Contents

1 State of the art

At the present time we are surrounded by devices that run complex software and operating systems. The number of these devices is still rising and their complexity is growing by the increasing amount computational resources, functionalities and interfaces. Also many of these devices are taking over important tasks such as smart locks or even cardiac pacemakers. Since most of these devices are connected to the public Internet they cannot only be accessed by their owners but anyone. These facts raise the question if we can trust these devices. Many cases in the present have shown that nearly all of these devices have security issues. Since the used systems are too complex to guarantee the absence of software errors the common practice is patching these devices when an error has been discovered. Unfortunately the approach of fixing errors has not proven reliable. Updates are often not or only for a short time available and often do not find their way to the end user. A difference to this practice are sectors where manufacturers are liable for the correctness of their products including software parts. This correctness is achieved by the separation of components and their individual evaluation and verification. The downside of this approach are the high costs, especially for a high amount of components. To reduce the costs multiple functionalities are often combined in one component. To securely separate the functionalities from each other separation kernels are used. Since those need to be secure they only consist of a low complexity and a few thousand lines of code. This reduction comes at the cost of flexibility at runtime which often lets vendors choose more complex but easy to use and configure systems. These often have a large trusted computing base (TCB) of multiple million lines of code. This leads to the question how this contradiction between flexibility and security can be resolved.

1.1 Genode

Genode aims to solve this problem through multiple ways. Instead of a monolithic kernel it uses microkernels. They provide a greater flexibility than separation kernels why keeping the size of the trusted computing base at a manageable level. This is accomplished for example by running device drivers as separate user space processes. User space applications in Genode are also managed by capabilities. They provide a way to allow and deny access to a certain number of system resources without the need for a global complex policy. Additionally to capabilities resources can be managed in

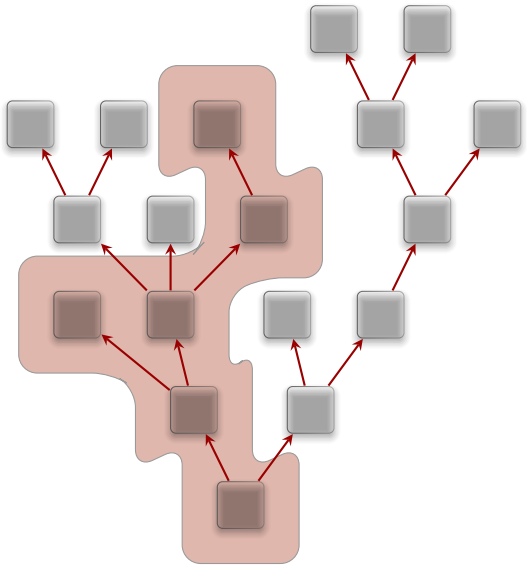

Figure 1.1: Application based TCB [1]

a hierarchical way. Software components can be be partitioned into parts of different complexity. This helps to exclude complex parts from the trusted computing base and therefore reduce the amount of code that needs to be verified. The continuation of this approach leads to an application specific trusted computing base that consists of the application itself and all the parts it depends on [1.1]. [1]

The Genode OS framework is the implementation of this architecture for highly secure special purpose operating systems. Mainly written in C++ it scales from embedded systems with only a few megabytes of memory to big systems with dynamic workloads. The architecture consists of a recursive sandbox structure. Those are controlled from their parents and separated from their siblings. But they can create additional sandboxes and subtrees of children. This creates hierarchical structures and reduces the attack surface of the system since a compromised component cannot access resources of other components. These components are small blocks that can create complex systems without unnecessarily increasing the trusted computing base. Beside being programs they can also provide other functionalities such as device drivers or protocol stacks.

Genode supports multiple CPU architectures such as Intel i386 and ARM. Multiple microkernels such as NOVA and seL4 can be used. Additionally Genode provides a custom kernel implementation that in comparison to other kernels further reduces the trusted computing base. The current framework also provides many usable driver and application components. Beside common periphery drivers such as USB and Intel wireless this also includes complex application frameworks such as Qt5. [1]

To debug components in the development process Genode can also be started on the

current running Linux kernel. This is a good approach for applications but is also limited. Some resources cannot be used since the Linux kernel doesn't provide them to the user space, for example memory mapped IO or direct memory access. To develop and debug native Genode device drivers the kernel needs to act as a microkernel and provide access to these resources in Genode.

1.2 User space drivers on Linux

There have already been different approaches on using suer space drivers on Linux. A, in the Linux world widely known, project is Filesystem in Userspace (FUSE). It's target is to provide users with the ability to use custom file systems without the need to require root privileges or edit kernel code. To achieve this FUSE consists of two parts, a kernel module and a user space library. The library communicates with the kernel module to provide the necessary functionality to the user. While this does not access any hardware resources it shows a concept how kernel resources can be accessed from the user space. [2]

Another approach is Userspace I/O (UIO). UIO addresses industrial systems that often require I/O cards which only need some mapped memory and interrupts. These cards are available as device files on Linux which provide access to the address space of each device. Contrary to FUSE UIO drivers require a small kernel module that deploys access to hardware resources. But the driver logic can be implemented in user space. This gives a number of advantages for both the programmer and the user. Instead of kernel modules these drivers can use many tools and libraries that are not available in the kernel. Also bugs and crashes of the driver do not influence the kernel itself. While updates of kernel modules often require the recompilation of the kernel or at least a reboot of the system updates of these drivers can be done more easily. [3]

1.3 Rump kernels

A rump kernel provides a platform to use component based drivers. It presents an instance of a set of components that runs in the user space. Drivers on top of a rump kernel only need to interface with it and not the underlying kernel. That makes this concept portable to highly different platforms such as monolithic or micro kernels. This is achieved by stripping away all unneeded components of a kernel and only leaving those required for drivers to work. At the lower part of the kernel exists a small and well defined portability layer to interface with the underlying environment. These concepts were implemented first by the NetBSD project. [4][5][6]

Rump applications and drivers are created using the so called Hypercall API. There are two main parts of this API. The first one are basic calls for applications such as those for allocating memory or using threads. The second is about handling I/O operations and is splitted into a virtual and a physical part. The virtualized calls access services provided by the host operating system such as network interfaces and protocols. For hardware drivers the physical hypercalls are more interesting. While those are referenced in the platforms description where also bare hardware is mentioned as a supported platform there was no reference to the actual hypercall API for physical operations. [7] [8]

2 Requirements

3 Design

3.1 Integration into Genode

The integration into Genode is done by creating a base directory. This is used to build a core binary which then is executed by the kernel. Since there is already a core that is able to use the Linux kernel it would be redundant to rewrite these parts. Reusing the already available code can be done in three different ways.

3.1.1 Copy base-linux

The simplest way to build upon base-linux is creating a copy it in a new base directory. The advantage of this approach lies in the fact that there is, beside including the new directory, no need to modify upstream code. On the other hand this leads to code duplication and any updates of base-linux probably need to be ported to its modified copy.

3.1.2 Include base-linux

Creating a new base directory and including the needed contents from base-linux via Makefile fragments. While this would include the advantages of the copy based approach there is less to no code duplication. The disadvantage here is the complexity due to the build system that does not provide a standard way to include parts of other base directories.

3.1.3 Extend base-linux

To prevent both code duplication and complex includes the already existing base directory could be used and extended. While this approach seems to be easier it has two caveats. Since it requires modifications of the original base-linux core the current functionality needs to be maintained. This can be done either by creating a hybrid core that is able to run like the current one and on a bare Linux kernel or a switch needs to be added that chooses the target on build time. Additionally this code needs

to be brought upstream as it cannot be plugged into the source tree like a separate base directory.

4 Implementation

5 Evaluation

6 Conclusion

Bibliography

[1] Norman Feske. Genode Operating System Framework 17.05 Foundations. Technical report, May 2017.

[2] libfuse/libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. `https://github.com/libfuse/libfuse`. Accessed: 05.11.2017.

[3] Hans-Jürgen Koch and Linutronix. The Userspace I/O HOWTO. `https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html`, December 2006. Accessed: 05.11.2017.

[4] rumpkernel. `https://wiki.netbsd.org/rumpkernel/`. Accessed: 09.11.2017.

[5] Interview: Antti Kantee: The Anykernel and Rump Kernels. `https://archive.fosdem.org/2013/interviews/2013-antii-kantee/`. Accessed: 09.11.2017.

[6] Antti Kantee. The Design and Implementation of Anykernel and Rump kernels. Technical report, August 2016.

[7] rumpuser - NetBSD Manual Pages. `http://netbsd.gw.com/cgi-bin/man-cgi?rumpuser++NetBSD-current`, . Accessed: 07.12.2017.

[8] Platforms - rumpkernel/wiki. `https://github.com/rumpkernel/wiki/wiki/Platforms`, . Accessed: 07.12.2017.