



Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Hacker Tools, Techniques, and Incident Handling (Security 504)"
at <http://www.giac.org/registration/gcih>

RapidTriage: Automated System Intrusion Discovery with Python

GIAC (GCIH) Gold Certification

Author: Trenton Bond, trent.bond@gmail.com

Advisor: Hamed Khiabani, Ph.D.

Abstract

Incident handlers may find themselves in situations where they need to validate a potential compromise but do not have administrative access to the systems in question or in situations where many systems need to be triaged quickly. This may leave the incident handler trying to relay commands to a system administrator or taking valuable time to triage each system individually. This communication and initial triage can be time sensitive and may be inaccurate if the data collection commands are not run as directed. This paper introduces the RapidTriage Python tool which can be used to automate intrusion discovery, speeding up the initial triage and ensuring consistency in the collected results across multiple systems and different platforms.

1. Introduction

There are six major incident handling phases typically used to manage information security incidents: preparation, identification, containment, eradication, recovery, and lessons learned. The identification phase of the process is critical as this is where all available information is collected and triaged. “The goal of the identification phase is to gather events, analyze them, and determine whether we have an incident” (Skoudis, 2008). According to the 2013 Verizon Breach Report, some 70% of all breaches in 2012 were discovered and reported by external parties who notified the victim (Verizon, 2013). It seems reasonable based on these statistics to assume that information security practitioners and system administrators will need to be prepared to respond quickly, often with little evidence beyond what is reported. In these situations incident handlers may focus initially on indicators of compromise (IOCs) identified from network firewalls, heuristic netflow data, and network intrusion detection events, but the chain of evidence will usually lead to a system (or set of systems) that must be carefully examined to confidently determine that an incident has occurred. Collecting and analyzing system evidence and mining available event data from suspect systems is an effective technique used to identify IOCs that strengthen the management of the remaining incident handling phases such as containment and eradication.

1.1. The Problem

There are existing tools that can assist a handler with system intrusion discovery such as rootkit identifiers “chrootkit” (Murilo, 2014) and “rkhunter” (Boelen, 2014). Tools like these are specifically designed to examine operating system binaries and configurations for known rootkit residue or modifications. However, they are not designed to collect system state information or system events for contextual analysis by the handler. For example, while investigating a potentially compromised Linux system, incident handlers will likely want to review active network connections to identify any anomalous traffic flows. Another example is the collection and review of command histories like system bash histories for suspicious commands executed by the root account. Rootkit discovery tools will not analyze network connections or report the contents of the bash history files for review.

Although tools like these rootkit discovery tools have a place, a critical component in any incident handler’s arsenal is a set of system intrusion discovery commands that can be used to

harvest events and potential system IOCs. The SANS Institute publishes several cheat sheets with commands that can be used to help with initial system triage including the “Intrusion Discovery Cheat Sheet for Windows” and the “Intrusion Discovery Cheat Sheet for Linux” (SANS, 2014). While these guides (or even a custom discovery cheat sheet) are an invaluable resource for anyone responsible for handling an incident, in practice, there are often logistical issues that may arise. For example, how much time will it take to manually run and collect results for each system command? If triage is required across several systems, will the system sources examined be consistent? What if the incident handler doesn’t have administrative access to run the commands? What if the system administrator or incident handler is not as familiar with a particular operating system platform? What is normal when reviewing systems state and sources?

1.2. A Possible Solution

RapidTriage was developed as one possible solution to the potential system intrusion discovery difficulties handlers face. This Python script automates the collection of critical system information from the following key operating system areas:

- General System Information
- Network State
- Process, Service, and Module State
- Unusual Files, Directories, and Registry Keys
- Scheduled Task Information
- Account and User State
- Histories and Log Data

Organizations “should be prepared beforehand to properly respond to incidents and investigate them in the shortest time possible (Vacca, 2013).” Attackers may be establishing a greater foothold in the environment or may be actively exfiltrating sensitive data while the system administrators and incident handlers try to identify the source of the breach. Often the process of collecting system information is manual and can be time intensive, particularly when multiple systems are suspect. However, the RapidTriage tool may be deployed quickly by a system administrator to many systems at once. The results can then be analyzed relative to other systems to help provide context or to prioritize containment phase efforts and eradication.

Occasionally, incident handlers may not have immediate access to the systems in question and establishing it can cost valuable time. RapidTriage can be given to an authorized system administrator to collect the critical information on their behalf, then provide the results for analysis.

Besides being faster to deploy and providing a way to deal with access barriers, the other benefits of RapidTriage include:

- Ability to add/modify collection commands or event sources as necessary
- Consistent results and output format
- Ability to choose specific operating system areas from which to collect
- Single collection script to maintain for multiple operating systems

2. RapidTriage – A Python Intrusion Discovery Tool

The Python RapidTriage discovery tool architecture consists of the following four major components (Figure 1):

- The User Interface
- Platform Detection
- The Collection Engine
- Reporting

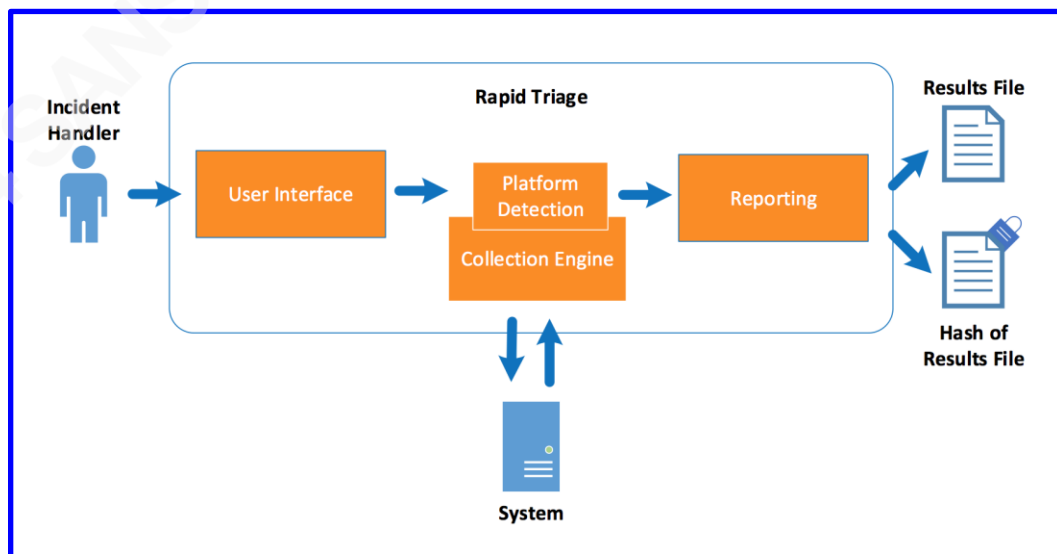


Figure 1

Why Python? While almost any programming language including Java, Visual Basic,

Python, PHP, or Perl could have been used to script the collection of system information. Python was chosen because it is freely available, open source, easy to learn, flexible, and works across Linux, FreeBSD, Windows, and Mac OS X systems. Most of the more popular distributions of Linux come with Python in the default installation. Starting in version 10.2, Mac OS X also has Python installed by default. Python is not installed by default with FreeBSD; though, the package can be easily added using the “`pkg_add -r python`” command (Python – Using on Unix Platforms, 2014). Windows also does not install Python by default; however, there are tools such as the py2exe “Distutils” extension that can convert Python scripts into a Windows executable (Heller, 2014).

2.1. The User Interface

The only RapidTriage interface available to the user is from the command-line. There are several system collection commands that require administrator rights to execute. Thus, RapidTriage must be executed with administrator rights or root privileges. In Linux, FreeBSD, and OS X this can be done with the root account or with “`sudo`” privileges. In Windows this can be done by opening a command terminal as “administrator” and running the RapidTriage script from there.

The “`optparse`” module from the Python standard library is used to handle and parse all command-line options. “`optparse`’s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a help value for each option, and optionally a short usage message for your whole program” (Python – `optparse`, 2014). RapidTriage runtime options and arguments can be viewed by simply not including any arguments or by issuing “`-h, - -help`” (Figure 2).

```

Nyx~/Downloads$ sudo python RapidTriage.py
Usage: python RapidTriage.py [argument(s)] -o <filename>

Options:
-h, --help            show this help message and exit
-o OUTFILE            specified file will contain the results of RapidTriage

Arguments (one or more system areas required):
-a, --all_areas       collect information from all areas
-f, --filesystem      collect filesystem related information
-l, --log_events       collect histories and log data
-n, --net_stats       collect network stats and config information
-p, --process          collect process, service, and module information
-t, --sched_tasks     collect scheduled task information
-u, --user             collect user account and configuration information

Optional:
-m, --md5sum          generate an md5 hash of the results file (<filename>)
                     and place in <filename>-hash
Nyx~/Downloads$

```

Figure 2

There are two required arguments to run the script. The first is a user defined output filename that is required at execution using the “-o <filename>” parameter. The second is one or more systems areas to collect information from using the “-aflnptu” arguments. When the “-a” option is chosen, all system areas will be collected regardless of other system areas selected. The “-m” option is not required, but when chosen at execution the results file will be hashed using md5. The md5 hash will then be stored in a file based on the output filename given with the “-o” parameter. Below are a few examples of how these arguments and options can be used.

1. Collect “all” available Linux system information and store the results in a file named “all_results.txt” (Figure 3).

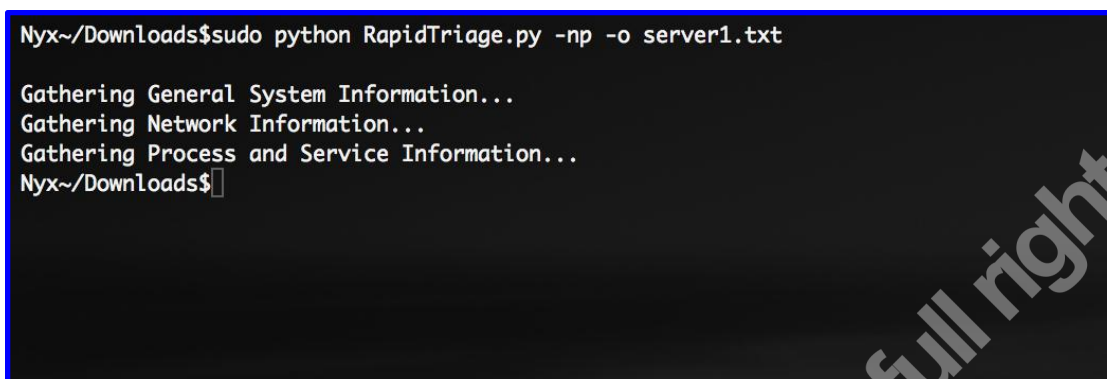
```

[Sun Feb 02 21:36:24 ~/Desktop ] $ sudo python RapidTriage.py -a -o all_results.txt
Gathering General System Information...
Gathering Network Information...
Gathering Process and Service Information...
Gathering Information on Unusual Files...
Gathering Scheduled Task Information...
Gathering Account and User Information...
Gathering History Files and Log Data...
[Sun Feb 02 21:36:39 ~/Desktop ] $

```

Figure 3

2. Collect just network and process state information from OS X and store the results in a file named “server1.txt” (Figure 4).

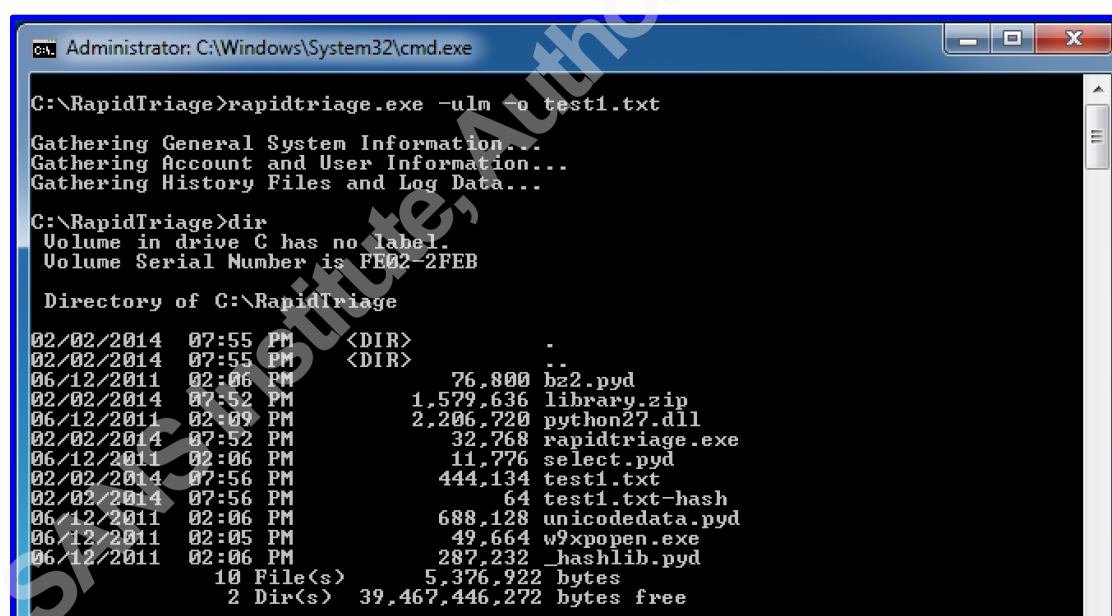


```
Nyx~/Downloads$ sudo python RapidTriage.py -np -o server1.txt

Gathering General System Information...
Gathering Network Information...
Gathering Process and Service Information...
Nyx~/Downloads$
```

Figure 4

3. Collect only user state and log events from a Windows system via a command terminal running as “administrator”. Store the results in a file named “test1.txt”. Hash the results file “test1.txt” and store the hash value in “test1.txt-hash” (Figure 5).



```
Administrator: C:\Windows\System32\cmd.exe

C:\RapidTriage>rapidtrriage.exe -ulm -o test1.txt

Gathering General System Information...
Gathering Account and User Information...
Gathering History Files and Log Data...

C:\RapidTriage>dir
Volume in drive C has no label.
Volume Serial Number is FE02-2FEB

Directory of C:\RapidTriage

02/02/2014  07:55 PM  <DIR>          .
02/02/2014  07:55 PM  <DIR>          ..
06/12/2011  02:06 PM             76,800  bz2.pyd
02/02/2014  07:52 PM          1,579,636  library.zip
06/12/2011  02:09 PM          2,206,720  python27.dll
02/02/2014  07:52 PM          32,768  rapidtrriage.exe
06/12/2011  02:06 PM          11,776  select.pyd
02/02/2014  07:56 PM          444,134  test1.txt
02/02/2014  07:56 PM             64  test1.txt-hash
06/12/2011  02:06 PM          688,128  unicodedata.pyd
06/12/2011  02:05 PM          49,664  w9xpopen.exe
06/12/2011  02:06 PM          287,232  _hashlib.pyd
                10 File(s)          5,376,922 bytes
                2 Dir(s)  39,467,446,272 bytes free
```

Figure 5

2.2 Platform Detection

To execute the appropriate system collection commands, RapidTriage must know what the underlying platform is. The Python “sys” library and “platform” module are used in the script to identify the operating system. The following are the supported systems values that Python can test for and the specific values that RapidTriage uses to evaluate whether the system is Windows, OS X, FreeBSD, or Linux (Table 1 and Figure 6):

System	“platform” value
Linux (2.x and 3.x)	linux<major version>
Windows	win32
Mac OS X	darwin
FreeBSD	freebsd<major version>

Table 1

```
#####
# Platform Detection
#####

from sys import platform as _platform
if _platform.startswith('linux'):
    os_type="linux"
elif _platform.startswith('freebsd'):
    os_type="freebsd"
elif _platform == "darwin":
    os_type="osx"
elif _platform == "win32":
    os_type="windows"
```

Figure 6

Note the “startswith()” function when evaluating Linux. “Since lots of code check for `sys.platform == 'linux2'`, and there is no essential change between Linux 2.x and 3.x, `sys.platform` is always set to 'linux2', even on Linux 3.x. In Python 3.3 and later, the value will always be set to 'linux', so it is recommended to always use the startswith idiom...” (Python – `sys.platform`, 2014). When evaluating for FreeBSD, the “startswith()” function is also used because the major release version is always appended. In the case of Windows, a value of “win32” is evaluated, but what about base64 installations? A “sys” patch was introduced in May 2000 so that the platform function would always return “win32” even for 64 bitwise Windows systems (Mick, 2000).

2.3 The Collection Engine

The main objective of RapidTriage is to collect information from key operating system sources that help incident handlers identify IOCs. The collection portion of the script is broken up into major source areas such as General System Information, Network State, User Account State, etc. Within each area are four specific command lists for each Linux, OS X, FreeBSD, and Windows such as those shown for the “Network State” area in figure 7 below.

```

if os_type is "linux":
    cmds = [
        'Network Interface Configuration::ifconfig -a',
        'Network Interfaces in Promiscuous Mode::ip link |grep PROMISC',
        'Route Table::netstat -rn',
        'Firewall Configuration::iptables -L',
        'ARP Table::arp -e',
        'Listening Ports and Associated Command::lsof -i',
        'Active Network Connections::netstat -natp',
        'Count of Half Open Connections::netstat -ant |grep "svn_recv" |wc -l',
        'Count of Open Connections::netstat -ant |grep "established" |wc -l',
        '/etc/hosts Contents::cat /etc/hosts'
    ]
elif os_type is "windows":
    cmds = [
        'Network Interface Configuration::ipconfig /all',
        'Route Table::route print',
        'Firewall Configuration::netsh advfirewall firewall show rule all',
        'ARP Table::arp -a',
        'Listening Ports::netstat -ano |find /i "listening"',
        'Established Connections::netstat -ano |find /i "established"',
        'Active/Listening Connections and Associated Command::netstat -anob',
        'Count of Half Open Connections::netstat -ano |find /i /c "syn_received"',
        'Count of Open Connections::netstat -ano |find /i /c "established"',
        '/etc/hosts Contents::type %SystemRoot%\System32\Drivers\etc\hosts',
        'Sessions Open to Other Systems::net use',
        'Local File Shares::net view \\\\127.0.0.1',
        'Available Local Shares::net share',
        'Open Sessions with Local Machine::net session'
    ]
elif os_type is "osx":
    cmds = [
        'Network Interface Configuration::ifconfig -a',
        'Route Table::netstat -rn',
        'Firewall Configuration::ipfw list',
        'ARP Table::arp -a',
        'Listening Ports and Associated Command::lsof -i |grep -i listen',
        'Active Network Connections::lsof -i tcp',
        'Count of Half Open Connections::netstat -ant |grep -i "svn_recv" |wc -l',
        'Count of Open Connections::netstat -ant |grep -i "established" |wc -l',
        '/etc/hosts Contents::cat /etc/hosts'
    ]
elif os_type is "freebsd":
    cmds = [
        'Network Interface Configuration::ifconfig -a',
        'Route Table::netstat -rn',
        'Firewall Configuration::ipfw list',
        'ARP Table::arp -a',
        'Listening Ports and Associated Command::lsof -i |grep -i listen',
        'Active Network Connections::lsof -i tcp',
        'Count of Half Open Connections::netstat -an |grep -i "svn_recv" |wc -l',
        'Count of Open Connections::netstat -an |grep -i "established" |wc -l',
        '/etc/hosts Contents::cat /etc/hosts'
    ]

```

Figure 7

Each command in the Python lists is specifically formatted as follows:

<description>::<command>****

Additional system commands can easily be added to any of the lists or modified by the user if they have a preferred command. For example, the following description and command could be added to the “Network State” section in the Linux commands list to get a count of the number of network connections in “closed” state.

Count of Closed Connections::netstat -ant |grep -i “closed” |wc -l

The Python “subprocess” module and “Popen” class are integral to processing the system command lists. “The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes”. “The underlying process creation and management in this module is handled by the Popen class” (Python – subprocess, 2014). A single script function called “run_cmds” was developed to handle the processing of each system command. The description is split from the actual command so it can be run and the results written to the user provided filename in a standard output format (Figure 8).

```
# Execute given commands and write the results to the user specified outfile
def run_cmds(list_cmds):
    for cmd in list_cmds:
        split_cmd=cmd.split(":")
        outfile.write("\n")
        outfile.write(timestamp()+"\t"+split_cmd[0]+":\n")
        outfile.write("-----\n\n")
        p = subprocess.Popen(split_cmd[1], stderr=subprocess.STDOUT, stdout=subprocess.PIPE, shell=True)
        for line in p.stdout.readlines():
            outfile.write("\t"+line)
```

Figure 8

Another important RapidTriage script function is the “timestamp”. This function is used to add a timestamp to the results file right before the execution of each system command. This can be helpful when troubleshooting the script; for instance, when the script is taking a significant amount of time to run. In such cases, the suspect command can be modified and removed, or the entire area can be skipped using the command-line arguments (Figure 9).

```
# Provide a timestamp when necessary
def timestamp():
    now = "["+time.strftime("%H:%M:%S")+"]"
    return now
```

Figure 9

The following section is a closer look at each system area in RapidTriage and some of the

highlights from the collection engine portion of the script.

2.3.1 General System Information

The General System area is unique in that the system commands included here are executed regardless of the command-line arguments issued. If the script successfully runs it will always return the results of this system area. It also cannot be selected independently of other system area arguments. At least one other system area must be selected at run time. At a minimum the system hostname, effective user (whoami), and system type (uname -a) are all collected (Figure 10). This information is important for the incident handler to have during an investigation (or as a reminder later) to document what system(s) the information was collected from, by what user, and the associated operating system. This data is of particular importance to collect when the incident handler is not the user running the script.

```
if os_type is "linux":
    cmds = [
        'System Name::hostname',
        'Effective User::whoami',
        'Runlevel::runlevel',
        'System Type::uname -a'
    ]
elif os_type is "windows":
    cmds = [
        'System Name::hostname',
        'Effective User::whoami',
        'System Type::for /F "delims= tokens=1-2" %a in (\wmic os get Caption /format:list^|find "Caption"\') do @echo %b'
    ]
elif os_type is "osx":
    cmds = [
        'System Name::hostname',
        'Effective User::whoami',
        'System Type::uname -a'
    ]
elif os_type is "freebsd":
    cmds = [
        'System Name::hostname',
        'Effective User::whoami',
        'System Type::uname -a'
    ]
```

Figure 10

Besides the standard information collected above, there are several more system-specific commands that have been included to seed the General System section (Figure 11). For example, file system disk space, memory usage, and environment variables are also collected by default. These kinds of commands are useful from an overall system health perspective and may also yield evidence indicative of a compromise. A file system with no disk space should be further investigated for an operational problem or possibly an IOC as attackers may be using the system to store significant amounts data.

Trenton Bond, trent.bond@gmail.com


```

if os_type is "linux":
    cmds = [
        'Filesystem Disk Space Usage::df -al',
        'Memory Usage::vmstat -s',
        'Uptime and Load Average::uptime',
        'Environment Variables::export'
    ]
elif os_type is "windows":
    cmds = [
        'Filesystem Disk Space Usage::wmic logicaldisk get caption, size, freespace',
        'Memory Usage::wmic os get totalvisiblememorysize, freephysicalmemory, totalvirtualmemorysize, freevirtualmemory/format:list',
        'Load Average::wmic path win32_processor get deviceid, loadpercentage',
        'Environment Variables::set'
    ]
elif os_type is "osx":
    cmds = [
        'Filesystem Disk Space Usage::df -al',
        'Memory Usage::vm_stat',
        'Uptime and Load Average::uptime',
        'Environment Variables::env'
    ]
elif os_type is "freebsd":
    cmds = [
        'Filesystem Disk Space Usage::df -al',
        'Memory Usage::vmstat -s',
        'Uptime and Load Average::uptime',
        'Environment Variables::env'
    ]

```

Figure 11

It is also important to note that while there is a standard function for processing the command lists that could have been used, we actually handle the command processing directly in this section. This was done because of the unique output format that was desired just for this system area.

2.3.2 Network State

When collecting network related system information there are several important system commands. This section was, of course, seeded with many of the example commands found from the SANS Intrusion Discovery reference cheat sheets. As with the other sections, commands here can be modified or added as desired by the user by simply changing the command list for the appropriate system type. For the RapidTriage script, a number of additional network state commands were included. Below are a few examples of commands that were added beyond the SANS guides.

The following command added to the RapidTriage script will pull the firewall configuration for a Linux system. This information could be useful during initial triage to understand what kind of network access would be allowed into or out of the system. Is a system firewall enabled in the first place? Did the attacker add a rule to exfiltrate data or to allow an outbound call home?

Trenton Bond, trent.bond@gmail.com

Firewall Configuration::iptables -L

This Windows command is used to collect the contents of the system /etc/hosts file. The results of which can be reviewed for suspicious entries or interesting mappings that may be used to help identify a compromise. “For instance, some malware spreads to computers with shared accounts or targets systems that are listed in the ‘/etc/hosts’ file on the compromised system” (Malin, 2008).

/etc/hosts Contents::type %SystemRoot%\System32\Drivers\etc\hosts

Often looking at established network connections to suspicious IP addresses can produce IOCs; but what if they all look like random foreign IP addresses? If a typical number of established connections happen to be known before an investigation, the count of established connections can be an interesting barometer. The following OS X command grabs the total number of established connections and writes it to the results file.

Count of Open Connections::netstat -ant |grep -i "established" |wc -l**2.3.3 Process, Services, and Module State**

This section of the collection script is for handling the harvest of process and service information. It’s also for the collection of information about loaded kernel modules in the case of Linux, FreeBSD, and OS X as well as the collection of installed packages and patches. Why kernel modules? “Kernel rootkits usually utilize loadable kernel modules to modify kernel functionality without requiring a kernel recompilation” (Mookhey, 2005). A review of the loaded kernel modules (LKMs) may provide clues of a compromise based on the sizes or other attributes retrieved. Below is an example of how RapidTriage extracts a list of loaded kernel extensions from OS X.

Loaded Kernel Extensions::kextstat

To collect all of the open files associated with each process ID in Linux the following command has been included in the default RapidTriage command list. These open files could help lead an incident handler to backdoors or other malicious files.

Open Files Associated with each PID::ps aux |awk 'NR!=1 {print \$2}' |while IFS= read pid; do echo " "; ps \$pid; lsof -p \$pid; done**2.3.4 Unusual Files, Directories, and Registry Keys**

Trenton Bond, trent.bond@gmail.com

The next section of RapidTriage is meant to be a collection of unusual files, unusual directories, and key Windows registry keys used to assist the handler or system administrator with intrusion discovery. The EC-Council suggests in their book “Computer Forensics: Investigating Data and Image Files” that one effective method for detecting steganography is looking for large files. “The investigator should look for large files in the system, as they can be used as carrier files for steganography. If the investigator finds a number of large duplicate files, then it is possible that are used as carrier files” (EC-Council, 2009). Below is a Windows “for loop” that identifies files greater than 50M. The user can adjust the target size as necessary by simply adjusting the value in the command.

Large Files >50M::for /R c:\ %i in (*) do @if %~zi gtr 50000000 echo %i %~zi

One of the favorite tactics of attackers is to take advantage of SUID and GUID root files. These types of system files are allowed to function with root (user or group) rights instead of the privileges of the user who executed the program. Incident handlers will often want to know what system files have these bits set. The following Linux SUID find command introduced in the SANS Linux Intrusion Discovery guide was modified in RapidTriage to include files with the GUID bit also set.

Files with SUID/GUID bits set::find / -type f \(-perm +4000 -o -perm +2000 \) -exec ls -l {} \; 2>/dev/null

2.3.5 Scheduled Task Information

As is suggested in the SANS intrusion discovery guides, system scheduled tasks are a vital system area from which to collect information and to triage. “Intruders may create a cron job on a compromised system to periodically launch a backdoor or beacon to enable them to regain entry. This is an old but extremely popular and effective way to maintain access” (Casey, 2009). For Windows systems, scheduled tasks and startup items are included in the tool and collected with the following commands.

Scheduled Tasks::schtasks

Startup Items::wmic startup list full

2.3.6 Account and User State

Usually system intrusion discovery would not be complete without a thorough review of

the current state of user accounts and logins. This section of the RapidTriage script will, with the seeded system commands, collect information about who is currently logged into the systems, local system accounts, the last login attempts (successful and failed), and group membership. In OS X the following two commands are seeded in the script and result in the retrieval of all local user accounts and groups.

User Accounts::dscacheutil -q user

Groups:: dscacheutil -q group

Unsuccessful Linux login attempts are typically logged to the (“/var/log/btmp”) log file. Where this binary file exists, it can be read with the “lastb” system command and is collected by RapidTriage by default.

Unsuccessful Login Attempts::lastb

2.3.7 Histories and Log Data

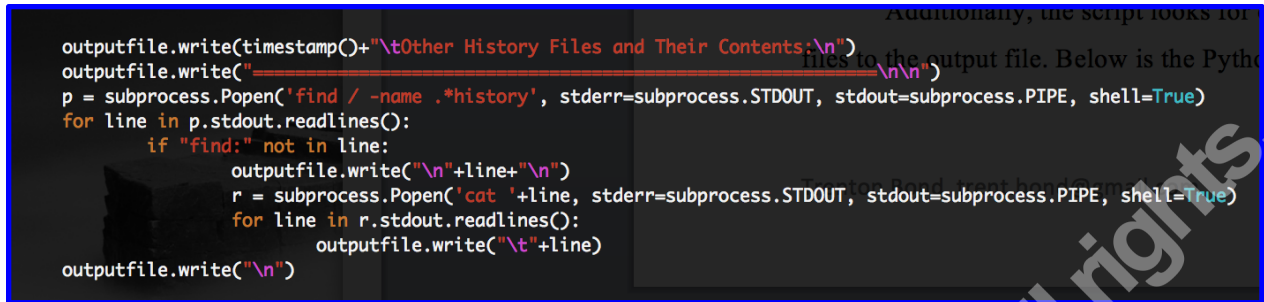
The last section of the collection engine is responsible for extracting history file data and log data. Because Windows, FreeBSD, OS X, and Linux all have unique log data sources and history files, they are handled individually in the script based on the operating system type. Similar to the General System Information collection, some of the command execution is processed here rather than with the general “run_cmds” function due to the unique output format that was desired.

System history files are always interesting places for incident handlers to look for IOCs. “A savvy criminal may open a terminal and use UNIX commands in the commission of a crime. This is most likely where an examiner will find evidence of a network intrusion. The bash shell ... is the command line to a file named .bash_history” (Olivier, 2006). In RapidTriage the script collects this information from FreeBSD systems with the following history commands.

Bash History::bash -i -c "history -r; history"

C Shell History::csh -i -c "history -r; history"

Besides bash histories, the script also looks for other system history files and writes the contents of the files to the output file. Below is the Python code used for history file collection in OS X (Figure 12).



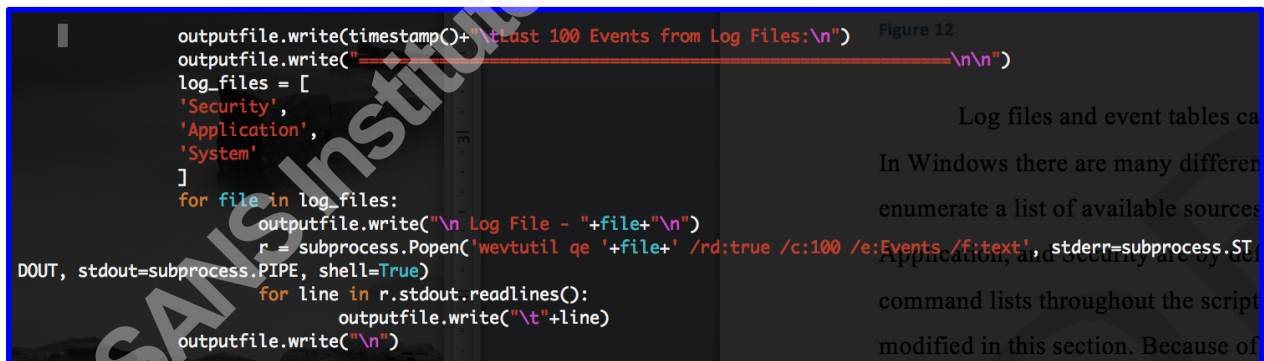
```

outputfile.write(timestamp()+"\tOther History Files and Their Contents:\n")
outputfile.write("-----\n\n")
p = subprocess.Popen('find / -name .*history', stderr=subprocess.STDOUT, stdout=subprocess.PIPE, shell=True)
for line in p.stdout.readlines():
    if "find:" not in line:
        outputfile.write("\n"+line+"\n")
        r = subprocess.Popen('cat '+line, stderr=subprocess.STDOUT, stdout=subprocess.PIPE, shell=True)
        for line in r.stdout.readlines():
            outputfile.write("\t"+line)
outputfile.write("\n")

```

Figure 12

Log files and event tables can also be of tremendous benefit while hunting system IOCs. In Windows there are many different event repositories and the “wevtutil el” command can be used to enumerate a list of available sources. The traditional Windows event sources like System, Application, and Security are by default used in the RapidTriage script. Like the system command lists throughout the script, the log file sources for any operating system type can be modified in this section. Because of the enormous number of system events generated by Windows, only the last 100 are collected and written to the output file using this Python logic and code, but that can also be modified as necessary (Figure 13).



```

outputfile.write(timestamp()+"\tLast 100 Events from Log Files:\n")
outputfile.write("-----\n\n")
log_files = [
    'Security',
    'Application',
    'System'
]
for file in log_files:
    outputfile.write("\n Log File - "+file+"\n")
    r = subprocess.Popen('wevtutil qe '+file+' /rd:true /c:100 /e:Events /f:text', stderr=subprocess.STDOUT, stdout=subprocess.PIPE, shell=True)
    for line in r.stdout.readlines():
        outputfile.write("\t"+line)
    outputfile.write("\n")

```

Figure 13

2.4 Reporting

The reporting component of Rapid Triage is straightforward. The results of the collection engine are written to the user-specified output file as they are collected. The user-specified file is opened for writing with the “open” function before collection begins and the “close” function after collection. To write to the specified output file, the “write” function is used. Below are examples of how opening, writing, and closing are done in the script.

Open

```
# Open the user specified outfile
outfile=open(options.outfile,"a")
```

Figure 14

Close

```
outfile.write("""
#####
#
#   General System Information
#
#####
""")
```

Figure 15

Write

```
#Close the user specified outfile
outfile.close()
```

Figure 16

The outfile is organized the same way that the script is organized. The “General System” is at the beginning of the report followed by the collected information from the other system areas selected at runtime.

When the md5 hash command line option is selected at runtime, a new hash file is opened after all of the results are collected. The outfile is reopened, read and then hashed using the Python “hashlib” module and the “md5” construct. The hash is saved in the new hash file and then closed.

```
#MD5 hash the output file
if options.hash:
    hashfile=open(options.outfile+"-hash","a")
    results=open(options.outfile,'rb')
    hashresults=results.read()
    results.close()
    md5 = hashlib.md5()
    md5.update(hashresults)
    md5sum=md5.hexdigest()
    hashfile.write(md5sum)
    hashfile.close()
```

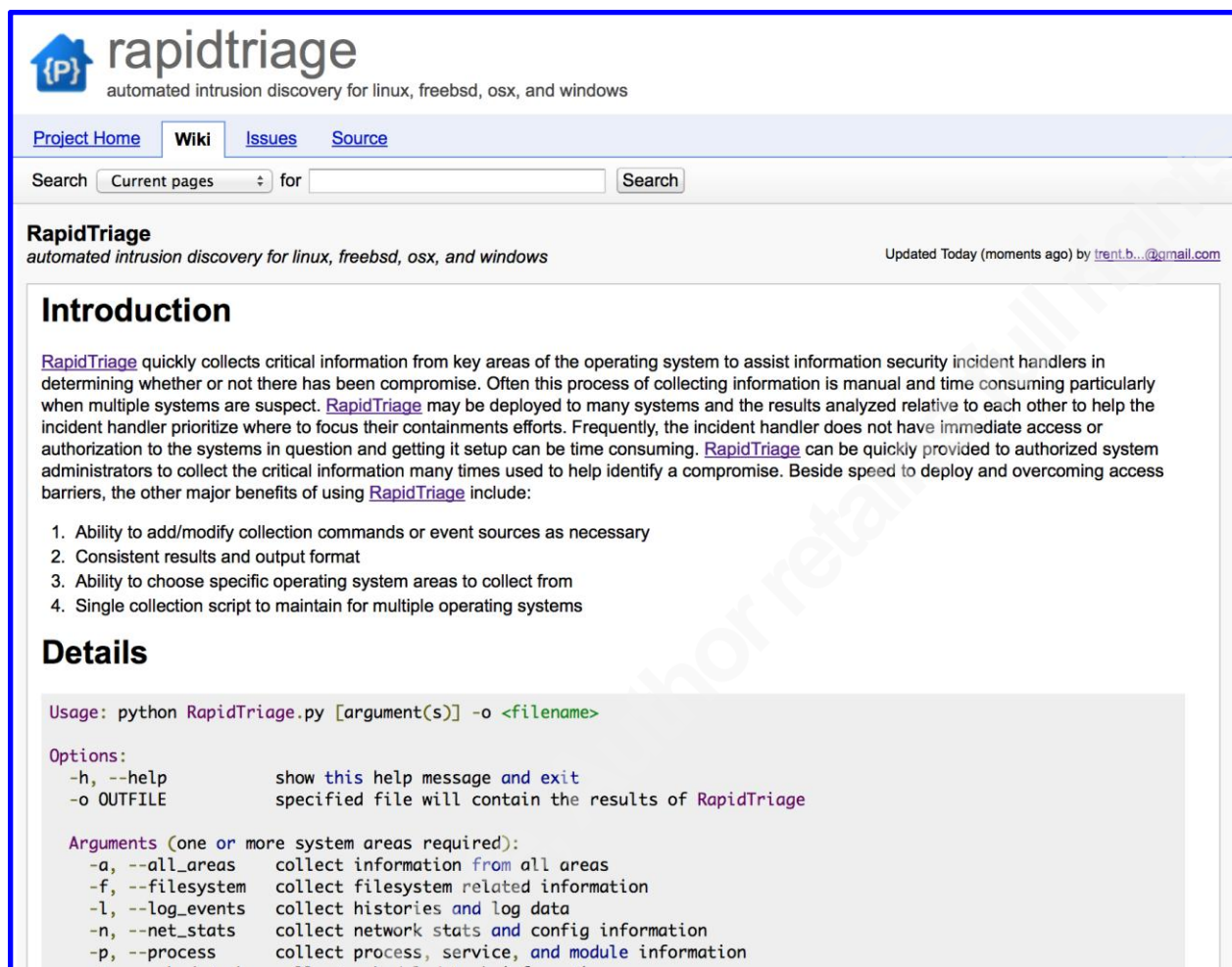
Figure 17

3. Use and Support

The real value of an intrusion discovery tool like RapidTriage is when it's used in comparison with a known "normal" state. Concerning the intrusion discovery cheat sheets, Ed Skoudis suggests, "they require the System Administrators to know the 'normal' state of their systems. The Cheat Sheets identify common areas of deviation from normal that a knowledgeable Sys Admin can spot. However, without a good gut feel of the normal state, these techniques won't work" (Skoudis, 2008). This "normal" state can be established with even a single archived RapidTriage result file or by scheduling a task to constantly run the script. The review and comparison of the results can then be operationalized to highlight anomalies and IOCs much more quickly.

RapidTriage code has been posted to Google Code as a project for the benefit and use of the community (Figure 18). The latest code revisions and documentation for the project can be found at the following location.

<https://code.google.com/p/rapidtriage/wiki/RapidTriage>



rapidtrriage
automated intrusion discovery for linux, freebsd, osx, and windows

Project Home Wiki Issues Source

Search Current pages for Search

RapidTriage
automated intrusion discovery for linux, freebsd, osx, and windows

Updated Today (moments ago) by trent.bond@gmail.com

Introduction

[RapidTriage](#) quickly collects critical information from key areas of the operating system to assist information security incident handlers in determining whether or not there has been compromise. Often this process of collecting information is manual and time consuming particularly when multiple systems are suspect. [RapidTriage](#) may be deployed to many systems and the results analyzed relative to each other to help the incident handler prioritize where to focus their containments efforts. Frequently, the incident handler does not have immediate access or authorization to the systems in question and getting it setup can be time consuming. [RapidTriage](#) can be quickly provided to authorized system administrators to collect the critical information many times used to help identify a compromise. Beside speed to deploy and overcoming access barriers, the other major benefits of using [RapidTriage](#) include:

1. Ability to add/modify collection commands or event sources as necessary
2. Consistent results and output format
3. Ability to choose specific operating system areas to collect from
4. Single collection script to maintain for multiple operating systems

Details

Usage: `python RapidTriage.py [argument(s)] -o <filename>`

Options:

- h, --help show this help message and exit
- o OUTFILE specified file will contain the results of [RapidTriage](#)

Arguments (one or more system areas required):

- a, --all_areas collect information from all areas
- f, --filesystem collect filesystem related information
- l, --log_events collect histories and log data
- n, --net_stats collect network stats and config information
- p, --process collect process, service, and module information

Figure 18

Currently RapidTriage supports Python version 2.7 and is known to work with the following versions of Windows, FreeBSD, Linux, and OS X. As already discussed, the Python script command lists and log locations can be easily modified to support other versions or even additional operating systems.

Operating Systems:

- Linux (2.6.x)
- Mac (OSX 10.2.x)
- Windows 7
- FreeBSD 9.2

This tool will not collect all the information an incident handler will need to identify an incident. If there are other system security tools installed such as rootkit hunters, consider using this tool to execute them and then extract information from them at the same time.

4. Conclusion

System intrusion discovery and the identification of IOCs is a critical component of the identification phase of incident handling. These time sensitive efforts can be impeded by access restrictions, expertise, no understanding of normalcy, and inconsistency.

The RapidTriage Python tool detects the operating system platform then automates the retrieval of information from key system areas. This automation allows the script to be handed to less skilled administrators or others who already have administrative system access to quickly collect data on behalf of the incident handler. Additionally, it can be deployed consistently to many systems at once and with a common output format that can be used for comparison purposes. The script was designed to be modified by the incident handler with preferred commands or event source locations. If used to establish a normal state before an incident, the output results can be compared by an incident handler to more quickly highlight system indicators of compromise.

5. References

- Boelen, Michael (2014). *rkhunter – rootkit scanner*. Retrieved from website:
<http://www.rootkit.nl>
- Casey, Eoghan (2009). *Handbook of Digital Forensics and Investigation*. Burlington, MA:
 Elsevier Academic Press
- EC-Council (2009). *Computer Forensics: Investigating Data and Image Files*. Clifton Park, NY:
 Cengage Learning
- Heller, Thomas & Retzlaff, Jimmy & Hammond, Mark (2014). *py2exe – distutils extensions
 which convert Python scripts in executable Windows programs*. Retrieved from website:
<http://www.py2exe.org>
- Malin, Cameron H. & Casey, Eoghan & Aquilina, James M. (2008). *Malware Forensics:
 Investigating and Analyzing Malicious*. Burlington, MA:Syngress
- Mick, Trent (2000). *[Patches] use “win32” for sys.platform on Win64*. Retrieved from website:
<https://mail.python.org/pipermail/patches/2000-May/000648.html>
- Mookhey, K.K. & Burghate, Nilesh (2005). *Linux Security, Audit and Control Features*. Rolling
 Meadows, IL:ISACA
- Murilo, Nelson & Steding-Jessen, Klaus (2014). *chkrootkit – locally checks for signs of a rootkit*.
 Retrieved from website: <http://www.chkrootkit.org>
- Olivier, Martin S. & Sheno, Sujeet (2006). *Advances in Digital Forensics II*. New York, New
 York: Springer
- Python Software Foundation (2014). *optparse – Parser for command line options*. Retrieved
 from Python website: <http://docs.python.org/2/library/optparse.html>
- Python Software Foundation (2014). *Python – Using on Unix Platforms*. Retrieved from Python
 website: <http://docs.python.org/2/using/unix.html>
- Python Software Foundation (2014). *sys.platform – Function to determine platform*. Retrieved
 from Python website: <http://docs.python.org/2/library/sys.html>
- Python Software Foundation (2014). *subprocess – Interprocess Communication and Networking*.
 Retrieved from Python website: <http://docs.python.org/2/library/subprocess.html>
- SANS Institute (2014). *Windows - Intrusion Discovery Cheat Sheet v2.0*. Retrieved from SANS
 website: <http://pen-testing.sans.org/retrieve/windows-cheat-sheet.pdf>

SANS Institute (2014). *Linux - Intrusion Discovery Cheat Sheet v2.0*. Retrieved from SANS website: <http://pen-testing.sans.org/retrieve/linux-cheat-sheet.pdf>

Skoudis, Ed (2008). *SANS Security 504: Hacker Techniques, Exploits and Incident Handling*. SANS course

Vacca, John R. (2013). *Managing Information Security*. Waltham, MA:Elsevier

Verizon (2013). *2013 Data Breach Investigation Report*. Retrieved from Verizon website: <http://www.verizonenterprise.com/DBIR/2013/>