Normalizing Logs to Canonical Form

The audit log contains an immense number of raw low level system events that do not readily reveal the causal relationship between system entities. Hence, it cannot be coupled with the high level artifacts in CTI reports for the reasons discussed below. We also discuss our workarounds for generating the normalized event log form.

A. Asymmetric System Call Arguments

Every system call (syscall) has its kind of arguments, which are not symmetrically structured across the audit log. Listing 1 shows examples of system call records from the event log for Ubuntu 14.04 (64 bit). For instance, listing 1(a) shows an event log record for `execve` syscall ($syscall = 59$) with the syscall arguments are distributed in `a0` (/bin/bash), `a1` (/usr/share/gh0st) and `a2` fields. On the other hand, it is clear that `fork` syscall ($syscall = 56$) allocates its main argument, the process ID (PID) of the new forked sub-process, in the `exit` field (see Listing 1(b)). The `chown` syscall family attributes the owner (user) id and group id in fields `a2` and `a3`, respectively. We can see in Listing 1(g) that the owner id and group id are both 0, which refers that the owner is now root. We need to fix this to represent system entities and the relationship between them in symmetric structure to map them to the high level descriptions in the CTI reports.

B. System Entities Unique Identifiers

We also observe that no explicit unique identifier is provided for the system entities in the event log record. Even if the PID can uniquely identify a running process at the runtime, the same PID can be later reused by another process after the earlier process terminates. Hence, the PID or the PPID cannot be used to uniquly identify a process and its parent across the system runtime.

In addition, for particular syscalls, such as `execve`, the executed process inherits the PID of the origin (source) process. So technically, the origin process is not clearly identified in the `execve` syscall event. We run a backward tracing [1] on all previously run processes to pinpoint the recent process with the same PID before the new process is started.

C. System Call Dependencies

For specific syscalls, event records are dependent on previous calls (e.g., `read` and `write` depend on `open`). As in Listings 1(d) and 1(e), the process `scp` reads and writes an object which is not included in the corresponding log record. A backward scan for the event logs shows that the `open` syscall record (see Listing 1(f)) contains the object to be read or written to, with its identifier (inode). We complement `read` and `write` syscall records with that missing information in our normalized records.

D. Data Format/Encoding

The audit log presents syscall arguments in different formats as evident in Listing 1, including: decimal values for `pid, ppid` in all syscalls, hex values for `a1, a2, a3` in `chmod` (see Listing 1(h)), and string values for `exe`. Besides, the same attribute can be in different format based on the syscall; for example `a2, a3` are represented in decimal values for the `chown` syscall family, but in hex for `chmod`. Once the data format representation is determined, it is important to understand the encoding used with every attribute before being able to reveal its value. As an example, `a2, a3` for the `chmod` syscall family are represented in hex and need to be converted into octal format to reveal the permissions granted for the object. Other attributes in the event logs contain encoded packed values. For example, in `connect` (see Listing 1(c)) and `accept` syscalls, the `saddr` attribute which includes the socket information (IP Address and Port) is encoded in hex packed format which needs to be processed to unpack it and retrieve the corresponding IP Address and Port.

To overcome the above challenges, we provide a canonical representation of the audit log with the following features:

- Compact format: we combine records that have the same syscall ID (indicating that they belong to the same operation), in one record in the compact form. For example, `execve` spwans at least the following record types in the audit log: `SYSCALL`, `EXECVE`, `CMD`, `PATH`, and `PROCTITLE`. We correlate all these events and include all the important information in one compact record.
- Unique identifier representation: we create a universal unique identifier (UUID) for every entity (e.g., processes, files, and sockets) in the system to uniquely identify the entity across the runtime. This requires analysis of different syscalls and their arguments and attributes.
- Symmetrical structure: we symmetrically structure every record in the new canonical form across different syscalls. The new record, in our proposal, contains 9 fields which capture the needed information for preserving causality among system entities during the system runtime: Timestamp, ID, SubjectUUID, SubjectProcess, Action, ObjectUUID, ObjectName, ActionDetails, and Hostname.

Our devised canonical log representation provides all the important information needed in attack forensics and real-time attack artifacts detection. For further reducing the log size while preserving the causality relationship between system entities, we applied a causality preserved reduction technique as in [2]. The core idea is to merge excessive events between the same pair of entities. Every type of event between a pair of entities is maintained in a stack. Every time the same type of event is occurred between the same pair of entities, the event is checked if it can be aggregated with the events in the

Listing 1 The audit log - system call record examples.

(a) Execve Syscall

```
type=SYSCALL msg=audit(1618071287.244:687514):syscall
=59 success=yes exit=0 a0=1579088 a1=1532a88
a2=163c008 a3=598 ppid=2077 pid=2083 exe="/bin/bash"
type=EXECVE msg=audit(1618071287.244:687514):
a0="/bin/bash" a1="/usr/share/gh0st" a2="start"
type=PATH msg=audit(1618071287.244:687514): item=0
name="/usr/share/gh0st" inode=792704
```

(b) Fork Syscall

```
type=SYSCALL msg=audit(1618072037.116:827848):syscall
=56 success=yes exit=7177 a0=1200011 a1=0 a2=0
a3=7fbec78bd9d0 ppid=7162 pid=7176 exe="/usr/bin/scp"
```

(c) Connect Syscall

```
type=SYSCALL msg=audit(1618071198.080:665224):
syscall=42 success=yes exit=0 a0=aa
a1=7fe95d5b17b0 a2=10 a3=0 items=0 ppid=1407
pid=3461 exe="/usr/lib/firefox/firefox"
type=SOCKADDR msg=audit(1618071198.080:665224):
saddr=020000000DE1BD3D0000000000000000
```

(d) Read Syscall

```
type=SYSCALL msg=audit(1618072049.092:828551):
syscall=0 success=yes exit=221 ppid=7162 pid=7176
exe="/usr/bin/scp"
```

(e) Write Syscall

```
type=SYSCALL msg=audit(1618072049.092:828550):
syscall=1 success=yes exit=80 a0=1 a1=7fff327bd760
a2=50 a3=1 items=0 ppid=7162 pid=7176
exe="/usr/bin/scp"
```

(f) Open Syscall

```
type=SYSCALL msg=audit(1618072049.092:828543):
syscall=2 success=yes exit=3 a1=800 a2=0 a3=8
items=1 ppid=7162 pid=7176 exe="/usr/bin/scp"
type=CWD msg=audit(1618072049.092:828543):
cwd="/home/ubuntu"
type=PATH msg=audit(1618072049.092:828543):
item=0 name="/etc/hosts" inode=2359455
```

(g) Fchownat Syscall

```
type=SYSCALL msg=audit(1617660032.907:154792795):
syscall=260 success=yes exit=0 a0=ffffff9c
a1=1da5cb0 ppid=114171 pid=114172 exe="/bin/chown"
type=PATH msg=audit(1617660032.907:154792795):
name="priv_key.txt" inode=2495043
```

(h) Fchmodat Syscall

```
type=SYSCALL msg=audit(1617660053.219:154794519):
syscall=268 success=yes exit=0 a0=ffffffffffffff9c
a1=11570f0 a2=1ff a3=3c0 ppid=114176 pid=114177
exe="/bin/chmod"
type=CWD msg=audit(1617660053.219:154794519):
cwd="/home/ubuntu"
type=PATH msg=audit(1617660053.219:154794519):
name="gh0st.sh" inode=2495042
```

stack. The aggregation is done if the two events (the new occurred event and the event in the stack) have the same backward and forward trackability. Any two events between two nodes are said to have same backward trakability if no other incoming events to the first node has occured between the end time of the two events in question. On the other hand, any two events between two nodes are said to have same forward trakability if no other outgoing events from the second node has occured between the start time of the two events in question. If both conditions match, the two events are said to have the same causal dependency and are then merged together. This helps in reducing the intense bursts of semantically similar events which are produced by system daemons and other several applications [2]. This is the last step of log processing done by the LogCore engine before building the whole system provenance graph. A significant point about our provenance graph is that because it is a highly compact version of the audit log, it requires less memory which facilitates real-time ingestion of events and generation of the graph over a long period of time. On this provenance graph, we apply the generated attack behavior queries to pinpoint attack behaviors as in the CTI reports.

References

[1] N. Michael, J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates, "On the forensic validity of approximated audit logs," in *Annual Computer Security Applications Conference*, 2020, pp. 189–202.

[2] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 504–516.