Automatic Policy Generation for BPFContain

Carleton University - COMP 4905 - Midterm Report

Jake Jazokas – 101083496

Supervised by: Anil Somayaji

March 4th, 2022

Abstract

The main objective of this honors project is to research and implement an application that will automatically generate security policies for the container security daemon BPFContain. The purpose of these security policies is to limit the behaviour and interactions that a process can make within a containerized system. The focus of these policies will be to provide protection against unusual program behaviour. eBPF will be used to give the application access to the calls being made in kernel space. This will allow us to view what applications are doing and will give our application the information it needs to automatically generate security policies. Once the security policies have been generated by the implementation, they will be sent to BPFContain, which will then execute and enforce them.

Contents

1 Background Information 3
1.1 The Linux Kernel and eBPF 3
1.2 Kernel Probes, User Probes and Tracepoints 3
1.3 BPFContain 3
2 Related Works 3
2.1 Audit2Allow 3
2.2 IPCDump 3
2.3 Dtrace 3
2.4 SystemTap 4
3 Attempts 4
3.1 Tracing 4
3.1.1 BCC 4
3.1.2 bpftrace 4
4 Current Progress 6
4.1 Tracing Program 6
4.2 Translation Program 7
4.3 Results 8
4.3.1 Example (Trimmed) Trace File 9
4.3.2 Example Generated Policy for Bash 10
5 Future Progress 11
5.1 Tracing Program 11
5.2 Translation Program 12
6 References 13

1 Background Information

1.1 The Linux Kernel and eBPF

The Extended Berkeley Packet Filter (*eBPF*) allows for the extension of kernel capabilities to privileged user space processes [1]. eBPF provides the ability to perform both kernel and user level tracing and offers the ability to trace event execution system wide with the use of different probes.

1.2 Kernel Probes, User Probes and Tracepoints

Kernel probes (*kprobes*) are used to trace any function calls within the kernel [2]. Whereas user probes (*uprobes*) allow for arbitrary function tracing in the user space [3]. Tracepoints can be used to hook specific functions. Each time the function specified in the tracepoint is executed, a function associated with the traced function is also executed [4].

1.3 BPFContain

BPFContain [5] is the base application that will be used to run the security policies generated by this project. BPFContain [5] reads and executes YAML formatted security policy files, that define the resources that a program is allowed or restricted from accessing.

2 Related Works

2.1 Audit2Allow

Audit2Allow is a Linux tool that generates SELinux policy based on rules, that are inferred from logs of denied operations for a given program [6].

2.2 IPCDump

IPCDump [7] is a Linux tool that can be used to trace inter process communication (*IPC*). It relies on BPF to probe specific areas of the kernel using kprobes and tracepoints [7]. To interact with BPF, IPCDump uses the go library gobpf [8]. After using this tool and going through the source code, I was able to determine which probes were used to trace specific events within the kernel. This provided me with insight into the probes needed to capture specific events.

2.3 Dtrace

Dtrace is an analysis tool that allows for kernel and user level tracing. Dtrace was the predecessor to eBPF [1] and bpftrace [9] and provided kernel level tracing before eBPF was implemented in the Linux kernel [10].

2.4 SystemTap

SystemTap [11] is like bpftrace in that it used kernel and user level probes to trace system events. However, SystemTap [11] instructions are compiled into C code that is loaded as a module in the Linux kernel. This adds additional security risks as it is not verified that the program can be trusted and will not crash the kernel. Whereas bpftrace [9] scripts are compiled into BPF code, which ensures that the program is safe to run in the kernel [12].

3 Attempts

3.1 Tracing

3.1.1 BCC

My first attempt at tracing the system was using the BCC framework [13]. This framework provides a python front end that interacts with BPF and provides the ability to do kernel level tracing. To use BCC [13], I had to build it from source, as the version available on the ubuntu package manager is outdated. After dealing with numerous issues from dependencies when building BCC [13], I decided to switch to using another tool, bpftrace [9], which could be easily installed via. The package manager DNF [14] on Fedora v35.

3.1.2 bpftrace

On my second attempt, I used bpftrace to implement the tracing program [9]. bpftrace provides a high-level language that can be used to interact with the BPF system [9]. Although a current version of bpftrace can be easily installed on Fedora, the kernel must be built with specific flags that allow access to BPF, kprobes, uprobes, and tracing capabilities [15].

Using bpftrace [9], I first attempted to trace filesystem operations as I assumed this could easily be done by using kprobes. These kprobes would get data from read and write operations within the virtual filesystem (*VFS*). These probes are activated any time a program attempts to read or write bytes from a given file. The function definitions for these two operations, found in the Linux header file `fs.h` are as follows:

```
extern ssize_t vfs_read(struct file *, char __user *, size_t, loff_t *);
extern ssize_t vfs_write(struct file *, const char __user *, size_t, loff_t *);
```

Knowing the input parameters to each of these functions allowed me to recognize that the first argument, the file pointer struct, could be used to get information on the file that is being accessed. We can extract the name of the file by traversing the file pointer. First, we get the `path` struct `f_path` contained within the file pointer. This `path` struct then gives us access to the `dentry` pointer, which is a specific part of the overall path to the given file. Finally, we can access the `qstr` struct `d_name` to get the name of this specific path component. Which in this case is the name of the file that is passed to either `vfs_read()` or `vfs_write()`.

```
$fileName = str(((struct file *)arg0)->f_path.dentry->d_name.name);
```

The next challenge was to extract the full path of the given file. This is done similarly to how the file name is resolved, except in this case we want to iterate through the `dentry` pointer `d_parent`, until we hit the root directory. Each of these `d_parent` objects contain the name of the parent directory for a given `dentry`. Since unbounded loops are not allowed to be run in the kernel, we assume a max depth of 10 folders. This limit has not been reached in testing thus far but could be increased if needed in the future.

```
$parentDir = ((struct file *)arg0)->f_path.dentry->d_parent;
@fullPath[0] = str($parentDir->d_name.name);
...
while($count < 10){
    $parentDir = $parentDir->d_parent;
```

All parent directories are stored within the `fullPath` array, which we then iterate through, to log the full path to the file. Filesystem access is denoted with `VFS:` at the beginning of the log line, and also contains information on if the file was read from or written to. An example of this output can be seen below.

```
VFS: Path: /null, Program: bash, Mode: Write
VFS: Path: /null, Program: bash, Mode: Read
```

The next events I attempted to trace were signals. I took inspiration from the bpftrace script `killsnoop.bt` [16]. This script uses the two tracepoints, `sys_enter_kill` and `sys_exit_kill`. These are activated when the `kill()` system call begins to send a signal to a process, and when it is finished sending the signal [17]. The `sys_enter_kill` tracepoint stores the process being sent the signal as well as the signal type in two separate arrays, within the bpftrace script. When the `sys_exit_kill` tracepoint is activated, we know a signal was sent successfully and therefore we log the information gathered in the `sys_enter_kill` tracepoint for a specific process id (*PID*). Signals are denoted with `Signal:` at the beginning of the log line, and also contains information on the PID that sent/received the signal, the process name, the number representing the signal being sent, and the result. An example of this output can be seen below.

```
Signal: PID: 3516 COMM: bash SIG: 9 TPID: 3516 RESULT: 0
```

One downside to this is that if `sys_exit_kill` or `sys_exit_kill` are never activated because an error occurs or for any other reason, the tracing program will not output correct data. This is because it relies on both tracepoint events to create the log entry. This also has the limitation of only being able to trace signals sent through the `kill()` system call. This can be improved by adapting the strategy used by IPCDump [7], which adds additional tracepoints for signal generation. This tracepoint is `signal_generate`, which is called any time a signal is generated [18]. The arguments supplied to the `signal_generate` tracepoint give us the same information as `sys_exit_kill` and `sys_exit_kill`, however, it is more general as it is called any time a signal is generated.

The next events that I attempted to trace were operations on pipes. This is the starting step for generating rules for inter process communication (*IPC*). The implementation took inspiration from IPCDump [7] and uses the `pipe_read` and `pipe_write` kprobes to log information. The function definitions for these two operations, found in the Linux source file `pipe.c` are as follows:

```
static ssize_t pipe_read(struct kiocb *iocb, struct iov_iter *to);
static ssize_t pipe_write(struct kiocb *iocb, struct iov_iter *from);
```

Information is extracted, whenever these kprobes are activated, from the first argument, the `kiocb` struct pointer. As it is possible for the pipe to not be within the filesystem, we instead aim to retrieve the inode of the pipe, by accessing the `file` pointer struct within the `kiocb` pointer.

```
$pipe_inode = ( ((struct file *)(((struct kiocb *) arg0)->ki_filp))->f_inode->i_ino);
```

Pipes are denoted with `Pipe:` at the beginning of the log line and contains information on the process name that is using the pipe, the inode of the pipe, and the operation which can be either read or write. An example of this output can be seen below.

```
Pipe: Comm: grep PipeInode: 44744 Operation: READ
Pipe: Comm: bpftrace PipeInode: 44744 Operation: WRITE
```

An additional modification that can be made to improve tracing pipes, would be to add kernel return probes (*kretprobe*) to the `pipe_read` and `pipe_write` functions. These kernel return probes would be activated when either function returns and would allow us to retrieve information on the last process to read or write to a specific pipe.

Finally, the last events I attempted to trace were capabilities. Capabilities are how Linux distributes specific privileges to processes. I took inspiration from the bpftrace script `capable.bt` [19]. This script uses a kprobe on the `cap_capable` kernel function which activates whenever a security check is made within the kernel. The function definition, found in the Linux header file `security.h`, is as follows:

```
extern int cap_capable(const struct cred *cred, struct user_namespace *ns,
                       int cap, unsigned int opts);
```

The numerical representation of the capability can be extracted using the data passed to the int `cap` argument. This numerical argument is transformed into the string representation of the given capability. If the unsigned int opts has a value of 0, we know that the given process has the given capability. Capabilities are denoted with `CAP:` at the beginning of the log line and contains information on the UID running the process, the PID and name of the process, the capability being checked, and the audit which is the numerical representation of the permitted set of operations.

```
CAP: UID:1000 PID:4772 COMM:sudo CAP:24 NAME:CAP_SYS_RESOURCE AUDIT:0
CAP: UID:1000 PID:4772 COMM:sudo CAP:7 NAME:CAP_SETUID AUDIT:4
```

4 Current Progress

4.1 Tracing Program

The current implementation of the tracing program can trace all events mentioned in section 3.1.2. The tracing program is written in the bpftrace [9] language. The tracing program takes in an optional argument that allows for the tracing of a program with a specific name. This is done by limiting each probe to only log data when the executing process matches the input argument. In bpftrace this is done by adding the following to each probe:

```
/comm == argName/
```

Where `argName` is the name of the desired program to trace. Each probe or tracepoint distinguishes itself in the log by specifying the type of the trace at the beginning of each line. Currently, the trace program can log VFS, Signal, Pipe, and Capability events.

4.2 Translation Program

The current implementation of the translation program is written in Python. It uses a variety of regex strings to match and extract different information within the log, this information is then translated into a YAML security policy for BPFContain. Although the trace program can log VFS, Signal, Pipe, and Capability events, the translation program currently only supports the creation of allow rules for filesystems and signals.

The first class in the translation program is the `TraceFile` class. The `TraceFile` class is used to open the trace log and split it into sections based on the type of activity. Currently, it splits the data into VFS traces and Signal traces.

```
class TraceFile():

    def __init__(self, fileName, proc) -> None:
        # Open the trace file
        with open(fileName) as f:
            traceLines = f.readlines()
        # Remove the first two lines as they are cmd output
        traceLines = traceLines[2:]
        # Split Signals and VFS, filter by process
        self.vfsTraces = [x for x in traceLines if proc in x and 'VFS:' in x]
        self.sigTraces = [x for x in traceLines if proc in x and 'Signal:' in x]
        self.procName = proc
        # Remove duplicates from VFS
        self.vfsTraces = list(set(self.vfsTraces))
        # Remove duplicates from Signals
        self.sigTraces = list(set(self.sigTraces))
```

The `TraceFile` object is then passed into the `TraceToPolicy` class, which generates the output YAML file through several functions. The first function being, `generate_policy_start()`. This function generates the start of the output string that will be written to the YAML file. It includes the process name and the command used to execute the process. Currently, this path is hardcoded manually, but in the future, I would like to output this data in the log file and then grab it from there.

Next, the `TraceToPolicy` class calls the function `generate_policy_allow()`, which calls all the functions associated with creating `allow` rules for the policy. The first of these functions is `generate_device_access()`. This function reads through the array of all VFS traces and grants access to null and/or terminal devices, based on the paths that the process interacted with while it was being traced.

The next function called by `generate_policy_allow()` is `generate_read_write_access()`. This function uses the regex strings below, to match and separate filesystem read and write events.

```
# Regex Strings - Group 0 is the path
read_regex = r"(?<=Path: )([^,]*)(?=, Program: "+traceFile.procName+r", Mode: Read)"
write_regex = r"(?<=Path: )([^,]*)(?=, Program: "+traceFile.procName+r", Mode: Write)"
```

These separated event arrays are then iterated through, and each path is added as a key to a dictionary. This dictionary stores the key value pair `path->[accessArray]`, where the value for each key is an array of access permissions (*read, write, etc.*). We then iterate through this dictionary to create file and filesystem `allow` rules for the traced program. Currently, this function can create rules for any file within the filesystem, however, it is only able to create filesystem rules for the main root directory `/`. Filesystems are denoted by `fs` and files are denoted as `file` in BPFContain.

Finally, the last function called by `generate_policy_allow()` is `generate_self_signals()`. This function uses the regex strings below, to match and separate all the information from the array of traced signal events.

```
# Group 0 and 3 contain the PIDS
# Group 1 is the process name (comm)
# Group 2 is the signal and group 4 is the result
regex_string = r"(?:PID: )(\S*)(?: COMM: )(\S*)(?: SIG: )(\S*)(?: TPID: )(\S*)(?: RESULT: )(\S*)"
```

Then, we iterate through the regex matches and check if our traced program sends any signals to itself. We do this by comparing group 0 and group 3 from the regex match, if a program is sending signals to itself, these two groups will be the same. When we find a valid self-signal we add the process name as a key, to a dictionary. This dictionary stores the key value pair `processName->[signals]`, where the value for each key is an array of signals that can be sent. We then iterate through this dictionary to create signal `allow` rules for signals that the traced program can send to itself. In the future, this will be expanded to be able to send signals to other processes once the trace program is updated to include the `signal_generate` tracepoint.

4.3 Results

Currently, the whole implementation can automatically generate BPFContain security policies for VFS read and write operations, as well as self-signals. Though the tracing script provides additional support for pipes and capabilities, these need to be refined and expanded to fully capture all aspects of inter process communication, before inter process communication is added to the translation program.

Using the program `bash` with a test workload of Creating/Reading/Writing to files, Sending Signals to itself, Reading/Writing to/from a pipe, and using its capabilities. The tracing program generates the log in section 4.3.1. Note that the log has been trimmed to show it can trace each event mentioned in section 3.1.2. This log file can then be fed into the translation program to generate the security policy shown in section 4.3.2.

4.3.1 Example (Trimmed) Trace File

```
Attaching 9 probes...
Tracing VFS calls, Signals, Capibilities, Pipes ... Hit Ctrl-C to end.
VFS: Path: /0, Program: bash, Mode: Read
VFS: Path: /0, Program: bash, Mode: Write
VFS: Path: /, Program: bash, Mode: Read
VFS: Path: /, Program: bash, Mode: Write
Pipe: Comm: bash PipeInode: 154080 Operation: WRITE
Pipe: Comm: bash PipeInode: 154083 Operation: READ
VFS: Path: /null, Program: bash, Mode: Write
VFS: Path: /2, Program: bash, Mode: Read
VFS: Path: /2, Program: bash, Mode: Write
Pipe: Comm: bash PipeInode: 154108 Operation: READ
VFS: Path: /home/jakejazokas/Desktop/namedPipe, Program: bash, Mode: Write
Signal: PID: 7947 COMM: bash SIG: 10 TPID: 7947 RESULT: 0
Signal: PID: 8012 COMM: bash SIG: 9 TPID: 8012 RESULT: 0
CAP: UID:1000 PID:7402 COMM:bash CAP:2 NAME:CAP_DAC_READ_SEARCH AUDIT:0
CAP: UID:1000 PID:7402 COMM:bash CAP:1 NAME:CAP_DAC_OVERRIDE AUDIT:0
VFS: Path: /usr/lib64/libtinfo.so.6.2, Program: bash, Mode: Read
VFS: Path: /usr/lib64/libc.so.6, Program: bash, Mode: Read
VFS: Path: /etc/authselect/nsswitch.conf, Program: bash, Mode: Read
VFS: Path: /etc/passwd, Program: bash, Mode: Read
VFS: Path: /usr/share/terminfo/x/xterm-256color, Program: bash, Mode: Read
VFS: Path: /root/.bashrc, Program: bash, Mode: Read
VFS: Path: /etc/bashrc, Program: bash, Mode: Read
VFS: Path: /etc/profile.d/bash_completion.sh, Program: bash, Mode: Read
Pipe: Comm: bash PipeInode: 154662 Operation: READ
VFS: Path: /etc/bash_completion.d/dog, Program: bash, Mode: Read
VFS: Path: /etc/bash_completion.d/gluster, Program: bash, Mode: Read
VFS: Path: /etc/bash_completion.d/lilv, Program: bash, Mode: Read
```

4.3.2 Example Generated Policy for Bash

```
name: bash
cmd: /usr/bin/bash

defaultTaint: false

allow:
  - device: terminal
  - device: null

  - file: {path: /usr/lib64/libc.so.6, access: r}
  - file: {path: /etc/profile.d/which2.sh, access: r}
  - file: {path: /etc/profile.d/debuginfod.sh, access: r}
  - file: {path: /usr/share/bash-completion/bash_completion, access: r}
  - file: {path: /etc/bashrc, access: r}
  - file: {path: /etc/bash_completion.d/python-argcomplete, access: r}
  - file: {path: /etc/bash_completion.d/redefine_filedir, access: r}
  - file: {path: /etc/profile.d/colorxzgrep.sh, access: r}
  - file: {path: /etc/bash_completion.d/lilv, access: r}
  - file: {path: /etc/profile.d/colorls.sh, access: r}
  - file: {path: /etc/profile.d/gawk.sh, access: r}
  - file: {path: /etc/profile.d/nano-default-editor.sh, access: r}
  - file: {path: /etc/bash_completion.d/gluster, access: r}
  - file: {path: /etc/authselect/nsswitch.conf, access: r}
  - file: {path: /root/.bash_history, access: rw}
  - file: {path: /etc/profile.d/colorgrep.sh, access: r}
  - file: {path: /etc/profile.d/vte.sh, access: r}
  - file: {path: /etc/profile.d/colorzgrep.sh, access: r}
  - file: {path: /etc/bash_completion.d/authselect-completion.sh, access: r}
  - file: {path: /usr/lib64/libtinfo.so.6.2, access: r}
  - file: {path: /etc/profile.d/less.sh, access: r}
  - file: {path: /root/.bashrc, access: r}
  - file: {path: /etc/passwd, access: r}
  - file: {path: /etc/bash_completion.d/abrt.bash_completion, access: r}
  - file: {path: /etc/inputrc, access: r}
  - file: {path: /etc/bash_completion.d/dog, access: r}
  - file: {path: /etc/profile.d/bash_completion.sh, access: r}
  - file: {path: /etc/profile.d/lang.sh, access: r}
  - file: {path: /usr/share/terminfo/x/xterm-256color, access: r}
  - file: {path: /etc/profile.d/PackageKit.sh, access: r}
  - file: {path: /etc/profile.d/flatpak.sh, access: r}
  - file: {path: /home/jakejazokas/Desktop/namedPipe, access: w}

  - fs: {pathname: /, access: rw}

  - signal: {to: bash, signals: [sigUsr1, sigKill]}
```

5 Future Progress

5.1 Tracing Program

To have a fully working tracing program that can capture all of the events specified in BPFContain [5], a few more features need to be added.

The first of which is to implement checks for access, modify, delete, and execute permissions on both files and filesystems. Execute permissions could be checked by tracing the `execve()` system call. Since it does not return on success, we can check if the call returns, and if it doesn't, grant execute permissions for the traced program. Delete permissions could be checked by using the `vfs_unlink` and `vfs_rmdir` kprobes. These kprobes are activated when a file or directory is deleted/removed/unlinked. The second argument of both kernel functions gives us access to a `dentry` struct pointer, which we can use to determine the file being deleted. A possible solution to get access and modify permissions would be to probe the `vfs_open` kernel function. The third argument of this function gives us access to a `cred` struct pointer, which we can use determine the traced programs permitted capabilities. From these permissions, we could then determine if the traced program can access or modify the file [20]. An additional modification to the filesystem tracing would be to implement support for access to devices, given by their major number. As devices are a type of special file, we can add an additional check that determines if the file being accessed is a device special file. If it is a device file, the output should be changed to start with `NumberedDevice:` rather than `VFS:`. In this case, we would also modify the output to display the major number of the device, rather than the path.

The second feature is to implement checks for different kinds of inter process communication. Currently, we can trace pipes and self-signals. But the program is still lacking general signals and sockets. Section 3.1.2 outlines a possible method to implement more general signals by using the `signal_generate` tracepoint. Unix sockets could be traced by using the `unix_stream_sendmsg` and `unix_stream_recvmsg` kernel probes. Each of these functions takes in a `socket` struct pointer as the first argument [21]. This struct can then be used to find the processes that are sending and receiving data. BPFContain only differentiates between two kinds of inter process communication, using Signal rules and a general IPC rule. This inter process communication could contain pipes and/or sockets. So, signals, pipes, and sockets all need to be traced to deduce security rules regarding which processes the traced program is allowed to communicate with.

The last feature is to implement restrictions and tainting. A possible implementation of this could use return probes for all the traced functions mentioned in this paper. This would allow us to check if every operation was successful or not. If an operation is not successful, it will be denoted, so that the translation program can add it to its list of restrictions.

If there is time remaining after the features above have been implemented, I would like to implement support for networking. This would most likely entail probing `TCP` and `UDP`, `connect` and `accept` functions. However, more research is still needed to determine the best approach.

5.2 Translation Program

Once all appropriate events have been captured by the trace program. The translation program will need to be modified to interpret the new logs.

A new function `generate_numbered_device_access()` will need to be created. This function will take in a list of traces generated through interaction with numbered devices and will create a string that will be included in the final output. The string will contain both the device number and the access that the traced program has with the numbered device. A new array containing the device interaction traces will be required in the `TraceFile` class as well.

The function `generate_read_write_access()` will have to be modified to include the permission bits: access (*a*), modify (*m*), and execute (*x*). This modification entails changing the loop that adds access permissions to the dictionary entries for a given path. The new loop will consider the new permission bits given in the trace, on top of the existing read (*r*) and write (*w*) permissions.

A new function `generate_ipc()` will need to be created. This function will take in all of the inter process communication data in the trace log file. This data will include both pipes and Unix sockets. Inter process communication rules can then be determined based on if any other programs are interacting with the pipes or sockets that the traced program is also using. Also, new arrays will be required in the `TraceFile` class to store the pipe and Unix socket traces.

A new function `generate_general_signals()` will need to be created. This function will be a modified version of `generate_self_signals()`. However, it will not include the check that the sending PID is equal to the receiving PID.

A new function `generate_capabilities()` will need to be created. The input data for this function already exists in the trace, but the `TraceFile` class does not yet read it into an array. Once it is stored in an array in the `TraceFile` class, the `TraceToPolicy` function `generate_capabilities()` will simply format the array of capabilities into a BPFContain readable rule.

Currently, the `__init__` method of the `TraceToPolicy` class calls two functions, `generate_policy_start()` and `generate_policy_allow()`. Two additional functions `generate_policy_restrict()` and `generate_policy_taint()` will be created and then called here. The `generate_policy_restrict()` function will translate any denied operations in the trace log, into the appropriately formatted BPFContain rules. The `generate_policy_taint()` function will have to define rules, that if executed, will cause the process to become tainted. Once a process is tainted, its security policy becomes stricter [5]. Generally, these rules should restrict when a program tries to perform an operation that would make it less trusted [5]. A basic networking taint rule could be determined by if a policy with no allowed network access, tries to access a network. However, filesystem and inter process communication rules may be harder to automate and will require further research, as a custom taint tracking system will need to be developed. Current research in the field of taint tracking includes papers with implementations such as CONFLUX [22] and TaintEraser [23].

If there is time remaining after the features above have been implemented, and network activity tracing is functioning as intended. Then I will attempt to implement a function that translates the networking

events into BPFContain policy. BPFContain policy for networking is a simple set of rules that outline if the process can act as a client or server and if it can send or receive networked data [5].

6 References

[1] "eBPF," [Online]. Available: https://ebpf.io/.

[2] "Kernel Probes (Kprobes) - The Linux Kernel documentation," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[3] "Uprobe-tracer: Uprobe-based Event Tracing - The Linux Kernel documentation," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/uprobetracer.html.

[4] "Using the Linux Kernel Tracepoints - The Linux Kernel documentation," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[5] W. P. Findlay, "A Practical, Lightweight, and Flexible Confinement Framework in eBPF," Carleton University, Ottawa, 2021.

[6] "audit2allow(1) - Linux man page," [Online]. Available: https://linux.die.net/man/1/audit2allow.

[7] "guardicore/IPCDump - Github," [Online]. Available: https://github.com/guardicore/ipcdump.

[8] "iovisor/gobpf - Github," [Online]. Available: https://github.com/iovisor/gobpf.

[9] "bpftrace: High-level tracing language for Linux eBPF," [Online]. Available: https://github.com/iovisor/bpftrace.

[10] "About DTrace," [Online]. Available: http://dtrace.org/blogs/about/.

[11] "SystemTap wiki," [Online]. Available: https://sourceware.org/systemtap/wiki.

[12] E. Rocca, "Comparing SystemTap and bpftrace," 13 April 2021. [Online]. Available: https://lwn.net/Articles/852112/#:~:text=The%20important%20design%20distinction%20between,is%20then%20compiled%20to%20BPF.

[13] "BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more," [Online]. Available: https://github.com/iovisor/bcc.

[14] "Using the DNF software package manager," [Online]. Available: https://docs.fedoraproject.org/en-US/quick-docs/dnf/.

[15] "INSTALL.md at master · iovisor/bpftrace," [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/INSTALL.md.

[16] "killsnoop.bt - Github," [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/tools/killsnoop.bt.

[17] "Driver Basics - sys_kill," [Online]. Available: https://docs.kernel.org/driver-api/basics.html?highlight=kill#c.sys_kill.

[18] "signal.h File Reference," [Online]. Available: https://docs.huihoo.com/doxygen/linux/kernel/3.7/include_2trace_2events_2signal_8h.html.

[19] "bpftrace/capable.bt - Github," [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/tools/capable.bt.

[20] "Credentials in Linux - The Linux Kernel documentation," [Online]. Available: https://www.kernel.org/doc/html/v4.15/security/credentials.html.

[21] "include/linux/net.h - Linux source code (v4.4)," [Online]. Available: https://elixir.bootlin.com/linux/v4.4/source/include/linux/net.h#L110.

[22] K. H. e. J. Bell, "A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships," *ACM Trans. Softw. Eng. Methodol.,* vol. 31, no. 2, 2021.

[23] J. J. D. S. T. K. e. D. W. D. (yu) Zhu, "TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking," *ACM SIGOPS Operating Systems Review,* vol. 45, no. 1, 2011.