scenarios with limited scanning resulted in less than twenty megabytes of logs with over thirty-two thousand unique log entries. The scope of testing and duration of a pentest must be considered to ensure that enough storage is available and that the log configuration supports collecting logs for the duration of the test. Utilizing a Syslog server could be a solution to this log storage problem as well.

With the modified auditd configuration, it was possible to identify which commands were executed, who executed them, what time they were executed, the source IP and port, and the target IP and port. This means that just two auditd rules are needed to generate logs that can be used to recreate all of the network activity generated from a pentester's system. `ausearch's` ability to search for specific processes, specific applications, or specific times periods makes it relatively simple to find specific network activities. Simple scripts can be developed to generate a more user-friendly output of the logs, such as the script in Appendix A of this paper. Nevertheless, there was still one significant missing piece to fully tracking a pentester's activities: what happened after the initial network connection.

3.2.4. System Call Logging Considerations

Auditd logs create very detailed records of what a program requested the system to do on its behalf through syscalls. The difficulty in parsing these logs lies in how applications make system calls. Syscalls are functions that take data as arguments, and each argument is represented in the log as encoded hexadecimal notation ("Understanding Audit Log Files Red Hat Enterprise Linux 6," n.d.). Using the previous auditd rules, syslogs of an SSH connection show a "connect" syscall followed by a "getsockname" syscall. For the attacker machine to send data over that connection, the "write" syscall is used to send data to the file descriptor that is joined to a network address during the "connect" syscall. Even when the "write" syscall is monitored through auditd, the resulting log only shows its buffer argument as encoded hexadecimal instead

of readable plain text, as shown in Figure 16. This means that syslogs are not going to contain the strings of commands that were sent through a successful connection to a target machine. Instead, some other methods must be used to track pentest activities that happen after the attacker makes a successful connection to a target system.

```
proctitle=ssh vagrant@192.168.38.108
saddr={ fam=inet laddr=192.168.38.108 lport=22 }
arch=x86_64 syscall=connect success=yes exit=0 a0=0x3 a1=0x563da80aca10 a2=0x10
a3=0x563da76db848 items=0 ppid=1527 pid=5530 auid=root uid=root gid=root euid=ro
ot suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts0 ses=2 comm=ssh e
xe=/usr/bin/ssh subj=unconfined key=network-access
----
proctitle=ssh vagrant@192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=49934 }
arch=x86_64 syscall=getsockname success=yes exit=0 a0=0x3 a1=0x7ffccbce3b50 a2=0
x7ffccbce3b4c a3=0x7fe41285bac0 items=0 ppid=1527 pid=5530 auid=root uid=root gi
d=root euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts0 se
s=2 comm=ssh exe=/usr/bin/ssh subj=unconfined key=network-access
----
proctitle=ssh vagrant@192.168.38.108
saddr={ fam=inet laddr=192.168.38.100 lport=49934 }
arch=x86_64 syscall=getsockname success=yes exit=0 a0=0x3 a1=0x7ffccbce3f40 a2=0
x7ffccbce3f3c a3=0x20 items=0 ppid=1527 pid=5530 auid=root uid=root gid=root eui
d=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts0 ses=2 comm=s
sh exe=/usr/bin/ssh subj=unconfined key=network-access
----
proctitle=ssh vagrant@192.168.38.108
arch=x86_64 syscall=write success=yes exit=32 a0=0x3 a1=0x563da80ad2a0 a2=0x20 a
3=0x563da76e63cd items=0 ppid=1527 pid=5530 auid=root uid=root gid=root euid=roo
t suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts0 ses=2 comm=ssh ex
e=/usr/bin/ssh subj=unconfined key=write
```

Figure 16 – syscall argument as encoded hex

3.3. Additional Tracking Tools

There are several possible methods to fill the gap of tracking actions that pentesters take during interactive sessions with target machines. Each method has its pros and cons, and no single method can collect all of the information needed to create a full picture of pentest activities.

The `script` command can be used to capture a typescript of everything displayed on a terminal regardless of where the command execution happens, such as in an interactive shell to a victim machine ("script(1) - Linux manual page," n.d.). To ensure each new terminal is logged using the script command, `script` can be added to the `.profile` file, which allows it to be launched when a new shell is launched. But,

launching `script` automatically with every shell instance requires strict adherence to the manual's instructions to protect against infinite loops.

Weaknesses in the `script` command lie in how the program is launched and the format in which it stores its logs. When a user tries to exit the terminal that was started using the `script` command, the first `exit` or `control+D` causes the `script` command to exit. Now the user is in the terminal session, which initially spawned the terminal being monitored by `script`, and their actions are no longer being logged. While this is not necessarily a frequent occurrence or one with a high likelihood of impact, the pentester might move on to another task and come back to the terminal that is not being logged and begin a separate task.

The manual pages also explicitly state that `script` does not deal well with some interactive commands such as `vi`. This is because `script` captures control characters as well as regular text. When trying to review the output of the `script` log using a tool that understands how to interpret typescript, some characters are interpreted during the review process. This can create some output that is unreadable because of how it is displayed to the user when control characters are interpreted.

3.3.1. Command Aliases

Bash aliases can also be helpful in tracking tool outputs. Of the small set of tools used for this project, many can output logs but do not, by default. Nmap, dirb, and wpscan have options that allow the creation of output logs. Command aliases can be configured to append the output options to the command if the user does not manually include it. For example, setting `alias wpscan='wpscan -o wpscan_$(date "+%Y-%m-%d-%T") '` would ensure that a tester running the wpscan command would generate a log file named wpscan followed by the current date and time. A specific file location can also be included in the alias for more convenient log aggregation. Inserting aliases into the `.bashrc` or the `.bash_profile` files ensures that the aliases exist in all terminals. Aliases can cause unexpected operation of some commands, so care is needed to make sure the alias is performing the intended action correctly.

Josh Arey, joshuaarey@gmail.com

3.3.2. Bash History

Bash history is another tool that can be easily configured to have a much better output than what is available by default. Thomas Laurenson's guide on creating an improved Bash history sets options to append instead of overwriting, formats multiline commands into one command, and appends each command to history immediately after it is executed (Laurenson, 2018). Laurenson's guide also shows how to include a timestamp and set the history file size to unlimited. The resulting configuration in Figure 17 can be added to `/etc/bash.bashrc` to apply the configuration to all users. While command executions are logged with auditd, having an accurate history file with timestamps can also be useful in correlating pentester activities.

```
HISTTIMEFORMAT='%F %T '
HISTFILESIZE=-1
HISTSIZE=-1
HISTCONTROL=ignoredups
shopt -s histappend
shopt -s cmdhist
export PROMPT_COMMAND="${PROMPT_COMMAND:+$PROMPT_COMMAND$"\n"}history -a;
history -c; history -r"
```

Figure 17 – Bash history modifications

3.3.3. Screen Recording

A final option for recording pentester activities is to run a screen recording application for the duration of test activities. The 2019.4 build of Kali Linux has a screen recorder called `recordmydesktop`, which is pre-installed and can directly save in MP4 format. Recording video can consume a large amount of disk space, but setting a lower framerate and using the MP4 format resulted in less than 500 Megabytes per hour of storage consumption. While screen recording does not record network traffic and is not a simple format to review unless it has time markers set in the video, it does provide a thorough record of all events that happened on the pentester's machine. The most crucial benefit screen recording provides documenting actions taken in GUI interfaces. No other tool can accurately capture what happens outside of a terminal. Besides storage, the only other issue is making sure the recording is initiated and saved correctly.

Josh Arey, joshuaarey@gmail.com

4. Recommendations and Implications

The ideal way to capture every action a pentester takes is to make three configuration changes to the attacking system. The first change is to configure auditd to capture the "connect," "accept," "getsockname," and "execve" syscalls. The second is to create command aliases in the pentester's user profile that append logging options to commands that have logging capabilities as well as modify the Bash history configuration. The third is to configure screen recording during all pentest activities. With all three modifications, there is a high degree of certainty that all actions taken by the penster can be recreated by analyzing the logs that were automatically created by the system.

4.1. Implications for Future Research

Due to the niche requirement of tracking a high level of detail about actions taken on a system, it is not likely that any of the tools discussed in this paper will be modified by their developers to incorporate additional logging capabilities. However, it might be possible to create a tool that launches all the tools needed to track the activity on a system and create a report. Any effort to create such a tool would be an enormous undertaking. Still, if such a tool were able to produce a user-friendly output, that tool would be a great asset to anyone trying to meet regulatory compliance by tracking pentester activities in great detail.

5. Conclusion

Unfortunately, there is no single tool that can track pentest activities in a way that would allow an auditor to recreate every action a pentester took during a test. This research showed that default logging is not sufficient for regulatory compliance. That does not mean that tracking pentest activities cannot be accomplished with some work. Many tools have been successful in capturing a specific part of the actions that are taken during a pentest. For the time being, these tools simply need to be used in unison. With the installation of auditd and the reconfiguration of several system settings, it is possible

to track every action that is performed during a pentest. Some of the resulting logs are not necessarily easy to parse and correlate, but it is absolutely possible. These configuration changes are not unreasonable to ask of pentesters because it results in easier compliance with regulations and provides better evidence of what pentesters have accomplished for their reporting.

Josh Arey, joshuaarey@gmail.com

References

Dean, A. K. (2018). *Linux Administration Cookbook*. Packt Publishing.

Enterprise techniques. (n.d.). Retrieved 28 November, 2019, from https://attack.mitre.org/techniques/enterprise/

Fletcher, D. (2016, October 24). How to Not Suck at Reporting (or How to Write Great Pentesting Reports). Black Hills Information Security. https://www.blackhillsinfosec.com/how-to-not-suck-at-reporting-or-how-to-write-great-pentesting-reports

Gerhards, R. (2009, March). RFC 5424 - The Syslog Protocol. https://tools.ietf.org/html/rfc5424

Koch, M. (2016, November 7). *Implementing Full Packet Capture* [White paper]. Retrieved 2 February, 2020, from https://www.sans.org/reading-room/whitepapers/forensics/implementing-full-packet-capture-37392

Laurenson, T. (2018, July 2). Improved BASH history for multiple sessions, unclean shell exits and timestamps for every command executed. https://www.thomaslaurenson.com/blog/2018/07/02/better-bash-history/

Penetration Test Guidance Special Interest Group. (2015). *Information supplement: Penetration testing guidance*. PCI Security Standards Council. https://www.pcisecuritystandards.org/documents/Penetration_Testing_Guidance_March_2015.pdf

Scarfone, K., Souppaya, M., Cody, A., & Orebaugh, A. (2008). *Technical guide to information security testing and assessment: recommendations of the National Institute of Standards and Technology* (SP 800-115). National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf

script(1) - Linux manual page. (n.d.). https://man7.org/linux/man-pages/man1/script.1.html

The Places database. (2019, October 30). https://developer.mozilla.org/en-US/docs/Mozilla/Tech/Places/Database

Josh Arey, joshuaarey@gmail.com

Understanding Audit Log Files Red Hat Enterprise Linux 6. (n.d.). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/sec-understanding_audit_log_files

Valsorda, F. (n.d.). Searchable Linux Syscall Table for x86 and x86_64. https://filippo.io/linux-syscall-table

Van Doorn, H. and Spithoff M. (2018, February 11). *Pentest Accountability by Analyzing Network Traffic & Network Traffic Metadata* [White paper]. Retrieved 30 October, 2019, from https://pdfs.semanticscholar.org/59d0/a3b5f16672ef734f356c19b14c8b61418a5a.pdf

Josh Arey, joshuaarey@gmail.com

Appendix A

Audit Log Parsing Script

```python
#!/usr/bin/env python3

from subprocess import check_output,PIPE

# ausearch command to get pids
ausearchPIDsCMD = 'ausearch -k network-access -i -sv yes | grep " pid" | cut -d" " -f15 | cut -d"=" -f2 | sort -u'

# ausearch prefix for next set of commands
ausearchPrefix = 'ausearch -i -p '

# ausearch command to get actual login user ID
ausearchAUID = ' -k exec | grep auid | cut -d" " -f16 | cut -d"=" -f2-'

# ausearch command to get command that was executed
ausearchEXEC = ' -k exec | grep proctitle | cut -d" " -f5- | cut -d"=" -f2-'

# ausearch command to get network access source and destination IP:PORT
ausearchNetworkAccess = ' -k network-access| awk -F[\ =] \'{OFS=":"} $10=="connect" || $10=="getsockname" {print $4 " "$5" "$10"\\t"ip,port }{ip=$12} {port=$14}\''

# get all the pids that created network connections into a list
pids = check_output(ausearchPIDsCMD, shell=True, universal_newlines=True)

# iterated through each PID and run ausearch commands to get the COMMAND, USER, and NETWORK access information
for pid in pids.split():
    # get user login info
    auidCMD = ausearchPrefix + pid + ausearchAUID
    user = check_output(auidCMD, shell=True, universal_newlines=True)

    # get command info
    execCMD = ausearchPrefix + pid + ausearchEXEC
    executedCommand = check_output(execCMD, shell=True, universal_newlines=True)

    # get network access
    netAccessCMD = ausearchPrefix + pid + ausearchNetworkAccess
    netAccess = check_output(netAccessCMD, shell=True, universal_newlines=True)

    # print the results
    print('{} executed {} resulting in: \n{}'.format(user,executedCommand,netAccess))
```

Figure 18 – Auditd log parsing script

Josh Arey, joshuaarey@gmail.com

```
root
 executed /usr/bin/ruby /usr/bin/wpscan --url http://192.168.38.113/wordpress
 resulting in:
audit(01/11/2020 19:25:23.560:31016) connect    192.168.38.113:80
audit(01/11/2020 19:25:23.568:31017) getsockname        192.168.38.100:35512
audit(01/11/2020 19:25:25.552:31018) connect    192.168.38.113:80
audit(01/11/2020 19:25:25.556:31019) getsockname        192.168.38.100:35514
audit(01/11/2020 19:25:30.880:31021) getsockname        192.168.38.100:35516
audit(01/11/2020 19:25:30.880:31022) connect    192.168.38.113:80
audit(01/11/2020 19:25:30.880:31023) getsockname        192.168.38.100:35518
audit(01/11/2020 19:25:30.884:31024) connect    192.168.38.113:80
audit(01/11/2020 19:25:30.880:31020) connect    192.168.38.113:80
audit(01/11/2020 19:25:30.884:31025) connect    192.168.38.113:80
audit(01/11/2020 19:25:30.884:31026) getsockname        192.168.38.100:35522
audit(01/11/2020 19:25:30.900:31027) connect    192.168.38.113:80
audit(01/11/2020 19:25:30.900:31028) getsockname        192.168.38.100:35524
audit(01/11/2020 19:25:30.900:31029) getsockname        192.168.38.100:35520

root
 executed java -jar /usr/bin/burpsuite
 resulting in:
audit(01/11/2020 19:29:09.476:31153) connect    ::ffff:192.168.38.113:80
audit(01/11/2020 19:29:09.496:31154) getsockname        ::ffff:192.168.38.100:35552
audit(01/11/2020 19:29:09.936:31158) connect    ::ffff:192.168.38.113:80
audit(01/11/2020 19:29:09.940:31159) getsockname        ::ffff:192.168.38.100:35556
audit(01/11/2020 19:29:10.472:31174) connect    ::ffff:192.168.38.113:80
audit(01/11/2020 19:29:10.472:31175) getsockname        ::ffff:192.168.38.100:35568
audit(01/11/2020 19:29:10.496:31177) connect    ::ffff:192.168.38.113:80
audit(01/11/2020 19:29:10.500:31178) getsockname        ::ffff:192.168.38.100:35570
audit(01/11/2020 19:29:10.508:31179) connect    ::ffff:192.168.38.113:80
audit(01/11/2020 19:29:10.508:31180) getsockname        ::ffff:192.168.38.100:35572
audit(01/11/2020 19:29:10.516:31181) connect    ::ffff:192.168.38.113:80
audit(01/11/2020 19:29:10.520:31182) getsockname        ::ffff:192.168.38.100:35574
```

Figure 19 – Sample audit parser output

Josh Arey, joshuaarey@gmail.com