Security Analysis of Linux Kernel Features for Embedded Software Systems in Vehicles

Ludwig Thomeczek

Security Analysis of Linux Kernel Features for Embedded Software Systems in Vehicles

Ludwig Thomeczek

***Abstract*—This paper describes different safety and security mechanisms implemented in the Linux kernel to prevent and protect against accidental or malicious misbehaviour in user applications.**

First, we present a generic system model for user applications with different levels of criticality and deterministic behaviour. From this, a theoretical model for failure modes and attack scenarios on the stability of the operating system and concurrently executed user applications is derived. Then, technologies in the Linux kernel to counter the identified failure modes and attack scenarios are examined and compared.

Current work in progress is to implement requirement-based tests for these security measures and assess their effectiveness, efficiency and limits.

***Keywords*—security, safety, embedded, vehicle, operating systems, Linux**

I. Introduction

With the on-going miniaturization and improvement both in computing power and energy efficiency, electronics in vehicle systems become more integrated and connected. Real-time reporting of process metrics and the ability to alter system settings as well as remote servicing and software updates often call for connection to remote corporate networks, commonly realised via internet connections. This presents a challenge to current vehicle software and hardware designs, which are optimized for safe deterministic behaviour and small footprint size to run in an embedded environment where CPU power and memory are scarce (either due to cost savings, energy concerns or the plain size requirements).

Often new features and services are introduced into these conventional vehicle systems without proper understanding of their implications. In industrial systems, this removal of the air gap between embedded systems and the internet to allow remote access is one of these features. As certain systems never were designed to run in any other environment than a hermetically-sealed network, several prominent industrial SCADA vendors had to deal with vulnerabilities exposing complete power plants and other apparel on the internet for everybody to control [CER] [SE].

With more powerful hardware and an ever-increasing amount of features needed, software designs for new industrial and vehicle components include general-purpose operating systems, in contrast to the traditional software designs with specialised real-time operating systems or bare-metal applications, i.e., application code directly accesses the hardware without using a hardware abstraction layer. General-purpose operating systems provide up-to-date security mechanisms, and useful features for new technological trends, such as Internet-of-Things, Industry 4.0 or Industrial Internet.

An established and widely-used general-purpose operating system is the Linux kernel and its surrounding software ecosystem, often just referred to as Linux or GNU/Linux. As Linux is deployed in systems with a wide range of computing capacities, from large HPC systems to low-power mobile phones, the Linux kernel is versatile and can be extensively adapted for various use cases. Its source code and documentation is provided under an open-source license and is continuously developed for over 20 years.

It can readily be adapted to be deployed onto embedded computers that provide relatively low computing power, comparable to current low-end smartphones. These embedded computing units (CU) are integrated into a certain system environment. In this paper, the assumed system environment is a vehicle, but we recognised that many insights also apply to other industrial environments. The assumed system environment is a network of embedded computers and devices, such as internet gateways or control computers.

The paper's main focus are means for configuration of the Linux Kernel to achieve a safe and secure system in the assumed environment. In this paper, we limit the scope of our analysis: we assume the use of an Preempt-RT Linux kernel and we do not consider special-purpose hardware, hard-disk encryption, redundant disk storage and the bootloader.

II. Theoretical Model based on POSIX

Figure 1 shows a schematic representation depicting a basic system with the most important interacting entities.

Several processes running on the platform (labeled P_1,P_2 $\dots P_N$) are executed simultaneously with multicore and preemptive symmetric multiprocessing (SMP) scheduling. The kernel controls all external interactions, hardware accesses and execution flow of these processes.

Applications request hardware access from the kernel using abstract functions defined by the POSIX API, e.g., the pthread or signals API. For interaction with the storage devices (DISK), the kernel provides a file system (Files) with the corresponding abstract functions to the processes. To use networking hardware (NET) and access the connect networks, Berkeley sockets (Sockets) are presented to the application.

While the underlying hardware may be different depending on the platform, the POSIX abstraction provided to the applications stays the same. For CPU or memory, no abstractions

Ludwig Thomeczek is currently studying at the University of Applied Sciences Landshut and works on his master's thesis at BMW Car IT GmbH.

His supervisors are Tilmann Ochs and Lukas Bulwahn from BMW Car IT and Gernot Hillier and Gudrun Schiedermeier from the University of Applied Sciences Landshut.

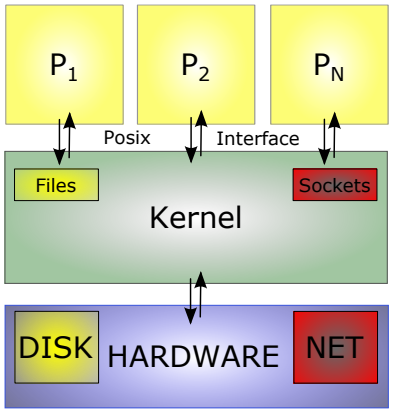

Fig. 1. POSIX based system model

are presented. Each application is agnostic to other running processes, as the kernel and the MMU arrange everything for the application to perceive as if it is the only one running on the system. This is indeed a gross oversimplification, but the separate address space and the interrupt behaviour that facilitate this are commonly not perceived by the application.

III. Technical Use Cases

Based on the system model from section II and the context provided in I, the main system workflow is built around the following key use cases.

A. SMP Software Execution

The basic state of the system as depicted in Figure 1 consists of the processes P_1, P_2, ... P_N running on the system, sharing the common resources.

Managing the usage of these is the applications responsibility, the OS only providing upper boundaries. As all applications assume that they are the only running program on the system, the kernel has to provide the SMP scheduling functionality by rotating control of the CPU between the currently executing processes.

B. Filesystem Usage

Using the filesystem abstraction to access data stored on the disk is one use case.

Configuration data, executables and other data needs to be stored permanently on the system, and applications need to read and write this data. Access to this data is provided with the POSIX filesystem abstraction, which the kernel then translates to disk read/writes.

C. External Communication

Programs running on the system often actively need to establish connection to external services, mostly IP-based connection through common internet infrastructure. Furthermore, external services often need to contact the running processes and request information or send control commands. Examples for this use case are system updates, cloud-based synchronisation or remote control services.

D. Internal Communication

As seen in the context model(section I), the CU is not the only control component in the vehicle. Therefore, another use case is communication to other CUs, actuators or sensors on the same vehicle. In this environment, real-time boundaries may apply to some of these communication paths.

E. Machine-Local Communication

Several different software components will be running simultaneously on the CU represented by the processes P_1,P_2 ... P_N, thus internal communication between the processes needs to be available.

IV. Safety Analysis

The safety analysis is not the main focus of this paper. Hence, only the most relevant requirements for the Linux kernel are highlighted here. The assumed safety requirements are:

1) Correct software code execution
2) Correct integration and execution of hardware acceleration
3) Correct execution of POSIX API calls and kernel functions
4) Freedom from interference between POSIX API calls and kernel functions
5) Freedom from interference between software components
6) Freedom from interference between hardware components

For this paper, correct integration and execution of hardware acceleration is out of scope. Correct software or Kernel/POSIX call execution (Items 1 and 3) can only be proven through formal verification and validation, or a sufficient trust level can be achieved with thorough testing. Freedom from interference by POSIX calls or kernel functions may need to be tested and handled on the source-code level. Freedom from interference between software components (Item 5) is analogous to the SMP use-case described in section III-A. For the last two safety requirements (Items 5 and 6), there exist several technologies in the Linux kernel that may help provide them. We discuss these in the next section.

Freedom from interference between software components: One of the OS scheduler's main objective is to guarantee interruption-free execution of high-priority tasks. To optimize for the specific real-time requirements, programs requiring realtime scheduling have to run at highest priority under a realtime-capable scheduler (SCHED_FIFO or SCHED_RR) [KZL+].

To further minimize unexpected latencies due to migrations between cores, the program can be pinned to a specific CPU core, isolated from other processes [Core Isolation].

To prevent memory or CPU cycle shortage due to demands by other tasks, the cgroup controllers for cpusets, memory and block IO can enforce limits for other processes [CGroups].

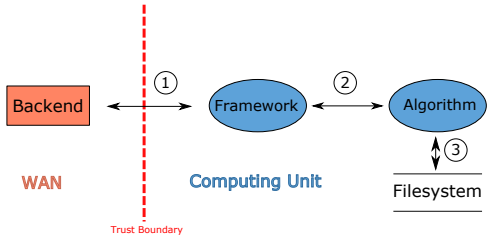

Fig. 2. External Communication Threat Model

Freedom from interference by hardware components: The Linux kernel can only slightly reduce the hardware's influence because many components work independently (e.g., DMA controllers, network cards) and may interrupt the CPU or block other resources at any time.

One possibility to minimize interruptions by hardware is to mask interrupts on a CPU core running real-time critical software. The interrupts then are handled on a different core and hence, do not interrupt critical code. However, not all interrupts can be masked [Core Isolation].

V. Security Analysis

For the security analysis, we determine assets and threats with the STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege, the threats used in the analysis) approach.

The threat modeling approach STRIDE developed by Microsoft models a system and its threats from an attacker's perspective by representing the system with a Data Flow Diagram (DFD). Elements in this diagram (either a process, data flow, data store or interactor) reside inside security-relevant regions that are separated by trust boundaries. Data flows may cross these trust boundaries. However, if they do they are especially prone to threats.

Following a first draft of the diagram, the processes and stores are either further broken down into sub-diagrams until a desired level of complexity is reached, or analysed directly according to their corresponding STRIDE threats. The DFD in Figure 2 depicts some of the assets examined in this analysis, based on the use cases defined in Section III-C. For each use case, we perform a basic security analysis.

Further detailed information about the STRIDE approach is described in [HLOS].

Based on the DFD in Figure 2, we review all components against the applying STRIDE metrics. For example, the "Framework" process in the DFD is vulnerable to all STRIDE threats, and we evaluate for mitigations. For example, picking tampering (the T in STRIDE) as a threat, which in this case most likely happens through buffer overflow or similar memory tampering techniques on running processes, we search for available technologies to counter that threat, in this case ASLR or DEP. We repeat this for the remaining threats (SRIDE), and all other elements of the DFD. Ultimately, we achieve a certain level of security by applying mitigations for each threat. If a higher confidence is required, the components and the DFD can be further refined and re-examined. A full analysis covering all use cases and DFD elements will be described in the author's master thesis.

VI. Technologies

This section provides an condensed overview of most safety and security features present in the kernel that were identified in the analysis.

Adress Space Layout Randomisation

ASLR is a mitigation technique to reduce the probability of a successful exploitation of buffer overflow weaknesses.

Control Groups

Cgroups are a Linux kernel subsystem to partition tasks into different logical groups. These groups then can be subjected to rules regarding resource consumption or allocation.

Core Isolation

Real-time critical programs can be isolated onto one CPU core and interrupts generated by the hardware can be routed to CPU cores not assigned to real-time critical program execution to minimize program execution interruptions.

Data Execution Prevention and W⊕X

DEP encompasses several features to prevent the execution of memory that is not designed to be executable, such as static data segments or program buffers. This protects against attacks which insert malicious code and trick the program into executing it.
W⊕X is the principle to never allow writeable memory to be executable at the same time, preventing effective code injection.

Dm-verity

Dm-verity provides block-based integrity checking for read-only filesystems.

Grsecurity

Grsecurity is an extensive patchset for the Linux kernel to improve its resilience against known attack vectors, adding a Role-based Access Control system (RBAC), *chroot* restrictions and a wide array of miscellaneous hardening features.

Integrity Measurement Architecture / Extended Verification Module

The IMA/EVM subsystem provides file-level monitoring against accidental or malicious changes.

Linux / Posix Capabilities

Capabilities in Linux implement a least privilege principle for processes needing superuser access by allowing finer granularity granting rights.

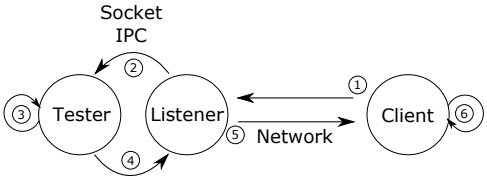

Fig. 3. Test Program Schematic

Linux Security Modules

With Linux security modules, the kernel implements a framework to allow the inclusion of one or several security extensions facilitating Mandatory Access Control (MAC), RBAC or other access control mechanisms that restrict access to internal kernel objects.

Namespaces

Namespaces are a Linux-specific resource isolation feature to compartmentalize system resources, such as Process IDs or networks.

Secure computing filters

Secure Computing (Seccomp) Filtering allows to restrict the use of system calls by processes through a Berkeley Packet Filter (BPF)-based program language, minimizing the kernel's attack surface.

VII. Evaluation of Selected Features

Current work in progress is to cover the selected kernel features chosen from the techniques described in section VI in-depth, extending and completing the initial assessment. This includes prototypes and tests designed to test the effectiveness, impact on system efficiency and deployment footprint.

A. Test Program Design and Implementation

The test program for the effects of the different kernel configurations/technologies is shown schematically in Figure 3. Each circle depicts a running process: Listener and Tester run on the machine under test; the Client process runs on a testing machine. (1) To take a measurement, the client sends a request for a test run to the listener process, noting the timestamp. (2) The Listener accepts the request on the network side and passes it through to the Tester via Inter Process Communication (IPC). (3) The tester then runs the interruption/preemption test to check for any occuring interruption that could occur while an algorithm is processing data for a response. (4) After a successful test run, the Tester replies to the Listener, which (5) then notifies the Client. The client again takes a timestamp, computes the difference with the first timestamp to acquire the elapsed runtime, and stores it in a histogram. (6) After a specified time, the test run is started again until a sufficient number of measurements are obtained.

After finishing the test runs, the client requests the interruption histogram from the server and stores it locally for further analysis. This test program depends highly on the type of machine both tested server and testing client are deployed on, and results are only comparable, when one parameter is changed between test runs on the same hardware setup.

So far, as of the time writing this workshop proposal, the tests have not been performed on the target system, and there is no evaluation to present at the current stage. However, we expect to present tests and an evaluation in time for the workshop in September, and update this paper if possible.

VIII. Related Work

Most security assessments are done with a concrete product in mind and usually not published as they are considered differentiating intellectual property of the assessors. Hence, we believe that as a consequence, the development of a more generic security analysis of an abstract system model has not been in the main focus of current security assessments. Our paper provides a basic analysis to argue the use of certain Linux kernel features for security means. As the Linux kernel is open-source software, there is also a large amount of literature describing the Linux kernel in versatile technical depth. Considering security features, we are only aware of the kernel developer James Morris' overview of the Linux kernel's security features [MORR].

IX. Conclusion

We have describe a generic system model, derived possible attack vectors on this system model and selected the different safety and security mechanisms implemented in the Linux kernel to prevent and protect against these possible attacks. In our current work in progress, we are implementing tests for these safety measures and evaluate their effectiveness, efficiency and limits.

With our effort to make these preliminary results and work in progress public, we hope to provide first steps for a generic and common security assessment for future vehicles, which can be publicly assessed by security experts. This collaborative activity hopefully increases the security in all vehicles and reduces damage and harm of the public due to compromised remote-controlled vehicles.

References

[CER] ICS CERT. Advisory (icsa-11-356-01) - Siemens Simatic Hmi Authentication Vulnerabilities. https://ics-cert.us-cert.gov/advisories/ICSA-11-356-01. accessed 2015-06-24. http://www.trendmicro.com/vinfo/us/threat-encyclopedia/web-attack/54/stuxnet-malware-targets-scada-systems. accessed 2015-06-24.

[SE] Louis-F. Stahl and Ronald Eikenberg. Fuenf nach zwoelf - Die Gefahr im Kraftwerk ist noch nicht gebannt. http://www.heise.de/ct/ausgabe/2013-15-Die-Gefahr-im-Kraftwerk-ist-noch-nicht-gebannt-2319254.html. accessed 2015-06-24.

[HLOS] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack. Uncover security design flaws using the stride approach. https://msdn.microsoft.com/en-us/magazine/cc163519.aspx. accessed 2015-04-20.

[KZL+] M. Kerrisk, P. Zijlstra, J. Lelli, et al. Linux programmer's manual - overview of scheduling APIs. http://man7.org/linux/man-pages/man7/sched.7.html. accessed 2015-06-22.

[MORR] J. Morris Overview of Linux Kernel Security Features. http://www.linux.com/learn/docs/727873-overview-of-linux-kernel-security-features/. accessed 2015-06-22.