Mrs V.Roopa, 2025, 13:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology

An Open Access Journal

Research Article
System core and mesh analyser

Mrs V.Roopa, Hariharan R, Anbarasu P, Eraiamudhan VD.
1. A/P Department of Cyber Security.
2,3,4 UG of Department of Cyber Security.

Abstract- The Python programs serve as comprehensive system diagnostic and security auditing tools. The first script utilizes the psutil and platform libraries to collect and display detailed system information, including operating system details, CPU specifications, memory usage, storage data, and boot time. It ensures the required psutil package is installed dynamically, making the script portable and robust. This tool is useful for monitoring resource usage and understanding system performance in real time.The second script focuses on basic security auditing and vulnerability scanning. It checks for the presence of known vulnerable packages (e.g., older versions of gitpython), scans the filesystem for common misconfigurations such as hardcoded secrets and insecure file permissions, and categorizes the associated risk levels. Additionally, it ensures that the pkg_resources module is available by managing the installation or upgrade of setuptools if needed.

Keywords: System Monitoring, Security Audit, Vulnerability, Detection RCE (Remote Code Execution), Hardcoded Secrets, Python Automation System Info, Python Security Tool

I. INTRODUCTION

This project introduces two essential Python scripts aimed at system monitoring and security auditing. The first script is focused on gathering real-time system diagnostics using the psutil and platform libraries. It provides detailed insights into system specifications such as CPU usage, core count, memory status, disk usage, system boot time, and operating system details. This script is especially useful for developers, administrators, or any users who need a quick snapshot of their system's performance and health. It ensures that the necessary psutil package is available, installing it if needed, and handles potential permission errors gracefully when accessing disk partitions. On the other hand, the second script targets basic security checks, helping to identify potential risks in the system. It starts by ensuring that setuptools and the pkg_resources module are installed so that installed packages and their versions can be properly assessed. It then checks for outdated or vulnerable versions of known Python packages (such as gitpython), which could be susceptible to Remote Code Execution (RCE) vulnerabilities. In addition to dependency checks, the script also scans the file system for common misconfigurations, such as hardcoded secrets within Python files or unsafe permissions on sensitive files like .env or config.py. It flags these issues and classifies them based on their risk level—high, medium, or low—providing clear guidance for remediation. Together, these scripts offer a simple but powerful toolkit for maintaining a secure and efficient computing environment. They can be valuable in personal projects, educational settings, or lightweight DevOps workflows, serving as a starting point for more advanced monitoring and security tools. Their modular design and readability make them easy to extend or integrate into larger automation frameworks, demonstrating the power and flexibility of Python for real-world system administration tasks. System Monitoring Script

Overview

The first script is a compact, yet powerful tool that helps in understanding how a system is currently performing. Its core functionality is built around psutil, a library that allows access to system-level information in a platform-independent way.

Key features of the system monitoring script include:

System Information:

The script retrieves and displays general system details such as the operating system name, version, machine type, processor info, and node (host) name. This is useful for quickly identifying the environment the script is running on.

Boot Time:
It reports the last boot time of the machine. This information can be helpful in determining uptime or identifying if a system has restarted unexpectedly.

CPU Details:
The script calculates and displays the number of physical and logical CPU cores. It also reads the CPU frequency and tracks per-core and total CPU usage. This is especially useful in environments where performance bottlenecks may be CPU-related.

Memory Usage:
Memory status is shown, including total, available, used memory, and the percentage of memory currently in use. This is critical in assessing whether the system has enough resources available to run applications efficiently.

Disk Usage:
It iterates through all disk partitions and provides data on their total size, used and free space, and usage percentage. It gracefully handles permission errors, which can occur if certain partitions are restricted or system-reserved.

All of this information is gathered and printed in a human-readable format, making it a handy diagnostic tool. Whether you're setting up a new server, troubleshooting an issue, or just curious about your system's performance, this script provides the snapshot you need.

Security Audit Script Overview
While performance monitoring is important, security cannot be overlooked—especially in today's threat landscape. Systems that are not regularly audited may unknowingly harbor vulnerabilities, either due to outdated software or poor configuration practices.

This is where the second script comes in. Its primary goal is to detect known package vulnerabilities and unsafe coding or configuration practices, helping the user proactively identify and mitigate security risks.

Here's what the script does:
Dependency Check & Auto-install:
Before performing any analysis, the script checks for the presence of the pkg_resources module, which is part of setuptools. If it is missing, the script attempts to install or upgrade setuptools automatically. This ensures it can safely retrieve version data for installed Python packages.

Vulnerability Detection:
The script has a built-in dictionary of packages and their safe version thresholds. For instance, it checks if an older, vulnerable version of the gitpython package is installed. If such a package is found, the script lists it and highlights the risk. This could be extended to support more packages and integrate with known vulnerability databases like CVE or PyUp.

Code Misconfiguration Checks:
One of the more advanced features is scanning the file system (starting at /) for potential misconfigurations:

Hardcoded Secrets: It looks for keywords like API_KEY or SECRET in Python source files. These could represent sensitive credentials that were accidentally committed to the codebase.

nsecure Permissions: The script checks whether certain config files, like config.py or .env, are too permissive (e.g., world-readable or writable), which could pose a security risk.

Risk Assessment:
For each issue found, the script assigns a risk level—high, medium, or low—based on the nature of the problem. This helps prioritize fixes.

Execution Summary:
At the end of its run, the script prints out all findings along with the time taken to complete the scan. This makes it easy to integrate into automated environments or log for future reference.

Scripts Matter

In modern computing environments, especially where DevOps practices are prevalent, automation plays a central role. These scripts, though simple, reflect the kind of automation that can save time, reduce errors, and improve security.

From a learning standpoint, they also demonstrate how powerful Python is for scripting real-world system tasks. The psutil and subprocess libraries alone open up a wide range of capabilities, from monitoring to automated installations and checks.
For anyone managing a Linux server, a cloud VM, or even a personal development machine, these tools can provide quick insights and early warnings about system issues or vulnerabilities.

II. WORKING PROCESS

Working Process of System Monitoring and Security Auditing Scripts

The system monitoring and security auditing scripts work synergistically to ensure the health and security of the system. These scripts are critical for system administrators, developers, or anyone working with a Python-based environment, as they provide a comprehensive approach to both performance and security analysis. By focusing on key system metrics and known vulnerabilities, these scripts allow for easy monitoring and vulnerability management without requiring heavy third-party tools or complex configurations. This content delves into the step-by-step workings of both scripts, explaining the critical components, their roles, and how they interact with the underlying system.

System Monitoring Script: Comprehensive Resource Management
The system monitoring script is designed to provide real-time data on the various critical resources of a computer system, ensuring that administrators or users can make informed decisions when optimizing their systems for performance or troubleshooting issues. The core objective of the monitoring script is to track the health of essential system resources such as CPU, memory, storage, and system uptime, which are the primary indicators of system performance.

Installing Required Libraries
One of the first steps the script takes is checking for the required library, psutil, which is essential for retrieving system performance metrics. psutil is a powerful Python library that can interface with the underlying system to fetch real-time data about CPU usage, memory usage, disk usage, and more. If the library is not installed on the system, the script automatically handles the installation process via Python's package manager, pip. This ensures that the script is always ready to run, regardless of the environment, without requiring manual dependency installation from the user.

Gathering General System Information
Once the required libraries are available, the script gathers general system information. This includes the operating system, node name (which refers to the system's hostname), release and version numbers, machine type, and processor details. This data is crucial for administrators to get an overview of the system's architecture, ensuring that it aligns with the intended usage and configurations. The system's name and version are particularly helpful when determining compatibility with specific tools, software, or operating environments.

System Uptime and Boot Time
System uptime is a critical aspect of monitoring, as it provides insight into the last time the system was rebooted. Knowing the system's boot time can help diagnose performance issues, as systems that have been running for long periods without a reboot may start showing degradation in performance or stability. Using psutil, the script retrieves the exact timestamp when the system was last booted. This timestamp is converted into a human-readable format, making it easy for users to understand when the system was last restarted. This metric can also be used to correlate system failures or crashes with uptime, identifying whether the system is running as expected or if it needs a reboot for optimal performance.

CPU Usage and Performance Metrics

One of the most important resources on any system is the CPU, as it handles all computational tasks. The script gathers detailed information about the system's CPU performance, such as the number of physical and logical cores. Physical cores refer to the actual hardware units responsible for processing, while logical cores account for the CPU's ability to perform multiple tasks concurrently via hyper-threading.

The script also tracks CPU frequencies, reporting the maximum, minimum, and current operating frequencies of the CPU. This data helps in identifying if the system is being throttled or if the CPU is operating below its expected performance range. The script provides real-time CPU usage statistics per core and an overall total CPU usage percentage, allowing users to see which cores are under load and how the workload is distributed across them. These metrics are essential for detecting performance bottlenecks, high CPU utilization, or identifying if certain processes are monopolizing system resources.

Memory Usage Monitoring

Memory usage is another vital system resource that needs continuous monitoring. The script uses psutil to retrieve virtual memory data, including total, available, and used memory. Memory consumption can impact the overall performance of the system, particularly in environments running memory-intensive applications. High memory usage can cause the system to slow down or even crash if it runs out of available RAM.

The script presents the memory usage in human-readable terms such as megabytes (MB), making it easier to understand and track usage trends over time. It also calculates and displays the percentage of memory being utilized. High memory usage may indicate that certain applications or services are consuming an abnormal amount of resources, which may require intervention to ensure smooth system operation.

Disk Usage and Storage Analysis

Disk storage is a finite resource, and it is crucial to monitor the available space to avoid system crashes or performance degradation. The system monitoring script analyzes all mounted disk partitions and gathers information such as the total size, used space, and free space available for each partition. It also calculates the usage percentage for each partition, alerting users to partitions that are nearing full capacity.

Knowing the available storage on critical system partitions is important, especially for systems running databases, log files, or applications that require constant write access. If the disk space becomes too limited, it can lead to data loss, file corruption, or system failures. Therefore, this part of the script ensures that users have the most up-to-date information about their storage, allowing them to take proactive measures before running out of disk space.

Security Auditing Script: Identifying Vulnerabilities and Misconfigurations

The security auditing script is designed to detect potential security vulnerabilities in the system's configuration and package dependencies. By focusing on security risks such as outdated software versions, misconfigurations, and hardcoded secrets, this script helps administrators maintain a secure environment by ensuring that the system is not susceptible to known vulnerabilities.

Ensuring Necessary Libraries Are Installed

Just as with the system monitoring script, the security auditing script starts by ensuring that necessary Python modules are available. One critical module for this script is pkg_resources, which is part of the setuptools package. This module is used to inspect the versions of installed packages. If pkg_resources is not available, the script automatically installs or upgrades setuptools, ensuring that the script can access the required functionality to perform vulnerability checks.

Vulnerability Scanning of Installed Packages

The script checks for known vulnerabilities in installed packages by maintaining a list of packages that are known to have security flaws. For example, older versions of certain libraries may have security vulnerabilities that allow attackers to exploit them for remote code execution (RCE) or other malicious activities. The script compares the installed versions of critical packages with a predefined list of vulnerable versions, flagging any outdated packages.

By identifying vulnerable packages, the script helps administrators take timely action, either by upgrading the vulnerable packages or by finding secure alternatives. This proactive approach helps reduce the system's exposure to potential security risks.

Scanning for Misconfigurations and Hardcoded Secrets
Misconfigurations are a common source of security breaches, as they can inadvertently expose sensitive information or grant excessive permissions. The security auditing script scans the entire file system for Python files that may contain hardcoded secrets, such as API keys, passwords, or tokens. These secrets can easily be exposed in public code repositories or shared inadvertently with unauthorized individuals. The script looks for specific keywords like API_KEY or SECRET within the code, identifying files that may contain sensitive information.

The script also checks the permissions of critical files such as .env and config.py. These files often contain sensitive configuration settings, and it is essential to ensure that they are not overly permissive. The script evaluates file permissions and flags any files that are accessible by unauthorized users, helping administrators secure these files by adjusting their permissions.

Risk Assessment and Reporting
Once vulnerabilities and misconfigurations are identified, the script assigns a risk level to each issue. For example, hardcoded secrets are typically classified as high risk, as they can lead to significant data breaches if exposed. Insecure file permissions are usually classified as medium risk, as they may allow unauthorized users to access sensitive configuration data but do not necessarily guarantee a full security compromise. The script categorizes the severity of each issue, making it easier for users to prioritize remediation efforts based on the potential impact.

The security auditing script then provides a detailed report, listing all identified vulnerabilities and misconfigurations, along with their associated risk levels. This allows administrators to understand the current security posture of their system and take corrective action as necessary.

III. RESULTS AND DISCUSSION

This discussion evaluates the results and implications of running two Python-based scripts developed for system monitoring and basic security auditing. These scripts demonstrate how lightweight, open-source Python libraries can be leveraged to gather detailed system diagnostics and identify potential security issues, such as misconfigurations and outdated or vulnerable packages.

System Monitoring Script
The first script utilizes the psutil and platform modules to extract and display a variety of system-level metrics. The script dynamically checks for and installs psutil if it's not already available, ensuring it can run in environments with minimal setup.
Operating System and Hardware Information
The platform.uname() function is used to extract basic system information, including:

- System Name: Identifies the operating system (e.g., Windows, Linux).
- Node Name: The device or hostname.
- Release and Version: Helps determine compatibility with software or tools.
- Machine Type and Processor Info: Indicates architecture (e.g., x86_64, ARM) and CPU details.
- These details are important for both compatibility and audit purposes, especially in environments where systems vary significantly. It allows administrators to have a clear record of what kind of hardware and software their applications are running on.

Boot Time and Uptime
By using psutil.boot_time(), the script converts the system's boot timestamp into a human-readable

format. This tells us when the system was last restarted. This is useful for understanding how long a system has been running continuously (uptime), which can be relevant for performance degradation assessments, patch cycle planning, and general maintenance routines.

CPU Information

The script uses several psutil features to report:

Physical and Logical Cores: Helpful for understanding CPU virtualization.

CPU Frequencies: Shows minimum, maximum, and current frequencies, allowing for detection of throttling or frequency scaling.

Per-Core CPU Usage: Essential for identifying overloaded or underutilized cores.

Total CPU Usage: Gives an overall load snapshot.

These details help in performance analysis. For instance, if one core is consistently at high usage while others are idle, it may indicate a single-threaded application that needs optimization.

Memory Usage Statistics

Using psutil.virtual_memory(), the script reports

- Total RAM
- Used RAM
- Available RAM
- Percentage of RAM in Use

These values are especially important when evaluating the performance of applications that consume large amounts of memory or diagnosing memory leaks.

Disk and Storage Information

Finally, the script uses psutil.disk_partitions() and psutil.disk_usage() to retrieve disk usage stats:

Total, Used, and Free Disk Space

Disk Usage Percentages per partition

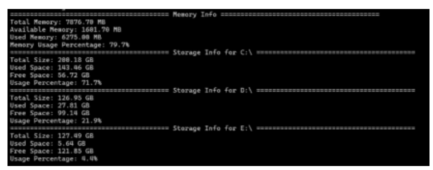
This provides insight into whether storage is nearing capacity — a critical point in preventing data loss or downtime due to insufficient space. It gracefully handles partitions where permission is denied, ensuring it doesn't crash.

Summary of System Monitoring Script

The script successfully offers a holistic view of system health and resources. In a production environment, this can be particularly helpful for setting performance baselines, conducting pre-deployment checks, and performing post-incident diagnostics. It is also a valuable tool for system administrators and developers who need real-time insights into the environments their applications run on.

Output:

```
C:\Users\Hariharan\Documents>python s.py
======================================== System Information ========================================
System: Windows
Node Name: DESKTOP-OCANSNG
Release: 11
Version: 10.0.26100
Machine: AMD64
Processor: Intel64 Family 6 Model 186 Stepping 3, GenuineIntel
======================================== Boot Time ========================================
Boot Time: 2025/4/13 13:40:49
======================================== CPU Info ========================================
Physical cores: 10
Total cores: 12
Max Frequency: 1300.00Mhz
Min Frequency: 0.00Mhz
Current Frequency: 1300.00Mhz
CPU Usage Per Core:
Core 0: 0.0%
Core 1: 12.1%
Core 2: 3.1%
Core 3: 0.0%
Core 4: 0.0%
Core 5: 0.0%
Core 6: 0.0%
Core 7: 0.0%
Core 8: 0.0%
Core 9: 0.0%
Core 10: 0.0%
Core 11: 0.0%
Total CPU Usage: 1.7%
======================================== Memory Info ========================================
Total Memory: 7876.70 MB
Available Memory: 1601.70 MB
Used Memory: 6275.00 MB
```



```
======================================== Memory Info ========================================
Total Memory: 7876.70 MB
Available Memory: 1601.70 MB
Used Memory: 6275.00 MB
Memory Usage Percentage: 79.7%
======================================== Storage Info for C:\ ========================================
Total Size: 200.18 GB
Used Space: 143.46 GB
Free Space: 56.72 GB
Usage Percentage: 71.7%
======================================== Storage Info for D:\ ========================================
Total Size: 126.95 GB
Used Space: 27.81 GB
Free Space: 99.14 GB
Usage Percentage: 21.9%
======================================== Storage Info for E:\ ========================================
Total Size: 127.49 GB
Used Space: 5.64 GB
Free Space: 121.85 GB
Usage Percentage: 4.4%
```

Security and Vulnerability Auditing Script

The second script is geared towards identifying basic security flaws and vulnerabilities in a Python environment. This script is not a substitute for enterprise-level vulnerability scanners, but it effectively demonstrates the potential of Python to assist in security hygiene monitoring.

Package Vulnerability Check

The script checks the environment for known vulnerable packages. In this case, it looks for gitpython versions older than 3.1.30, known for potential remote code execution (RCE) vulnerabilities. By comparing installed versions using pkg_resources, the script identifies any packages that may pose a threat due to unpatched vulnerabilities. Misconfiguration Detection

The script scans the entire filesystem recursively to check for common misconfigurations:

Hardcoded Secrets in Python Files: By scanning .py files for keywords like API_KEY or SECRET, the script flags locations where sensitive information may be exposed. Secrets hardcoded in source files can easily leak through version control or deployment, making them one of the most dangerous yet common coding mistakes.

File Permission Issues: Files like .env and config.py, which typically contain sensitive environment variables or credentials, are scanned for insecure file permissions. Permissions such as 777 or 755 can make these files readable or writable by unauthorized users.

Risk Assessment Logic

The script uses simple rules to classify findings into:

High Risk: For secrets exposed in code

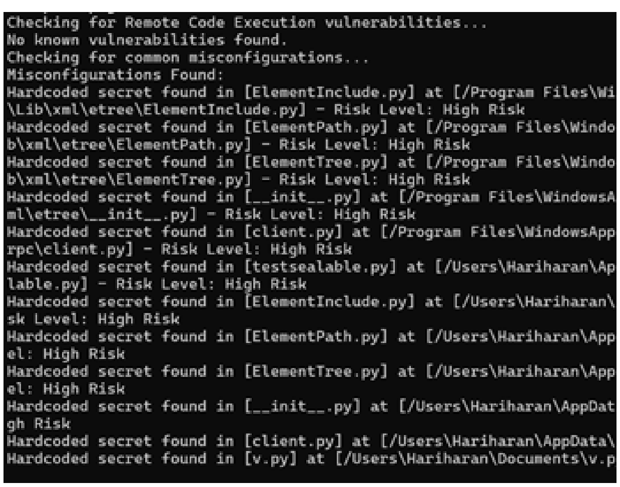
Medium Risk: For misconfigured permissions

Low Risk: Placeholder for any other minor concerns

While basic, this form of risk classification introduces security awareness and highlights the importance of prioritizing remediation steps based on severity.

When run together, these two scripts form a solid foundation for automated diagnostics and risk analysis. They are lightweight, easily extendable, and offer several benefits

Output:



```
Checking for Remote Code Execution vulnerabilities...
No known vulnerabilities found.
Checking for common misconfigurations...
Misconfigurations Found:
Hardcoded secret found in [ElementInclude.py] at [/Program Files\Wi
\Lib\xml\etree\ElementInclude.py] - Risk Level: High Risk
Hardcoded secret found in [ElementPath.py] at [/Program Files\Windo
b\xml\etree\ElementPath.py] - Risk Level: High Risk
Hardcoded secret found in [ElementTree.py] at [/Program Files\Windo
b\xml\etree\ElementTree.py] - Risk Level: High Risk
Hardcoded secret found in [__init__.py] at [/Program Files\WindowsA
ml\etree\__init__.py] - Risk Level: High Risk
Hardcoded secret found in [client.py] at [/Program Files\WindowsApp
rpc\client.py] - Risk Level: High Risk
Hardcoded secret found in [testsealable.py] at [/Users\Hariharan\Ap
lable.py] - Risk Level: High Risk
Hardcoded secret found in [ElementInclude.py] at [/Users\Hariharan\
sk Level: High Risk
Hardcoded secret found in [ElementPath.py] at [/Users\Hariharan\App
el: High Risk
Hardcoded secret found in [ElementTree.py] at [/Users\Hariharan\App
el: High Risk
Hardcoded secret found in [__init__.py] at [/Users\Hariharan\AppDat
gh Risk
Hardcoded secret found in [client.py] at [/Users\Hariharan\AppData\
Hardcoded secret found in [v.py] at [/Users\Hariharan\Documents\v.p
```

Practical Benefits

1. Improved Visibility: Knowing your system's CPU, memory, disk usage, and software versions helps in operational decision-making and capacity planning.
2. Early Detection of Issues: Detecting secrets and misconfigurations early in development or deployment can prevent major security breaches.
3. Automation-Ready: These scripts can be easily integrated into DevOps or CI/CD pipelines to ensure that system state and code security meet predefined requirements before software moves through development stages.
4. Learning Tool: These tools are also ideal in academic or training settings, helping students understand system internals and secure coding practices.

Limitations

Despite their usefulness, there are some limitations to note:

- Static Vulnerability Database: The package check relies on a hardcoded list of vulnerable packages. Without integration with CVE feeds or vulnerability APIs, it will quickly become outdated.
- Performance Bottleneck: Scanning the entire filesystem for .py files can be slow and resource-intensive. Optimizing it to scan only known source code directories would be more efficient.
- False Positives: Secret detection uses keyword matching, which may flag non-sensitive variables. This can result in false alarms.
- Permissions: Certain directories and files may require elevated privileges to read or scan, potentially reducing the completeness of the scan on a restricted user account.
- Opportunities for Enhancement
- To make these scripts more robust and production-ready, the following improvements are suggested:
- Configurable Scan Paths: Allow the user to define which directories to scan for faster and more relevant results.
- Integration with CVE/NVD Feeds: Automate vulnerability detection using real-time data from official security databases.

- Use of Secret Detection Libraries: Integrate with tools like detect-secrets, truffleHog, or git-secrets for advanced secret scanning.
- Reporting: Export results in .txt, .json, or .html formats for audit trails or automated reporting.
- Scheduled Execution: Integrate with task schedulers like cron or Windows Task Scheduler for regular scans.
- GUI or Web Interface: Build a dashboard using Tkinter, Flask, or Streamlit to visualize the results in real time.

IV. CONCLUSIONS

The system monitoring and security auditing scripts work together to provide a comprehensive solution for maintaining both the performance and security of a system. The system monitoring script helps users keep track of vital resources, such as CPU, memory, and storage, ensuring the system runs efficiently. By offering real-time data on how each of these resources is being used, it helps identify any performance issues early on, allowing users to take corrective action before they become critical.

At the same time, the security auditing script focuses on identifying vulnerabilities and misconfigurations within the system. It scans for outdated or vulnerable software packages, detects hardcoded secrets within code, and checks for inappropriate file permissions. This proactive approach to security helps prevent potential breaches and ensures sensitive information is protected. By assessing the severity of any identified issues, it enables users to prioritize remediation efforts based on the potential risks to the system.

Together, these scripts form a powerful combination for ensuring a system remains both efficient and secure. They can be seamlessly integrated into regular workflows, whether in a development environment or a production setup, and help administrators or developers maintain a healthy and protected system. Their automation makes them an invaluable asset in the ever-evolving landscape of system management.

REFERENCES

1. L. He, D. Li, and C. Shen, "A System Resource Monitoring Method Based on Data Mining in Cloud Computing," IEEE Access, vol. 6, pp. 2591–2601, 2018.
   doi: 10.1109/ACCESS.2017.2778047
2. M. Almorsy, J. Grundy, and A. S. Ibrahim, "Collaboration-Based Cloud Computing Security Management Framework," 2011 IEEE 4th International Conference on Cloud Computing, Washington, DC, USA, 2011, pp. 364–371.
   doi: 10.1109/CLOUD.2011.34
3. L. Chen and Y. Ye, "Automated Software Vulnerability Detection with Machine Learning," 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Paris, France, 2018, pp. 347–352.
   doi: 10.1109/WETICE.2018.00067
4. B. Saltaformaggio, R. Biedermann, X. Zhang, and D. Xu, "ACDC: An Automatic Cybersecurity Data Collection Framework for Android Devices," 2015 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Rio de Janeiro, Brazil, 2015, pp. 383–394.
   doi: 10.1109/DSN.2015.43
5. S. Roy, H. Shirazi, and K. Salah, "A Machine Learning Approach for Intrusion Detection on Cloud Virtual Machines," 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 2018, pp. 2905–2914.
   doi: 10.1109/BigData.2018.8621860
6. A. H. Lone and R. B. Mir, "A Comparative Study of Various Machine Learning Techniques for Detecting Malicious URLs," 2019 5th International Conference on Computing Communication and Automation (ICCCA), Greater Noida, India, 2019, pp. 1–6.
   doi: 10.1109/ICCCA47527.2019.8958656
7. Z. Xiao and Y. Xiao, "Security and Privacy in Cloud Computing," IEEE Communications Surveys & Tutorials, vol. 15, no. 2, pp. 843–859, Second Quarter 2013.
   10.1109/SURV.2012.060912.00182
8. C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A Survey of Intrusion Detection Techniques in Cloud," Journal of

Network and Computer Applications, vol. 36, no. 1, pp. 42–57, 2013.
doi: 10.1016/j.jnca.2012.05.003

9. A. H. Sung and S. Mukkamala, "Identifying Important Features for Intrusion Detection Using Support Vector Machines and Neural Networks," Proceedings of the 2003 Symposium on Applications and the Internet, Orlando, FL, USA, 2003, pp. 209–216.
doi: 10.1109/SAINT.2003.1183050
10. M. Rouse and D. Pham, "Security Auditing in Cloud-Based Infrastructures," 2020 IEEE International Conference on Cloud Engineering (IC2E), Sydney, Australia, 2020, pp. 71–79.
doi: 10.1109/IC2E48784.2020.00019