Perf events and tool security

Overview

Usage of Performance Counters for Linux (perf_events) [1] , [2] , [3] can impose a considerable risk of leaking sensitive data accessed by monitored processes. The data leakage is possible both in scenarios of direct usage of perf_events system call API [2] and over data files generated by Perf tool user mode utility (Perf) [3] , [4] . The risk depends on the nature of data that perf_events performance monitoring units (PMU) [2] and Perf collect and expose for performance analysis. Collected system and performance data may be split into several categories:

1. System hardware and software configuration data, for example: a CPU model and its cache configuration, an amount of available memory and its topology, used kernel and Perf versions, performance monitoring setup including experiment time, events configuration, Perf command line parameters, etc.
2. User and kernel module paths and their load addresses with sizes, process and thread names with their PIDs and TIDs, timestamps for captured hardware and software events.
3. Content of kernel software counters (e.g., for context switches, page faults, CPU migrations), architectural hardware performance counters (PMC) [8] and machine specific registers (MSR) [9] that provide execution metrics for various monitored parts of the system (e.g., memory controller (IMC), interconnect (QPI/UPI) or peripheral (PCIe) uncore counters) without direct attribution to any execution context state.
4. Content of architectural execution context registers (e.g., RIP, RSP, RBP on x86_64), process user and kernel space memory addresses and data, content of various architectural MSRs that capture data from this category.

Data that belong to the fourth category can potentially contain sensitive process data. If PMUs in some monitoring modes capture values of execution context registers or data from process memory then access to such monitoring modes requires to be ordered and secured properly. So, perf_events performance monitoring and observability operations are the subject for security access control management [5] .

perf_events access control

To perform security checks, the Linux implementation splits processes into two categories [6] : a) privileged processes (whose effective user ID is 0, referred to as superuser or root), and b) unprivileged processes (whose effective UID is nonzero). Privileged processes bypass all kernel security permission checks so perf_events performance monitoring is fully available to privileged processes without access, scope and resource restrictions.

Unprivileged processes are subject to a full security permission check based on the process's credentials [5] (usually: effective UID, effective GID, and supplementary group list).

Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities [6] , which can be independently enabled and disabled on per-thread basis for processes and files of unprivileged users.

Unprivileged processes with enabled CAP_PERFMON capability are treated as privileged processes with respect to perf_events performance monitoring and observability operations, thus, bypass *scope* permissions checks in the kernel. CAP_PERFMON implements the principle of least privilege [13] (POSIX 1003.1e: 2.2.2.39) for performance monitoring and observability operations in the kernel and provides a secure approach to performance monitoring and observability in the system.

For backward compatibility reasons the access to perf_events monitoring and observability operations is also open for CAP_SYS_ADMIN privileged processes but CAP_SYS_ADMIN usage for secure monitoring and observability use cases is discouraged with respect to the CAP_PERFMON capability. If system audit records [14] for a process using perf_events system call API contain denial records of acquiring both CAP_PERFMON and CAP_SYS_ADMIN capabilities then providing the process with CAP_PERFMON capability singly is recommended as the preferred secure approach to resolve double access denial logging related to usage of performance monitoring and observability.

Prior Linux v5.9 unprivileged processes using perf_events system call are also subject for PTRACE_MODE_READ_REALCREDS ptrace access mode check [7] , whose outcome determines whether monitoring is permitted. So unprivileged processes provided with CAP_SYS_PTRACE capability are effectively permitted to pass the check. Starting from Linux v5.9 CAP_SYS_PTRACE capability is not required and CAP_PERFMON is enough to be provided for processes to make performance monitoring and observability operations.

Other capabilities being granted to unprivileged processes can effectively enable capturing of additional data required for later performance analysis of monitored processes or a system. For example, CAP_SYSLOG capability permits reading kernel space memory addresses from /proc/kallsyms file.

Privileged Perf users groups

Mechanisms of capabilities, privileged capability-dumb files [6], file system ACLs [10] and sudo [15] utility can be used to create dedicated groups of privileged Perf users who are permitted to execute performance monitoring and observability without limits. The following steps can be taken to create such groups of privileged Perf users.

1. Create perf_users group of privileged Perf users, assign perf_users group to Perf tool executable and limit access to the executable for other users in the system who are not in the perf_users group:

```
# groupadd perf_users
# ls -alhF
-rwxr-xr-x  2 root root  11M Oct 19 15:12 perf
# chgrp perf_users perf
# ls -alhF
-rwxr-xr-x  2 root perf_users  11M Oct 19 15:12 perf
# chmod o-rwx perf
# ls -alhF
-rwxr-x---  2 root perf_users  11M Oct 19 15:12 perf
```

2. Assign the required capabilities to the Perf tool executable file and enable members of perf_users group with monitoring and observability privileges [6] :

```
# setcap "cap_perfmon,cap_sys_ptrace,cap_syslog=ep" perf
# setcap -v "cap_perfmon,cap_sys_ptrace,cap_syslog=ep" perf
perf: OK
# getcap perf
perf = cap_sys_ptrace,cap_syslog,cap_perfmon+ep
```

If the libcap [16] installed doesn't yet support "cap_perfmon", use "38" instead, i.e.:

```
# setcap "38,cap_ipc_lock,cap_sys_ptrace,cap_syslog=ep" perf
```

Note that you may need to have 'cap_ipc_lock' in the mix for tools such as 'perf top', alternatively use 'perf top -m N', to reduce the memory that it uses for the perf ring buffer, see the memory allocation section below.

Using a libcap without support for CAP_PERFMON will make cap_get_flag(caps, 38, CAP_EFFECTIVE, &val) fail, which will

lead the default event to be 'cycles:u', so as a workaround explicitly ask for the 'cycles' event, i.e.:

```
# perf top -e cycles
```

To get kernel and user samples with a perf binary with just CAP_PERFMON.

As a result, members of perf_users group are capable of conducting performance monitoring and observability by using functionality of the configured Perf tool executable that, when executes, passes perf_events subsystem scope checks.

In case Perf tool executable can't be assigned required capabilities (e.g. file system is mounted with nosuid option or extended attributes are not supported by the file system) then creation of the capabilities privileged environment, naturally shell, is possible. The shell provides inherent processes with CAP_PERFMON and other required capabilities so that performance monitoring and observability operations are available in the environment without limits. Access to the environment can be open via sudo utility for members of perf_users group only. In order to create such environment:

1. Create shell script that uses capsh utility [16] to assign CAP_PERFMON and other required capabilities into ambient capability set of the shell process, lock the process security bits after enabling SECBIT_NO_SETUID_FIXUP, SECBIT_NOROOT and SECBIT_NO_CAP_AMBIENT_RAISE bits and then change the process identity to sudo caller of the script who should essentially be a member of perf_users group:

```
# ls -alh /usr/local/bin/perf.shell
-rwxr-xr-x. 1 root root 83 Oct 13 23:57 /usr/local/bin/perf.shell
# cat /usr/local/bin/perf.shell
exec /usr/sbin/capsh --iab=^cap_perfmon --secbits=239 --user=$SUDO_USER -- -l
```

2. Extend sudo policy at /etc/sudoers file with a rule for perf_users group:

```
# grep perf_users /etc/sudoers
%perf_users    ALL=/usr/local/bin/perf.shell
```

3. Check that members of perf_users group have access to the privileged shell and have CAP_PERFMON and other required capabilities enabled in permitted, effective and ambient capability sets of an inherent process:

```
$ id
uid=1003(capsh_test) gid=1004(capsh_test) groups=1004(capsh_test),1000(perf_users) context=unconfined_u:unconfined_r:unconf
$ sudo perf.shell
[sudo] password for capsh_test:
$ grep Cap /proc/self/status
CapInh:        0000004000000000
CapPrm:        0000004000000000
CapEff:        0000004000000000
CapBnd:        000000ffffffffff
CapAmb:        0000004000000000
$ capsh --decode=0000004000000000
0x0000004000000000=cap_perfmon
```

As a result, members of perf_users group have access to the privileged environment where they can use tools employing performance monitoring APIs governed by CAP_PERFMON Linux capability.

This specific access control management is only available to superuser or root running processes with CAP_SETPCAP, CAP_SETFCAP [6] capabilities.

Unprivileged users

perf_events *scope* and *access* control for unprivileged processes is governed by perf_event_paranoid [2] setting:

-1:
: Impose no *scope* and *access* restrictions on using perf_events performance monitoring. Per-user per-cpu perf_event_mlock_kb [2] locking limit is ignored when allocating memory buffers for storing performance data. This is the least secure mode since allowed monitored *scope* is maximized and no perf_events specific limits are imposed on *resources* allocated for performance monitoring.

>=0:
: *scope* includes per-process and system wide performance monitoring but excludes raw tracepoints and ftrace function tracepoints monitoring. CPU and system events happened when executing either in user or in kernel space can be monitored and captured for later analysis. Per-user per-cpu perf_event_mlock_kb locking limit is imposed but ignored for unprivileged processes with CAP_IPC_LOCK [6] capability.

>=1:
: *scope* includes per-process performance monitoring only and excludes system wide performance monitoring. CPU and system events happened when executing either in user or in kernel space can be monitored and captured for later analysis. Per-user per-cpu perf_event_mlock_kb locking limit is imposed but ignored for unprivileged processes with CAP_IPC_LOCK capability.

>=2:
: *scope* includes per-process performance monitoring only. CPU and system events happened when executing in user space only can be monitored and captured for later analysis. Per-user per-cpu perf_event_mlock_kb locking limit is imposed but ignored for unprivileged processes with CAP_IPC_LOCK capability.

Resource control

Open file descriptors

The perf_events system call API [2] allocates file descriptors for every configured PMU event. Open file descriptors are a per-process accountable resource governed by the RLIMIT_NOFILE [11] limit (ulimit -n), which is usually derived from the login shell process. When configuring Perf collection for a long list of events on a large server system, this limit can be easily hit preventing required monitoring configuration. RLIMIT_NOFILE limit can be increased on per-user basis modifying content of the limits.conf file [12] . Ordinarily, a Perf sampling session (perf record) requires an amount of open perf_event file descriptors that is not less than the number of monitored events multiplied by the number of monitored CPUs.

Memory allocation

The amount of memory available to user processes for capturing performance monitoring data is governed by the perf_event_mlock_kb [2] setting. This perf_event specific resource setting defines overall per-cpu limits of memory allowed for mapping by the user processes to execute performance monitoring. The setting essentially extends the RLIMIT_MEMLOCK [11] limit, but only for memory regions mapped specifically for capturing monitored performance events and related data.

For example, if a machine has eight cores and perf_event_mlock_kb limit is set to 516 KiB, then a user process is provided with 516 KiB * 8 = 4128 KiB of memory above the RLIMIT_MEMLOCK limit (ulimit -l) for perf_event mmap buffers. In particular, this means that, if the user wants to start two or more performance monitoring processes, the user is required to manually distribute the available 4128 KiB between the monitoring processes, for example, using the --mmap-pages Perf record mode option. Otherwise, the first started performance monitoring process allocates all available 4128 KiB and the other processes will fail to proceed due to the lack of memory.

RLIMIT_MEMLOCK and perf_event_mlock_kb resource constraints are ignored for processes with the CAP_IPC_LOCK capability. Thus, perf_events/Perf privileged users can be provided with memory above the constraints for perf_events/Perf performance monitoring purpose by providing the Perf executable with CAP_IPC_LOCK capability.

Bibliography

[1] https://lwn.net/Articles/337493/

[2] (*1*,*2*,*3*,*4*,*5*,*6*,*7*) http://man7.org/linux/man-pages/man2/perf_event_open.2.html

[3] (*1*,*2*) http://web.eece.maine.edu/~vweaver/projects/perf_events/

[4] https://perf.wiki.kernel.org/index.php/Main_Page

[5] (*1*,*2*) https://www.kernel.org/doc/html/latest/security/credentials.html

[6] (*1*,*2*,*3*,*4*,*5*,*6*) http://man7.org/linux/man-pages/man7/capabilities.7.html

[7] http://man7.org/linux/man-pages/man2/ptrace.2.html

[8] https://en.wikipedia.org/wiki/Hardware_performance_counter

[9] https://en.wikipedia.org/wiki/Model-specific_register

[10] http://man7.org/linux/man-pages/man5/acl.5.html

[11] (*1*,*2*) http://man7.org/linux/man-pages/man2/getrlimit.2.html

[12] http://man7.org/linux/man-pages/man5/limits.conf.5.html

[13] https://sites.google.com/site/fullycapable

[14] http://man7.org/linux/man-pages/man8/auditd.8.html

[15] https://man7.org/linux/man-pages/man8/sudo.8.html

[16] (*1*,*2*) https://git.kernel.org/pub/scm/libs/libcap/libcap.git/