CROWDSTRIKE

CrowdStrike Services

VMware vSphere Adversary Hunting Guide Quick Reference Guide

Version 1.0

Dated: October 13, 2025

Copyright © 2025 CrowdStrike, Inc. All rights reserved.

Table of Contents

Overview 4
Containment Actions 5
Threat Hunting and Investigation 6
Network Traffic Analysis 6
Investigate Traffic from Edge Network Appliances 6
Investigate Network Traffic from vCenter 6
VirtualGhost Hunting 6
Python Script 6
PowerShell Script 7
VMware Log Analysis 7
Secure Shell (SSH) 7
DNS Requests 7
vCenter Malware: *BRICKSTORM* 8
vCenter Malware: Webshells 10
ESXi Malware: *Junction* 11
Forensic Evidence Collection on vSphere 12
Support Bundle Collection 12
ESXi Support Bundle Collection 12
vCenter Support Bundle Collection 13
vCenter Image Collection 13
Recommendations for Log Retention and Monitoring 15
Forward vCenter and ESXi Logs to a SIEM 15
Schedule and Retain Recurring Backups of the PostgreSQL Database 15
STS Token Monitoring 16
Log DNS Queries 16
SSH Key Auditing 16
Monitor Internal network Traffic 17
CrowdStrike Falcon NG-SIEM 17
Network Segmentation 18
Appendix A: Recommended auditd Configuration Modifications 19
Appendix B: GHOSTHUNTER Python Script 20
Script Actions 20

Extracting the Script 21
Executing the Script 21
Example Invocation 21
Script Contents 22

Overview

CrowdStrike has observed China-nexus adversary groups exploiting publicly accessible edge network appliances to gain initial access to victim environments. During its response to these intrusions, CrowdStrike has observed these adversary groups move quickly to target the victim's VMware vSphere infrastructure, including vCenter servers and ESXi hypervisors hosting virtualized systems, to achieve persistent access to the victim environment.

Adversaries will often target VMware vSphere infrastructure and the datastores that contain the virtualized systems' local storage. By impacting ESXi hosts, threat actors can increase the scope of affected systems within a target environment while avoiding traditional endpoint protection solutions deployed to the hosted virtual machines, by interacting directly with virtual machines (VMs) from the hypervisor. Adversaries can also target data stored within VMs, and exfiltrate it directly from vSphere infrastructure.

This Quick Reference Guide (QRG) provides instructions on how to hunt for China-nexus adversary activity within VMware vSphere clusters based on CrowdStrike's observations. Additionally, this QRG highlights additional logging considerations in advance of an intrusion.

For broad guidance on general evidence collection, hardening, and real-time monitoring best practices, please reference CrowdStrike's VMware vSphere Evidence Collection and Containment QRG and CrowdStrike's VMware vSphere Hardening Guide QRG.

Containment Actions

In instances where VMware infrastructure is determined to be compromised, CrowdStrike recommends the following be taken into consideration:

- Do not restore VMs to known compromised vSphere environments until any malicious processes on either vCenter or ESXi have been identified and killed.
- When possible, rebuild ESXi hosts from a verified clean image into separate and isolated infrastructure and bring assets back into the isolated cluster.
- Restrict access to ESXi hosts to a small number of approved systems. Additionally, ensure all systems granted ESXi access have Endpoint Detection and Response (EDR) software installed and running.
- Ensure SSH access to the ESXi systems is disabled where possible.
    - Enable **Normal Lockdown Mode** to further restrict access.
- Ensure each ESXi host has a unique `root` password and ensure this is different to `root` and administrative credentials in vCenter.
    - These passwords should be strong and complex using a combination of letters, special characters, and numbers. Avoid using dictionary words and "1337" speak. Random passwords are ideal.
- Explore enabling the `execInstalledOnly` option, which is designed to prevent binaries being executed on the system that weren't signed by VMware. This will require a reboot of the system to enable and if using third-party software on the system it could cause issues with said software. However, enabling this option will help prevent non-signed binaries from executing on ESXi systems. Refer to the Broadcom support document on this option for more details.[1]
- Rotate credentials for the `vpxuser` account for each ESXi host. To do this, temporarily change the parameter `VirtualCenter.VimPasswordExpirationInDays` to `1` then wait one day for the change to take effect.[2] Once the passwords have expired and been changed automatically by vCenter, set this parameter back to its default value (typically `30`).
- Rotate vCenter Security Token Service (STS) Certificate.[3]
- Remove `vpxuser` shell access on vSphere 8+.[4]
- Restrict outbound internet access from the vSphere environment, including ESXi hosts and vCenter appliances. For more information, see Network Segmentation.

---

[1] https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/7-0/vsphere-security-7-0/securing-esxi-hosts/securing-the-esxi-configuration/managing-a-secure-esxi-configuration/enable-or-disable-the-execinstalledonly-enforcement-for-a-secure-esxi-configuration.html

[2] https://knowledge.broadcom.com/external/article/320773/configure-expiry-settings-for-the-vpxuse.html

[3] https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/8-0/refresh-an-sts-certificate-using-the-vsphere-client.html

[4] https://knowledge.broadcom.com/external/article/396941/to-deactivate-shell-access-for-nonroot-e.html

Threat Hunting and Investigation

Evidence of China-nexus adversary groups targeting an organization can be found in a variety of VMware and infrastructure logs adjacent to VMware related systems.

NETWORK TRAFFIC ANALYSIS

INVESTIGATE TRAFFIC FROM EDGE NETWORK APPLIANCES

CrowdStrike has observed China-nexus adversary groups moving laterally from compromised edge network appliances directly to VMware vSphere infrastructure via secure shell (SSH). Review all traffic from edge network appliances, with a focus on management interfaces, to ensure it is expected activity with a focus around traffic to VMware vSphere infrastructure.

Management interfaces for edge devices should not be internet-facing, as this greatly increases their attack surfaces.

INVESTIGATE NETWORK TRAFFIC FROM VCENTER

CrowdStrike has responded to intrusions that have included significant volumes and breadth of network traffic emanating from compromised vCenter Server Appliance (vCSA) hosts. This has included targeting of internal services, such as source code, wikis, ticketing systems, and build infrastructure. Furthermore, adversaries may proxy network traffic to external services, such as Microsoft 365, with the purpose of obscuring the true source of this malicious activity and evading location-based authentication controls and alerts (e.g., avoid impossible travel alerts, leverage Microsoft Entra ID trusted locations, etc.).

Any outbound network traffic from a vCenter system to infrastructure that is not directly related to vSphere, such as Broadcom telemetry or other ESXi hosts on legitimate management ports, should be considered suspicious and will warrant further investigation.

VIRTUALGHOST HUNTING

CrowdStrike has observed sophisticated threat actors leveraging the ESXi shell to surreptitiously power on VMs without the use of the ESXi or vCenter web consoles. Notably, these VMs are not registered, and do not appear if an administrator logs in to a web console. CrowdStrike named this technique *VirtualGHOST*. **VMs created in this manner are known as ghost VMs, as they can be used by threat actors to maintain hidden persistence within a vSphere environment, rendering other remediation methods ineffective**.

PYTHON SCRIPT

CrowdStrike has previously developed a script, `Detect-VirtualGHOST.ps1`, that leverages the VMware PowerCLI PowerShell library to detect currently running ghost VMs within a vSphere environment.

As a result of continued Threat Actor operations leveraging the *VirtualGHOST* technique, CrowdStrike has additionally developed a Python script called `GHOSTHUNTER` that can be used to triage a vCenter server for signs of compromise. An initial version of `GHOSTHUNTER` and instructions for use has been attached to this document and can be found in Appendix B: GHOSTHUNTER Python Script.

The `GHOSTHUNTER` script complements `Detect-VirtualGHOST.ps1` by **detecting the presence of ghost VMs that are not currently running**, as well as performing several other checks to identify a **wide range** of vCenter compromise and ghost VM scenarios.

If a threat actor accessed the vSphere environment during its intrusion, CrowdStrike strongly recommends running the `GHOSTHUNTER` script and reviewing its output. If any ghost VMs are detected or you require assistance in interpreting the results, inform your CrowdStrike team immediately to discuss next steps to preserve evidence and plan remediation.

POWERSHELL SCRIPT

`Detect-VirtualGHOST.ps1` leverages the VMware PowerCLI PowerShell library to detect **currently running** ghost VMs within a vSphere environment. If a threat actor accessed the vSphere environment during its intrusion, CrowdStrike strongly recommends running this triage script and reviewing its output. If any ghost VMs are detected, inform your CrowdStrike team immediately to discuss next steps to preserve evidence and plan remediation.

The output of this script should be carefully analyzed. The VMware Cluster Service (vCLS) can trigger false positive outputs, typically shown as VMs with names starting with `vCLS` followed by a globally unique identifier, CrowdStrike has observed sophisticated threat actors naming their ghost VMs according to the same format. Pay close attention to the VM configuration file paths, as CrowdStrike has observed threat actors placing their VMs in hidden directories. Hidden directories begin with a period; CrowdStrike has observed China-nexus adversaries use legitimate VMFS datastore configuration directories such as `/.sdd.sf`.

VMWARE LOG ANALYSIS

SECURE SHELL (SSH)

CrowdStrike has observed both nation state-sponsored and eCrime threat actors access vCenter and ESXi hosts via SSH. Carefully analyze SSH logs for the following suspicious patterns of activity:

- Any SSH activity to an ESXi host with the `vpxuser` user account must immediately be treated as suspicious, as this is never required as part of legitimate vSphere activity. The `vpxuser` account should only be used by vCenter to authenticate to ESXi hosts under management, and not via SSH.
- SSH access to a vCenter appliance with any account (particularly `root`) originating from any system that is not a systems administrator's workstation as part of their legitimate business need.
- SSH access to an ESXi host with any account (particularly `root`) originating from any system that is not a systems administrator's workstation as part of their legitimate business need. SSH from a vCenter appliance to an ESXi host should be considered particularly suspicious.

DNS REQUESTS

Key path: `/var/log/vmware/dnsmasq.log.*`

vCenter records local DNS requests proxied by the dnsmasq DNS server running on the vCenter appliance. These files can serve as clear evidence of a malware implant on vCenter, including *BRICKSTORM*. When the adversary proxies their own web browser traffic through *BRICKSTORM* and the infected vCenter system, some of the adversary's browser's DNS lookups can be recorded in dnsmasq logs.

These log files ordinarily should only contain references to other vSphere infrastructure (e.g., ESXi hosts), and legitimate Broadcom infrastructure for telemetry, update checks, and licensing. Other unexpected DNS names

being present in these logs can indicate an adversary using the vCenter appliance as a network traffic proxy. Look out for access to cloud services, such as Microsoft 365 (including domains such as `login.microsoftonline.com` and `sharepoint.com`), as these can indicate an adversary attempting to bypass impossible travel checks by proxying cloud traffic through the victim's own network infrastructure.

Example Suspicious Log Entry

```
Mmm DD HH:MM:SS dnsmasq[1895]: query[A] login.microsoftonline.com from
127.0.0.1
```

VCENTER MALWARE: *BRICKSTORM*

BRICKSTORM is a China-nexus malware variant that allows the operator to remotely access vCenter and proxy network traffic through it. This can disguise the true origin of malicious activity by having it appear to originate from a customer's own network infrastructure.

Review the following for evidence of *BRICKSTORM* activity.

LOCALHOST UI ACCESS

Key log file: `/var/log/vmware/vsphere-ui/logs/access/localhost_access_log`

Once deployed to a vCenter system, *BRICKSTORM*'s proxying functionality is typically used for further access to vCenter. This results in the web access logs recording access from the loopback address, `127.0.0.1`, in the first two fields of each access log entry. Note that the first field is always shown as the loopback address due to the use of a reverse proxy within vCenter, and therefore only lines that show the loopback address **twice** should be considered suspicious.

Example suspicious log entry

```
127.0.0.1 127.0.0.1 - - [DD/Mmm/YYYY:HH:MM:SS +0000] "GET
/ui/webconsole.html?vmId=vm-1234&vmName=ImportantVM
&numMksConnections=0&serverGuid=648c9545-f716-4087-93dc-0d96f865ad36&locale=en-
US HTTP/1.1" 200 7323 200718 78349563 http-nio-5090-exec-2162 0
```

Any substantial access the web console, including searches in the UI, and viewing of pages for Virtual Machines, is highly likely to be adversary activity if it originates from the loopback address.

NOTE: the following requests are **benign** even when they originate from the loopback address and can be safely ignored. Any other loopback requests should be treated with suspicion.

- `POST /ds/vapi HTTP/1.1`
- `GET /apigw/resourcebundle HTTP/1.1`
- `GET /ui/resources/vsphere-ui-resource-bundle.zip HTTP/1.1`
- `POST /api/ui/session/sso/local-domain/saml-token HTTP/1.1`

SUSPICIOUS ELF BINARIES

Key paths:

- `/etc/sysconfig/*`

- `/etc/sysconfig/network/*`
- `/etc/sysconfig/init`

In recent intrusions, CrowdStrike has observed *BRICKSTORM* deployed to the `/etc/sysconfig/` directory on vCenter. While the filename and hash vary between samples and victims, **any ELF file within this directory should be considered suspicious**.

Additionally, CrowdStrike has observed adversaries persist *BRICKSTORM* by adding its path to the legitimate script `/etc/sysconfig/init`. The persistence is typically achieved by placing the name of the malware binary within this file surrounded by backtick characters, which trigger the binary's execution. **Any reference to an executable file within this script should be considered likely adversary activity.**

Notably, data from `/etc/sysconfig/` is backed up when a file-level backup of vCenter is performed and is included in VMware support bundles. While this eases detection, it can result in the transfer of persistence from one vCenter appliance to another if a backup of an infected appliance is restored to a clean one.

PROCSTATE DATA

Key path: `/var/log/vmware/procstate-*.gz`

vCenter regularly runs a process listing command and saves the output to disk in compressed files named according to the date they were generated on. These files can contain evidence of historic and/or ongoing suspicious process executions. The files are not retained for long, but any available procstate data can be invaluable to an investigator.

To hunt for *BRICKSTORM* within procstate data, search for a file executed by the `root` user that does not include a full file path in the command line. Any binary that does not have a file path should be considered suspicious even if the binary's name mirrors that of a legitimate VMware process (e.g., `envoy`, `updatemgr`, etc.).

Example benign entry

```
updatem+  12500  0.1  4.7 1732123 5321232 ?      Sl   MmmDD 124:11
/usr/lib/vmware-updatemgr/bin/updatemgr /usr/lib/vmware-updatemgr/bin/vci-
integrity.xml
```

Note: full file path is shown in the procstate output

Example malicious entry

```
root 2241234 4121234 0.0 0.0 721234 11234 ? Sl MmmDD HH:MM:SS 0 239 updatemgr
```

Note: no file path is provided for the binary, despite the name mirroring a legitimate VMware process.

AUDITD DATA

Key paths:

- `/var/log/audit`
- `/var/log/journal`

vCenter logs the `fchmodat` syscall via auditd, which can provide evidence of a threat actor marking a new executable (e.g., a malware binary) as executable via a command such as `chmod +x /path/to/binary`. CrowdStrike has observed this action during the deployment of malware binaries, including *BRICKSTORM*.

```
type=PROCTITLE msg=audit(MM/DD/YY YY:MM:SS.123:171234) : proctitle=chmod +x
vmafd
type=PATH msg=audit(MM/DD/YY YY:MM:SS.123:171234) : item=0 name=vmafd
inode=2626019 dev=fe:00 mode=file,644 ouid=root ogid=root rdev=00:00
nametype=NORMAL cap_fp=none cap_fi=none cap_fe=0 cap_fver=0 cap_frootid=0
type=CWD msg=audit(MM/DD/YY YY:MM:SS.123:171234) : cwd=/etc/sysconfig
type=SYSCALL msg=audit(MM/DD/YY YY:MM:SS.123:171234) : arch=x86_64
syscall=fchmodat success=yes exit=0 a0=0xffffff9c a1=0x55920abf5120 a2=0755
a3=0x49 items=1 ppid=123456 pid=123457 auid=root uid=root gid=root euid=root
suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts1 ses=81234
comm=chmod exe=/usr/bin/chmod subj=unconfined key=perm_mod
```

The adversarial commands that add *BRICKSTORM* persistence to `/etc/sysconfig/init` (typically via the `sed` command) may also be recorded by auditd.

```
type=PROCTITLE msg=audit(MM/DD/YY YY:MM:SS.123:171234) : proctitle=sed -i
$a\SETCOLOR_WARNING="echo -en `/etc/sysconfig/vmafd`\\033[0;33m" init
type=SYSCALL msg=audit(MM/DD/YY YY:MM:SS.123:171234) : arch=c000003e
syscall=fchmodat success=yes exit=0 a0=4 a1=7f23224300b3 a2=54aa7fcdb770 a3=1c
items=1 ppid=3213234 pid=3245145 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0
egid=0 sgid=0 fsgid=0 tty=pts0 ses=18193 comm="sed" exe="/usr/bin/sed"
subj=unconfined key="perm_mod"
```

In addition, auditd entries that show activity with a current working directory of `/tmp` (`CWD=/tmp`) should also be scrutinized.

VCENTER MALWARE: WEBSHELLS

Key potential webshell paths:

- `/usr/lib/vmware-sso/vmware-sts/webapps/ROOT/*`
- `/usr/lib/vmware-sso/vmware-sts/work/Catalina/localhost/ROOT/*`

Key log path:

- `/var/log/vmware/envoy/*`

Sophisticated adversaries may install webshells on vCenter appliances as forms of additional persistence and remote access. These webshells will typically be uploaded to directories accessible on vCenter without authentication, including the two key paths provided above.

CrowdStrike recommends hunting on the logs associated with the vCenter Envoy service for request URLs that include the string `..;/`, as this can be indicative of a threat actor leveraging path traversal abuse to access uploaded webshells.

Furthermore, adversaries may leverage webshells to upload arbitrary code and binaries to the `/tmp` directory. Binaries executed out of this path may also appear in procstate Data.

ESXI MALWARE: *JUNCTION*

A sophisticated adversary has developed a malware implant CrowdStrike has named *Junction*. *Junction* is a Go binary that allows the adversary to marshal data into and out of ESXi guest VMs via VSOCK sockets.[5] Network traffic destined from a guest VM can be sent to ESXi via a standard HTTP request, and *Junction* will route the datagrams to a guest VM of the adversary's choosing via a VSOCK, and return the response to the adversary. This powerful technique can allow a threat actor to reach into a running VM and leave minimal forensic evidence of this activity.

Review the following evidence sources to identify samples of *Junction*, or the effects of its use:

- SSH access to ESXi hosts from vCenter appliances.
- SSH access to ESXi hosts using the `vpxuser` account.
- Use of the `chmod +x` command against files in the ESXi `/tmp` directory.
- All other evidence of binary execution out of the ESXi `/tmp` directory.
- Evidence of execution of any running processes with the command line argument `-r /etc/vmware/rhttpproxy/config.xml`, except for the legitimate `rhttpproxy` binary that legitimately runs on ESXi.
- Large amounts of outbound traffic from an ESXi host routed to a vCenter appliance over port 443.

[5] A VSOCK is a special family of sockets that allows hypervisors to directly exchange data with guest virtual machines (VMs) without passing network packets through a standard network interface. It is the process by which VMware Tools (and similar functionality) can pass data to and from a VM's hypervisor. See: https://man7.org/linux/man-pages/man7/vsock.7.html.

Forensic Evidence Collection on vSphere

If systems in the vSphere environment were identified as compromised and/or impacted during an adversary intrusion, your CrowdStrike team may request forensic evidence as part of an Incident Response investigation. There are several ways that forensic evidence can be gathered from vSphere environments, with each method yielding a different dataset, so follow the below guidance based on the guidance of your CrowdStrike team.

SUPPORT BUNDLE COLLECTION

VMware has included functionality within the vSphere products to export Support Bundle packages that contain log files pertinent to the functionality of the appliance. While primarily intended for technical support, these logs can contain valuable forensic data that can be used by the CrowdStrike team to identify evidence of threat actor activity in the vSphere environment.

ESXI SUPPORT BUNDLE COLLECTION

To export a Support Bundle from an ESXi host, follow these steps:

1. Authenticate to the vCenter web interface.
2. Navigate to `Inventory`.
3. In the `Hosts & Clusters` view, select the ESXi host for which you intend to export a Support Bundle.
4. Click on the `ACTIONS` menu, then select `Export System Logs...`
5. Click on the `EXPORT LOGS` button.

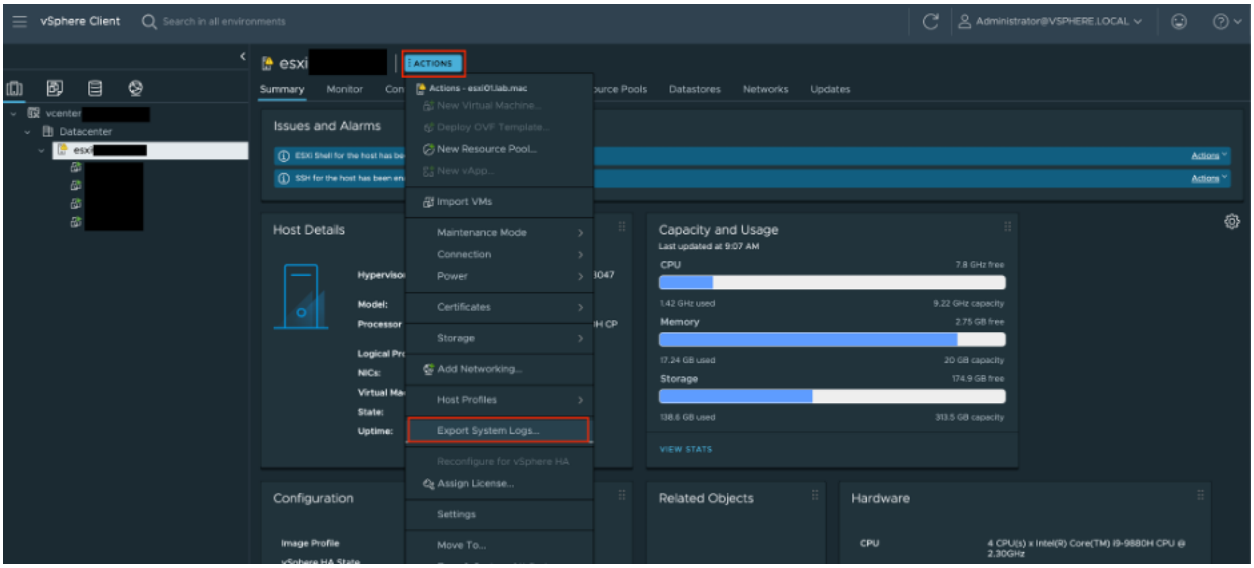

Figure 1: Generating an ESXi Support Bundle from the vCenter Web Console

The resulting archive (`.zip`) file can then be uploaded to the CrowdStrike team for further analysis.

For more information on collecting Support Bundles from ESXi hosts, see Broadcom's documentation.[6]

VCENTER SUPPORT BUNDLE COLLECTION

vCenter Support Bundles can be downloaded either from the vSphere Client, or the vCenter Server Appliance Management Interface (VAMI).[7, 8] The resulting archive (`.tgz`) file can then be uploaded to the CrowdStrike team for further analysis.

When exporting a vCenter Support Bundle, the CrowdStrike team will usually request you to select the option that includes all Support Bundles from associated ESXi hosts so that you do not need to obtain each ESXi host's Support Bundle separately. These Support Bundle export tasks can take a while to complete as vCenter executes the collection of, and aggregates, all the Support Bundles from each managed ESXi host.

VCENTER IMAGE COLLECTION

The vCenter Server Appliance (vCSA) virtual machine is stored as a series of VMDK files on a datastore mounted to an ESXi host. These VMDK files can be retrieved from the datastore for analysis by CrowdStrike through the following steps:

1. Authenticate to the vCenter web interface.
2. Navigate to `Inventory`.
3. In the `Datastores` view, select the datastore that the vCSA virtual machine is stored on.
4. The VMDK filenames follow traditional naming conventions (e.g., `vcenter01.vmdk`, `vcenter01_1.vmdk`, etc.). Select and download **all** the associated VMDK files from the datastore.

The resulting archive (`.zip`) file of VMDKs can then be uploaded to your CrowdStrike team for further analysis.

NOTE: vCSA can comprise of as many as 17 individual VMDKs depending on the version and configuration, and each VMDK contains different data. CrowdStrike will generally request a copy of all vCenter/vCSA VMDKs; however, in some cases, the vCenter PostgreSQL database can grow to several terabytes in size, and its corresponding VMDK can consequently become prohibitively large. If this is the case, consult with your CrowdStrike team to identify if the largest VMDK (typically this database) is required for your scenario. When investigating sophisticated adversaries, this database can contain valuable forensic data and should be provided by default where possible.

---

[6] https://knowledge.broadcom.com/external/article/326299/collecting-diagnostic-information-for-es.html

[7] The vCenter Server Appliance Management Interface (VAMI) is commonly hosted at `https://vCSA-IP-Address-or-FQDN:5480/`

[8] For instructions on exporting vCenter Support Bundles, see Broadcom's documentation here: https://knowledge.broadcom.com/external/article/330178/collecting-diagnostic-information-for-vm.html

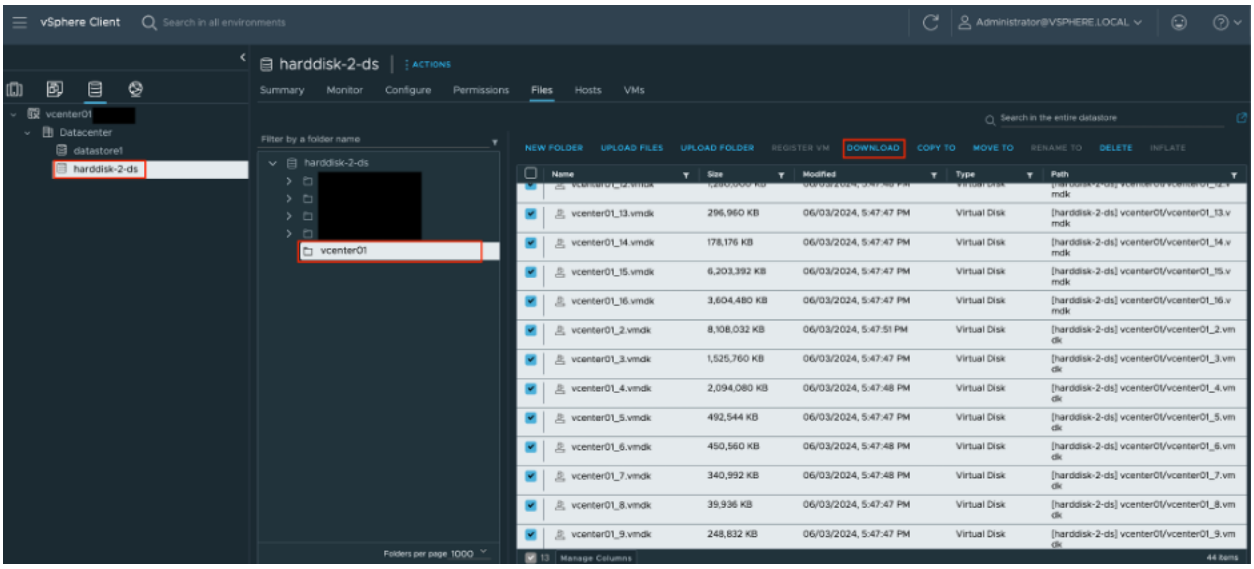

Figure 2: Exporting Disk Images of vCenter Server Appliance from the vCenter Web Console

Recommendations for Log Retention and Monitoring

FORWARD VCENTER AND ESXI LOGS TO A SIEM

Centralizing VMware vSphere logs in a Security Information & Event Management (SIEM) system represents a critical defensive measure against sophisticated adversaries targeting virtualized infrastructure. Given the observed tactics of China-nexus threat actors who target VMware environments for persistence and to evade security tools, organizations must implement comprehensive logging strategies that provide real-time visibility into their vSphere infrastructure.

The integration of vCenter and ESXi logs into a SIEM platform enables security teams to detect unauthorized access attempts, privilege escalation activities, and suspicious administrative actions that may indicate compromise. This centralized approach facilitates correlation between virtualization events and broader network security telemetry, creating a more complete picture of potential threats. Furthermore, sophisticated threat actors will attempt to tamper with and/or delete logs from local systems; centralizing of logs can provide critical defense against this type of activity.

CrowdStrike recommends forwarding all syslog data from vCenter and ESXi to a SIEM and retaining this data for a minimum of one year. CrowdStrike's Next-Gen SIEM solution natively supports syslog sent from both vCenter and ESXi, with in-built parsers and threat hunting rules designed specifically to identify malicious activity in vSphere.[9]

SCHEDULE AND RETAIN RECURRING BACKUPS OF THE POSTGRESQL DATABASE

The PostgreSQL database serves as the foundational data repository for VMware vCenter Server, containing critical configuration information, inventory data, performance statistics, and security-relevant metadata. Sophisticated adversaries understand that compromising or corrupting the vCenter database can severely impact an organization's ability to manage virtualized infrastructure and may provide opportunities for data manipulation or destruction as part of their operational objectives.

Database backups serve as critical forensic evidence during security incidents involving VMware infrastructure compromise. Backup files captured before, during, and after suspected compromise provide investigators with snapshots of system configuration and data that can reveal the scope and timeline of adversary activities.

The PostgreSQL database can be backed by selecting the `Stats, Events, and Tasks` option when performing or scheduling a vCenter Server backup. CrowdStrike recommends scheduling recurring backups at least once every two weeks and retaining each backup for at least one year, and ideally two years.[10]

---

[9] See CrowdStrike's blog post for more information: https://www.crowdstrike.com/en-us/blog/falcon-next-gen-siem-protects-against-vmware-vcenter-attacks/.

[10] https://techdocs.broadcom.com/us/en/vmware-cis/vsphere/vsphere/7-0/vcenter-server-installation-and-setup-7-0/file-based-backup-and-restore-of-a-vcenter-server-environment/file-based-vcenter-server-appliance-backup.html

STS TOKEN MONITORING

CrowdStrike has observed both China-nexus and eCrime adversary groups targeting VMware Security Token Service (STS) certificate databases, and leveraging them for privilege escalation, lateral movement, and persistent access. The STS issues cryptographically signed authentication tokens that can grant a user access to vSphere infrastructure, and therefore theft of the STS secrets can allow an adversary to forge certificates on behalf of administrative users within the vSphere environment, including the inbuilt `administrator@vsphere.local` account.

STS tokens represent a critical authentication mechanism within vSphere environments and serve as the primary method of establishing and maintaining authenticated sessions between vSphere components, ESXi hosts, and administrative clients. These threat actors understand that compromising or manipulating STS tokens can provide persistent access to virtualized environments while evading traditional authentication monitoring focused solely on username and password combinations.

It is critical that organizations monitor for anomalous activities involving the STS certificate storage locations & properly secure vSphere backups (as they contain a copy of the STS certificate as well). CrowdStrike recommends following the guidance in Appendix A: Recommended auditd Configuration Modifications to enable auditd rules on vSphere appliances that are designed specifically to identify potential STS certificate theft.

Several indicators may suggest that STS tokens have been compromised or are being used maliciously. These include:

- Multiple concurrent vCenter sessions with the same session token originating from difficult network locations, as this may suggest token theft or replay attacks.
- Tokens used for administrative activities that are inconsistent with user roles or responsibilities may indicate privilege escalation or unauthorized access.
- Failed token validation events may indicate attempts to use expired, forged, or corrupted tokens for unauthorized access.
- Patterns of access from suspicious source locations targeting sensitive resources (e.g., domain controllers, identity infrastructure, etc.).

LOG DNS QUERIES

DNS query logging represents a critical security monitoring capability for detecting and analyzing the sophisticated tactics employed by China-nexus adversaries targeting VMware vSphere infrastructure. DNS queries generated by compromised vCenter servers, ESXi hosts, and virtual machines provide valuable intelligence about adversary infrastructure, communication patterns, and operational objectives that can significantly enhance threat detection and incident response capabilities.

SSH KEY AUDITING

CrowdStrike has observed adversaries adding SSH keys to vCenter Server Appliances to obtain persistent access. The persistent nature of SSH key authentication creates significant security risks when keys are not properly managed, monitored, or rotated. Adversaries who gain access to SSH private keys can maintain persistent access to vCenter systems even after password changes, account lockouts, or other traditional security responses to suspected compromise.

Adversaries may configure SSH keys for password-less authentication to administrative accounts including the `root` account and service accounts used for automated operations. These keys are stored in user home directories within the `~/.ssh/authorized_keys` files within each user profile directory and provide persistent

authentication capabilities that bypass traditional password controls. The `root` account's `authorized_keys` file represents a particularly high-value target for adversaries seeking to establish persistent administrative access to vCenter infrastructure.

SSH keypairs are not created for ESXi or vCenter user accounts by default, but Broadcom provides documentation to configure SSH keypairs.[11] Therefore, **the presence of any `authorized_keys` files on vCenter or ESXi may be suspicious if the use of SSH keys is not part of your vSphere security baseline or standard operating procedure.**

MONITOR INTERNAL NETWORK TRAFFIC

CrowdStrike has observed adversaries moving laterally across victim environments after obtaining persistence in vSphere. While logging of network traffic at the perimeter is extremely useful to identify command-and-control and exfiltration activity ("north/south" traffic), it is equally crucial to monitor internal (i.e., "east/west") network traffic to identify where an adversary has moved within a compromised network environment.

The following network traffic patterns may be indicative of China-nexus activity. Ensure that your internal traffic logging has visibility into, and can record, these traffic patterns:

- Network traffic from vCenter systems to internal services such as code repositories, wikis, ticketing systems, and build infrastructure.
- Network traffic on port `9999`.
- Network traffic on port `10022` to an unknown system (potential ghost VM).
- Unexpected SSH (port `22`) activity from vCenter to ESXi hosts, or any other SSH activity outside of the vSphere cluster (e.g., SSH from one vCenter system to another vCenter system in a different cluster, or from a vCenter system to an ESXi host managed by a different vCenter system).

CROWDSTRIKE FALCON NG-SIEM

For CrowdStrike Next-Gen SIEM customers, CrowdStrike recommends forwarding vCenter Logs to Falcon Next-Gen SIEM. CrowdStrike has released Next-Gen SIEM rule templates and dashboards to help safeguard enterprises from vCenter and ESXi attacks.

[11] https://knowledge.broadcom.com/external/article/313767/allowing-ssh-access-to-vmware-vsphere-es.html

Network Segmentation

Implementing restrictive outbound firewall rules for vCenter Server represents a critical security control for defending against China-nexus and other adversaries who exploit compromised vCenter systems for command-and-control communications, data exfiltration, and lateral movement activities. These sophisticated threat actors understand that vCenter servers typically operate with elevated network privileges and may have less restrictive outbound access controls compared to other enterprise systems. By establishing comprehensive outbound firewall rules that permit only necessary communications while blocking unauthorized traffic, organizations can significantly limit adversary capabilities following initial compromise and create additional detection opportunities through traffic analysis. Broadcom publishes a list of the minimum required outbound connectivity for VMware products.[12]

[12] https://knowledge.broadcom.com/external/article/320929/public-ip-list-for-vmware-update-manager.html

Appendix A: Recommended auditd Configuration Modifications

Create a file named `/etc/audit/rules.d/CrowdStrike-vCenter.rules` and place the following contents there:

```
# CrowdStrike auditd rules for enhanced vCenter monitoring

# Key used to encrypt vpxuser password inside Postgresql database
-a always,exit -F path=/etc/vmware-vpx/ssl/symkey.dat -F perm=r -F euid!=vpxd -
F auid!=unset -k CrowdStrike-vpxuser-password

# Secure Token Service (STS) certificate storage locations
-a always,exit -F path=/storage/db/vmware-vmdir/data.mdb -F perm=r -F
auid!=unset -k CrowdStrike-STS-Cert-vmdir
-a always,exit -F path=/storage/db/vmware-vmafd/afd.db -F perm=r -F
euid!=vmafdd-user -F auid!=unset -k CrowdStrike-STS-Cert-vmafd

# Execution from /tmp directory
-a always,exit -F dir=/tmp -F perm=x -k CrowdStrike-execution-from-tmp
```

The additional rules in the above file will be loaded by auditd during system startup. To apply the rules immediately without having to restart, execute the following four `auditctl` commands with administrative (`root`) privileges:

```
auditctl -a always,exit -F path=/etc/vmware-vpx/ssl/symkey.dat -F perm=r -F
euid!=vpxd -F auid!=unset -k CrowdStrike-vpxuser-password

auditctl -a always,exit -F path=/storage/db/vmware-vmdir/data.mdb -F perm=r -F
auid!=unset -k CrowdStrike-STS-Cert-vmdir

auditctl -a always,exit -F path=/storage/db/vmware-vmafd/afd.db -F perm=r -F
euid!=vmafdd-user -F auid!=unset -k CrowdStrike-STS-Cert-vmafd

auditctl -a always,exit -F dir=/tmp -F perm=x -k CrowdStrike-execution-from-tmp
```

Appendix B: GHOSTHUNTER Python Script

CrowdStrike has developed a Python script that leverages pyVmomi (the Python SDK for the VMware vSphere Management API) to triage vCenter Servers for signs of suspicious activity. It is designed to complement the original PowerShell script `Detect-VirtualGhost.ps1` that CrowdStrike previously created to identify the presence of VirtualGHOST VMs.

The script was tested using Python 3.13 against vCenter 8. It requires two Python modules which can be installed using Python's default package manager `pip`:

- `requests`
- `pyvmomi`

SCRIPT ACTIONS

This script will perform the following actions:

- Review events stored in vCenter's embedded database for evidence of:
  - Any SSH activity
  - Logins
  - Virtual machine removal
- Any events that satisfy the following criteria will be flagged for the user:
  - SSH involving the `vpxuser` account
  - SSH or web UI logins originating from the vCenter IP address
- All events that were retrieved will be saved to the output directory for further review. The script will default to writing the output to the directory containing the script file, but the output directory can be specified via the `--outdir` command line switch.
- Search all datastores for any files that appear to be virtual machine log files. For each identified log file, the script will:
  - Download the log file to a temporary directory
  - Perform three checks on each line of the log file:
    - Hunt for suspicious environment variables
    - Hunt for a suspicious virtual machine command line i.e., `/bin/vmx -x`)
    - Hunt for references to virtual machines files in suspicious directories
- Any log files that satisfy one of the above criteria will be copied to the output directory for further review. A message will additionally be shown on screen to indicate where the files were stored so that the suspicious VM can be reviewed and uploaded to CrowdStrike for review as desired.

EXTRACTING THE SCRIPT

Original filename: `GHOSTHUNTER.py`

To avoid the need to copy and paste the script from the Script Contents section below, the script has also been attached to the PDF as an embedded text file that can be saved to disk using commercially available PDF reader software such as Adobe Acrobat and Foxit Reader.

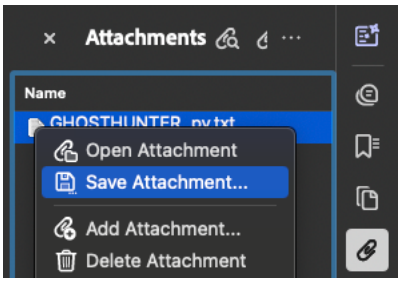

Figure 3: Exporting the GHOSTHUNTER Script from this Document Using Adobe Acrobat

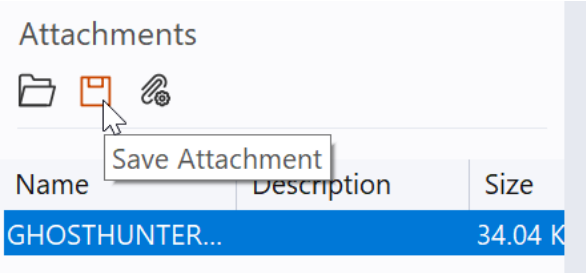

Figure 4: Exporting the GHOSTHUNTER Script from this Document Using Foxit Reader

Once the file has been exported, rename `GHOSTHUNTER_py.txt` to `GHOSTHUNTER.py` to continue with these instructions.

EXECUTING THE SCRIPT

Script arguments and usage is available by using the `--help` command line switch.

EXAMPLE INVOCATION

The following command will run the GHOSTHUNTER script against `vcenter.example.com` using the `administrator@vsphere.local` account. Full debug logging will be written to screen, SSL/TLS certificate checks will be disabled (required for self-signed vCenter certificates), and output will be written to `C:\Temp\CrowdStrike_GHOSTHUNTER_Results`.

```
python3 ./GHOSTHUNTER.py --server vcenter.example.com --user
administrator@vsphere.local --log-level DEBUG --no-ssl-check --outdir
C:\Temp\CrowdStrike_GHOSTHUNTER_Results
```

NOTE: You do not need to execute the script with the `administrator@vsphere.local` account specifically, but you will need to use a user account that has elevated privileges within vCenter.

SCRIPT CONTENTS

Original filename: `GHOSTHUNTER.py`

The contents of the embedded GHOSTHUNTER script are provided below.

```
#!/usr/bin/env python3
"""
Description: VirtualGHOST & General Threat Actor detection script for vSphere environments
Author: CrowdStrike
Date: October 9, 2025
Acknowledgements: Heavily inspired by the great work from the community at https://github.com/vmware/pyvmomi-community-samples/
"""

import argparse
import sys
import logging
import re
import tempfile
import os

from pyVim.connect import SmartConnect
from pyVmomi import vim
from typing import Optional, Union
import requests
import time
from datetime import datetime
from pathlib import Path
import shutil
from collections import Counter
import getpass
import socket


def setup_logging(log_level, log_file: Path):
    """Configure logging based on specified level and return a logger object"""
    logger = logging.getLogger()
    logger.setLevel(log_level)

    # TODO: Adjust this so we can provide an extra= dictionary argument & display it
    formatter = logging.Formatter(
        "%(asctime)s UTC | %(levelname)s | %(funcName)s | %(message)s",
        datefmt="%Y-%m-%dT%H:%M:%S",
    )
    formatter.converter = time.gmtime  # Use UTC

    # File logger setup
    file_handler = logging.FileHandler(log_file)
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    # Also stream to standard out
    consoleHandler = logging.StreamHandler(sys.stdout)
    consoleHandler.setFormatter(formatter)
    logger.addHandler(consoleHandler)

    return logger


def search_datastore_for_logfiles(
    datastore: vim.Datastore,
) -> list[vim.host.DatastoreBrowser.SearchResults]:
    logger.debug(f"Searching datastore '{datastore.name}' for log files")
    base_path = f"[{datastore.name}]"
    searchSpec = vim.host.DatastoreBrowser.SearchSpec()
    searchSpec.matchPattern = ["vmware*.log"]
    logger.debug(f"Using match pattern: {searchSpec.matchPattern}")
    searchSpec.details = vim.host.DatastoreBrowser.FileInfo.Details(
        fileSize=True, modification=True
    )
    browser = datastore.browser
    logger.debug(f"Creating search task for datastore '{datastore.name}'")
    start_time = time.time()
    task = browser.SearchSubFolders(base_path, searchSpec)
```

```
    results = wait_task(task)
    search_duration = time.time() - start_time
    logger.debug(f"Search complete for datastore '{datastore.name}' in {search_duration:.2f}s")
    if results:
        logger.debug(f"Found {len(results) if isinstance(results, list) else 1} result folders")
    return results


def get_vsphere_cookie(si: SmartConnect) -> dict:
    logger.debug("Retrieving vSphere session cookie")
    # Get the cookie built from the current session
    client_cookie = si._stub.cookie
    # Break apart the cookie into it's component parts - This is more than
    # is needed, but a good example of how to break apart the cookie
    # anyways. The verbosity makes it clear what is happening.
    cookie_name = client_cookie.split("=", 1)[0]
    cookie_value = client_cookie.split("=", 1)[1].split(";", 1)[0]
    cookie_path = (
        client_cookie.split("=", 1)[1].split(";", 1)[1].split(";", 1)[0].lstrip()
    )
    cookie_text = " " + cookie_value + "; $" + cookie_path
    # Make a cookie
    cookie = dict()
    cookie[cookie_name] = cookie_text

    logger.debug(f"Session cookie name: {cookie_name}")
    logger.debug("Cookie retrieved successfully")

    return cookie


def generate_individual_files(
    search_results: list[vim.host.DatastoreBrowser.SearchResults],
    datacenter: vim.Datacenter,
) -> list[dict]:
    """Given a list of datastore search hits, generate a list of dictionaries with the necessary information to download the files
parsed out

    Args:
        search_results (list[vim.host.DatastoreBrowser.SearchResults]): Search hits

    Returns:
        list[dict]: Individual file records
    """
    logger.debug(f"Processing search results for {search_results.folderPath}")
    results = []
    # Pattern to match [datastore] path/format
    pattern = r"\[(.*?)\]\s+(.*?)$"

    match = re.search(pattern, search_results.folderPath)
    if match:
        datastore = match.group(1)  # Captures text inside brackets
        folder = match.group(2)  # Captures text after brackets and whitespace
        logger.debug(f"Matched datastore path: [{datastore}] {folder}")
    else:
        logger.warning(f"Failed to match datastore path pattern for: {search_results.folderPath}. Files include: {',
'.join([item.path for item in search_results.file])}")
        return None

    file_count = len(search_results.file) if hasattr(search_results, 'file') else 0
    logger.debug(f"Processing {file_count} files in this folder")

    for file in search_results.file:
        results.append(
            {
                "datastore": datastore,
                "folder": folder,
                "filename": file.path,
                "filesize": file.fileSize,
                "modification": file.modification,
                "datacenter": datacenter.name,
            }
        )
    logger.debug(f"Generated {len(results)} file records")
    return results


def get_hypervisors(content: vim.ServiceInstanceContent) -> list[vim.HostSystem]:
    logger.debug("Retrieving hypervisor hosts from vSphere environment")
```

```
    host_view = content.viewManager.CreateContainerView(
        content.rootFolder, [vim.HostSystem], True
    )
    hosts = list(host_view.view)
    logger.debug(f"Retrieved {len(hosts)} hypervisor hosts")
    host_view.Destroy()
    return hosts


def wait_task(task: vim.Task) -> object:
    """Waits for task to complete"""
    logger.debug(f"Waiting for task '{task.info.descriptionId}' to complete")
    start_time = time.time()
    info = task.info
    while info.state in ["queued", "running"]:
        logger.info(f"Waiting on task to complete.... Current state: {info.state}")
        time.sleep(5)
        info = task.info

    elapsed_time = time.time() - start_time
    logger.debug(f"Task completed with state: {info.state} in {elapsed_time:.2f} seconds")

    if info.state == "error":
        logger.error(f"Task failed with error: {info.error}")
        raise info.error

    return info.result

def get_vsphere_ips(content: vim.ServiceInstanceContent):
    try:
        hostname = get_vsphere_setting(content, 'VirtualCenter.FQDN')

        addr_info = socket.getaddrinfo(hostname, None)
        ip_addresses = list(set([info[4][0] for info in addr_info]))
        logger.info(f"All IP addresses for {hostname}: {ip_addresses}")

        return ip_addresses

    except socket.gaierror:
        logger.error(f"Could not resolve hostname: {hostname}")
    except Exception as e:
        logger.error(f'Unable to get vSphere IP addresses: {e}')



def get_interesting_events(
    service_instance: vim.ServiceInstanceContent,
) -> list[vim.event.Event]:
    """Gets interesting events from vSphere Event API

    Args:
        host (vim.HostSystem): Individual hypervisor/host to search for events from
        service_instance (vim.ServiceInstanceContent): Python object for accesessing vSphere APIs

    Returns:
        list[vim.event.Event]: List of events satisfying the 'interesting' criteria

    """
    logger.debug("Starting event retrieval with filter")
    eventIds = [
        "esx.audit.ssh.session.closed",
        "esx.audit.ssh.session.opened",
        "esx.audit.ssh.enabled",
        "vim.event.VmOrphanedEvent",
        "vim.event.VmRemovedEvent",
        "vim.event.UserLoginSessionEvent"
    ]
    logger.debug(f"Using event filter for event types: {eventIds}")

    event_filterspec = vim.event.EventFilterSpec(eventTypeId=eventIds)

    start_time = time.time()
    event_collector = service_instance.eventManager.CreateCollectorForEvents(event_filterspec)
    events_per_call = 1000
    logger.debug(f'First event pull, retrieving {events_per_call} events at a time')
    events = event_collector.ReadNextEvents(events_per_call)

    # Need to loop here because the default limit is 1000 and we dislike oddly round numbers
    while True:
```

```
        these_events = event_collector.ReadNextEvents(events_per_call)
        if len(these_events) == 0:
            logger.debug('Out of events!')
            event_collector.DestroyCollector() # We'll be nice and try to save resources
            break
        logger.debug(f'Retrieved {len(these_events)} events')
        events.extend(these_events)

    query_time = time.time() - start_time
    logger.debug(f"Event query completed in {query_time:.2f} seconds")
    logger.info(f"Retrieved {len(events)} events")

    event_types = Counter([type(item) for item in events])
    logger.info(f"Event types found: {event_types}")
    logger.debug(
        f"Stats for events filtered {Counter(item.fullFormattedMessage for item in events)}"
    )
    return events


def extract_event_data(events, fields_to_extract):
    """
    Extract specified fields from a list of events, handling missing fields gracefully.

    Args:
        events (list[vim.event.Event]): List of vSphere events
        fields_to_extract (list[str]): Fields to extract from each event

    Returns:
        list[dict]: List of dictionaries with extracted fields
    """
    logger.debug(f"Extracting {len(fields_to_extract)} fields from {len(events)} events")
    result = []

    for event in events:
        event_data = {}
        event_id = getattr(event, 'key', 'unknown_event_id')

        for field in fields_to_extract:
            # Handle nested fields like 'host.name'
            if "." in field:
                parts = field.split(".")
                value = event
                try:
                    for part in parts:
                        if value is not None:
                            value = getattr(value, part, None)
                except Exception as e:
                    logger.debug(f"Error extracting nested field '{field}' from event {event_id}: {e}")
                    value = None
            else:
                # Handle regular fields
                value = getattr(event, field, None)

            event_data[field] = value

        result.append(event_data)

    logger.debug(f"Successfully extracted data from {len(result)} events")
    return result

def check_suspicious_environment_vars(log_line):
    """
    Extract all environment variables from a log line and check which ones aren't allowed.

    Args:
        log_line (str): A log line containing environment variables

    Returns:
        dict: Dictionary containing all_vars (all found variables) and suspicious_vars (disallowed ones)
    """
    # Extract environment variables in the format "NAME=VALUE"
    env_var_pattern = r'"([^="]+)=([^"]*)"'

    # Find all environment variables in the line
    all_vars = {}
    if "Environment:" in log_line:
        matches = re.finditer(env_var_pattern, log_line)
        for match in matches:
```

```
            var_name = match.group(1)
            var_value = match.group(2)
            all_vars[var_name] = var_value

    # Check which variables aren't in the allowed list
    allowed_env_vars = ["USER", "HOME", "PATH", "LANG", "SHELL", "LD_LIBRARY_PATH"]
    logger.debug(f"Performing environment variable check using allowed environment variables: {allowed_env_vars}")
    suspicious_vars = {name: value for name, value in all_vars.items()
                       if name not in allowed_env_vars}

    return {
        "all_vars": all_vars,
        "suspicious_vars": suspicious_vars
    }

def has_suspicious_environment_vars(log_line):
    """
    Check if a log line contains suspicious environment variables.

    Args:
        log_line (str): A log line to check

    Returns:
        bool: True if suspicious variables found, False otherwise
    """
    if "Environment:" not in log_line:
        return False

    result = check_suspicious_environment_vars(log_line)
    return len(result["suspicious_vars"]) > 0

def search_file_for_patterns(file_path, patterns) -> dict:
    """
    Search a file for multiple regex patterns and print matching lines.

    Args:
        file_path (str): Path to the file to search
        patterns (list): List of regex pattern strings

    Returns:
        dict: Dictionary with keys match_count and matched_lines
    """
    logger.debug(f"Searching file '{file_path}' for {len(patterns)} patterns")
    file_size = os.path.getsize(file_path) if os.path.exists(file_path) else 0
    logger.debug(f"File size: {file_size} bytes")

    # Compile all patterns for efficiency
    compiled_patterns = [re.compile(pattern) for pattern in patterns]
    match_count = 0
    matched_lines = []

    try:
        start_time = time.time()
        with open(file_path, "r") as file:
            for line_num, line in enumerate(file, 1):
                # Check if line has suspicious environment variables
                if "Environment:" in line and has_suspicious_environment_vars(line):
                    matched_lines.append(f"Line {line_num}: {line.rstrip()}")
                    match_count += 1
                    logger.debug(f"Suspicious environment variables found in line {line_num}")
                    continue

                # Check other patterns
                for i, pattern in enumerate(compiled_patterns):
                    if pattern.search(line):
                        matched_lines.append(f"Line {line_num}: {line.rstrip()}")
                        match_count += 1
                        logger.debug(f"Pattern {i} matched line {line_num}")
                        break
        search_duration = time.time() - start_time
        logger.debug(f"Search completed in {search_duration:.2f}s, found {match_count} matches in {line_num} lines")
        return {"match_count": match_count, "matched_lines": matched_lines}

    except FileNotFoundError:
        logger.error(f"Error: File '{file_path}' not found")
        return None
    except Exception as e:
        logger.error(f"Error processing file: {e}", exc_info=True)
        return None
```

```
def search_logfile_for_ghosts(
    targetfile: dict, si: SmartConnect, session: requests.Session, outdir: Path
):
    if targetfile['folder'].endswith('/'):
        folder_filename = f'{targetfile['folder']}{targetfile['filename']}'
    else:
        folder_filename = f'{targetfile['folder']}/{targetfile['filename']}'
    logger.debug(f"Examining log file: {folder_filename} (size: {targetfile.get('filesize', 'unknown')} bytes)")
    indicators_found = False

    GHOST_PATTERNS = [
        r"/\.sdd",
        r"Command line:.+vmx\".+\"-x\"",
    ]

    logger.debug(f"Using {len(GHOST_PATTERNS)} patterns plus environment variable checks for malicious activity detection")

    start_time = time.time()
    with tempfile.NamedTemporaryFile(mode="w+b", delete=False) as temp:
        # Download file
        url = f"https://{si._stub.host}/folder/{folder_filename}"
        logger.debug(f"Downloading file from URL: {url}")
        try:
            logger.debug(f"Sending POST request with parameters: dcPath={targetfile['datacenter']},
dsName={targetfile['datastore']}")
            response = session.post(
                url,
                stream=True,
                params={
                    "dcPath": targetfile["datacenter"],
                    "dsName": targetfile["datastore"],
                }
            )
            response.raise_for_status()  # Raise exception for HTTP errors
            logger.debug(f"Download request successful, status code: {response.status_code}")

            # Write to temp file
            bytes_downloaded = 0
            for chunk in response.iter_content(chunk_size=8192):
                temp.write(chunk)
                bytes_downloaded += len(chunk)
            temp.flush()
            temp.seek(0)
            tempfilename = temp.name
            download_time = time.time() - start_time
            logger.debug(f"Download complete: {bytes_downloaded} bytes in {download_time:.2f}s ({bytes_downloaded/download_time:.2f}
bytes/sec)")
        except requests.exceptions.RequestException as e:
            logger.error(f"Error downloading file: {e}", exc_info=True)
            logger.error(f"Error context: URL={url}, datacenter={targetfile['datacenter']}, datastore={targetfile['datastore']}")
            # Skipping the re-raise as we don't want to stop the scan if a single file fails to download
            return None

    logger.debug(f"Beginning scan of downloaded file: {tempfilename}")
    hits = search_file_for_patterns(tempfilename, GHOST_PATTERNS)

    if hits and hits.get("match_count") >= 1:
        indicators_found = True  # Set the indicators found flag
        print('\n')
        logger.warning(f"Hits found in: {targetfile['folder']}{targetfile['filename']}")
        for hit in hits["matched_lines"]:
            logger.warning(hit)

        # Copy the original file
        try:
            new_filename = outdir.joinpath(
                *[item for item in targetfile["folder"].split("/") if item],
                targetfile["filename"],
            )  # Unpacking with * so we get each folder from the datastore individually
            logger.debug(f"Creating directory structure: {new_filename.parent}")
            new_filename.parent.mkdir(exist_ok=True)
            logger.debug(f"Copying suspicious file to: {new_filename}")
            shutil.copy2(tempfilename, new_filename)
            logger.info(f"Copied logfile to {new_filename}")
        except Exception as e:
            logger.error(f"Error saving suspicious file: {e}", exc_info=True)
    else:
```

```
        logger.debug(f"No indicators found in file: {targetfile['folder']}{targetfile['filename']}")

    # Remove the temporary file
    try:
        logger.debug(f"Removing temporary file: {tempfilename}")
        os.remove(tempfilename)
    except Exception as e:
        logger.warning(f"Failed to remove temporary file {tempfilename}: {e}")

    total_time = time.time() - start_time
    logger.debug(f"File examination completed in {total_time:.2f}s")
    return indicators_found


def parse_arguments():
    """Parse command line arguments"""
    parser = argparse.ArgumentParser(description="Python script developed by CrowdStrike Professional Services to access vSphere
APIs and identify signs of potential compromise")

    # Required arguments
    parser.add_argument("-s", "--server", help="vSphere server hostname", required=True)
    parser.add_argument("-u", "--user", help="vSphere username", required=True)
    parser.add_argument("-p", "--password", help="vSphere password", default=None)

    # Optional arguments
    parser.add_argument(
        "-v", "--verbose", action="store_true", help="Enable verbose output"
    )
    parser.add_argument(
        "--log-level",
        choices=["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"],
        default="INFO",
        help="Set the logging level",
    )
    parser.add_argument(
        "-d",
        "--no-ssl-check",
        dest="disable_ssl_verify",
        action="store_true",
        help="Disable SSL verification for HTTPS requests",
    )
    parser.add_argument(
        "--outdir",
        help="Directory to store output logs and messages. Defaults to the script directory",
        default=Path(__file__).parent,
    )

    return parser.parse_args()

def get_vsphere_setting(service_instance: vim.ServiceInstanceContent, property:str) -> Union[str, float, None]:
    try:
        this_value = service_instance.setting.QueryView(property)[0].value
        return this_value
    except Exception as e:
        logger.debug(f'Unable to retrieve vSphere setting {property}. {e}')
        return None

def main():
    """Main function"""
    script_start_time = time.time()
    script_start_file_formatted = datetime.now().isoformat().replace(":", "")
    global logger  # Declare logger as global at the beginning of the function
    local_logger = None  # Use a local variable for tracking before global assignment

    try:
        args = parse_arguments()
        logger_initialized = False  # Flag to track if logger has been initialized

        output_dir = Path(args.outdir)
        output_dir.mkdir(exist_ok=True, parents=True)
        logger_debug_enabled = args.log_level == "DEBUG" or args.verbose

        # Prompt for password if not provided as argument
        password = args.password
        if password is None:
            password = getpass.getpass(f"Enter password for {args.user}@{args.server}: ")

        # Setup logging
        log_filename = output_dir.joinpath(
```

```
            f'CrowdStrike_GHOSTHUNTER_{args.server}_{script_start_file_formatted}.log'
        )
        logger = setup_logging(getattr(logging, args.log_level), log_filename)
        local_logger = logger  # Keep a reference in case we need to check initialization
        logger_initialized = True

        # Log script start with version and platform info
        logger.info(f"GHOSTHUNTER starting execution at {datetime.now().isoformat()}")
        logger.debug(f"Python version: {sys.version}")
        logger.debug(f"Platform: {sys.platform}")
        logger.debug(f"Arguments: {args}")
        logger.debug(f"Output directory: {output_dir.absolute()}")

        # Connection phase
        logger.info(f"Attempting connection to vSphere server {args.server}")
        logger.debug(f"Using username: {args.user}")
        logger.debug(f"SSL verification: {'disabled' if args.disable_ssl_verify else 'enabled'}")

        connection_start = time.time()
        si = SmartConnect(
            host=args.server,
            user=args.user,
            pwd=password,
            disableSslCertValidation=args.disable_ssl_verify,
        )
        connection_time = time.time() - connection_start
        logger.debug(f"Connection established in {connection_time:.2f}s")
        logger.info(f"Successfully connected to {args.server}")

        vsphere_service_content = si.RetrieveContent()
        logger.debug("Retrieved service content from vSphere server")

        # Get some data that's useful for scoping and general knowledge
        logger.info(f'Database retention settings: Events - {get_vsphere_setting(vsphere_service_content, 'event.maxAge')} | Tasks -
{get_vsphere_setting(vsphere_service_content, 'task.maxAge')}')

        # Get all of the hypervisors in the environment - we'll use this object elsewhere a few times
        host_retrieval_start = time.time()
        all_hosts = get_hypervisors(vsphere_service_content)
        host_retrieval_time = time.time() - host_retrieval_start
        logger.debug(f"Host retrieval completed in {host_retrieval_time:.2f}s")
        logger.info(f"Found {len(all_hosts)} hypervisor hosts in the environment")

        # Setup a flag to indicate levels of concern that we need to alert the user about
        INDICATORS_FOUND = False
        NEEDS_REVIEW = False

        # Event searches
        logger.info("Starting event search...\n")
        event_search_start = time.time()
        all_events = get_interesting_events(vsphere_service_content)
        event_search_time = time.time() - event_search_start
        logger.debug(f'Retrieved {len(all_events)} events in {event_search_time:.2f}s')

        # Process events
        logger.debug("Filtering events for vpxuser SSH activity")
        vpxuser_ssh = [
            item for item in all_events if "vpxuser" in item.fullFormattedMessage
        ]

        logger.debug("Filtering events for SSH activity originating from a vSphere IP")
        vsphere_ips = get_vsphere_ips(vsphere_service_content)
        vsphere_ssh = [
            item for item in all_events if any(ip in item.fullFormattedMessage for ip in vsphere_ips)
        ]

        remaining_events = [item for item in all_events if item not in vpxuser_ssh  and item not in vsphere_ssh]
        logger.debug(f"Found {len(vpxuser_ssh)} vpxuser SSH events and {len(remaining_events)} other events")

        event_fields_to_display = [
            "createdTime",
            "userName",
            "host.name",
            "fullFormattedMessage",
            "eventTypeId",
            "objectName",
            "chainId",
        ]
```

```
        # Process vpxuser SSH events (high severity)
        if vpxuser_ssh:
            INDICATORS_FOUND = True
            logger.debug("Processing vpxuser SSH events (high severity indicators)")
            print("\n")
            logger.warning(
                f"===== There were {len(vpxuser_ssh)} SSH events involving the vpxuser account. These are NOT normal and are likely
indications of Threat Actor activity. Specific instances will be displayed ===== "
            )
            print("\n")
            extracted_events = extract_event_data(vpxuser_ssh, event_fields_to_display)
            for event in extracted_events:
                event_string = " | ".join(
                    [
                        f"{key}: {value}"
                        for key, value in event.items()
                        if value is not None
                    ]
                )
                logger.warning(f"Event details: {event_string}")

        # Process SSH events involving vSphere IP (high severity)
        if vsphere_ssh:
            INDICATORS_FOUND = True
            logger.debug("Processing events involving a vSphere IP (high severity indicators)")
            print("\n")
            logger.warning(
                f"===== There were {len(vsphere_ssh)} events likely originating from a vSphere IP. These are NOT normal and are
likely indications of Threat Actor activity. Specific instances will be displayed ===== "
            )
            print("\n")
            extracted_events = extract_event_data(vsphere_ssh, event_fields_to_display)
            for event in extracted_events:
                event_string = " | ".join(
                    [
                        f"{key}: {value}"
                        for key, value in event.items()
                        if value is not None
                    ]
                )
                logger.warning(f"Event details: {event_string}")

        # Process other suspicious events (medium severity)
        if remaining_events:
            NEEDS_REVIEW = True
            logger.debug("Processing other suspicious events (medium severity indicators)")
            print("\n")
            logger.warning(
                f"===== There were {len(all_events)} remaining events that are somewhat suspicious in nature. Statistics will be
displayed for a quick triage. ===== "
            )
            print("\n")
            event_stats = Counter(item.fullFormattedMessage for item in remaining_events)
            logger.warning(f'{event_stats}')
            print("\n")

        if all_events:
            event_filename = output_dir.joinpath(f'CrowdStrike_GHOSTHUNTER_DetectedEvents_{script_start_file_formatted}.log')
            all_event_printable = extract_event_data(all_events, event_fields_to_display)
            with open(event_filename, 'w') as f:
                for event in all_event_printable:
                    event_string = " | ".join(
                    [
                        f"{key}: {value}"
                        for key, value in event.items()
                        if value is not None
                    ]
                )
                    f.write(event_string + '\n')
            logger.info(f'Writing out a file containing each event to {event_filename}')

        # Datastore searches
        logger.info('Beginning the process of searching datastores for VM log files & examining them for signs of suspicious
activity.')
        targets = []

        # Get the list of all datacenters we have available to us
        logger.debug("Retrieving datacenter list")
        datacenters_start = time.time()
```

```
        datacenters_object_view = (
            vsphere_service_content.viewManager.CreateContainerView(
                vsphere_service_content.rootFolder, [vim.Datacenter], True
            )
        )
        dc_count = len(datacenters_object_view.view)
        logger.info(f"Found {dc_count} datacenters")

        # List all the datastores
        total_datastores = 0
        for dc_idx, dc_obj in enumerate(datacenters_object_view.view, 1):
            logger.info(f"Processing datacenter {dc_idx}/{dc_count}: {dc_obj.name}")
            temp_datastores = []
            datastores_object_view = (
                vsphere_service_content.viewManager.CreateContainerView(
                    dc_obj, [vim.Datastore], True
                )
            )
            ds_count = len(datastores_object_view.view)
            logger.info(f"Found {ds_count} datastores in datacenter {dc_obj.name}")
            total_datastores += ds_count

            for ds_obj in datastores_object_view.view:
                temp_datastores.append(ds_obj)
            # Add all of the datastores to the target object. We need the datacenter information for later
            targets.append({"datacenter": dc_obj, "datastores": temp_datastores})
            datastores_object_view.Destroy()

        datacenters_time = time.time() - datacenters_start
        logger.info(f"Datacenter and datastore enumeration completed in {datacenters_time:.2f}s")
        logger.info(f"Identified {total_datastores} datastores across {dc_count} datacenters")

        # Clean up the views now that we have what we need
        datacenters_object_view.Destroy()

        # VMware logfile scanning
        files_to_scan = []
        logger.info("Beginning scan for logfiles in datastores")
        scan_start_time = time.time()

        # Loop over the targets list. These represent a list of the datacenters & associated datastores available
        for target_idx, target in enumerate(targets, 1):
            dc_name = target["datacenter"].name
            logger.debug(f"Processing datacenter {target_idx}/{len(targets)}: {dc_name}")

            for ds_idx, datastore in enumerate(target["datastores"], 1):
                logger.info(f"Scanning datastore {ds_idx}/{len(target['datastores'])}: {datastore.name} in datacenter {dc_name}")
                # Build list of logfiles on the datastore
                logfiles = search_datastore_for_logfiles(datastore)
                if logfiles:
                    logger.debug(f"Processing {len(logfiles) if isinstance(logfiles, list) else 1} search results")
                    for search_result in logfiles:
                        # Break out the list of logfiles & create a list of files to scan (including the datacenter since we need
that)
                        new_files = generate_individual_files(search_result, target["datacenter"])
                        if new_files:
                            logger.info(f"Found {len(new_files)} log files to scan in {datastore.name}")
                            files_to_scan.extend(new_files)
                else:
                    logger.info(f"No log files found in datastore {datastore.name}")

        scan_duration = time.time() - scan_start_time
        logger.info(f"Log file discovery completed in {scan_duration:.2f}s, found {len(files_to_scan)} files to scan")

        # For each file to scan, call the search function & determine if there were any hits
        logger.info(f"Scanning {len(files_to_scan)} log files for indicators of compromise")
        files_with_hits = 0

        # Get a Requests Session object that we can reuse it rather than issuing numerous individual requests for downloading files

        session = requests.Session()
        session.cookies.update(get_vsphere_cookie(si))
        if args.disable_ssl_verify:
            session.verify = False
            logger.debug("SSL certificate verification disabled for download")
            # disable urllib3 warnings
            requests.packages.urllib3.disable_warnings(
                requests.packages.urllib3.exceptions.InsecureRequestWarning
            )
```

```
        for file_idx, log_file in enumerate(files_to_scan, 1):
            if logger_debug_enabled and file_idx % 10 == 0:  # Log progress every 10 files if in debug mode
                logger.debug(f"Scanning file {file_idx}/{len(files_to_scan)} ({(file_idx/len(files_to_scan)*100):.1f}%)")

            logfile_ghost_scan_return = search_logfile_for_ghosts(
                log_file, si, session, output_dir
            )
            if logfile_ghost_scan_return:
                # If we had hits from the vmware logfile scan
                files_with_hits += 1
                INDICATORS_FOUND = True

        logger.info(f"Log file scanning complete. Found indicators in {files_with_hits} files")

        # Provide the final message(s) to the user for their review
        script_duration = time.time() - script_start_time
        logger.info(f"GHOSTHUNTER execution completed in {script_duration:.2f} seconds")

        if INDICATORS_FOUND:
            print(
                "\a\n"
            )  # Generate a noise (where able) & add in a newline to make it extra obvious
            logger.warning(
                f"===== One or more indicators of potential compromise found. ====="
            )
            print("\n")
        if NEEDS_REVIEW:
            logger.warning(
                f"===== One or more items of potential concern were identified. These items do not necessarily indicate a
compromise, but should be investigated to ensure that there is a legitimate reason for them to have occurred. ====="
            )
        logger.info(
            f"Please review the logfile that was generated at {log_filename} for additional details."
        )

        # Return success
        return 0

    except Exception as e:
        # Enhanced error reporting with context
        error_context = {
            'timestamp': datetime.now().isoformat(),
            'operation': sys._getframe().f_code.co_name
        }

        # Add args context if available
        try:
            if 'args' in locals():
                error_context.update({
                    'server': getattr(args, 'server', 'unknown'),
                    'user': getattr(args, 'user', 'unknown')
                })
        except Exception:
            pass

        # Use logger if initialized, otherwise fall back to standard logging
        if local_logger and logger_initialized:
            logger.error(f"Error: {e}", exc_info=True)
            logger.error(f"Error context: {error_context}")
        else:
            logging.error(f"Error: {e}", exc_info=True)
            logging.error(f"Error context: {error_context}")
        return 1


if __name__ == "__main__":
    sys.exit(main())
```

CROWDSTRIKE

CrowdStrike Services

If you require additional assistance, please reach out to CrowdStrike's support team at **support@crowdstrike.com** or contact us via phone:

Americas/Canada	+1 888 512 8906
UK/Ireland	+44(0) 118 453 0400
Australia/New Zealand/APAC	(+61) 1300 245 584
Middle East/Turkey/Africa	+9714 429 5829

If further help is required and you would like to engage CrowdStrike's Incident Response team, please contact Professional Services by completing the form on **https://www.crowdstrike.com/experienced-a-breach**, or contact us via phone:

Americas/Canada	+1 855 276 9347
UK/Ireland	+44(0) 800 0487187
France	+33 801840073
Germany	+49 (0800) 3252669
Australia	(+61) 1800 290 853
Japan	+81 800 170 5401
India	+91 1800 040 3447
Saudi Arabia	+966 8008803012
UAE	+971 8000320534
Qatar	+974 800101302

Customers with an active CrowdStrike Services retainer should notify the Services team in accordance with the process outlined in your retainer agreement.

CrowdStrike — the world's leader in Endpoint Protection. We stop breaches.

Copyright © 2025 CrowdStrike, Inc. All rights reserved.