Kaylyn King, Tasmania Tamboli, Jean-Charles Hekamanu, Ujjwal Rana Magar
CSCE 5550
Spring 2025

Step 4: Monitoring

Libraries

Our script uses the python watchdog library to actively monitor our “critical” folder. Along with watchdog, we added the “auditd” command to our Linux machine to get more information about the processes used during this time.

Auditd

“Auditd” is a good command to track fast processes like the “curl” command we are using in our ransomware script.

The following code runs the command “systemctl is-active –quiet auditd” to start the auditing process on Linux.

```
# Run auditd to catch the curl process
def run_auditd():
    try:
        subprocess.run(["systemctl", "is-active", "--quiet", "auditd"],
check=True)
    except subprocess.CalledProcessError:
        print("starting auditd")
        try:
            subprocess.run(["sudo", "systemctl", "start", "auditd"],
check=True)
            print(f"auditd started successfully")
        except subprocess.CalledProcessError:
            print(f"failed to start auditd")
```

The following code runs the command “auditctl -w “./critical” -p rwxa -k “critical watch” to add the audit rule to the user’s Linux machine if they have not already. The audit is set to watch the “critical” folder with a key of “critical watch” to make filtering easier.

```
def add_audit(path_to_watch):
    try:
        subprocess.run(["auditctl", "-w", path_to_watch, "-p", "rwxa",
"-k", AUDIT_KEY], check=True)
        print(f"rule added for {path_to_watch}")
    except subprocess.CalledProcessError:
```

```
        print(f"audit rule already exists")
```

The following code runs the command “sudo ausearch -k “critical watch” -ts recent to parse the audit to find the process id that altered the files in the “./critical” directory.

```
def pid_from_audit():
    try:
        output = subprocess.check_output(["ausearch", "-k", AUDIT_KEY,
"-ts", "recent"]).decode()
        for line in reversed(output.splitlines()):
            if "pid=" in line:
                parts = line.strip().split()
                for part in parts:
                    if part.startswith("pid="):
                        return int(part.split("=")[1])
    except subprocess.CalledProcessError:
        pass
    return None
```

Logging

Our script logs the data to a .json file, so it can be easier to transfer the data to a database if needed. The timestamp, what was done to the files (add, moved, deleted, modified), file path, process that modified the file, and what action was taken are all logged to the .json file for further inspection. For now, the action will be logging until we reach the mitigation stage.

The following is the code for writing to the .json file. For now, it also writes to the command prompt for debugging.

```
# Write to a json log file
def log_event(event_type, file_path, pid):
    log_entry = {
        "timestamp": datetime.now().isoformat(),
        "event_type": event_type,
        "file": file_path,
        "pid": pid,
        "action_taken": "logged"
    }
    with open(LOG_FILE, "a") as f:
        f.write(json.dumps(log_entry) + "\n")
    print(f"[{log_entry['timestamp']}] {event_type.upper()} | File:
{file_path} | pid: {pid}")
```

Kaylyn King, Tasmania Tamboli, Jean-Charles Hekamanu, Ujjwal Rana Magar
CSCE 5550
Spring 2025

Our Script

This is our python code for monitoring the “./critical” folder:

```
# Group 1
# Monitoring Assignment
# CSCE 5550
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
import time
import psutil
import json
import os
import subprocess
from datetime import datetime

# setup variables, watchdog will be watching the "critical" folder and
write to monitoring_log.json
LOG_FILE = "monitoring_log.json"
WATCH_PATH = os.path.abspath("./critical")
AUDIT_KEY = "critical_watch"

# Run auditd to catch the curl process
def run_auditd():
    try:
        subprocess.run(["systemctl", "is-active", "--quiet", "auditd"],
check=True)
    except subprocess.CalledProcessError:
        print("starting auditd")
        try:
            subprocess.run(["sudo", "systemctl", "start", "auditd"],
check=True)
            print(f"auditd started successfully")
        except subprocess.CalledProcessError:
            print(f"failed to start auditd")

# Write to a json log file
def log_event(event_type, file_path, pid):
```

```
    log_entry = {
        "timestamp": datetime.now().isoformat(),
        "event_type": event_type,
        "file": file_path,
        "pid": pid,
        "action_taken": "logged"
    }
    with open(LOG_FILE, "a") as f:
        f.write(json.dumps(log_entry) + "\n")
    print(f"[{log_entry['timestamp']}] {event_type.upper()} | File:
{file_path} | pid: {pid}")

# Add the auditd rule to linux to find fast processes
def add_audit(path_to_watch):
    try:
        subprocess.run(["auditctl", "-w", path_to_watch, "-p", "rwxa",
"-k", AUDIT_KEY], check=True)
        print(f"[AUDIT] Rule added for {path_to_watch}")
    except subprocess.CalledProcessError:
        print(f"Audit rule already exists")

# Find pid of the ransomware from the audit
def pid_from_audit():
    try:
        output = subprocess.check_output(["ausearch", "-k", AUDIT_KEY,
"-ts", "recent"]).decode()
        for line in reversed(output.splitlines()):
            if "pid=" in line:
                parts = line.strip().split()
                for part in parts:
                    if part.startswith("pid="):
                        return int(part.split("=")[1])
    except subprocess.CalledProcessError:
        pass
    return None

# Separate the actions
class MonitorHandler(FileSystemEventHandler):
    def handle_event(self, event_type, event):
```

```
        pid = pid_from_audit()
        if pid:
            time.sleep(0.2)
        else:
            pid = "Unknown"

        log_event(event_type, event.src_path, pid)

    def on_created(self, event):
        self.handle_event("created", event)

    def on_modified(self, event):
        self.handle_event("modified", event)

    def on_deleted(self, event):
        self.handle_event("deleted", event)

    def on_moved(self, event):
        self.handle_event("moved", event)

# Continuously monitor the folder
if __name__ == "__main__":
    if not os.path.exists(WATCH_PATH):
        os.makedirs(WATCH_PATH)

    run_auditd()

    add_audit(WATCH_PATH)

    handler = MonitorHandler()
    observer = Observer()

    path_to_watch = "./critical"
    observer.schedule(handler, path=WATCH_PATH, recursive=True)
    observer.start()

    print(f"Monitoring started on: {WATCH_PATH}")
    print(f"Logging to: {os.path.abspath(LOG_FILE)}")
```

```
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print(f"Monitoring stopped")
        observer.stop()

    observer.join()
    subprocess.run(["auditctl", "-W", WATCH_PATH, "-k", AUDIT_KEY])
```

The following is the bash script to compile and run the python code:

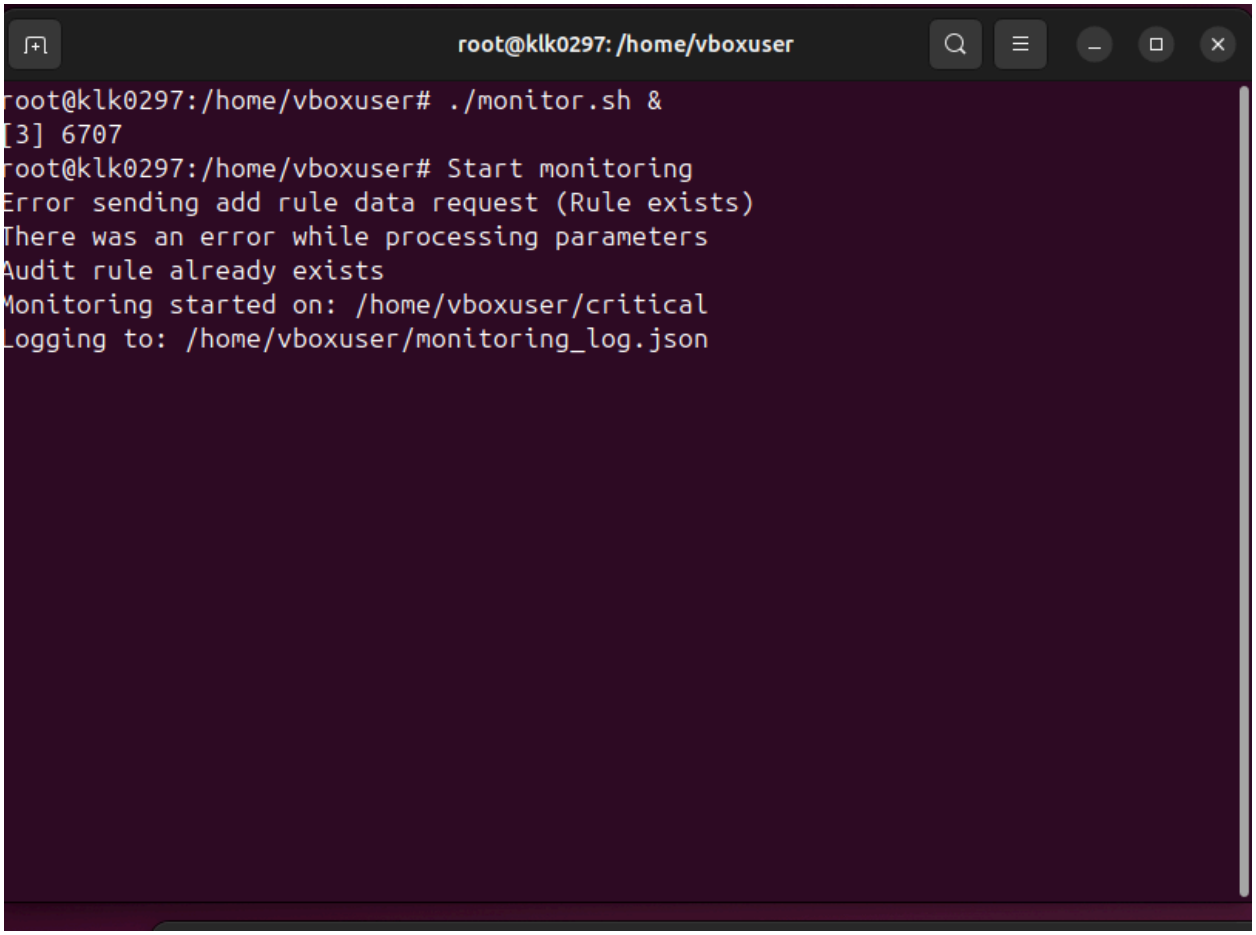
```
#!/bin/bash

echo "Start monitoring"

python3 ./monitor.py
```

Results

Compiling

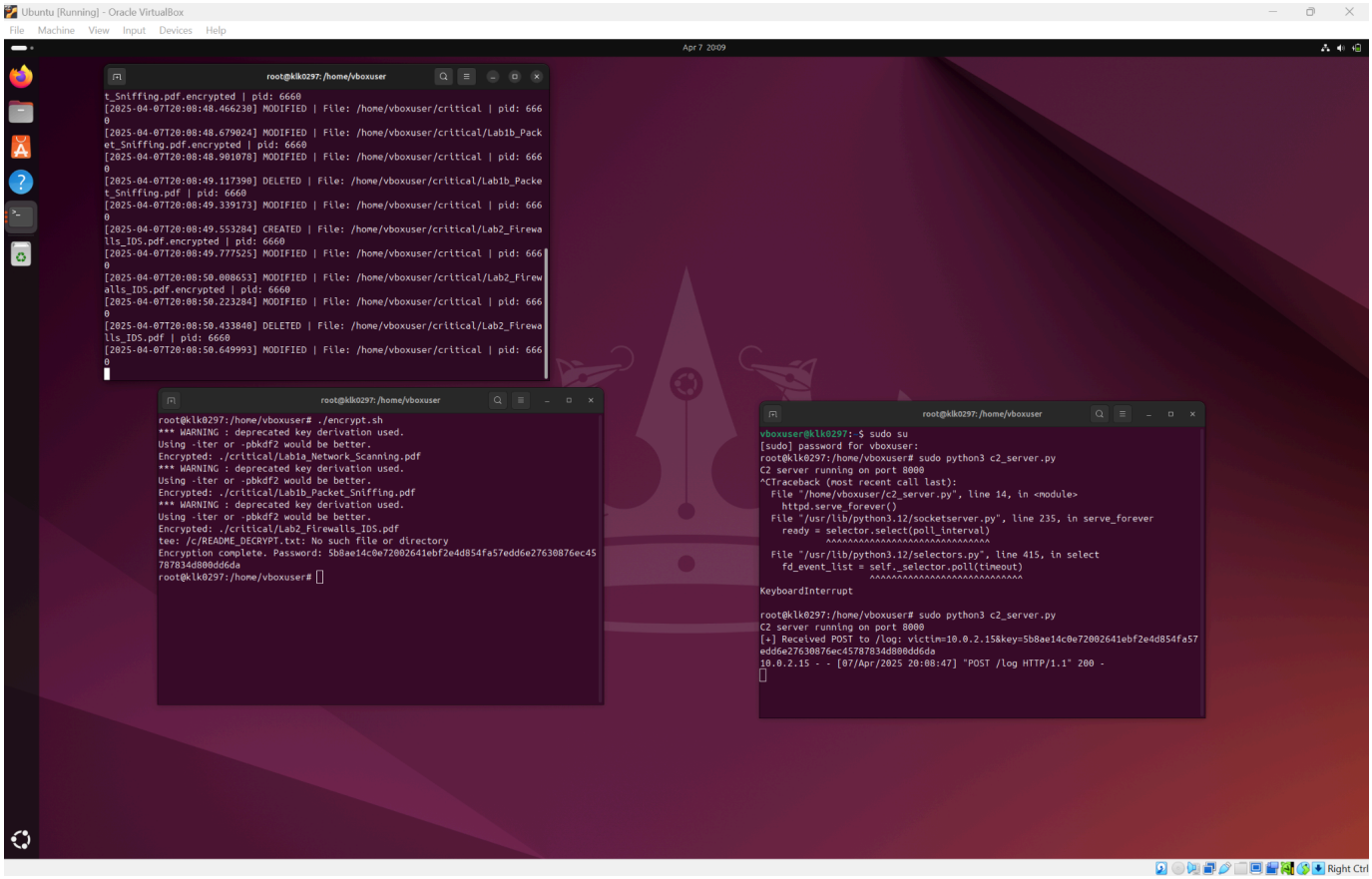
“./monitor.sh &” continuously runs the monitoring script without disrupting the current terminal from use. This should be used before running the encryption script. The following screenshot shows the result of running the command:

Kaylyn King, Tasmania Tamboli, Jean-Charles Hekamanu, Ujjwal Rana Magar
CSCE 5550
Spring 2025



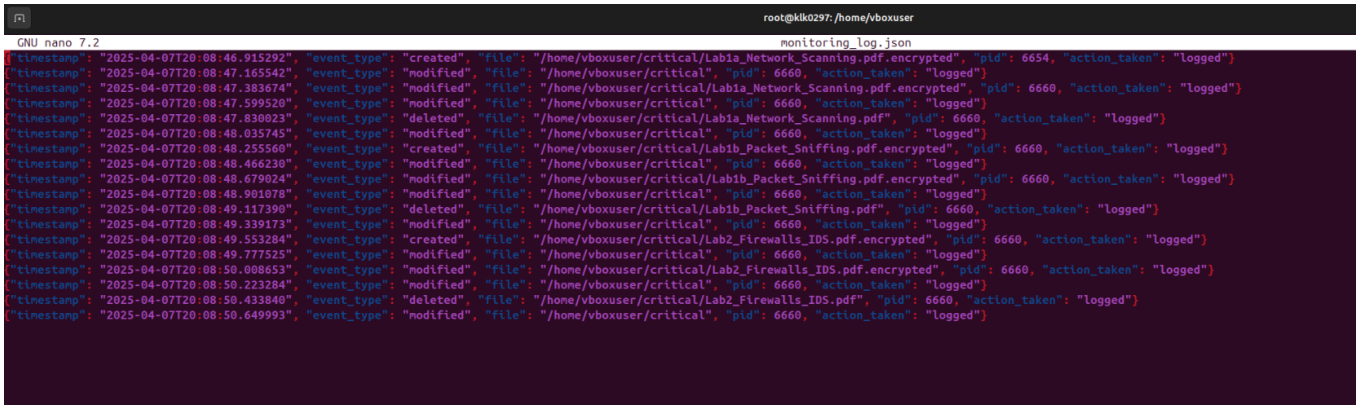
Kaylyn King, Tasmania Tamboli, Jean-Charles Hekamanu, Ujjwal Rana Magar
CSCE 5550
Spring 2025

Results on Terminal



.json log

The following screenshot shows the data saved in the .json file:



Kaylyn King, Tasmania Tamboli, Jean-Charles Hekamanu, Ujjwal Rana Magar
CSCE 5550
Spring 2025

Next Steps

In the next assignment, we will set permission rules to detect the malicious ransomware code with the information we gathered from monitoring the system.