



**CS5231**

**System Auditing Analysis Project**

**Team 28**

<b>Name</b>	<b>Matric No.</b>
Haziq Hakim Bin Abdul Rahman	A0216481H
Ng Jong Ray, Edward	A0216695U

# Abstract

System security is paramount for safeguarding information, resources and thwarting unauthorised access. Traditional system auditing, while valuable, faces challenges such as the manual configuration of rules and the complexity of log interpretation. Our project aims to leverage extended Berkeley Packet Filter to create a user-centric logging system. Our goal is to first designate a special folder for sensitive resources and then enable enhanced security logging to detect and identify programs accessing this special folder. Our methodology involves using BCC (BPF Compiler Collection) to create our own kernel tracing program, utilising the `openat` syscall to identify tasks accessing files. Our findings showcase the program's effectiveness in detecting both absolute and relative pathname access methods. We aim to contribute to a “user first” logging system, allowing users to easily understand logs and promptly respond to these potential threats.

# Introduction

System security is an essential part in ensuring that a system is using and accessing its resources as intended by the user of the system. This can all be compromised through malicious threats by those with unauthorised access to the system's resources, and in order to protect our system, we have multiple ways we can detect different forms of unauthorised access such as the use of system auditing.

System auditing is the collection of data about the use of its resources by the system. This can be in the form of data logs where actions and their respective user are logged onto a file for the purpose of monitoring who is using the resources and what they are doing with it. These logs are then stored in a file and directory that are both only accessible by an authorised user who can read the logs.

System auditing helps in protecting a system as all actions done in a system are recorded down in the logs for an authorised user to see. This way they can perform multiple tasks regarding security of a system which includes actions such as monitoring security related events happening on the system, detecting unauthorised access, discovering attempts to obtain sensitive data and many more. The user reading the audit logs will be able to do the correct task in response to any problem they may encounter from looking at the logs.

## **Problem Statement**

As stated above, system auditing provides relevant information of a machine which is beneficial for doing checks on whether a system is secure. However, there are some issues that can make such checks for a system's security to be difficult. One such issue is that logging the relevant information is dependent on the level of checking done by the user. For example, when using Auditd, we have to set our own rules on what to watch or what syscalls we want to see being logged. Even then, the logged information can be difficult to parse for a user who is not skilled in using Auditd. Another problem is that malicious threats are not easily detectable in audit logs as the logs tend to only record down low-level security actions and that the threats are rarely every one small action but a collection of many different actions. Hence, simply reading audit logs can make quickly identifying possible issues difficult at its current state.

## **Objective**

Given the challenges addressed previously in relation to the utilisation of Auditd, our objective aims to address these challenges by creating an eBPF (extended Berkeley Packet Filters) program to create logs with a user-centric design philosophy. For this project, we envision a designated special folder which the user wants to place their important files in. Our goal is to use an eBPF program which can help the user easily detect and identify which program has accessed files in this special folder and ascertain its operational origin. In this scenario, we designate the “/home/student/secret” directory as the special folder.

# Methodology

BCC (BPF Compiler Collection) serves as a toolkit to facilitate the creation of efficient kernel tracing programs and comes with other useful tools that can analyse the system. BCC makes integrating the use of eBPF (extended Berkeley Packet Filters) programs simple by providing a python interface to help compile valid BPF programs (iovisor, n.d.). This allows the developer to focus on writing the BPF program without worrying about the intricacies during the compilation phase (iovisor, n.d.). The streamlined compilation process prompted us to try developing our custom kernel tracing program on BCC instead of other tools such as libbpf-tools.

## Setting up the BCC file

The first phase when creating this BCC program entails identifying a system call (syscall) suitable for our purpose of tracing file access. For our BCC program `secret_snoop.py`, we decided to leverage the `openat` syscall to trace all open calls called by programs running in the system. It is noteworthy that the conventional `open` syscall was not able to trace open calls as accurately. This discrepancy is most likely attributed to `openat` usage of file descriptors `dirfd` to be able to better handle reading relative pathnames instead of the `open` syscall using the current working directory only to handle relative pathnames (Kerrisk, 2023).

The subsequent decision pertains to deciding the type of probe to attach our syscall to. Given that the primary objective of our BCC program is to trace `openat` syscalls, using `kprobe` (dynamic tracing of a kernel function call) is sufficient as opposed to using `kfunc` or `lsm_probe` (Keniston et al., n.d.). This will be discussed in further detail in the [Conclusions and Recommendations](#).

To attach our BPF function to the `openat` syscall, we can utilise BCC's python interface and call the `BPF.attach_kprobe` method. A critical consideration is that our BPF function name must have the `syscall__` prefix. This special prefix creates a `kprobe` for any syscall name provided, otherwise the `kprobe` will not recognise our custom BPF function and will not call it whenever an `openat` syscall is detected. The final function name attached to the `kprobe` is

called `syscall__openat_entry`. Moreover, this convention allows us to put in the arguments of `openat` syscall as function arguments of our custom BCC tracing program to utilise (iovisor, n.d.).

The BPF code is written as a python string and is attached via calling the BPF constructor in the python code. This can be seen in figure 1 below.

```
b = BPF(text=bpf_text)
b.attach_kprobe(event=b.get_syscall_fnname("openat"), fn_name="syscall__openat_entry")
```

Figure 1: Calling the python interface to attach the kprobe

## Writing the BPF Code

The main C function in the BPF code is `syscall__openat_entry`. The purpose of this function is to after detecting an `openat` syscall, our BCC program will extract the current running task using the BPF function `bpf_get_current_task()`. This `task_struct` is crucial in extracting vital information such as the current working directory as well as the task name.

To find the task name, we call `bpf_get_current_comm()` to save the current task name as a string. To find the current directory, we access the `dentry` struct through `task->fs->pwd.dentry`. The name of the current directory can be found by accessing `working_dentry->d_name.name`. Moreover, the `dentry` struct of the parent directory can be accessed through `working_dentry->d_parent`.

Subsequently, a function `check_path` was created to analyse the current task and filename with the objective of identifying whether the accessed file resides within the “secret” directory. This is done by examining both the current directory and the `filename` variable. The `filename` variable can be obtained from the function argument of the `openat` syscall which captures the program opening any file. For example, running the shell command `cat secret/secret.txt` from the “student” directory results in the `filename` as

“secret/secret.txt” and the current directory name as “student”. Another example would be to run the shell command `cat ../secret/secret.txt` from the “student/test” directory will result in the filename as “../secret/secret.txt” and the current directory as “test”. This exemplifies the correlation between filename and the directory name. The path in the shell command, stored in `filename`, and the directory name must be considered when tracing how a program accesses the “secret” directory. `bpf_probe_read_kernel()` is invoked to save the name of the working directory as a string called `name`. Then an `if-else-if` chain is used to compare `name` against the different directories that a program is anticipated to reside in when accessing the “secret” folder.

In instances where the program is executed from the root directory, then it is necessary for the program to call the absolute path to access files in the “secret” directory. Hence, checking if `filename` contains the path “/home/student/secret/” is sufficient.

```
if (filename[0] == '/') { //file path is absolute
    //program directly opening with absolute path
    long match = MY_STRNCMP(filename, sizeof(full_path)-1, full_path);
    if (match == 0) {
        bpf_trace_printk("%s has accessed %s from %s directory\\n", comm, filename, name);
    }
}
```

Figure 2: Code if filename contains absolute path

In instances where the program is executed from the “home” directory, a check for `filename` if it contains the path “student/secret/” will occur. If this is true, then the program will check if the parent directory is the root directory.

```
} else if (MY_STRNCMP(name, sizeof("home"), "home") == 0) {
    long match = MY_STRNCMP(filename, sizeof(home_path)-1, home_path);
    if (match == 0) {
        char parent[10];
        bpf_probe_read_kernel(parent, sizeof(parent), working_dentry->d_parent->d_name.name);
        match = MY_STRNCMP(parent, 1, "/");
        if (match == 0) {
            bpf_trace_printk("%s has accessed %s from %s directory\\n", comm, filename, name);
        }
    }
}
```

Figure 3: Code if current directory is in “home”

In instances where the program is executed from the “student” directory, a check for `filename` if it contains the path “secret/” will occur. If this is true, then the program will check if the parent directory is the “home” directory. The root directory is being assumed.

```

} else if (MY_STRNCMP(name, sizeof("student"), "student") == 0) {
    long match = MY_STRNCMP(filename, sizeof(stu_path)-1, stu_path);
    if (match == 0) {
        char parent[10];
        bpf_probe_read_kernel(parent, sizeof(parent), working_dentry->d_parent->d_name.name);
        if (MY_STRNCMP(parent, sizeof("home"), "home") == 0) {
            bpf_trace_printk("%s has accessed %s from %s directory\\n", comm, filename, name);
        }
    }
}

```

Figure 4: Code if current directory is in “student”

In instances where the program is executed from the “secret” directory, there is no need to check for the path in `filename`. The program will check if the parent directory is the “student” directory and if the parent of the “student” directory is the “home” directory. The root directory is being assumed.

```

} else if (MY_STRNCMP(name, sizeof("secret"), "secret") == 0) {
    //current directory is in the 'secure' folder
    //hence the filename will only have the file being read
    char parent[10];
    bpf_probe_read_kernel(parent, sizeof(parent), working_dentry->d_parent->d_name.name);
    working_dentry = working_dentry->d_parent;
    if (MY_STRNCMP(parent, sizeof("student"), "student") == 0) {
        char parentparent[10];
        bpf_probe_read_kernel(parentparent, sizeof(parentparent), working_dentry->d_parent->d_name.name);
        if (MY_STRNCMP(parentparent, sizeof("home"), "home") == 0) {
            bpf_trace_printk("%s has accessed %s from %s directory\\n", comm, filename, name);
        }
    }
}

```

Figure 5: Code if current directory is in “secret”

In instances where the program is executed from a directory that is out of these directories, this can indicate that the program is accessing the “secret” directory using a relative path, otherwise the program has to use the absolute path. These paths could include something like the previous example `cat ../secret/secret.txt`. This means that `filename` should be used as a “source of truth” regarding the potential access to the “secret” directory. In this context, the check here would be to iterate through `filename` to find the string “secret”. Although this would result in false positives, these false positives can be detected by following the relative



pathname from the current directory. More analysis on the efficacy of the program will be discussed in the [Findings and Analysis section](#).

```
} else {  
    //a program is accessing a secret folder from a relative path  
    //to try and catch programs like cat ../../secret/secret.txt  
    for (int i = 0; i < sizeof(filename); i++) {  
        if (MY_STRNCMP(filename + i, sizeof(stu_path)-1, stu_path) == 0) {  
            bpf_trace_printk("%s has accessed %s from %s directory\\n", comm, filename, name);  
            break;  
        }  
    }  
}
```

Figure 6: Code if current directory is not in the expected directories

Finally, for log generation purposes, the output will be directed to the terminal. Calling `bpf_trace_printk()` allows us to print out to “/sys/kernel/debug/tracing/trace\_pipe”. Leveraging BCC’s python interface, we can call `printb()` to then print them out on the terminal. The output can be redirected into any log or text file for any further analysis. The output of the text includes the timestamp, current task, the current pid and the message derived from debug statements.

```
while 1:  
    try:  
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()  
    except ValueError:  
        continue  
    except KeyboardInterrupt:  
        exit()  
    printb(b"%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```

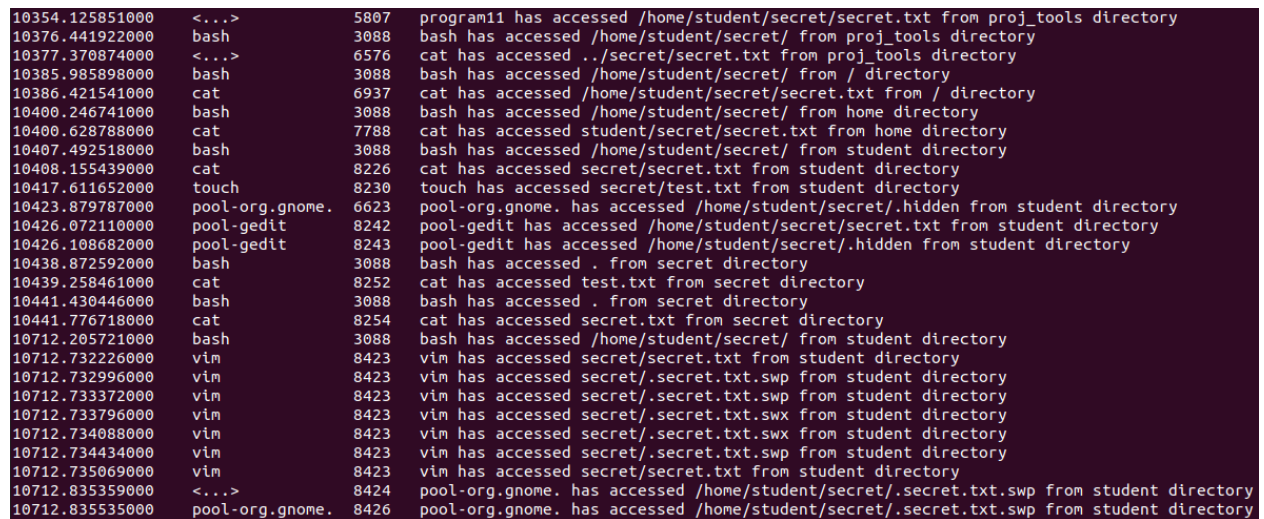
Figure 7: Python code to print debug statements to terminal

# Findings and Analysis

The steps to set up and use the BCC program can be found in [Annex A](#). For our test, we executed `sudo python3 secret_snoop.py > secret_logs.txt`. We then ran a series of diverse commands to test our BCC program across different directories and different methods of accessing the “secret” directory.

For the test, we first ran `./run-attack.sh`. Then we attempted to read the “secret.txt” file from various directories. This includes `root`, “/home”, “/home/student”, “/home/student/proj-tools” and “/home/student/secret”. Our testing employed various ways of reading “secret.txt”, using terminal commands such as `cat` and `touch` to read and create a new file in the “secret” directory. Moreover, we used both the terminal text editor `vim` and graphics user interface text editor `gedit`. The objective was to comprehensively evaluate the effectiveness and comprehensiveness of our BCC program.

The below figure is the result of calling `cat secret_logs.txt`. This can be called while `secret_snoop.py` is still running, capturing all accesses up to that point.



```
10354.125851000 <...> 5807 program11 has accessed /home/student/secret/secret.txt from proj_tools directory
10376.441922000 bash 3088 bash has accessed /home/student/secret/ from proj_tools directory
10377.370874000 <...> 6576 cat has accessed ../secret/secret.txt from proj_tools directory
10385.985898000 bash 3088 bash has accessed /home/student/secret/ from / directory
10386.421541000 cat 6937 cat has accessed /home/student/secret/secret.txt from / directory
10400.246741000 bash 3088 bash has accessed /home/student/secret/ from home directory
10400.628788000 cat 7788 cat has accessed student/secret/secret.txt from home directory
10407.492518000 bash 3088 bash has accessed /home/student/secret/ from student directory
10408.155439000 cat 8226 cat has accessed secret/secret.txt from student directory
10417.611652000 touch 8230 touch has accessed secret/test.txt from student directory
10423.879787000 pool-org.gnome. 6623 pool-org.gnome. has accessed /home/student/secret/.hidden from student directory
10426.072110000 pool-gedit 8242 pool-gedit has accessed /home/student/secret/secret.txt from student directory
10426.108682000 pool-gedit 8243 pool-gedit has accessed /home/student/secret/.hidden from student directory
10438.872592000 bash 3088 bash has accessed . from secret directory
10439.258461000 cat 8252 cat has accessed test.txt from secret directory
10441.430446000 bash 3088 bash has accessed . from secret directory
10441.776718000 cat 8254 cat has accessed secret.txt from secret directory
10712.205721000 bash 3088 bash has accessed /home/student/secret/ from student directory
10712.732226000 vim 8423 vim has accessed secret/secret.txt from student directory
10712.732996000 vim 8423 vim has accessed secret/.secret.txt.swp from student directory
10712.733372000 vim 8423 vim has accessed secret/.secret.txt.swp from student directory
10712.733796000 vim 8423 vim has accessed secret/.secret.txt.swx from student directory
10712.734088000 vim 8423 vim has accessed secret/.secret.txt.swx from student directory
10712.734434000 vim 8423 vim has accessed secret/.secret.txt.swp from student directory
10712.735069000 vim 8423 vim has accessed secret/secret.txt from student directory
10712.835359000 <...> 8424 pool-org.gnome. has accessed /home/student/secret/.secret.txt.swp from student directory
10712.835359000 pool-org.gnome. 8426 pool-org.gnome. has accessed /home/student/secret/.secret.txt.swp from student directory
```

Figure 8: Output of BCC program

As described in the [Methodology](#), the output produced by our BCC program will include information such as the timestamp, current task, its pid and the debug statement. In the debug statement, it reveals the name of the current program, `filename` providing insights of what the

path being called is and the current directory which the program resides in. This information is instrumental for a comprehensive understanding of the executing process, which will be described later in this section.

BCC program is able to detect that “program11” is the malicious program which has accessed “secret.txt” when executing “run-attack.sh”. The access method employed by “program11” to access “secret.txt” can be discerned by the absolute path. These logs allow us more in-depth analysis of how a malicious program is trying to access our sensitive files within the “secret” directory.

Subsequently, it is observed that `bash` has accessed the “secret” directory as well. This was a result of trying to use the terminal’s autocomplete feature by pressing “Tab” to autofill a command by going through the existing files in a certain directory. This observation complements the expected behaviour of detecting activity within the “secret” folder from different text editing softwares and file creations attempts. This exemplifies the robustness of our BCC program, showcasing its proficiency in detecting more subtle and indirect ways of accessing and reading the secret directory.

As described in our [Methodology](#), we created an `if-else-if` chain to monitor programs accessing the “secret” folder from different directories. Recognising the inherent limitations of a naive approach, which relies solely on checking the absolute path of the syscall, we acknowledge the potential for circumvention. This can be done particularly through attempts to access files using relative paths, notably through the utilisation of “../” in the path. In adherence to Kerckhoff Principle, an adversary possessing knowledge of our filesystem structure and BCC code could exploit this understanding and gain access to the “secret” directory through the previously discussed technique. This can be seen in the third line of Figure 8. We can see how the command `cat ../secret/secret.txt` was called from the “proj\_tools” directory. Therefore, the final `else` section is successful at monitoring these types of file accesses at the cost of creating more false alerts. An access now to any directory called secret would be flagged out and logged, even if the directory “secret” is not the same one as “/home/student/secret”.

However, the false alerts can be detected by using the relative pathname and the current directory. By combining both of these details from the message, we would be able to predict if the “secret” directory is being accessed, thus enabling accurate removal of false positives from the log.

## **Conclusions and Recommendations for Future Work**

From our findings and analysis, our BCC program is able to detect the simulated attack and log information pertaining to the malicious attack which exfiltrated data from the “secret.txt” file. When an authorised user checks the logs, they would be able to detect that there was a malicious attack on their system which is important for discovering potential vulnerabilities in their system.

A recommendation for future work would be to implement `lsm_probe` as a way to implement MAC security policies in BPF. This macro instruments an LSM hook as a BPF program and allows us to deny any security operation. This means that if during run time we detect a malicious program accessing the “secret” folder, we would be able to instantly deny its access by restricting its permissions. To make this process more transparent to the user, a Graphic User Interface could be created to ask the user if they want to allow or deny that operation. This could be similar to Windows implementation of allowing administrative access.

Another recommendation would be to create a network snoop BCC program to create user friendly network logs. This is because looking into the shell script of “run-attack.sh”, we can see that a `curl` command was run to download the programs from the attacker’s web server on localhost:8080. Having a network snoop BCC program could help to provide more information of where certain malicious programs come from.

## References

- iovisor. (n.d.). *bcc/docs/reference\_guide.md at master · iovisor/bcc*. GitHub. Retrieved December 6, 2023, from [https://github.com/iovisor/bcc/blob/master/docs/reference\\_guide.md#8-system-call-tracepoints](https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#8-system-call-tracepoints)
- iovisor. (n.d.). *BCC - Tools for BPF-based Linux analysis*. GitHub. Retrieved December 6, 2023, from <https://github.com/iovisor/bcc>
- iovisor. (n.d.). *tutorial\_bcc\_python\_developer*. GitHub. Retrieved December 6, 2023, from [https://github.com/iovisor/bcc/blob/master/docs/tutorial\\_bcc\\_python\\_developer.md](https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md)
- Keniston, J., Panchamukhi, P. S., & Hiramatsu, M. (n.d.). *Kernel Probes (Kprobes) — The Linux Kernel documentation*. The Linux Kernel documentation. Retrieved December 6, 2023, from <https://www.kernel.org/doc/html/latest/trace/kprobes.html>
- Kerrisk, M. (2023, 04 03). *open(2) - Linux manual page*. man7.org. Retrieved December 6, 2023, from <https://www.man7.org/linux/man-pages/man2/openat.2.html>

## Annex A (Using the BCC program)

1. On the terminal run

```
sudo apt-get install bpfcc-tools linux-headers-$(uname -r)
sudo apt-get install libbpf-dev
```

2. Run the BCC program and save the output to secret\_logs.txt

```
sudo python3 secret_snoop.py > secret_logs.txt
```

3. Run the simulated attack

```
./proj_tools/run_attack.sh
```

4. Check the logs

```
cat secret_logs.txt
```

5. Run other commands to read secret.txt file and check the logs