Extending Non-Repudiable Logs with eBPF

Avery Blanchard\textsuperscript{1}, Gheorghe Almasi\textsuperscript{2}, James Bottomley\textsuperscript{2} and Hubertus Franke\textsuperscript{2}

\textsuperscript{1} Duke University
\textsuperscript{2} IBM Research

November 13th, 2023
Visibility into System State with eBPF

Userspace

- Application
  - bytecode

Operating System

- bpf()
- eBPF
- Verifier
- JIT compiler
- eBPF
- eBPF programs
- event!

Boot loader

Hardware
Visibility into System State with eBPF

The diagram illustrates the process of visibility into system state using eBPF. It shows the flow from the hardware to the boot loader, through the operating system, and into the userspace.

- **Application**: The application in userspace invokes a function `bpf()`.
- **bpf()**: This function is part of the `eBPF` framework.
- **Verifier**: The `eBPF` verifier checks the program.
- **JIT compiler**: The `eBPF` program is compiled just-in-time (JIT) into bytecode.
- **Event log**: The compiled `eBPF` programs can trigger events, and these are logged.

The `eBPF` programs interact with various parts of the system, allowing for detailed monitoring and analysis of system state.
Visibility into System State with eBPF

Can we trust this?
Visibility into System State with eBPF

The addition of non-repudiable logging to eBPF allows for the verification of workload/environment specific data.
Building a Chain of Trust

Userspace
- Application
  - bytecode

Operating System
- `bpf()`
- `eBPF` (Verifier)
- JIT compiler
  - `eBPF` programs
  - event!

Boot loader

Hardware
Building a Chain of Trust

Hardware

Boot loader

Operating System

Userspace

Application

bytecode

bpf()

Verifier

JIT compiler

eBPF programs

sha256: ...

measures

event!

measures

sha256: ...

Hardware
Rooting Trust in Hardware

- **Hardware**
  - TPM
  - EK
  - AK
  - PCRs
  - Cryptographic processor

- **Boot loader**
  - Measures
  - Stores

- **Operating System**
  - **Application**
    - bytecode
  - **bpf()**
  - **Verifier**
    - eBPF programs
  - **JIT compiler**
    - eBPF

- **Userspace**
Extending Measurements Through Runtime

Userspace

- Application A
- Application B
  - bpf()
  - JIT compiler

Operating System

- eBPF
- Verifier
- eBPF programs
  - event!

Linux IMA
- IMA log

Boot loader
- measures
- stores

Hardware
- TPM
- EK
- AK
- PCR
- Cryptographic processor

TPM
EK
AK
PCRs
Building Trust in Environments

Attesting Machine

Remote Verifier

Attestation evidence
Attesting System Properties

Attesting Machine

request → challenge ← evidence

response → evidence → result

Relying Party

Policy engine

Verifier

Policy engine
Non-repudiable Logging in eBPF Programs

Hardware

Operating System

Userspace

Application A

bytecode

bpf()

Verifier

JIT compiler

eBPF programs

Measurement and logging interface

IMA log

Boot loader

measures

stores

stores

measures

TPM
EK
AK
PCR
Cryptographic processor
Measurement Interface

Operating System

- eBPF programs
- `bpf_process_measurement()`

Kernel module

- Measurement
- Format log entry
- Append log
- Extend PCR

Log

Hardware

TPM
- EK
- AK
- PCRs
- Cryptographic processor
From the eBPF side

• Available to sleepable eBPF programs
• Programs can provoke the measurement and storage of formatted data and files
From the eBPF side

- Available to sleepable eBPF programs
- Programs can provoke the measurement and storage of formatted data and files

```c
#include "vmlinux.h"
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_helpers.h>
#include <string.h>

#define bpf_target_x86
#define bpf_target_defined
#define PROT_EXEC 0x84

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

struct ebpf_data {
  struct file *file;
  unsigned int ns;
};

extern int bpf_process_measurement(void *, int) __ksym;

SEC("/sys/fs/mmap_file")
int BPF_PROG(mmap_hook, struct file *file, unsigned int reqprot, 
  unsigned int prot, int flags)
{
  struct task_struct *task;
  unsigned int ns;
  int ret;

  if (!file)
    return 0;

  if (reqprot & PROT_EXEC)
    {
    task = (void *) bpf_get_current_task();
    ns = BPF_CORE_READ(task, nsproxy, uts_ns, ns.inum);

    struct ebpf_data data = { .file = file, .ns = ns };

    ret = bpf_process_measurement((void *) &data, 
      sizeof(&data));
    }
  return 0;
}
```
Example Use Case

**Hardware**
- TPM
- EK
- AK
- PCR
- Cryptographic processor

**Operating System**
- Userspace
  - Application A
    - bytecode
  - bpf()
  - eBPF
  - Verifier
  - JIT compiler
    - eBPF

- eBPF programs
- Linux IMA
  - event!
  - IMA log
  - stores
  - measures

**Boot loader**
- measures
- stores

**Application B**
Extending Linux IMA to Containers

**Hardware**
- **TPM**
- **EK**
- **AK**
- **PCR**
- **Cryptographic processor**

**Boot loader**
- Measures
- Stores

**Operating System**
- **Userspace**
  - Application A
  - Userspace
  - `bpf()`
  - Verifier
  - JIT compiler
- `bpf()`
- Measures
- Stores

**Linux IMA**
- eBPF programs
- Stores

**Container A**
- Application
- `bpf()`
- Stores

**Application**
- `bpf()`
- Stores

**Event!**
The Need for Namespace Support

Which of these measurements are from a container?
Extending Linux IMA to Containers

- **Userspace**
  - Application A
  - byte code
- **Operating System**
  - bpf(
  - Verifier
  - JIT compiler
- **Linux IMA**
  - eBPF programs
  - measures
  - stores
  - event!
- **Boot loader**
- **Hardware**
  - TPM
  - EK
  - AK
  - PCR
- **Isolated through namespaces and cgroups**
Adding Namespace Support to IMA
Resulting IMA Log
Resulting IMA Log
The PTRACE wrapper executed a file, stopping after the first instruction to isolate the measurement and TPM extension.
Enabling non-repudiable logging of workload/platform specific system properties using eBPF.

avery.blanchard@duke.edu

https://github.com/avery-blanchard/container-ima