SECURE FILE SYSTEMS USING eBPF

(Final Report)

Team - 7

TA: Nishchal Mehra
Prof: D JanakiRam

Problem Statement

- The task involves building a custom Linux kernel to strengthen file system security for managing sensitive files, ensuring their data remains concealed from unauthorized users.

Objective

- This project aims to develop a secure Linux kernel module that enhances file system security for sensitive files by utilizing eBPF filters.
- This module aims to implement fine-grained access control, real-time monitoring, and filtering of file operations on sensitive data.
- Additionally, kernel-level encryption and decryption will be applied to ensure that sensitive files remain unreadable and unusable by unauthorized users.

Kernel Hooks (Kprobes)

- Kernel hooking is a mechanism to intercept and extend the behavior of kernel functions without modifying the kernel code itself.
- These hooks allow custom code to run in response to specific kernel events, such as file access, network activity, or system calls.
- Kprobes is a debugging and profiling mechanism in the Linux kernel that allows you to break into any kernel routine and collect diagnostic information dynamically. Using Kprobes, developers can trace and intercept functions and events in the kernel for monitoring and performance tuning.

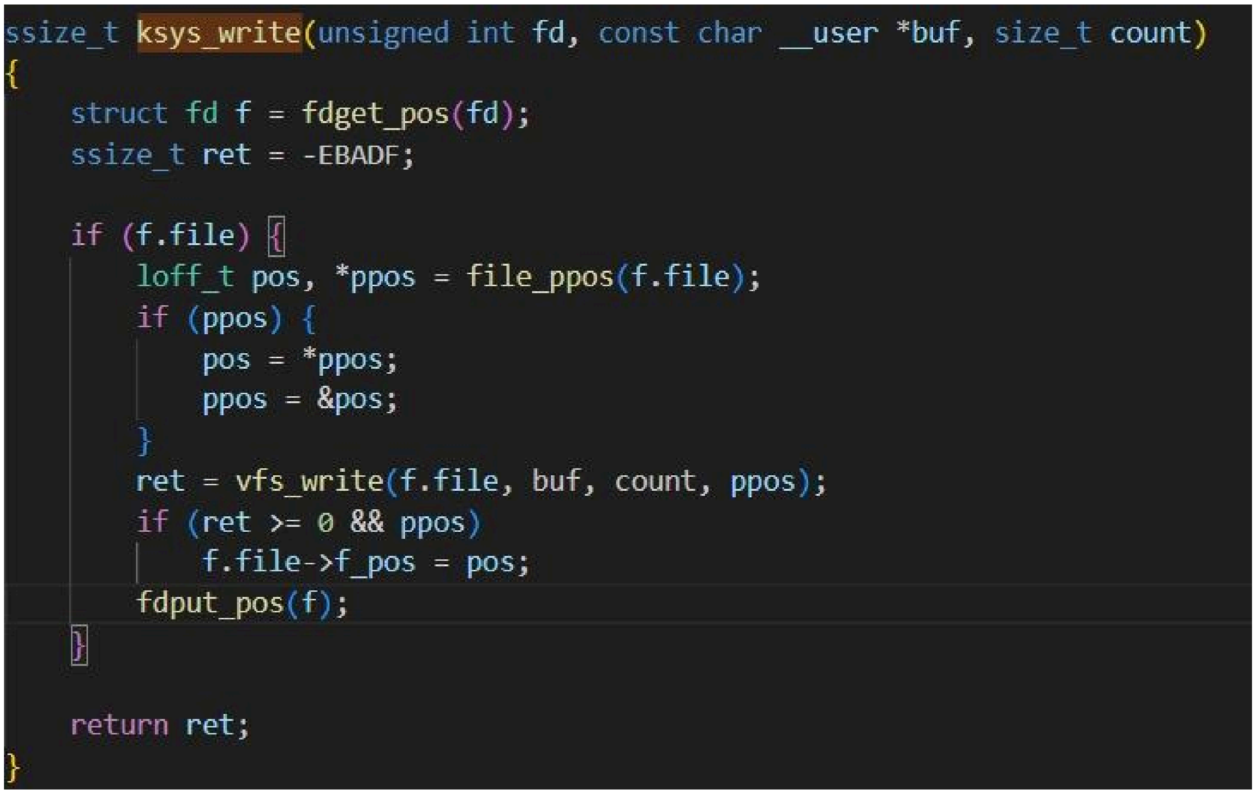
Attaching eBPF programs to Kprobes allows for:

- Enhanced Analysis: eBPF provides the ability to process and filter data on the fly, reducing the overhead on the kernel and improving performance.
- Non-intrusive Monitoring: Kprobes with eBPF do not require kernel modification, allowing it to collect kernel metrics and events without impacting stability.

Setting up eBPFs

- We want to write eBPF program that hook into the file system events where you want to enforce encryption/decryption. Typically, this would include:

On write operation: Apply encryption before the data is written to disk (at start)

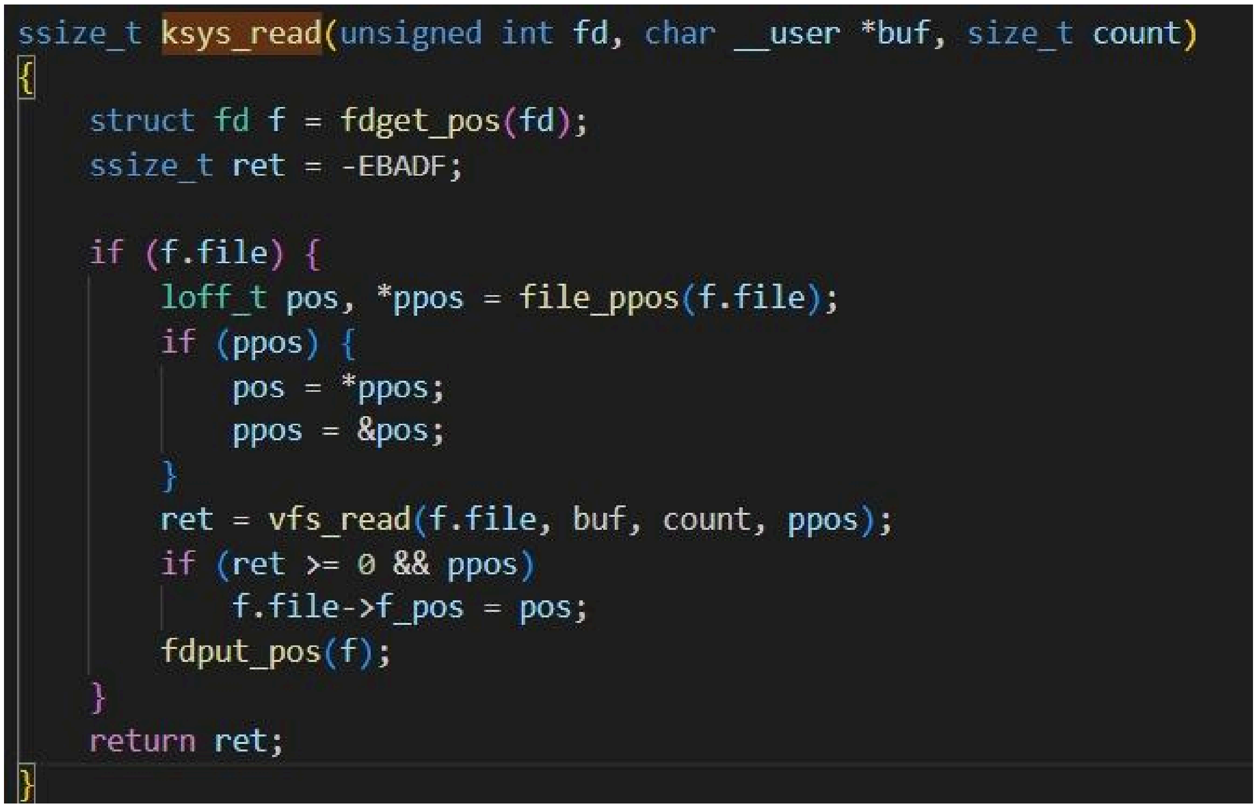


```
ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_write(f.file, buf, count, ppos);
        if (ret >= 0 && ppos)
            f.file->f_pos = pos;
        fdput_pos(f);
    }

    return ret;
}
```

- The eBPF program we implement will access the function arguments from the ksys_write.
- The eBPF program will have access to the buffer (which was actually storing the data that will be written into the file), and our encryption algorithm in the eBPF program will encrypt the data present in the buffer and rewrite the encrypted data into the buffer.
- The data in the modified buffer will now be written into the file.

On read operation: Apply decryption after the data is read from the file on disk

(after read call completed)



```
ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_read(f.file, buf, count, ppos);
        if (ret >= 0 && ppos)
            f.file->f_pos = pos;
        fdput_pos(f);
    }
    return ret;
}
```

- The eBPF program we implement will access the function arguments from the ksys__read.
- The eBPF program will have access to the buffer (which was actually storing the data which is present in our encrypted file), and our decryption key in eBPF program will decrypt the data present in the buffer and rewrite the decrypted data into the buffer.
- The data in the modified buffer will now be accessed while reading.

Crypto Algorithms for Encryption

1. AES (Advanced Encryption Standard)

- Type: Symmetric Key Encryption
- Description: AES is a widely used encryption standard that supports 128, 192, and 256-bit key sizes. It's efficient for encrypting large amounts of data and is considered secure for most applications.

2. RSA (Rivest–Shamir–Adleman) -- The Algo that we are going to implement in our eBPF

encryption program

- Type: Asymmetric Key Encryption
- Description: RSA is commonly used for secure data transmission. It relies on the difficulty of factoring large numbers and is often used in digital signatures and for encrypting small pieces of data like symmetric keys.

3. ECC (Elliptic Curve Cryptography)

- Type: Asymmetric Key Encryption
- Description: ECC is a public-key cryptography approach that offers high security with smaller key sizes, making it efficient for devices with limited processing power. It's widely used in mobile devices, IoT, and blockchain.

4. SHA-256 (Secure Hash Algorithm 256-bit)

- Type: Hash Function
- Description: SHA-256 is part of the SHA-2 family and is used for generating a fixed-length hash from any data input. It's widely used in blockchain for verifying data integrity.

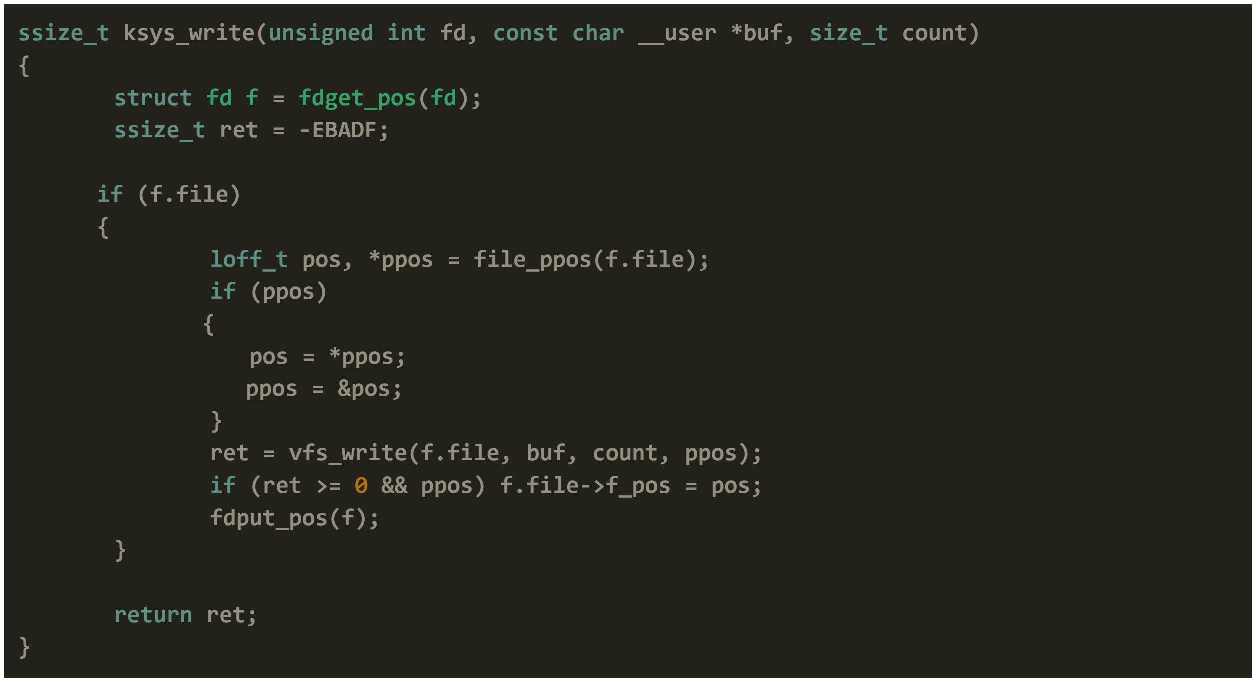
Our Idea for Implementation

➔ **Kernel Probing at Read system Call**

```
ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
{
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;
        if (f.file)
       {
                loff_t pos, *ppos = file_ppos(f.file);
                if (ppos)
                {
                   pos = *ppos;
                   ppos = &pos;
                }
                ret = vfs_read(f.file, buf, count, ppos);
                if (ret >= 0 && ppos) f.file->f_pos = pos;
                fdput_pos(f);
        }
        return ret;
}
```

- The eBPF program we implement will access the function arguments from the ksys__read.
- The eBPF program will have access to the buffer (which was actually storing the data present in our encrypted file), and our decryption key in the eBPF program will decrypt the data present in the buffer and rewrite the decrypted data into the buffer.
- The data in the modified buffer will now be accessed while reading.

➔ **Kernel Probing at Write system Call**



```
ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
       struct fd f = fdget_pos(fd);
       ssize_t ret = -EBADF;

      if (f.file)
      {
              loff_t pos, *ppos = file_ppos(f.file);
              if (ppos)
             {
                 pos = *ppos;
                 ppos = &pos;
              }
              ret = vfs_write(f.file, buf, count, ppos);
              if (ret >= 0 && ppos) f.file->f_pos = pos;
              fdput_pos(f);
       }

       return ret;
}
```

- The eBPF program we implement will access the function arguments from the ksys_write.
- The eBPF program will have access to the buffer (which was storing the data that is going to be written into the file), and our encryption algorithm in the eBPF program will encrypt the data present in the buffer and rewrite the encrypted data into the buffer.
- The data in the modified buffer will now be written into the file.
-

User Level Implementation

Requirements:

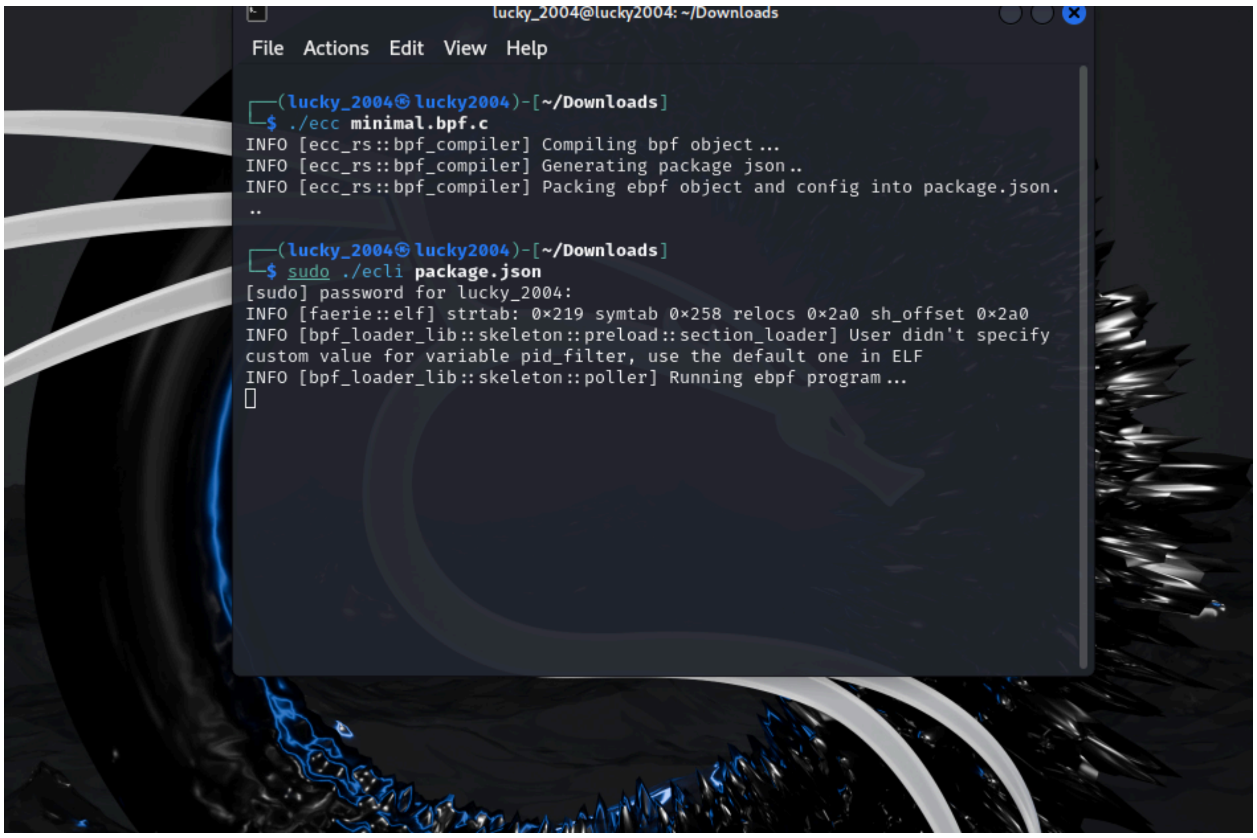
- Eunomia bpf developer tools (contains "ecli" and "ecc" executables to load eBPF bytecode into the kernel)

- ecli tool is required for running eBPF programs.

Minimal.bpf.c (source code)

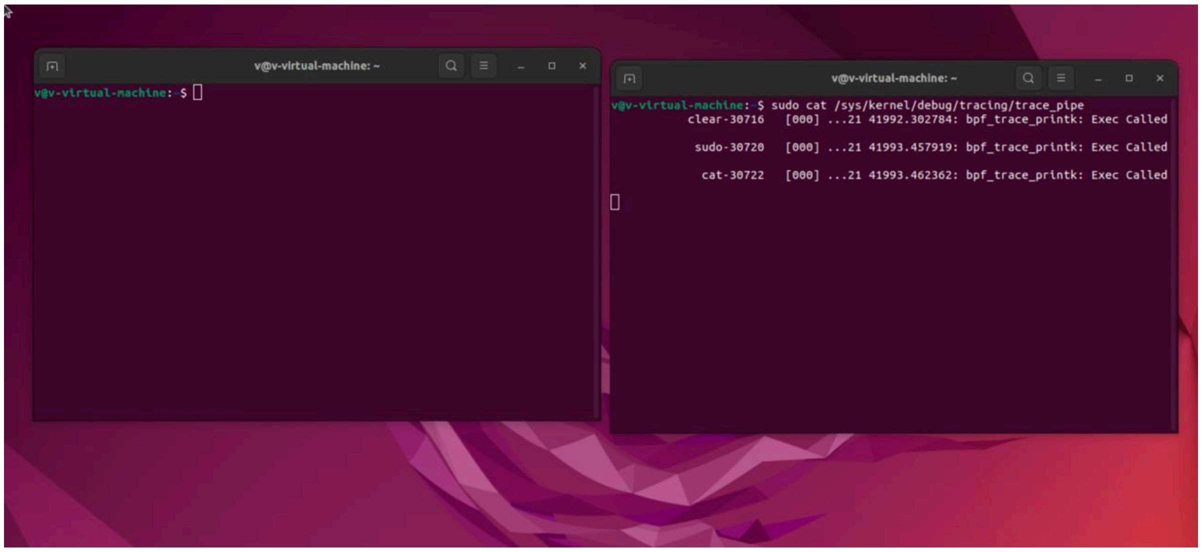
```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

SEC("tp/syscalls/sys_enter_execve")
int handle_execve(void *ctx)
{
    bpf_printk("Exec Called\n");
    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

- ./ecc minimal.bpf.c will compile the eBPF source code and generate the bytecode (object file) and package.json file.
- sudo ./ecli package.json command will load the bytecode into the kernel.


lucky_2004@lucky2004: ~/Downloads
File Actions Edit View Help

```
┌──(lucky_2004㉿lucky2004)-[~/Downloads]
└─$ ./ecc minimal.bpf.c
INFO [ecc_rs::bpf_compiler] Compiling bpf object...
INFO [ecc_rs::bpf_compiler] Generating package json..
INFO [ecc_rs::bpf_compiler] Packing ebpf object and config into package.json.
..

┌──(lucky_2004㉿lucky2004)-[~/Downloads]
└─$ sudo ./ecli package.json
[sudo] password for lucky_2004:
INFO [faerie::elf] strtab: 0x219 symtab 0x258 relocs 0x2a0 sh_offset 0x2a0
INFO [bpf_loader_lib::skeleton::preload::section_loader] User didn't specify
custom value for variable pid_filter, use the default one in ELF
INFO [bpf_loader_lib::skeleton::poller] Running ebpf program...
▯
```

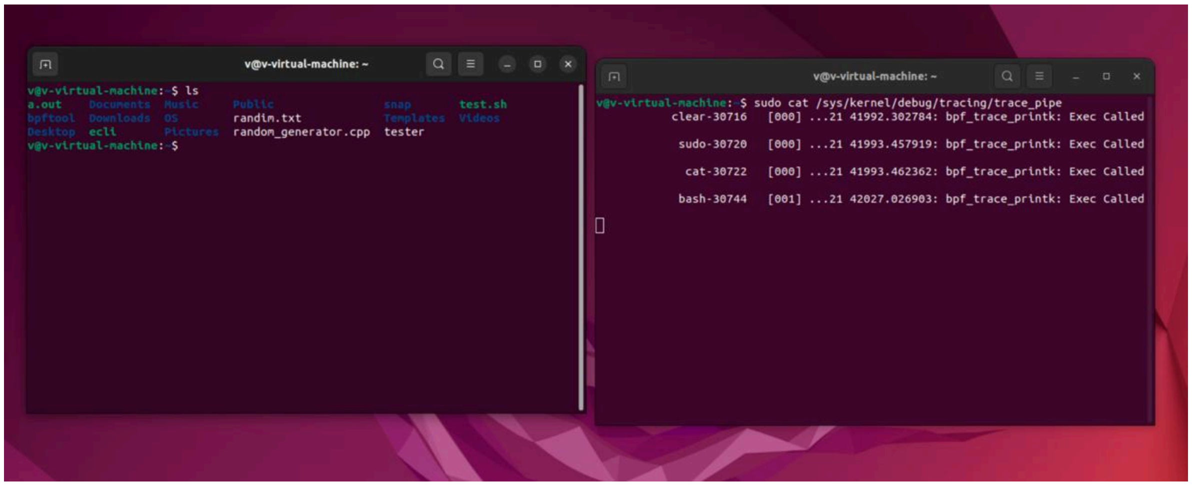
- After loading the bytecode into the kernel, we open the kernel and call the command prom which is shown in the terminal (right side) picture below.
- We can see the print statement we used "Exec Called" in our eBPF program to get printed the terminal screen below (it gets printed whenever execv sys called gets called during the kernel routine)



```
v@v-virtual-machine:~$
```

```
v@v-virtual-machine:~$ sudo cat /sys/kernel/debug/tracing/trace_pipe
           clear-30716   [000] ...21 41992.302784: bpf_trace_printk: Exec Called

            sudo-30720   [000] ...21 41993.457919: bpf_trace_printk: Exec Called

             cat-30722   [000] ...21 41993.462362: bpf_trace_printk: Exec Called
```

- We now called **ls command** in the left terminal and observed the following



```
v@v-virtual-machine:~$ ls
a.out    Documents  Music     Public                snap       test.sh
bpftool  Downloads  OS        randim.txt            Templates  Videos
Desktop  ecli       Pictures  random_generator.cpp  tester
v@v-virtual-machine:~$
```

```
v@v-virtual-machine:~$ sudo cat /sys/kernel/debug/tracing/trace_pipe
           clear-30716   [000] ...21 41992.302784: bpf_trace_printk: Exec Called

            sudo-30720   [000] ...21 41993.457919: bpf_trace_printk: Exec Called

             cat-30722   [000] ...21 41993.462362: bpf_trace_printk: Exec Called

            bash-30744   [001] ...21 42027.026903: bpf_trace_printk: Exec Called
```

- Once again the "Exec Called" print statement was printed on the right terminal. (As execv will be called internally during the kernel routine of **ls command**).

Kernel Level Implementation(Hooking up)

The Python code which we run on the system to hook up the eBPF program in **bpf__text** section

```
from bcc import BPF
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import os
import mmap

# eBPF program
bpf_text = """
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/dcache.h>

struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
} events SEC(".maps");

struct event {
    u32 pid;
    char filename[256];
    u32 syscall; // 1 for write, 2 for read
};

SEC("tracepoint/syscalls/sys_enter_write")
int trace_write(struct trace_event_raw_sys_enter *ctx) {
    struct event evt = {};
    u32 pid = bpf_get_current_pid_tgid() >> 32;

    // Get file descriptor and filename
    int fd = ctx->args[0];
    struct file *file = fd_file(fd);
    if (!file)
        return 0;

    struct dentry *dentry = file->f_path.dentry;
    bpf_probe_read_kernel_str(evt.filename, sizeof(evt.filename), dentry->d_name.name);

    evt.pid = pid;
    evt.syscall = 1; // Write syscall
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &evt, sizeof(evt));

    return 0;
}

SEC("tracepoint/syscalls/sys_enter_read")
```

```
int trace_read(struct trace_event_raw_sys_enter *ctx) {
    struct event evt = {};
    u32 pid = bpf_get_current_pid_tgid() >> 32;

    // Get file descriptor and filename
    int fd = ctx->args[0];
    struct file *file = fd_file(fd);
    if (!file)
        return 0;

    struct dentry *dentry = file->f_path.dentry;
    bpf_probe_read_kernel_str(evt.filename, sizeof(evt.filename), dentry->d_name.name);

    evt.pid = pid;
    evt.syscall = 2; // Read syscall
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &evt, sizeof(evt));

    return 0;
}

char LICENSE[] SEC("license") = "GPL";
"""

# Load eBPF program
bpf = BPF(text=bpf_text)

# Attach to write and read syscalls
bpf.attach_tracepoint("syscalls:sys_enter_write", "trace_write")
bpf.attach_tracepoint("syscalls:sys_enter_read", "trace_read")

# Generate RSA keys
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()

# Symmetric key for AES encryption (for file content)
symmetric_key = os.urandom(32)

# Encrypt data using RSA
def rsa_encrypt(data):
    return public_key.encrypt(
        data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

# Decrypt data using RSA
def rsa_decrypt(data):
    return private_key.decrypt(
        data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

# Encrypt file content using AES
def aes_encrypt(data):
    iv = os.urandom(16)  # Initialization vector
    cipher = Cipher(algorithms.AES(symmetric_key), modes.CFB(iv))
    encryptor = cipher.encryptor()
    return iv + encryptor.update(data) + encryptor.finalize()

# Decrypt file content using AES
def aes_decrypt(data):
    iv = data[:16]
    encrypted_content = data[16:]
    cipher = Cipher(algorithms.AES(symmetric_key), modes.CFB(iv))
    decryptor = cipher.decryptor()
    return decryptor.update(encrypted_content) + decryptor.finalize()

# Handle events from the perf buffer
def handle_event(cpu, data, size):
    event = bpf["events"].event(data)
    print(f"Syscall detected! PID={event.pid}, File={event.filename}, Syscall={'write' if
event.syscall == 1 else 'read'}")

    filepath = event.filename
    if event.syscall == 1:  # Write syscall
        if os.path.exists(filepath):
            with open(filepath, 'rb') as f:
                content = f.read()
            encrypted_content = aes_encrypt(content)
            with open(filepath, 'wb') as f:
                f.write(encrypted_content)
            print(f"Encrypted content written back to {filepath}")
    elif event.syscall == 2:  # Read syscall
        if os.path.exists(filepath):
            with open(filepath, 'rb') as f:
                encrypted_content = f.read()
            decrypted_content = aes_decrypt(encrypted_content)
            with open(filepath, 'wb') as f:
                f.write(decrypted_content)
            print(f"Decrypted content written back to {filepath}")

# Open perf buffer
bpf["events"].open_perf_buffer(handle_event)

print("Monitoring read/write syscalls. Press Ctrl+C to exit...")
while True:
    try:
        bpf.perf_buffer_poll()
    except KeyboardInterrupt:
        print("Exiting...")
        break
```

Overview of the Code

eBPF Hooking Mechanism

eBPF provides a powerful framework for dynamically tracing events within the Linux kernel. In this case, we hook into two tracepoints:

- `sys_enter_write`: Captures system calls that write data to files.
- `sys_enter_read`: Captures system calls that read data from files.

Hooking Details:

1. **Tracepoints**:
   - Tracepoints are predefined instrumentation points in the Linux kernel that expose events such as system calls. The program attaches to the `syscalls:sys_enter_write` and `syscalls:sys_enter_read` tracepoints.
2. **Instrumentation**:
   - The eBPF program intercepts arguments to the respective syscalls:
     - For `write` (`sys_enter_write`), it retrieves the file descriptor and resolves it to the file path.
     - For `read` (`sys_enter_read`), it performs a similar operation.
   - The file paths and process IDs (PIDs) are captured and stored in the eBPF event structure.
3. **Kernel Space to User Space**:
   - The `bpf_perf_event_output()` function sends these event structures to user space using a perf buffer.

Handling Events in User Space

The Python program uses the BCC library to manage and process the events received from the eBPF perf buffer.

Event Workflow:

1. **Syscall Detection**:
   - Events in the perf buffer are consumed using `bpf.perf_buffer_poll()` and passed to the `handle_event` function.
2. **File Operations**:
   - The `handle_event` function extracts syscall details (PID, filename, and syscall type). Based on the syscall type:
     - **Write Syscall**:
       - The program reads the file's content.

        - The content is encrypted using AES (a symmetric encryption algorithm).
        - The encrypted content is written back to the file, ensuring the data on disk is protected.
    - **Read Syscall**:
        - The program reads the encrypted content from the file.
        - It decrypts the content and writes the plaintext back to the file for application use.

Encryption and Decryption

The program integrates cryptographic operations to manage file data security:

1. **Encryption (Write Syscall)**:
    - AES is used for symmetric encryption.
    - A randomly generated initialization vector (IV) ensures data uniqueness.
    - The content is encrypted and stored in the format: `IV + Encrypted Data`.
2. **Decryption (Read Syscall)**:
    - The IV is extracted from the file.
    - The remaining content is decrypted using the symmetric key and the extracted IV.
3. **RSA Keys**:
    - RSA is used for asymmetric encryption of the symmetric AES key if needed (e.g., for sharing or secure storage).

Encryption using RSA and SHA with OAEP in detail

In modern cryptography, **RSA encryption** and **OAEP (Optimal Asymmetric Encryption Padding)** are widely used together to provide secure encryption and decryption of data. Below, we will break down the components of encryption using RSA, SHA (Secure Hash Algorithm), and OAEP in simple terms, while addressing how each element contributes to the overall process.

1. RSA Encryption Overview

RSA is an asymmetric encryption algorithm, which means it uses two separate keys:

- **Public key** (used for encryption)

- **Private key** (used for decryption)

The RSA algorithm works by using large prime numbers to generate two keys:

- The **public key** consists of an exponent (usually 65537) and a modulus n derived from the product of two large prime numbers.
- The **private key** contains the modulus n and a private exponent, which is mathematically related to the public exponent.

Key Steps in RSA Encryption:

1. **Encryption:** Data is encrypted using the **public key**. The data is converted into a number and raised to the power of the public exponent, then modulo n is applied.
2. **Decryption:** Data is decrypted using the **private key**. The encrypted data is raised to the power of the private exponent, modulo n.

RSA encryption typically operates on blocks of data, meaning the data is split into chunks that can be individually encrypted.

2. SHA (Secure Hash Algorithm)

SHA (Secure Hash Algorithm) refers to a family of cryptographic hash functions, where **SHA-256** is one of the most widely used. SHA-256 produces a fixed-length (256-bit) output called a **hash**, which is unique to the input data. Even a small change in the input will produce a completely different hash value.

SHA-256 is employed in RSA encryption for **padding** purposes in a scheme called **OAEP** (Optimal Asymmetric Encryption Padding). This enhances security by preventing certain attacks on the encryption scheme.

3. OAEP (Optimal Asymmetric Encryption Padding)

OAEP is a padding scheme used in RSA encryption to securely encrypt data. Padding ensures that the input to RSA encryption is of the correct length, but more importantly, it provides additional security by incorporating randomization.

In OAEP, the data is padded with extra bits before encryption to ensure it is the correct length for RSA. The main components of OAEP involve:

- **Hashing**: SHA-256 is used to generate a cryptographic hash of the input data.
- **Randomization**: OAEP incorporates randomization to ensure that encrypting the same data multiple times results in different ciphertexts, thus providing semantic security.

OAEP consists of two main functions:

1. **Padding Phase**: The plaintext data is padded with a combination of a **random seed** and **hash values**.

2. **Encoding Phase**: The padded data is then encoded with RSA.

The padding process prevents the RSA encryption algorithm from revealing any informa about the plaintext (such as length or structure), which would be a vulnerability in standard RSA.

Detailed Steps in OAEP:

1. **Message Preprocessing**: The message is divided into two parts:
    - **M**: The original message to be encrypted.
    - **Seed**: A random value used for encoding the message.
2. **Hashing**:
    - A **hash function (SHA-256)** is applied to the message and seed to generate new values.
3. **XOR Operations**:
    - The padded message is XORed with a masked version of the seed.
    - This ensures that even if the same message is encrypted multiple times, the ciphertext will differ due to the random seed, thus preventing attacks based on ciphertext patterns.
4. **Final Message Construction**: After XOR operations, the padded message is ready for RSA encryption.

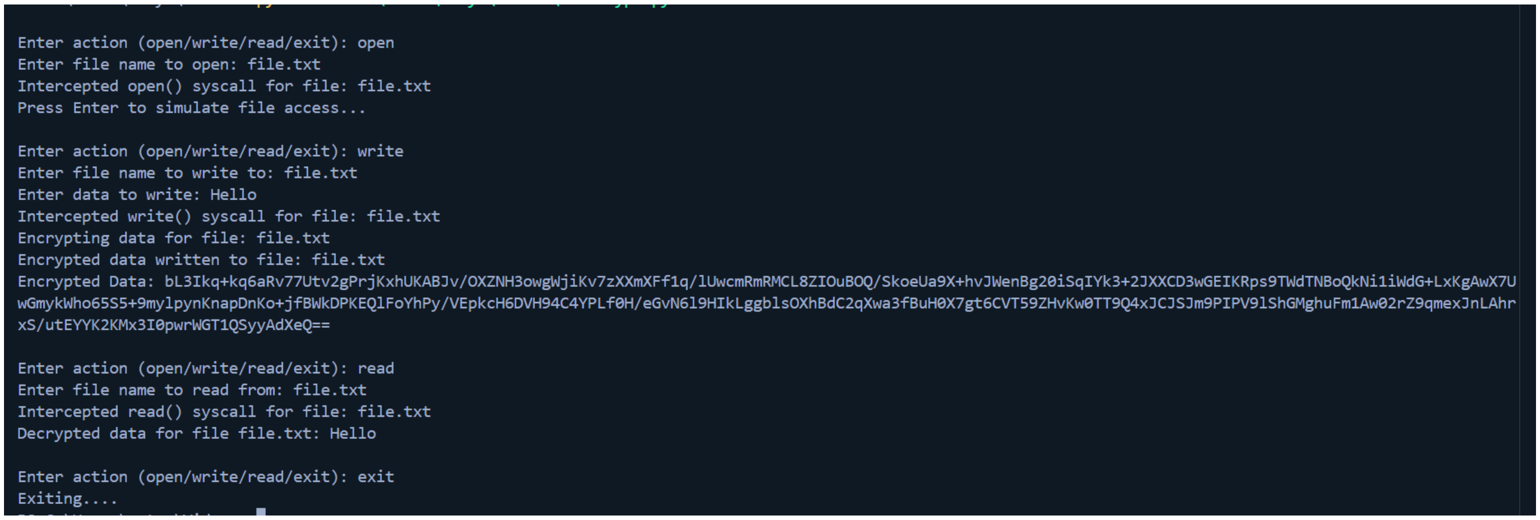
4. Use of RSA and OAEP in the Provided Code

In the code, the following happens:

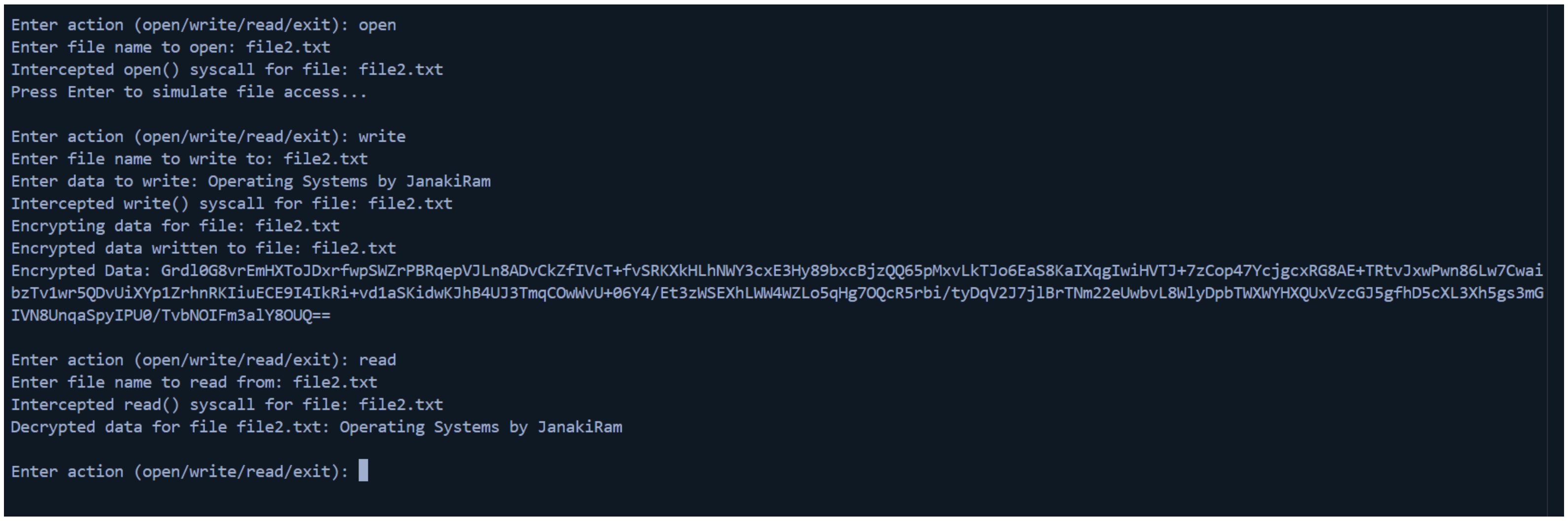
1. **Key Generation**:
    - A private key is generated using rsa.generate_private_key with a key size of 2048 bits and an exponent of 65537. The public key is then derived from the private key.
2. **RSA Encryption**:
    - **rsa_encrypt(data)**: This function encrypts the data using the RSA public key and OAEP padding. The data is encrypted with the public key using the OAEP scheme, which involves the following:
        - The data is first hashed using SHA-256.
        - The resulting hash is combined with a random seed and XORed with the message before being encrypted.
3. **RSA Decryption**:
    - **rsa_decrypt(data)**: This function decrypts the encrypted data using the RSA private key and OAEP padding. The decryption process is similar to encryption, but it uses the private key to recover the padded message, which is then unpadded.
4. **AES Encryption/Decryption**:

- The code also uses **AES** (Advanced Encryption Standard) for file content encryption, but this operates independently of RSA. AES encryption uses a symmetric key, while RSA operates with asymmetric keys.

Glimpses of the working mechanism



```
Enter action (open/write/read/exit): open
Enter file name to open: file.txt
Intercepted open() syscall for file: file.txt
Press Enter to simulate file access...

Enter action (open/write/read/exit): write
Enter file name to write to: file.txt
Enter data to write: Hello
Intercepted write() syscall for file: file.txt
Encrypting data for file: file.txt
Encrypted data written to file: file.txt
Encrypted Data: bL3Ikq+kq6aRv77Utv2gPrjKxhUKABJv/OXZNH3owgWjiKv7zXXmXFf1q/lUwcmRmRMCL8ZIOuBOQ/SkoeUa9X+hvJWenBg20iSqIYk3+2JXXCD3wGEIKRps9TWdTNBoQkNi1iWdG+LxKgAwX7U
wGmykWho65S5+9mylpynKnapDnKo+jfBWkDPKEQlFoYhPy/VEpkcH6DVH94C4YPLf0H/eGvN6l9HIkLggblsOXhBdC2qXwa3fBuH0X7gt6CVT59ZHvKw0TT9Q4xJCJSJm9PIPV9lShGMghuFm1Aw02rZ9qmexJnLAhr
xS/utEYYK2KMx3I0pwrWGT1QSyyAdXeQ==

Enter action (open/write/read/exit): read
Enter file name to read from: file.txt
Intercepted read() syscall for file: file.txt
Decrypted data for file file.txt: Hello

Enter action (open/write/read/exit): exit
Exiting....
```



```
Enter action (open/write/read/exit): open
Enter file name to open: file2.txt
Intercepted open() syscall for file: file2.txt
Press Enter to simulate file access...

Enter action (open/write/read/exit): write
Enter file name to write to: file2.txt
Enter data to write: Operating Systems by JanakiRam
Intercepted write() syscall for file: file2.txt
Encrypting data for file: file2.txt
Encrypted data written to file: file2.txt
Encrypted Data: Grdl0G8vrEmHXToJDxrfwpSWZrPBRqepVJLn8ADvCkZfIVcT+fvSRKXkHLhNWY3cxE3Hy89bxcBjzQQ65pMxvLkTJo6EaS8KaIXqgIwiHVTJ+7zCop47YcjgcxRG8AE+TRtvJxwPwn86Lw7Cwai
bzTv1wr5QDvUiXYp1ZrhnRKIiuECE9I4IkRi+vd1aSKidwKJhB4UJ3TmqCOwWvU+06Y4/Et3zWSEXhLWW4WZLo5qHg7OQcR5rbi/tyDqV2J7jlBrTNm22eUwbvL8WlyDpbTWXWYHXQUxVzcGJ5gfhD5cXL3Xh5gs3mG
IVN8UnqaSpyIPU0/TvbNOIFm3alY8OUQ==

Enter action (open/write/read/exit): read
Enter file name to read from: file2.txt
Intercepted read() syscall for file: file2.txt
Decrypted data for file file2.txt: Operating Systems by JanakiRam

Enter action (open/write/read/exit): 
```

- Open sys call takes file name as input and stores the name of file in a map, which stores the file names for all the files whose content has to be encrypted.

- The map stores the names of all files which have to be encrypted. The map gets updated with the file name as soon as the open system call is called.

- Write sys call take file name and ask for the contents of the file, once we provide t content, it will be encrypted using our encryption mechanism and the encrypted will be written back into the disk (I have made to print it to the screen for explanation purposes above...)

- Read sys call take file name and read the encrypted content in the file and decrypt the data in it by decryption mechanism and stores in the local buffer (I have made to print it to the screen for explanation purposes above...)

Our Team

Understanding virtual file system and system calls in Linux and eBPF kernel Hooking

Ch. Srikar - CS22B070

K A Vignesh - CS22B076

G Vignesh - CS22B025

K Vibhav - CS22B028

K Gireesh - CS22B031

Implementing a crypto algorithm

Malireddi Sri Sai Shanmukh raj - CS22B029

K Prem Sagar - CS22B039

Testing and Debugging

Navaneeth Paturi - CS22B009

M L Sri Charan - CS22B080

Updating weekly progress in PPTs and Documentation

Adarsh - CS22B024

B Abishek Praizy - CS22B068

G Prasanna Kumar - CS14B008