Threat mitigation on Genode

Genode Labs

October 5, 2016

Contents

Genode is an OS technology designated for application areas where high security and robustness are mandated. Being a component-based system, it applies strict privilege separation in a holistic way to the entire software stack, from low-level OS services like memory management, over device drivers and protocol stacks, to applications and application plugins. It thereby addresses security in a way that is fundamentally different from conventional operating systems, which generally consider security as an afterthought rather than a premise. The current state-of-the-art of securing commodity OSes is the application of an arsenal of threat-mitigation techniques combined with the timely installation of security fixes.

This document contrasts Genode's approach of structural resilience with the threat-mitigation practices employed by today's mainstream OSes. It furthermore discusses the costs and presumed benefits of incorporating those techniques into the Genode OS framework.

1 Introduction

Producers of security equipment generally prefer static to dynamic systems because the former are much easier to evaluate. However, users increasingly demand flexible, scalable, and feature-rich products that are inherently dynamic. With virtualization, there exists a middle ground where a highly complex dynamic system running in a virtual machine is embedded in a static system. However, this approach does not scale well enough. Dynamic functionalities are always exposed to the complex guest OS, which limits the implementation of applications that are both flexible and secure. If such applications are demanded, the use of commodity operating systems (predominately based on GNU/Linux) seems to be inevitable. In contrast to rigid static systems, however, a realistic assessment of the security implications of using a commodity OS is impossible. Security apparently has to be traded against flexibility.

Genode[1] is an OS architecture that is designed from the ground up to align high security with dynamic application workloads, ultimately resolving the conflict between security and flexibility. Genode's architecture is explained in detail in the book "Genode Foundations", which can be downloaded at the Genode website:

Genode OS Framework

https://genode.org

For assessing the viability of Genode for next-generation security products, experts who are deeply familiar with commodity OS technology like Unix-based systems tend to struggle with getting a tangible feeling for the risks and benefits of an unfamiliar technology like Genode. Intuitively, it makes perfect sense to compare the state-of-the-art security features as found on familiar OSes with the situation on Genode. Because Genode's philosophy deviates from the threat-mitigation mindset that is behind most of the security features of commodity systems, however, if actually lacks security features that we take for granted in modern OSes. Genode disappoints in this discipline and may be rejected. This document addresses those concerns by discussing state-of-the-art mitigation techniques in the context of Genode.

1.1 Motivation behind mitigation techniques

Figure 1 illustrates a fairly mundane application scenario of using an email client like Mozilla Thunderbird on a GNU/Linux system. The Linux kernel is a highly complex program that contains all device drivers (e. g., network card, graphics, disk, USB) and protocol stacks (e. g., TCP/IP, various file systems) that are needed to accommodate feature-rich applications on top. Whereas the kernel is concerned with "low-level" details, the email client is concerned with "application-level" features. Besides providing user-facing functionality, the application contains highly complex protocol implementations:

[1] When talking about Genode within this document, we refer to Genode used with a microkernel like NOVA, seL4, or Genode's custom base-hw kernel. Genode also happens to run on the Linux kernel but the argumentation given herein does not apply to this version.

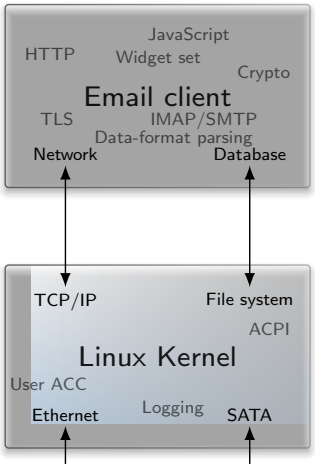

Figure 1: A feature-rich application running on GNU/Linux.

- It speaks protocols like IMAP and SMTP for talking to email servers. The network communication is performed via a socket interface provided by the kernel's TCP/IP stack.
- It takes care about transport-level security (TLS), e. g., based on OpenSSL. This, in turn, involves the execution of cryptographic functions.
- It manages email-account settings including the local storage of the login credentials needed to access the user's account on the email server.
- It interprets the email format, including the handling of attachments.
- It displays emails of various formats and encodings. In fact, to show HTML emails, it contains a web rendering engine that includes the ability to execute Javascript code.
- To show images contained in an email, it decodes various image formats like JPEG, PNG, GIF, etc.
- It manages all emails that are stored in local folders by accessing the file system provided by the kernel. This includes full-text indexing of the content of all emails.

- It provides a sophisticated user interface for navigating, viewing, and composing emails.
- If using GPG (via the Enigmail plugin), it requests the passphrase for the user's private keys.
- It interacts with the desktop environment to support copy-and-paste and drag-and-drop.
- To allow the user to add attachments to emails, it is able to access all of the user's files.

What could go wrong? The answer must consider two aspects: The likelihood for an attack (or bug), and the potential damage.

The **likelihood for an attack** depends on the attack surface of the application. Both the kernel and the email client are **outward-facing** and - with a code size in the order of millions of lines - **hugely complex**. Outward-facing means that a potential attacker is able to directly interact with them. For example, an attacker may

- Send manipulated network packets to the machine to make the kernel stumble,
- Insert a manipulated USB stick into the machine triggering a kernel bug while parsing file-system structures,
- Send an email with a manipulated JPEG image that triggers a buffer overflow in the email client,
- Manipulate the email client from another compromised program (web browser, PDF viewer) via ptrace, the X protocol, or desktop-integration protocols.

Because of the high complexity, vulnerabilities are abound. Because of the outward-facing nature of both programs, those vulnerabilities are exposed to attackers.

The **consequences of an attack** - if launched successfully - are so far-reaching that an assessment becomes impossible. An attacker that compromised the email client has gained the authority to:

- Access all of the user's emails and data stored on disk. This data can be sent over the network,
- Store files on disk, e. g., installing a permanent backdoor as an automatically started program of the user's desktop session,
- Install key loggers or take screenshots,
- Further escalate its privileges by attacking the machine's kernel or other machines from the compromised user application,
- Exploit computing resources like CPU time (crypto-currency mining) or network bandwidth (spam botnets).

In the light of this given architecture, the application of state-of-the-art **threat-mitigation** measures is **indispensable**. There is no doubt that features like address-space randomization, data-execution prevention, or stack-overflow protection help to hamper the outcome of attacks.

Still, most state-of-the-art mitigation techniques address symptoms instead of the root cause of the problem, which is the missing separation of duties (and privileges) in today's software stacks. Mitigation techniques are crafted with the same line of thinking as Kevin in the movie "Kevin home alone". He knows that the burglars are able to enter his house. There are too many vulnerabilities like unsecured windows, rusty locks, or weak cellar doors. To defend the house, Kevin spills marbles on the floor, installs trip wires, and constructs all kinds of funny traps within the house.

1.2 Compartments and structural resilience

Genode addresses security from a different angle. Instead of relying on trip wires and traps, and following "best practices" like putting cash at random places instead of the kitchen table, Genode focuses on

- Structural integrity of the building,
- Strong walls between rooms within the building,
- Minimizing of outside-facing doors and windows that may be misused to illegitimately enter the building (attack surface from the outside),
- Locked doors between rooms that cannot be circumvented without authorization,
- Storing treasures in vaults.

In Genode, the main purpose of the kernel is the creation of isolated compartments and the provisioning of controlled and explicitly authorized interactions between compartments. The less complex the kernel, the smaller is the chance of cracks in the walls between the compartments. With a microkernel of less than 15K lines of code, there is a realistic chance that the kernel is completely free from vulnerabilities. This claim is supported by the existence of seL4, which is a microkernel that is formally proven to have no bugs. The Genode framework on top of the kernel extends the microkernel-construction principles to the user level. E.g., the structural resilience of Genode is reinforced by the following considerations:

- The microkernel is not extensible by loadable kernel modules.
- The microkernel interface (that is exposed to all user-level components) is extremely rigid. E.g., the microkernel never touches any user memory and never interprets user-space-provided pointers. The user land can reference microkernel objects solely via "capabilities".

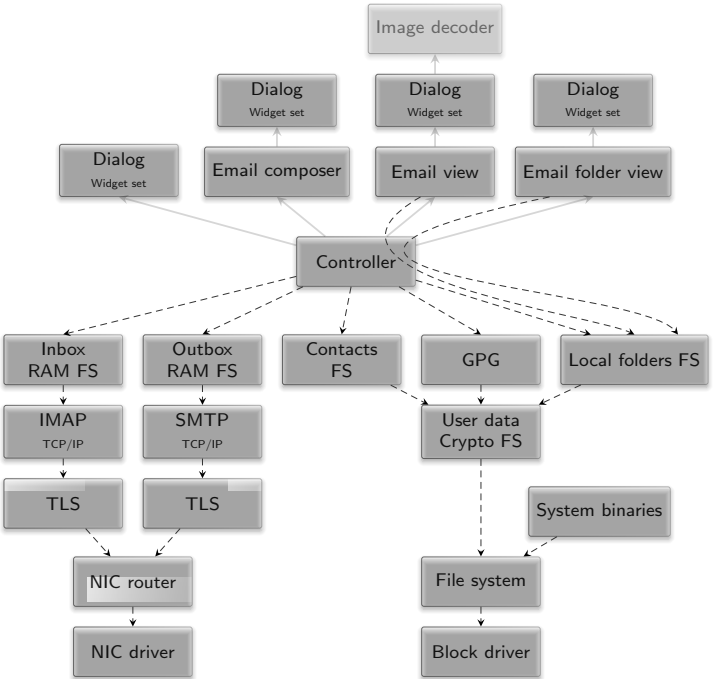

Figure 2: An email client as a multi-component application on Genode

- Code that is relied on by critical components is as low complex as possible, both in terms of the amount of code and the simplicity of data structures. This equally applies to the microkernel and critical user-level components.
- Even though Genode components are written in C++, they do not depend of a C runtime or C++ standard library. Hence, they do not inherit the questionable security legacies of POSIX such as format strings. The Genode API leverages the C++ type system to eliminate several classes of programming bugs at compile time.
- Components follow a functional programming style that largely avoids the reliance on side effects like global variables or anonymous memory allocations.
- There is no over-provisioning of physical resources. The consumption of physical memory is always accounted to user-level components.

Given the Genode architecture, systems and applications can be designed with the inherent assumption that **complex code will fail**. Complex code can still be leveraged as long as it is not responsible to uphold fundamental security properties. Figure [**?**] illustrates a canonical example of a Genode-based multi-component application.

- Instead of using a global file system, different parts of the application have completely different views. E.g., the inbox resides in a dedicated file system instance

that is completely decoupled from the backing store used for the local-folders database. Most components do not see any files at all.

- Each component has a dedicated and narrow purpose. E.g., the IMAP component fetches emails from a server to populate the inbox file system. It has no access to the local email folders.
- No component has access to both the network and persistent storage.
- Data that originates from an untrusted source (like a JPEG image contained in an email) is decoded in short-living sandboxes. The decoder is presented with the raw JPEG data and a way to report the resulting pixels but lives in an otherwise empty environment.
- TLS is realized by a bump-in-the-wire component. This eliminates the risk that the IMAP component accidentally sends unencrypted information to the outside world.
- The highly complex GUI widget rendering is decoupled from the low-complexity application logic.
- Different parts of the application (IMAP and SMTP) use dedicated TCP/IP stacks. Hence, the TCP/IP protocol implementation is not a single point of failure or prone to unintended information flows.
- Importing an email from the inbox to the local-folders database follows a formal procedure that can be triggered by the user only.
- The browser of the local folders has read-only access to the local-folders database.
- The central controller component that defines the interplay and information flow between the user-facing, the network-facing, and the storage-related components is relieved from the complexities of protocol handling, user-interface rendering, and file-format decoding. It can thereby be implemented at a very low complexity.

In contrast to the original monolithic design, the multi-component design allows for a realistic risk assessment. For any complex component, one may ask: What would happen in the event the component is compromised? For example, the outward-facing and complex IMAP component may be prone to attacks. But even in the worst case - being completely in the hands of the attacker - it cannot be exploited to leak any emails from the local folders. Because it has no access to any persistent storage where executable programs reside, it cannot permanently manifest itself in the system. Because IMAP is unrelated to the GUI, the attacker would not be able to log keys or take screenshots.

1.3 Pros and cons

Given the multi-component email scenario, it becomes clear that the provisioning of state-of-the-art mitigation techniques would not bring a substantial improvement because the complex code that would presumably be protected by the mitigation techniques is expected to fail anyway. That said, there are two very good arguments to equip Genode with state-of-the-art mitigation techniques nevertheless:

Real-world applications look different It is not realistic to redesign all software as multi-component applications. In fact, the scenario depicted in Figure 2 actually does not exist (yet).

Current-generation monolithic applications must be accommodated by Genode. For example, the TOR component as added by Genode 16.08 has the form of a single monolithic component that contains a staggering amount of 400K lines of code. It goes without saying that this complexity makes the component highly vulnerable. Since it communicates over the network it is directly exposed to potential attackers. Because it is unrealistic to redesign the component as a multi-component subsystem in the short term, Genode's structural resilience is unable to protect the critical security functions of the TOR component. Applying state-of-the-art mitigation techniques would be the best we can do.

Multiple lines of defense Even though we are confident in the implementation of the microkernel and Genode's base system, we know that bugs remain. Even if the entire code underwent formal verification, there is still the chance for hardware bugs. Uncertainties remain. Mitigation techniques would come into effect as a remedy in such situations.

Wouldn't it be reasonable to apply all state-of-the-art mitigation techniques to Genode then? Yes, but we have to consider that the addition of "security features" bears the risk of weakening Genode's structural resilience by making the foundation of the system more complex and less deterministic. Therefore an even-handed selection of techniques is desired.

1.4 Document structure

The remainder of the document is structured as follows. The landscape of threats and mitigations is extremely confusing and full of jargon. Section 2 tries to frame the discussion by classifying the different kinds of attacks. Section 3 reviews the mitigation techniques employed in today's modern operating systems and discuss how they relate to Genode. Section 4 takes a look at recently disclosed vulnerabilities of the Xen hypervisor and the Linux kernel. The section is meant to create a tangible feeling for the most pressing problems, motivating specific mitigation measures. Section 5 closes the document with a rough plan for improving Genode in the future.

2 Taxonomy of attacks

There exists a huge arsenal of threat mitigation techniques. Before we review the diverse approaches in Section 3, this section examines the threats in a systematic way. We differentiate the attacks into the categories/phases denial of service (Section 2.1), information gathering (Section 2.2), privilege escalation (Section 2.3), and exploitation (Section 2.4). Boot attacks are not covered.

2.1 Denial of service

A denial-of-service attack is the lowest hanging fruit for attackers. In fact, most vulnerabilities (as the ones studied in Section 4) bear the potential to halt the system via a kernel crash due to memory-safety issues, a failed assertion, a deadlock, division by zero, or similar bugs. Other forms of denial-of-service attacks drain the physical resources of the system to a point where it becomes unusable. E.g., a memory leak in the kernel may consume all physical memory, triggering the kernel out-of-memory (OOM) handling, a bug in a device driver may result in the generation of interrupts at a high rate (interrupt storm) keeping the kernel busy, or a computation-intensive kernel operation (like a cryptographic function) parameterized by an attacker may dominate the overall system performance.

The opportunities for such attacks are so manifold that resistance against denial-of-service problems is universally regarded as impossible on current-generation general-purpose OSes.

Genode is ultimately designed to withstand such problems. Thanks to its component-based architecture, resource leaks or crashes are constrained by component boundaries. In principle, a malfunctioning component can be restarted. In contrast to traditional OSes, Genode accounts the use of physical resources to components and provides means to trade those resources between components. There is no over-provisioning of resources that would put pressure on the OS kernel. That said, the current implementation is incomplete in this respect.

Known resource-exhaustion issues There are a few known resource-exhaustion denial-of-service attack vectors in Genode. In particular, capabilities are a system-global limited resource because a capability is a kernel object. The limit stems from the microkernel design or the dimensioning of the capability namespace of Genode's core component. Consequently, the allocation of capabilities should be regimented similarly to physical memory allocations. In contrast to physical memory where Genode employs its resource-trading scheme, this scheme remains unused for allocating capabilities. Another instance of the risk for resource exhaustion is the kernel memory of the NOVA kernel, which has a static limit. Genode's custom base-hw does not have this limitation.

Constraints of restarting components The ability to restart components is currently limited to components that do not provide a service to other components because

a disappearing server would affect clients that depend on it. Because of this constraint, restartable components must either modelled as RPC clients or interposed by a failsafe-guarding wrapper component. E.g., To make a network driver restartable there are two approaches:

- The bug-prone network driver can be executed as a child component of a robust NIC failsafe component. This idea is discussed at https://github.com/genodelabs/genode/issues/1592.
- Alternatively, network drivers could be designed as clients of a trusted (and non-restartable) NIC router instead of a NIC service as is the case today.

As of today none of these two options are realized.

2.2 Information gathering

As described very well by the following article, attacks are often pursued in multiple stages.

Bypassing PaX ASLR protection

http://phrack.org/issues/59/9.html

In a first stage, the attacker learns about the attacked system. For example, in order to circumvent address-space randomization, the attacker tries to get hold of the virtual-memory layout of the attacked program. Only after enough information is gathered, the attacker launches the next stage of the attack, e. g., via a chained return-oriented program (ROP) (Section 2.3). Consequently the first line of defense against privilege-escalation attacks is preventing the attacker to gather information. Typical attack vectors for the information-gathering phase are format-string attacks, kernel-information leakage (if the attacker is local), and error handling.

Besides leveraging information leaks to aid privilege-escalation attacks, an attacker may solely be after the information processed on the system. Cold boot attacks presume an attacker model where the attacker is able to gain access over the entire physical memory content in order to steal information

2.2.1 Format-string attacks

Format-string attacks can be launched against programs that can be tricked to process input supplied by the attacker as a format string, for example, if a part of the attacker-controlled input is printed via `printf`.

```
char const *id = get_request_id(...);
...
printf(id);
```

The program may expect `id` to be a number. But since `id` is controlled by the attacker, it could contain a string like "%x %x %x %x", which is a format string that outputs four hexadecimal numbers. The `printf` function is called without any further arguments. By parsing the format string, it expects four arguments on the stack. However, the stack positions where those arguments are expected actually belong to a different stack frame (because the caller of `printf` has pushed only one instead of four arguments to the stack). Consequently, the attacker tricks the program to print a backtrace, which includes interesting return addresses. If enough stack content is gathered, the attacker becomes able to determine the virtual addresses of known-to-be used library symbols such as the libc startup code. This information, in turn, is extremely valuable to craft an ROP attack. The correct way to make the program resilient against this attack is calling printf via a fixed format string:

```
printf("%s", id);
```

But because this requires awareness and discipline of each programmer, format-string vulnerabilities easily slip into programs.

Genode originally relied on the use of format strings. However with the introduction of the current text-output API, the former use cases of format strings are now covered by a type-safe C++ API so that format strings will eventually be removed from the Genode API. This way, native Genode components are immune against this form of information leakage. Still, higher-level components, in particular third-party software that relies on a C library remains vulnerable.

Work of removing format strings from the entire Genode code base is under way.

2.2.2 Information-leaking error messages

Error messages are often crafted with the intention to assist the developer with analysing the situation. The more information the developer gets about the circumstances of error conditions, the better. Unfortunately, to an attacker that can deliberately trigger error conditions, diagnostic messages are a welcome leak of program-internal or system-internal information. Examples of this problem exist on all levels of the software stack. For example, the Linux kernel prints a backtrace to the kernel log when a kernel thread crashes. The backtrace contains virtual addresses of kernel symbols. So if an attacker is able to trigger a kernel bug and observe the kernel log, this information can be exploited for a local privilege escalation. Another example is a web server that forwards the error messages from a database to the browser. The errors may be deliberately triggered by an attacker that issues malformed queries and reveal implementation details of the web server and the database.

2.2.3 Leaky parameter structures

The most prominent reason for the leak of kernel-internal information to the user space are incompletely initialized parameter structures passed between the kernel and the

user space, in particular when the user land interacts with the kernel via `ioctl` operations.

Structure holes and information leaks

https://lwn.net/Articles/417989/

In a situation where an `ioctl` operation returns information (e.g,, the properties of a device), the returned information is typically assembled in the form of a `struct` on the kernel stack, and then copied to the user land. If a device driver accidentally leaves a member of the struct uninitialized, the information previously stored on the member's stack position won't be overwritten. The stale information is eventually copied to the user land. This information may contain any kind of kernel-internal information - depending on the code that was previously executed via the kernel stack. This may include hints about the virtual address-space layout of the kernel (useful for circumventing kernel ASLR) or cryptographic credentials.

Such leaks may even occur if the driver explicitly initializes each single member of the parameter structure. In the presence of members of different sizes, the compiler may insert padding between the members to meet alignment constraints of the individual members. E.g., the following structure on a x86_64 machine will have a gap of 6 bytes between the 2-byte `device_id` member and the 8-byte-aligned `status_bits` member.

```
struct status {
  short device_id;
  long  status_bits;
};
```

Even when initializing both members, the gap between the members remains uninitialized. To avoid this situations, it is a good practice to clear the entire parameter structure via `memset`. However, since this practice depends on the discipline of each programmer, the problem persists.

Possible information leaks between Genode components Currently, Genode is prone to the same principle information-leak problem when structured data is passed between components, e. g., as arguments or return values of RPC calls.

For example, Genode's `Framebuffer::Session::mode` RPC function returns a POD (plain-old-data) object of type `Mode`. In this case, there is no padding between members but other RPC functions may be prone to leakage.

2.2.4 Cold-boot attacks

Instead of targeting an attack on the software running on a machine, an attacker may gain access to the machine's physical memory by a mechanism below the operating system. For example:

- An attacker with physical access to the machine may reset the machine, boot a minimal-complexity custom-made OS from a USB stick, which reads the physical memory content.
- A compromised base-band processor of a mobile-phone chip set may access the entirety of the application-processors physical memory.
- The firmware running on the Intel management engine (ME) co-processor may access the physical memory bus directly without being subjected to an IOMMU.
- A bus-level attack may origin from a custom device inserted into a laptop's PC-card slot. Such a device would be able to issue DMA transactions, accessing the physical memory unless an IOMMU prevents such accesses.

Given such a strong attacker model, the best software running on the machine can do to mitigate such attacks is leaving as little plain-text information in memory as possible. Authors of crypto libraries and certain security-sensitive applications pay attention to these attacker models by zeroing cryptographic material when no longer in use. However, commodity OSes don't attempt to mitigate such attacks on the operating system level.

Diminishing cold-boot attacks on Genode Currently, Genode does not consider cold-boot attacks. However, the following two improvements would greatly relieve the situation:

- Genode maintains the invariant that freshly allocated RAM dataspaces are zero-initialized. Today, dataspaces are cleared at the time of their allocation. However, after freeing a dataspace, the underlying physical memory retains stale information until it is allocated for a new dataspace. To anticipate the cold-boot attacker model, dataspaces could instead be cleared when freed.

  As a prerequisite to implement this idea, all physical memory must be cleared at boot time. Otherwise a dataspace allocated directly after booting would remain uncleared. Depending on the amount of memory installed in the machine, clearing the entire memory, however, delays the boot by several seconds. This problem could be solved by a change of Genode's way of clearing memory. In the current implementation, Genode's core component clears dataspaces "in band" with the dataspace allocation. Alternatively, a low-priority thread could perform memory purging out of band and maintain a second allocator structure that distinguishes dirty physical memory from cleared physical memory. Dataspace would exclusively be allocated from the clean physical memory pool. When freed, the backing store is handed over to the dirty memory pool. A dedicated thread would successively move memory from the dirty memory pool to the clean memory pool while erasing the memory. The thread would usually run at the lowest priority. If the clean memory pool comes under pressure by a dataspace allocation, the thread's

priority could be boosted to the one of the client that issued the dataspace allocation.

- Currently, Genode's heap does not clear memory blocks, which is consistent with the behavior of traditional heaps. However, in line with the dataspace handling described above, the heap could erase memory blocks upon release. When the heap allocates its backing store in the form of RAM dataspaces, the backing store is known to be clean. When freeing a block, the heap would zero the memory unless it is explicitly configured to skip the clearing phase, e. g., for performance reasons.

  For Genode's slab allocators, this line of argument does not apply because slabs are always used as a performance optimization. So the clearing is never anticipated.

2.3 Privilege escalation

Most attacks are motivated by the goal to expand the attacker's privileges in one form or another.

- Access information that normally not available to the attacker, e. g., login credentials, documents, blackmail material, cryptographic keys,
- Control a system that is normally beyond the attacker's control, e. g., sabotaging machinery,
- Use foreign computing resources, e. g., CPU time for crypto-currency mining or network bandwidth for spamming or launching distributed denial-of-service attacks.

We distinguish two different privilege escalation attempts.

Leveraging ambient authority

An attacker wants to enter a locked building but does not posses the key. However, he knows someone who has legitimate access to the building. So the attacker tricks this person to unlock the door for him. The attacker does not need steal the key. The tricked person is not harmed. But the attacker nevertheless manages to illegitimately enter the building.

In commodity operating systems, the tricked persons are system daemons that offer services to unprivileged users but that need special privileges to fulfill those services. For example, a user has no direct access to talk to the printer. In order to print a document, the user asks the printer daemon to talk to the printer on the user's behalf. The printer daemon has the privileges required to talk to the printer. But these privileges also enable the daemon to do many other things that an unprivileged user could not do. By supplying creative parameters to the daemon, the attacker tricks the daemon into applying its privileges in ways that are not intended by its designated role.

Exploiting vulnerabilities

To enter a locked building, the attacker would try to find a weakness like an unsecured window, a rusty lock, or an unlocked flue. To get into the building, a window may be shattered or a lock picked. Sometimes, the whole house may blow up in the process. The starting point of an attack is the gathering of information to identify weaknesses in the building's structural integrity. Once a weakness is identified, it serves as a starting point for getting into the building. Once in the building, the attacker cannot freely move yet. In order to go forward, the environment needs to be manipulated in some way, e. g., by switching on the lights to look around or disarming an alarm system. For an attacker it might be beneficial to cover the tracks. Once in the building, it becomes easy to weaken the building's structural integrity, e. g., by leaving a window ajar. This weakness will greatly ease the future access to the building.

The weaknesses of the structural integrity of an operating system is called "vulnerability", which is a mundane programming error such a missing bounds check. In today's commodity OSes and application stacks, the existence of abundant vulnerabilities is a given. In most cases, a vulnerability is not directly exploitable but merely serves as a starting point of a chain of attacks that build upon each other. At some stage, the attacker needs to import custom code into the attacked system that acts in the interest of the attacker. Attacks may have side effects like crashing processes, anomalies of resource consumption, or network traffic. To cover up those side effects, a part of the attack may be concerned with manipulating logs or monitoring systems. For installing a permanent back door, access to persistent storage and a way to manipulate the system's boot procedure are extremely valuable.

Ambient authority problems are best countered by a fine-grained separation of concerns. E.g., if the printer daemon has access to the printer but no other part of system, the damage it can cause when tricked by an attacker is limited to the printer. Mandatory access control frameworks like SELinux replace the notion of an almighty root user by fine-grained policies and thereby reduce the risks. The majority of threat-mitigation techniques discussed in this document, however, address the second category: The exploitation of vulnerabilities. To understand the various mitigation approaches and their limitations, we first discuss how such attacks work in principle.

2.3.1 Seizing control over a foreign program

At this stage, the immediate goal of the attacker is to cause the attacked program to leave its regular path of execution. This change of behavior is induced by supplying attacker-controlled input to the program that triggers a vulnerability. Typical vulnerabilities are:

Integer-overflow Certain input arguments (like the day of the month) are expected to intuitively lie in a range of values. The programmer does not expect an input

value that is deliberately chosen by the attacker to lie at nearby the limit of an integer type. An innocent looking conditional branch may reach a wrong conclusion.

```
if (day_of_month + days_per_week > days_of_this_month) {...
  // day is in the last week of the month
}
```

In this example, if the attacker supplied the value INT_MAX as `day_of_month` argument, the addition would trigger an integer overflow resulting in a low value (6), which satisfies the condition. If the subsequent code further uses the value of `day_of_month` (e. g., for dimensioning a buffer, or copying data), the program is vulnerable.

Missing bounds checks Low-level languages like C and C++ perform no bounds checks when accessing array elements. The responsibility lies with the programmer and is easy to miss. Classical examples are POSIX functions like `strcpy`, which do not even take an upper bound as argument. As another typical example, the interplay of unsigned and signed integer values is bug-prone:

```
if (i < upper_bound) ...
```

If `i` is a signed integer that originates from the attacker, the attacker may supply the value 0xffffffff (on 32-bit platforms). The check will then effectively compare `upper_bound` against `-1` and wrongly accept the condition.

Double fetch A program repeatedly obtains its parameters from an attacker-controlled buffer. The program expects that the buffer's content remains constant whereby the attacker deliberately changes the buffer values. E.g., if the program first performs a successful bounds check and then reads the value later on, the attacker may have replaced the checked value with a value that lies out of bounds, eventually causing a buffer-write operation outside the buffer boundaries. This specific problem is commonly referred to as "time-of-check-time-of-use" but the underlying double-fetch bugs can occur in various other forms.

Use after free A dynamically allocated memory block is freed but there still exist stale (dangling) pointers that refer to the former content of the memory block. If the attacker manages to trigger an allocation of a new block that uses the same backing store as the freed block, and to supply the content of the freshly allocated block, the dangling pointer will refer to attacker-controlled memory content. If the attacker is able to trick the attacked program into de-referencing such a pointer, the attacked program will operate on attacker-controlled data structures.

Refcount overflows If the attacked program uses reference counters to track the lifetime of objects and the attacker finds a way to increment the number of references (e. g., by repeatedly opening the same file without closing it), the reference counter may overflow to the value of zero. This may trigger an unintended destruction of the underlying object (producing dangling pointers), or the unintended double-allocation of the object's backing store (if the ref counter is used for the allocation of objects).

Section 4.1 reviews the most prominent vulnerabilities in more detail. An attacker uses any of those vulnerabilities to manipulate the internal state of the program, more specifically pointer values that are de-referenced during the subsequent execution of the program. Eventually, **the attacker needs to manipulate jump targets** to steer the control flow of the program away from its regular path. Jump targets are values in the program's memory that end up being loaded into the instruction-pointer register of the CPU. The following techniques are common for manipulating jump targets:

Return addresses (stack smashing) Each time a function is called, the caller's instruction pointer ("return address") is written to the program's stack. E.g., on x86, the saving of the return address is implicitly done by the `call` instruction. Once the function returns, the `ret` instruction conversely loads the instruction pointer with the return address as fetched from the stack. If the attacker manages to manipulate the stack position where the return address of the currently called function is stored, the `ret` instruction will load the instruction pointer with the value supplied by the attacher.

Local pointer overwrites If the attacked program stores a function pointer adjacent to a buffer that is prone to an overflow, the overflowing buffer may overwrite the pointer value. Later, when the pointer gets de-referenced the program resumes execution at an attacker-controlled instruction pointer. Note that function pointers may have the form of C++ vtables, e. g., if both an overflowing buffer and a C++ object with virtual functions appear as local function variables, the function may be vulnerable.

Heap overflows If a buffer that is stored on a dynamically allocated memory block is prone to overflow, memory adjacent to the memory block can be corrupted. This memory may contain pointers to the heap's meta data, which are eventually de-referenced by the heap (i. e., when freeing a block). Therefore, an attacker who can predict the locations of allocated memory blocks can deliberately manipulate the heap's metadata, tricking the heap into calling an attacker-provided function pointer.

Data-pointer write operations It is not always needed to overwrite a function pointer. Plain data pointers that are de-referenced for writing are almost as valuable for an attacker with a-priory knowledge of the program's virtual memory layout. E.g., one technique to attack Windows OS is to direct data writes into an in-kernel location where page tables are kept. So an attacker could modify page-table entries

directly (bypassing SMAP/SMEP). Another application of data-write operations are PLT pointer manipulations described next.

PLT pointer manipulation The procedure linkage table (PLT) is an array of function pointers that is present in each dynamically linked binary or shared object (ELF object). For each external library function called by the respective ELF object, the table holds a pointer to the corresponding function. The table is not static but filled by the dynamic linker on demand. When loading an ELF object, the dynamic linker initializes each PLT entry with a pointer to so-called jump-slot-relocation function provided by itself. When the ELF object calls a function the first time, it jumps through the PLT entry to the linker's jump-slot-relocation function. This function looks up the pointer to the actually to-be-called function in the loaded ELF objects, registers the function pointer in the corresponding PLT entry, and calls the function. All subsequent calls to the library function will jump through the PLT entry directly to the corresponding library function. Thanks to this "lazy bind" mechanism, the loading of shared objects - even very large ones - requires very little up-front setup costs. The fact that PLT entries are jump targets that lie within legitimately writable memory (because the dynamic linker has to modify them) at a predictable virtual base address (in contrast to stack locations or heap allocations) makes them an valuable target for an attacker that can trick the attacked program to perform an arbitrary write operation.

2.3.2 "Shellcode" injection

Now that the attacker has managed to make the foreign program stumble, it is time to fill the program's dead hull with new life. Otherwise the effort would remain a denial-of-service rather than a valuable privilege-escalation attack. The attacker needs to import custom-crafted executable code into the attacked program, which is then executed in place and equipped with the privileges of the original program.

The classical method to inject attacker-provided code ("shellcode") is to supply the program along with the content of an overflowing buffer. After the buffer overflow, the attacker's code will ultimately end up somewhere in the program's stack or heap area. When padding the area with enough `nop` instructions and with a halfway predictable stack and heap layout, there is a good chance for the attacker to overwrite a jump target with a value that points to the attacker's code.

However, with the introduction of non-executable memory mappings (as emulated by PaX or featured by current-generation CPUs) commonly referred to as data-execution prevention (DEP), heap and stack memory are no longer executable, which led to the advent of the so-called return-oriented programming (ROP) technique described in the next section.

In practice, the implementation of DEP is not all-encompassing. I.e., on account of backwards compatibility, the Windows kernel maintains an internal pool of executable memory. An attacker may leverage the fact that legacy drivers allocate kernel objects from this pool. If the attacker is able to define the memory content of such a kernel

object (e. g., a GDI color palette), it is possible to misuse the kernel object as a carrier of to-be-injected code. A further discussion of this topic can be found in Section 3.6.2.

2.3.3 Return-oriented programming (ROP)

Because of DEP, "shellcode" can no longer be injected into the attacked program. Instead of injecting new code into the attacked program, a ROP attack executes snippets of code that are already present in the program's address space, e. g., in the form of the executable binary or shared libraries.

In general, the attack presumes that the attacker has access to executable binaries of the program and the loaded shared libraries. This is realistic in scenarios where those programs are installed from binary packages as on most GNU/Linux distributions or proprietary software. The attacker scans the binaries for so-called "gadgets" - little snippets of a few instructions that are followed by a `ret` instruction. What the CPU instruction set is for a regular assembly programmer, the set of gadgets is for the ROP programmer.

A ROP-based shellcode is started by manipulating a jump target to point to a gadget, e. g., via one of the methods described in Section 2.3.1. Once the gadget returns (that is, the CPU executes the `ret` instruction), the CPU loads the next instruction pointer from the stack (which is normally the origin of a function call). By identifying gadgets that manipulate the stack pointer, the attacker is able precisely steer the execution towards different positions on the stack where pointers to further gadgets are placed. This way it is possible to execute gadgets in a row - a so-called chained ROP attack. Attackers may employ sophisticated tooling that craft chained return-oriented programs. Similar to how a regular compiler creates sequences of CPU instructions, such tools create ROP chains.

An excellent explanation of return-to-PLT exploits is given by the following article:

The advanced return-into-lib(c) exploits - PaX case study

http://phrack.org/issues/58/4.html

2.3.4 Stack Pivoting

Because ROP chains are essentially a batch of return addresses on the stack, the size of ROP programs is naturally bounded by the stack size. A common technique to overcome this limitation is the so-called stack pivoting, which is described in the following article:

Emerging 'Stack Pivoting' Exploits Bypass Common Security

https://blogs.mcafee.com/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security/

The attacker uses ROP to change the stack pointer to an attacker-controlled buffer. A possible gadget to achieve the stack pointer modification may look like (example taken from the blog entry above):

```
push eax
pop esp
pop ecx
movzx eax, ax
retn
```

The prepared stack starts a sequence of more ROP gadgets. This way, extremely complex ROP attacks can be launched.

Windows 8 introduced additional assertions regarding the stack pointer into sensitive functions to counter stack-pivoting attacks, i. e., those functions that manipulate the virtual address-space layout. Unfortunately, those checks can be circumvented as described in the following article:

Defeating Windows 8 ROP Mitigation

http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/

2.4 Exploitation

Once the attacker managed to hijack a foreign program, the attacker's code gains the authority of the subverted program. In the case the target is the kernel, the attacker gains ultimate authority over the entire system. But even if the attacker merely gains the authority of a plain user program, the opportunities for exploitation are abound. For the actual exploitation phase, the attacker does no longer need to jump through hoops like ROP. Instead, easy-to-develop "payload" can be smuggled into the attacked machine, which is then free to leverage the comfort of regular APIs.

Potential motives of such a "payload" are:

Persistent installation of malware This is straight-forward if any regular user program can store data persistently and install autostart entries in the user's desktop environment.

Hooking into the interesting parts of the system The payload may hook into any part of the system like a regular user. E.g., for executing a key logger, taking screenshots, browse the user's data stored in disk, capture audio.

Establishing a control channel to the attacker A control-channel allows the attacker to remotely control the machine, update the malware, or use the local malware as a base camp for launching further attacks like a local privilege escalation to gain root privileges, or to explore other machines connected via local network.

Covering the tracks It is the interest of the attacker to remain undetected. Hence, the attacker may try to clear system logs from any suspicious content, or circumvent AV scanning.

Note that current-generation commodity OSes are almost defenseless against the actual exploitation stage. In order to accommodate feature-rich user applications, regular user programs are able to persistently store data, communicate over the network, or interact with the user interface. Whereas the popular threat mitigation techniques are primarily concerned with preventing privilege-escalation attacks, the reach of malware payload - once installed - remains almost without bounds.

Note that the situation looks much different on Genode where the principle of least authority is rigorously applied not only to OS-level services but to user applications. E.g., when opening a PDF document, the PDF viewer can read the PDF data and has a facility to send the pixel data of the rendered document, but it can not store any data nor can it access the network. If a malicious PDF file corrupts the PDF viewer, the attacker would not be able to execute any of the payload functions mentioned above. Instead, the attacker's shellcode finds itself in an environment where no communication with the outside world is possible, with no "system" library function, no `execve` system call, no way to spawn a shell, and not even the notion of a root user.

3 Mitigation techniques

This section discusses threat-mitigation techniques commonly used on GNU/Linux systems. A very good starting point is the compilation of the security features present in Ubuntu Linux.

Security features of Ubuntu Linux

https://wiki.ubuntu.com/Security/Features

In the following sections, we present each technique, discuss how the approach relates to Genode, and - if applicable and beneficial - outline ways of incorporating it into Genode.

3.1 Stack-smashing protection

Stack-smashing protection (SSP) is a compiler-based mitigation feature for detecting buffer overflows that overwrite return addresses on the stack. The basic idea behind SSP is to place a so-called canary value adjacent to each return address stored on the stack. The canary value is an arbitrary value chosen at program start. Because this canary changes each time the program is executed, an attacker cannot easily predict its value. Each time a function is called, the canary value is placed on the stack frame. On function return, the value present on the stack frame is compared to the canary value. The `ret` instruction is executed only if the values are equal. Otherwise, the program aborts. In the event that a buffer overflow occurred in the function body, the canary value is overwritten by the overflowing buffer content before the overflow reaches the return address.

Originally, this idea was implemented by the StackGuard and ProPolicy features of GCC on the x86 architecture. The development culminated in the `-fstack-protector` and `-fstack-protector-all` flags introduced in 2005. In 2012, Google introduced `-fstack-protector-strong` to improve the balance of performance and security. This variant is available since GCC 4.9 and used on Android by default. (Genode's tool-chain is currently based on 4.9.2)

The following article provides a well-written explanation of stack smashing, stack canaries, and GOT/PLT exploitation:

Stack Smashing On A Modern Linux System (2012)

https://www.exploit-db.com/papers/24085/

Another good compilation of practical tool-chain hardening measures is provided by Gentoo's Hardening project:

Introduction to the Gentoo Hardened toolchain

https://wiki.gentoo.org/wiki/Project:Hardened/Toolchain

3.1.1 Function selection

Equipping each function with a canary check would unreasonably impede the performance. For this reason, heuristics built into the compiler select the potentially vulnerable functions where this technique is applied. The heuristics can be chosen via the compiler parameters `-fstack-protector-all` (cover all functions), `-fstack-protector` (cover functions with vulnerable objects, local arrays larger than a certain size, or buffers allocated via `alloca`), and `-fstack-protector-strong` (also covers functions that take addresses of local variables, use local arrays of any size, or use register local variables).

3.1.2 Canary-value protection

If an attacker can guess the canary value of a long-running program, a buffer-overflow attack could overwrite the canary field with the attacker-supplied canary value along with the forged return address. The stack overflow would then remain undetected. To limit this risk, the location of the global variable that stores the canary value can be randomized. Furthermore, the approach of so-called random-xor canaries scramble the canary value with contextual information local to the function. Because the canary is not the same for different function calls, it is harder to predict.

3.1.3 Limitations

Even with -fstack-protector and DEP in place, an attacker may use a buffer overflow to overwrite pointers within the local stack frame. When such a pointer is subsequently de-referenced for a write operation, the attacker can steer the write operation to a deliberately chosen virtual address. A worthwhile target is the program's PLT (Section [Seizing control over a foreign program]), which contains pointers to the library functions used by the ELF object. Instead of overwriting a return address on the stack, the attacker overwrites a PLT entry such that the next time, the PLT-entry's corresponding function is called, the program jumps to the attacker-provided instruction pointer. This attack relies on the attacker-known position of the (writable) PLT.

To mitigate this attack, the PLT as used by the ELF binary could be provided in a read-only mapping whereas the dynamic linker would maintain a writable alias mapping at a randomly chosen position.

3.1.4 Enabling SSP on Genode

The stack-guard mechanism relies on the ABI symbols. `__stack_chk_guard` and `__stack_chk_fail`.

```
uintptr_t __stack_chk_guard;
__attribute__((noreturn)) void __stack_chk_fail(void)
```

In principle, this is very simple to implement. We just need to initialize the guard variable with a random value. However, this initialization must not happen from within C code that is protected by SSP because the initialization function would be called with a different canary than the one used on return time. SPP would wrongly detect this situation as a buffer overflow. However, fortunately Genode's components are initialized in two stages, which use different stacks. The first stage uses an initial stack located in the BSS segment of the binary. This stage initializes the C++ runtime, establishes the access to the component's Genode environment, and sets up the stack used by the actual application-level code. In this stage, the component could obtain a random value from its parent and store it in a variable that is visible to the code that switches the stack to the actual application stack. Since we start the second stage with a fresh stack from this point on, we could safely initialize a new canary value at this point.

Review of the code generated by the compiler Most GNU/Linux on x86 take advantage of thread-local-storage (TLS) to store the canary value. The value is loaded via segment-relative addressing. The `gs` segment refers to a thread-specific memory area (that is re-configured during the context switch to a thread). The use of a field in the TLS area alleviates the need to store the canary value in a global variable. The following code is generated by Ubuntu's compiler (x86_64). It shows the epilogue of a function equipped with SSP.

```
        mov    -0x8(%rbp),%rax
        xor    %fs:0x28,%rax
        je     ok
        callq  __stack_chk_fail
ok:     leaveq
        ret
```

On Genode, this approach cannot be used because most microkernels do not provide a TLS mechanism based on `gs`-relative addressing. However, the Genode tool chain, which is configured to not rely on Linux TLS, generates a simpler version based on a global variable `__stack_chk_guard` (on x86_32):

```
        mov    -0xc(%ebp),%eax
        xor    __stack_chk_guard,%eax
        je     ok
        call   __stack_chk_fail
ok:     leave
        ret
```

On x86_64, it looks like this (apparently the canary value is accessed using IP-relative addressing):

```
        mov     -0x8(%rbp),%rax
        xor     __stack_chk_guard(%rip),%rax
        je      ok
        callq   __stack_chk_fail
ok:     leaveq
        retq
```

On ARM, the code looks as follows:

```
    ldr     r3, [pc, #28] ; __stack_chk_guard_ptr
    ldr     r2, [fp, #-8] ; obtain stack value
    ldr     r3, [r3]      ; obtain canary value
    cmp     r2, r3        ; compare canary value with stack value
    beq     ok
    bl      __stack_chk_fail
ok: sub     sp, fp, #4    ; return from function
    pop     {fp, lr}
    bx      lr

__stack_chk_guard_ptr: .word __stack_chk_guard
```

A pointer to `__stack_chk_guard` is kept nearby the function to obtain it via a PC-relative load instruction.

Consequently, this form of SSP can be enabled for Genode with little effort. The only open question is where the random canary initialization value comes from. It would be nice to place the `__stack_chk_guard` variable at a random position. This could, in principle be achieved by an ELF-binary randomization approach as discussed in Section 3.4.

3.2 Pointer obfuscation

As explained in Section 2.3.1, function pointers are dangerous as they allow an attacker to control the execution flow if overwritten via a vulnerable function. In glibc, many pointers are manually protected against such manipulation by using a macro called `PTR_MANGLE`. The macro XORs the pointer with a runtime-generated "key" value, similar to a stack-canary value. Since the attacker has no information about this value, it cannot redirect the pointer to an attacker-chosen address.

In C++, this technique could in principle be applied via smart pointers. However, modern C++ code rarely uses pointers explicitly. For the most part, pointers are implicitly used to represent references or vtable entries, which cannot be treated that way.

The PointGuard GCC extension "encrypts" pointers automatically. It is available on Windows, but apparently not widely used.

PointGuard Protecting Pointers From Buffer Overflow Vulnerabilities:

https://www.helpnetsecurity.com/dl/articles/pointguard_usenix_security2003.pdf

The idea behind PointGuard is closely related to binary randomization: Even though pointer-manipulation vulnerabilities are presumed to be possible, an attacker is unable to guess an interesting addresses to write to the manipulated pointer. But the two approaches address the problem from the opposite directions.

3.3 Heap-overflow detection / heap protection

As discussed in Section 2.3.1 and supported by the CVEs reviewed in Section 4, heap overflows remain one of the most prominent vulnerabilities. They are not covered by a general mitigation technique like SSP for stack overflows.

In principle, attackers rely on the fact that a buffer located at the programs heap will - when overflowing - corrupt data that is adjacent to the heap block. This data may be meta data of the heap itself (as typical for list-based allocators that store meta data in a header of the user data). But it may also be data structures (presumably containing pointers) relied on by the attacked program.

Unfortunately, there is no easy way to mitigate heap overflows via a general technique (unlike SSP for stack overflows). The best mitigation would be the use of programming languages that are not prone to memory-safety issues. However, since most low-level applications and libraries are written in unsafe languages, the problem is addressed by a variety of band-aids:

3.3.1 Integrity checks of the heap's metadata

Glibc's heap implementation uses pointer obfuscation to reduce the likelihood that an attacker can overwrite a metadata structure with meaningful pointer values. This makes the heap implementation less vulnerable but does not help in situations where other data is overwritten.

3.3.2 Randomization of heap allocations

Heap-overflow attacks depend on predictable allocation patterns. Only when knowing the relative locations of the heap block containing the overflowing buffer and the heap-block storing the to-be-manipulated data structure, the attacker can create the content of the buffer overflow. Heap manipulation makes the layout of the heap block harder to predict. On the other hand, it reduces the efficiency of the heap in several respects:

1. The higher the randomization (the entropy) of the block layout, the more fragmented the memory becomes. As a consequence, small gaps between memory blocks stay unused, which artificially inflates the memory demand of the program.
2. The computation of pseudo-random values consumes CPU time.

3. The amount of meta data required to keep track of the heap layout grows because there is very little likelihood for adjacent free ranges, which would normally be merged into a single range. Consequently, the average metadata overhead may increase approximately by the factor 2.
4. An increased amount of meta data implies a larger search space for allocations.

3.3.3 Guard pages between heap chunks

A heap uses coarse-grained memory objects as backing store. E.g., on Unix, a very large consecutive virtual memory area is populated with an on-demand-paged memory object (e. g., via `mmap` for anonymous memory). Instead of using one consecutive area, the heap may by organized in multiple chunks that are mapped to individual virtual-memory positions. By leaving at least one virtual-memory page unused between chunks, a heap-buffer overflow is constrained to the memory blocks allocated within the same chunk.

3.3.4 Cookies at the bounds of heap blocks

Similar to how stack canaries protect the integrity of their adjacent return address, a cookie value placed at the upper boundary of each heap block can in principle detect a heap-buffer overflow. But unlike SSP where there is a sensible time to check the consistency (just before executing the `ret` instruction of the current function call), there is no natural time when to perform heap integrity checks. E.g., checking the integrity of the cookie when freeing the block would not help to protect long-living blocks.

3.3.5 Protecting the heap on Genode

In the following, we discuss the current role and implementation of Genode's heap along with possible hardening approaches.

Genode base framework Unlike programs running on a commodity operating system where a heap is omnipresent, Genode components are not equipped with a heap by default. For critical components, it is good practice to avoid any form of dynamic memory management. By not using a heap, heap-overflows become a non-issue.

If components require dynamically allocated memory, the Genode API provides a set of mechanisms where the heap is one tool of the tool box. In particular, multiple instances of the heap may be used by different parts of the program (e. g., a service may use a dedicated heap per client). For coarse-grained allocations (e. g., a per-client session structure), the so-called sliced heap places each allocation in a dedicated virtual-memory area and thereby minimizes the potential of side effects on other memory objects. The heap implementation obtains its backing store in multiple independent chunks (dataspaces) that are individually attached to the component's address space. Hence, the heap would implicitly benefit from the randomization of the dataspace-attach operation and the provisioning of guard pages.

All Genode allocators including the heap and sliced heap implement the same `Genode::Allocator` interface. This interface is solely used as an argument to the `new` operator. Malloc-style byte-wise allocations are supported in principle but not used in practice. The `Genode::Heap` implementation keeps the heap's meta data separate from payload data. Hence, a pointer to a block's meta data cannot be calculated from the block's memory address. This is in contrast to traditional list-based heap implementations that intertwine the heap's meta data with the actual user data. There is no detection of accesses that lie outside the bounds of an allocated block. However, since the allocator is solely used in combination with the `new` operator for creating objects of a concrete type with a known object size, out-of-bounds accesses are unlikely with this pattern.

Possible improvements

Protection against the violation of Allocator constraints

Certain `Genode::Allocator` implementations accept allocations of limited sizes only. For example, a slab allocator returns blocks of a specific size regardless of the size argument specified. It is the responsibility of the allocator user to make sure that an allocator specified to `new` matches the object size. The violation of this invariant remains undetected.

A straight-forward improvement would be the addition of assertions to the `alloc` operation of each individual allocator. However, this approach would detect the programming error solely at runtime.

A better solution would be the detection at compile time. The suitability of an allocator for allocating a specific object could be enforced at compile time by leveraging the C++ type system. Allocators could be modelled as class templates that provide their constraints (i. e., maximum allocation size) as type traits. The object creation via `new` would be replaced by a template function that performs the static checks of the compatibility of the to-be-created type with the allocator used.

Genode issue #1571

"base/allocator.h: retire new operators, introduce create function template instead"

https://github.com/genodelabs/genode/issues/1571

Clearing memory on free

The current implementation of the `Heap` and `Slab` allocators do not clear the managed memory.

Similarly to the discussion of preventing cold-boot attacks (XXX ref) at the granularity of dataspaces allocated from the RAM service, the clearing of memory on free would be a consequent approach. Memory should be cleared by default with the option to explicitly override the default in performance-critical situations.

Storing the heap's meta data from the user data in distinct dataspaces

In the current implementation, the heap's meta data is allocated from the heap itself. Therefore, the meta data may and up in a memory range adjacent to a user data block. A buffer overflow inside the user data block could eventually corrupt the meta data. An attacker with the knowledge about the allocation patters could then predict or provoke such a situation.

Combined with the use of guard pages around attached dataspaces, this measure would prevent a buffer overflow to corrupt meta data that is stored in a subsequent chunk.

C runtime The C runtime uses a slab-based allocation scheme for small blocks and the `Genode::Heap` for large blocks.

The C runtime's malloc back end was introduced because the FreeBSD's original allocator relied on `sbrk` and on-demand-paged anonymous memory, which is not available on Genode. It would be worthwhile to investigate alternative and time-tested allocators that do not rely on Unix mechanisms and feature heap-integrity protection.

3.4 Address-space layout randomization (ASLR)

By randomizing the address space of a vulnerable program, ASLR reduces the likelihood that an attacker who has successfully seized control over the program (Section 2.3.1) steers the program's control-flow towards an attacker-provided shellcode (Section 2.3.2) or return-oriented program (Section 2.3.3). Without the knowledge about the location of ROP gadgets within the program's virtual address space, an attacker cannot launch a ROP attack.

There exist various opportunities for randomization:

3.4.1 Randomizing library load addresses

By loading shared libraries to random locations, short running programs become hard to attack. However, long running programs such as daemons may be attacked by first gathering the information (the virtual address of a single library symbol is enough) needed to compute gadget addresses, followed by a ROP attack using the gadgets.

On Genode, ELF objects are placed within the so-called linker area, which is a portion of the component's virtual address space that is managed manually by the dynamic linker. There exist two opportunities for randomizing load addresses: First the base address of the linker area could be randomly picked by the linker. Second the location of each ELF object within the linker area could be randomized.

3.4.2 Randomizing executable load addresses

Unlike shared libraries, ELF executables are normally loaded at a link address that is defined at build time. Hence, gadgets present within the executable have an

attacker-known location. However, modern tool chains allow the creation of position-independent binaries, which principally enable the randomization of the executable's load address. On Linux on x86_64, position-independent binaries are used by default.

On Genode, ELF executables are not position-independent but there is no technical reason to use predefined link addresses. So this could be changed easily. The randomization of the executable load address would be performed by the dynamic linker analogously to the description of shared libraries above. However, also the location of the dynamic linker should be picked randomly. The dynamic linker is loaded by its parent. Therefore, the randomization would need to be implemented in `Genode::Child::Process` of Genode's base library.

3.4.3 Randomized stack locations

The location of stacks can be randomized within certain constraints, e. g., stack alignment and the virtual-memory layout conventions of the OS.

On Genode, the randomization of the stack location can be pursued at the following levels:

Location of the stack area The stack area is a portion of the component's virtual address space that hosts all stacks. In the current implementation, its base address is at a fixed location. However, since component code does not depend on the specific location, the base address could be randomized at a granularity of 1 MiB.

Allocation of a slot within the stack area Each thread occupies a 1 MiB slot of virtual memory within the stack area. Its stack is hosted somewhere within this slot. Furthermore, a few contextual information of the thread are stored at the boundary of the slot. This thread-local information can be accessed by the thread by clearing the lower 10 bits of its stack pointer and thereby enable Genode to provide a thread-local storage mechanism that does not rely on dedicated register (like a `gs`-segment-relative virtual register as used on Linux) and thereby works across all kernels.

Currently, the slot used by a thread is determined by a deterministic bit allocator. Randomizing the slot allocation would add 8 bits of entropy (the stack area has 256 slots).

User-level stack positions within their stack-area slots The location of the stack within the thread's slot could be randomized as long as the stack remains within the bounds of the slot. The randomization at page-granularity would add up to 10 bits of entropy. The implementation of this idea would imply that we need different mappings for the thread's context information (at the end of the slot) and the actual stack. The former could be turned into a R/O mapping so that the pointers within the context cannot be forged by an attacker even it the pointer's location is known (or brute-forced).

start offset within the first page of the stack

Normally, the stack starts at the upper bound of the underlying 4 KiB page. However, the start address could be picked randomly as long as the alignment constraint of 16 bytes is met. Consequently, this idea would add 8 bits of entropy. Note that this change would require us to enlarge the stacks by 4 KiB because up to 4096 - 16 bytes may remain unused.

3.4.4 Randomization of memory mappings

When mapping files to a program's virtual address space via `mmap` with a zero-address argument, the kernel is free to pick an arbitrary value as the virtual base address of the memory mapping.

On Genode, the corresponding mechanism is the `Region_map::attach` operation that attaches a dataspace to the region map of the component, thereby making the dataspace's content visible within the component's virtual address space. If the `attach` operation is invoked without an explicitly provided virtual address, core's RM service picks a location. The current implementation uses a deterministic best-fit allocator for this purpose. In order to randomize the mapping addresses, there are two ways forward, either by adding randomization to core's region-map implementation or by virtualizing core's PD service through a new "shim" component that sits between a to-be-randomized component and core. This component would transparently intercept the attach operations and implement a custom (randomized) allocation strategy. The appeal of the latter approach is that Genode's core component can be left unmodified. However, the disadvantage is that the interception of core's PD service is possible only at higher parts of the Genode component tree, not for components directly started by the top-level init component (because init requests the PD sessions for its immediate child components directly from init's parent, which is core).

3.4.5 Randomizing the addresses of the heap chunks / the BRK boundary

If the heap allocates its backing store in multiple chunks rather than a single contiguous anonymous memory mappings, the position of each chunk can be randomized by the kernel's `mmap` operation. This lowers the risk for heap-buffer overflows.

The `brk` boundary is used on traditional Unix systems to define the location of heap allocations. By randomizing this boundary, dynamically allocated heap objects end up at less predictable positions.

Genode does not use a concept like `brk`. Instead, the backing store of the heap is allocated in multiple independent chunks (dataspaces). Thereby, the heap would implicitly benefit from the randomization of memory mappings as described above.

3.4.6 VDSO randomization

VDSO (virtual dynamic shared object) is a trampoline mechanism used by the Linux user land to issue system calls to the kernel.

Man page of the vdso mechanism

http://man7.org/linux/man-pages/man7/vdso.7.html

The VDSO memory object is provided by the kernel whereby the kernel maintains the freedom to tune the kernel-entry mechanism to a suitable underlying mechanism (such as `int 0x80` or `sysenter`) without the need to relink the application. It is primarily motivated by performance considerations. The VDSO mapping is installed at a random position that is provided by the kernel to the process via the so-called "initial auxiliary vector" (passed above the argument list and environment variables)

On Genode, there is no VDSO-like mechanism.

3.4.7 Kernel address-space layout randomization

What the randomization of executable load addresses is to regular user-level programs, kernel ASLR is to the Linux kernel. The base address of the kernel image in virtual memory is randomized. Still the content of the image has a static layout. With the known kernel image and the address of a single symbol, all symbol addresses can be calculated. (related to kptr_restrict)

On GNU/Linux, the feature is usually not enabled by default.

3.4.8 ELF layout randomization

The randomization of the code within a single ELF object is an interesting idea proposed in the following paper:

Marlin A ne grained randomization approach to defend against ROP attacks:

https://w3.cs.jmu.edu/kirkpams/papers/nss13-marlin.pdf

Similar to the randomization of the load address of executables and shared libraries, the idea addresses the mitigation of ROP attacks, but in a much more profound way. By merely randomizing the load address of an ELF object, the attacker needs to know the virtual address of just a single library symbol to calculate the addresses of all gadgets present in the library. By randomizing the layout of the code within the ELF object at its load time, this is no longer possible.

The paper presents a prototypical implementation but omits several details that would be required in practice:

- It relies on objdump to perform the disassembly of the ELF binary. We would need to have a fast instruction decoder that identifies the jmp locations to adjust.
- The paper does not address the use of shared libraries and the handling of the PLT and GOT

However, ELF layout randomization is an intriguing approach. Granted, the code shuffling is complex. So adding this feature to an OS kernel would imply an unwelcome increase of kernel complexity. However, **on Genode, the mechanism could be provided by a dedicated ROM service** component that transparently applies ELF shuffling to ELF images obtained from another ROM service. Complex components that are likely to be vulnerable could obtain their binaries via this component whereas other component may obtain their binaries from Genode's regular ROM services directly. This way the presence of this feature would not negatively affect the TCB complexity of such components.

3.4.9 Further opportunities to apply ASLR to Genode

This section reviews further ideas that may be possible in principle.

Allocations within the capability-selector space

Capability selectors are allocated via a deterministic bit allocator. This could be changed to a randomized allocation strategy with the effect that an attacker could not predict the meaning behind the selectors and thereby could not manually issue RPC calls by directly using the kernel interface. On the first attempt to invoke a capability with spurious arguments, the server would keep the RPC call in a blocking state, halting the attacked program.

One could argue that a random allocation of capability-selector slots adds performance costs but in practice, the import of new capabilities into a component's capability is a rare operation so that the overhead would be negligible.

The following ideas remain questionable and are presented solely for the sake of completeness.

Microkernel-stack randomization

The stacks of the microkernel (base-hw or NOVA) could in principle be placed at random virtual kernel addresses. However, this may unreasonably increase the microkernel complexity. In contrast to the kernel stacks of a monolithic kernel that are exposed to a variety of highly complex subsystems and an overly bread kernel interface, a microkernel's stack is several orders of magnitude less likely to be vulnerable (there is not much code in the kernel) and exposed to the user land only via an extremely narrow interface.

UTCB location of the initial thread

The so-called user-level control block (UTCB) is a kernel-provided thread-specific communication buffer that is used by the thread to pass/receive arguments to/from the kernel. For the initial thread of a component, the UTCB is located at a prior known position. The randomization of this position is difficult because we would somehow need to propagate this address to a new thread w/o the thread's ability to interact with the kernel beforehand.

Slab entry allocations within a slab block

In the line of the argumentation of heap randomization, the allocation of slab entries within slab blocks could be considered. However, slabs are usually employed as a deliberate performance optimization and designed for very quick allocations. The randomization would require a search to find the Nth free entry of a slab block. This defeats the purpose of using a slab allocator in the first place.

3.5 Fortify source

"Fortify-source" is a compiler-assisted C-library-level mechanism to protect the use of typically vulnerable functions:

```
memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat,
strncat, sprintf, vsprintf, snprintf, vsnprintf, gets
```

Like many threat-mitigation techniques, the principle direction was first suggested by the OpenBSD community in the form of a `-Wbounded` compiler flag. The current approach is described in the following article:

Enhance application security with FORTIFY_SOURCE

https://access.redhat.com/blogs/766093/posts/1976213

If enabled, the compiler emits warnings for failed static checks, and generates additional runtime checks for the given functions. The generated tests are effective even if the programmer misses to check the validity of the arguments. If failed, the execution of the program aborts. The mechanism works similar to manually written assertions placed right before each call of the respective function. But the compiler generates those assertions automatically. Note that the same effect could not be achieved by placing assertions in the called functions because the functions solely receive a pointer as argument and are unaware of the dimensions of the underlying buffer in memory. So the sanity checks have to be generated at the caller side.

The protection is limited by built-in heuristics about the known set of vulnerable functions. It offers no protection against custom crafted vulnerabilities such as `for` loops with out-of-bounds indices.

Fortify-source is designated to defeat pointer-manipulation attacks. In contrast to `-fstack-protector`, the checks prevent not only stack smashing but also heap overflows and the override of local pointers. The compiler feature is available since GCC version 4 and can be enabled by the `-D_FORTIFY_SOURCE=1` compiler argument. The mechanism could be enabled for Genode's C runtime with relatively little effort. For Genode's base system, the technique has no effect because Genode's API does not rest on C library functions.

3.6 MMU mechanisms

This section reviews protection mechanisms provided by the MMU (memory management unit) hardware. Being microkernel-based, Genode naturally relies on MMU mechanisms in the form of virtual address spaces. The primary purpose of MMUs is not discussed here. Instead, we focus on less traditional mechanisms, namely data-execution prevention and supervisor-mode access/execution protection. For the sake of completeness, we briefly cover zero-address protection in Section 3.6.3.

https://pax.grsecurity.net/docs/PaXTeam-H2HC12-PaX-kernel-self-protection.pdf

3.6.1 Data Execution Prevention (DEP)

Data execution prevention (DEP) prevents the execution of "shellcode" injected into the stack or heap of a vulnerable program (Section 2.3.2) by marking memory mappings for the program's data as non-executable in the MMU page tables. The ability to mark memory mappings as non-executable is commonly referred to as "NX" bit. Page-table entries are either marked as read-only (constant data sections), read-and-executable (code), or read-writable (regular bss and data sections), but never writable and executable at the same time.

DEP on the x86 architecture was pioneered by PaX (NOEXEC), which emulated the non-existing NX bit on the x86 32-bit architecture. The emulation effectively turns x86 into a software-loaded TLB architecture. It works by marking all non-executable pages as privileged (accessible only by the kernel). Once the user program tries to access the data, a page fault is triggered (because of the attempt to access a privileged mapping). Based on the information provided by the CPU along with the page-fault exception, the in-kernel page-fault handler is able to distinguish the reason of the fault. If the fault resulted from an instruction fetch, an attempt to execute injected code within a data area is detected and the program aborts. If the fault was caused by a legitimate data access, the page-fault handler temporarily marks the page as user page, populates the DTLB (data translation look-a-side buffer) cache of the MMU by accessing the page, and subsequently mark the page as privileged. Because the mapping is now present in the DTLB, the user code will be served by the new DTLB entry when accessing the data after the page-fault handler passes control back to the user land.

Fortunately, recent CPU architectures (ARM, 64-bit x86) are equipped with NX support in hardware. On Genode, we do not plan to implement the NX emulation mechanism for the 32-bit x86 architecture because there is no good reason to use Genode on x86-32 instead of x86-64.

The propagation of the non-executable page attribute must be done by both core and the microkernel. In the current version of Genode, NX remains unused. However, the implementation should be straight-forward for the NOVA and the base-hw kernel.

mprotect, sealing of Genode dataspaces, PLT protection On Linux, existing mappings can be downgraded via the `mprotect` system call. This is useful in situations

where certain parts of the address space are expected to be constant over the further lifetime of the program.

On Genode, there is no mprotect-like mechanism. To accommodate the use cases of mprotect, Genode's RAM could be extended by a mechanism to "seal" a RAM dataspace to become read-only. This operation is irreversible until the destruction of the dataspace. A practical application would be a ROM provider that would first populate a RAM dataspace with information but turns the RAM dataspace into a ROM dataspace before handing it out to its client. Another example would the dynamic linker, which would be able to change the PLT section of an BIND_NOW-loaded program to a read-only mapping after initializing all PLT entries.

3.6.2 Supervisor Mode Access/Execution Protection (SMAP and SMEP)

The supervisor-mode access/execution protection (SMAP/SMEP) feature of modern MMU hardware protects the OS kernel from accidentally accessing or executing user memory.

Intel CPU security features

https://github.com/huku-/research/wiki/Intel-CPU-security-features

In a traditional attack pattern, the attacker places "shellcode" in arbitrary user memory, enters the kernel (via a system call), triggers a vulnerable kernel function (e. g., missing user-pointer check), and then tricks the kernel into jumping directly to the shellcode. Consequently, the attacker-controlled code is executed with kernel privileges. SMEP counters this kind of attack by disallowing the kernel to execute user pages.

Prior to the hardware-based SMEP mechanism, the PaX UDEREF feature pursued a very similar idea that unmaps the userland when entering the kernel mode.

On monolithic-kernel OSes, SMEP can be circumvented by sneaking "shellcode" into the kernel-address space, e. g., as payload that is buffered in the kernel. Normally such payload will ultimately be stored in non-executable memory inside the kernel. However, in some circumstances, such payload is placed in executable memory pools (GDI palette object on Windows 8) as described in the following article:

Intel SMEP overview and partial bypass on Windows 8

http://blog.ptsecurity.com/2012/09/intel-smep-overview-and-partial-bypass.html

Also, 3rd-party drivers are sometimes unaware of NX memory pools and store objects in executable memory. Other attacks try to change a page-table entry at known virtual addresses in the kernel and transform a user page into a kernel page.

Modifying Paging Structures

https://www.coresecurity.com/system/files/publications/2016/05/Windows%20SMEP%20bypass%20U%3DS.pdf

SMAP/SMAP on Genode On Genode, the interaction between a user thread and the microkernel is based on an extremely minimalistic mechanism called UTCB (user-level thread control block). There is one UTCB per user thread. The UTCB is a single memory page that is pinned in the microkernel's virtual address space (via a privileged mapping), and is also mapped inside the user space (via a user mapping). The microkernel never accesses the user-space mapping but solely the kernel mapping. Since UTCBs are pinned, the kernel can never fault when accessing a UTCB (no need to handle recursive page faults). A UTCB is used like an enlarged register set where arguments can be transferred between the user thread and the kernel. The interaction between a user thread and the microkernel works as follows:

1. User code writes content to the UTCB (e. g., an IPC message)
2. Invoke a syscall (passing further arguments in CPU registers) → entering the kernel, the transition code switches to the kernel stack
3. Kernel consumes UTCB content, performs kernel operation, it may populate the UTCB with results (e. g., an received IPC message)
4. Transition from kernel to user
5. User code consumes the new UTCB content

Because a UTCB is not shared between threads or PDs, it is not prone to leak information between PDs. Pointers are never passed as arguments to system calls! Neither in registers nor via the UTCB. The kernel never interprets pointers originating from user space.

SMEP/SMAP protection is very simple to support in Genode because the kernel is not expected to touch the user space anyway.

Technicalities:

- Page-table entries are already setting the U/S (user mode) flag for user pages.
- Check if SMEP is supported via CPUID.
- CR4.SMEP, CR4.SMAP must be set to enable the feature.
- If a mapping is changed from kernel to user, the TLB must be invalidated.

3.6.3 0-address protection

The first virtual-memory page cannot be populated via a memory mapping so a dereferenced null-pointer produces an unresolvable page fault.

This restriction applies to all Genode components.

3.7 Seccomp

Seccomp is a sandboxing mechanism for Linux introduced in 2005.

Seccomp and sandboxing (2009)

https://lwn.net/Articles/332974/

A seccomp overview (2015)

https://lwn.net/Articles/656307/

After an initialization phase, the Linux process drops its privileges via a `prctl` or `seccomp` (since 2014) system call. After this point, seccomp prevents the process from invoking any system calls except `read`, `write`, `exit`, and `sigreturn`. On some versions of Linux, entering seccomp may also disable the `rdtsc` instruction. The sandboxed code is limited to plain computations and the interaction with the file descriptors that were initialized prior entering the sandbox.

Seccomp remained a fairly obscure feature for several years with Google being the most prominent early user for running "native client" (NaCl) applications in the Chromium web browser. In this scenario, all interactions of the application with the outside world have to be issued indirectly via a "monitor" process that listens on one of the file descriptors passed to the sandbox. This puts the monitor in a position where it can validate all requests before performing the operation on the behalf of the sandboxed application. This design greatly reduces the attack surface of the kernel or any global resources, which can no longer be accessed directly. Hence, attackers need to find a vulnerability in the relatively small "monitor".

According to the LWN article cited above, Google initiated the discussion to make seccomp more flexible by introducing different "modes" of operation. This idea was eventually implemented by seccomp-bpf (BPF stands for Berkeley Packet Filter), which introduced a configurable policy for filtering system calls and system-call arguments. Today, seccomp-bpf is used by OpenSSH, Docker (optional feature), Cjdns, Tor, LXD (according to Wikipedia). Since seccomp became configurable, it is conveniently used (e. g., by Docker) to enforce system-call filtering policies (actually not implementing the "monitor" model). So users of Docker can tune the system calls usable by a container.

The "monitor" model of seccomp has similarities to Genode's parent-child relationship where the parent is able to interpose all interactions of the child with the outside world. In Genode, all components, except core, are effectively executed like seccomp'ed processes. The performance drawback of the monitoring approach, as mentioned in the LWN article, is addressed by Genode's session concept, which alleviates indirections through the monitor (or several nested monitors) when a child interacts with a service.

3.8 POSIX capabilities

POSIX capabilities are attributes of executable files that express the program's ability to perform certain system calls. Instead of granting a program root privileges, POSIX

capabilities can be used to allow a certain operation that is normally privileged, e. g., creating a bind mount. This reduces the need to grant all-encompassing root privileges to programs and decreases the likelihood for ambient authority problems as described in Section 2.3.

Thanks to the fine-grained privilege separation, as facilitated by Genode's component architecture and its capability-based access-control model, Genode does not suffer from the problem addressed by POSIX capabilities. E.g., there is not even the notion of a root user to start with.

3.9 Mandatory Access Control (MAC)

Access control on traditional Unix systems is based on discretionary access control where each file carries information about its owner (user, group) and the access permissions (read, write, execute) for the respective user, group, and the public. In addition to this traditional access-control scheme, Linux supports several policy frameworks that allow the user/admin to define the access rights of programs (subjects) to resources (objects).

The framework uses a common Linux kernel infrastructure called Linux Security Modules as underlying mechanism and addresses the problem of ambient authority that is otherwise inherent to traditional Unix systems.

Linux Security Modules (LSM) Linux Security Modules (LSM) provide a mechanism to interpose system calls and introduce policy hooks (also called "upcall") for operations that are deemed to need policing. Each time such an operation ought to be performed, an upcall requests a policy decision based on the operation arguments and the subject that initiated the operation. A explicit policy decision is taken for each access individually, hence the name "mandatory access control". LSM is a moving target. As the kernel changes, the hooks change, and policies need to be adapted.

In contrast to LSM-based MAC frameworks, Genode does not employ a system-global policy but is based on a decentralized capability-based access-control model where policies are expressed at different levels of Genode's component hierarchy. In particular, there is no need for a global system administrator with an all-encompassing view of the entire system.

SELinux SELinux is an "inode-based" MAC framework. Each time a program issues an operation, the policy module is requested to take a policy decision based on the subject, the accessed objects, and the type of operation. As a framework for expressing policies, subjects and objects are subsumed into categories. Assigning "user", "role", and "domain" attributes to subjects; or "name", "domain", and "type" attributes to objects. The object identities are based on inodes (in contrast to paths as used by AppArmor). Different models of expressing policies are provided: type enforcement, role-based access control (RBAC), and multi-level security.

SELinux policies tend to become extremely complex because the criterion for a policy decision is complex and the policy has to capture a system-global view. The burden is left to the policy writer. One needs to know the semantics and the consequences of all operations that are equipped with policy hooks. This is difficult.

If a policy hook is missing from a code path that issues a critical operation, SELinux remains without effect.

It is not enough to equip the kernel with policy hooks. But all user-level services shared by differently-labeled subjects must be covered as well. E.g., the X server is such a service, which comes with a corresponding extension called XACE.

SELinux is nicely applicable to daemons and the parts of the system that do not change over time, but hard to apply for parts that are dynamically changing (e. g., creating a new user, hotplugging of USB devices).

The following study provides an insightful qualitative analysis of the MAC systems SELinux, AppArmor, and FBAC-LSM.

Towards Usable Application-Oriented Access Controls

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.669.2435&rep=rep1&type=pdf

AppArmor AppArmor limits ("confines") programs to a set of files and POSIX capabilities with respect to file access, the loading of shared libraries, the execution of child processes, network access, DBus API access, and ptrace access. Compared to SELinux, it follows a more pragmatic approach.

FBAC-LSM In FBAC-LSM, policies are expressed as hierarchies of functionalities.

FBAC-LSM - protect yourself from your apps

http://schreuders.org/FBAC-LSM/

Compared to SELinux and AppArmor, users are not bothered with too many technical details but can reason about policies on a more useful level.

SMACK SMACK is a MAC framework that provides a subset of SELinux but is implemented using LSM directly instead of using SELinux.

Simplified Mandatory Access Control Kernel

http://schaufler-ca.com/description_from_the_linux_source_tree

It uses extended file attributes "xattr" to attach labels to files. Labels are assigned to subjects (tasks) and objects, which is a simplifications compared to SELinux.

3.10 Information leakage prevention

3.10.1 /proc/$pid/maps protection

The information in /proc/$pid/maps used to be world-readable. However, a local attacker may use the knowledge about the virtual address-space layout of any process to circumvent ASLR for launching ROP attacks. For this reason, current Linux systems deny the global access of this information.

On Genode, there is no globally visible information like /proc.

3.10.2 Stack leakage in the padding in API data structures

Section 2.2.3 introduced the problem of leaking information via uninitialized parts of parameter structures.

There exists no generally applicable mitigation for this problem on commodity OSes. Within Genode, the problem may occur for RPC functions that pass structured data as RPC arguments. To make sure that all those types are void of any padding, static assertions may be leveraged. E.g.,

```
struct status {
  short device_id;
  long  status_bits;
};

static_assert(sizeof(device_id) + sizeof(status_bits)
              == sizeof(status),
              "possible information leak");
```

Furthermore, all types that are potentially passed as RPC arguments could be augmented with a `Rpc_arg` base class that clears the object at its construction time unless explicitly prevented. Skipping the clearing would require the programmer to take a cautious decision with an accompanied explanation (e. g., the absence of padding is shown by a static assertion). Genode's RPC mechanism could check the adherence of RPC arguments to this regime at compile time, allowing only basic types and types tagged as `Rpc_args` to be used as RPC arguments.

3.10.3 Kernel Address Display Restriction and dmesg restrictions

This mitigation measure tries to hide the address-space layout of the kernel from attackers so that exploits cannot adapt their operation to the present layout. If enabled, kernel addresses are not spilled in logs or via the /proc interface, and the read access to the kernel image (vmlinuz, System.map) is prevented.

kptr_restrict for hiding kernel pointers

https://lwn.net/Articles/420403/

The leakage of kernel pointers is prevented by using a dedicated format specifier "%pK" in the kernel. If kptr_restrict is enabled, any pointers printed via this format specifier will appear as "0". For format strings that don't use this format specifier, `kptr_restrict` is without effect.

Many kernel pointers are still present in the output of dmesg. The following article exemplifies how the kptr_restrict feature can be side-stepped.

Effectively bypassing kptr_restrict on Android

http://bits-please.blogspot.de/2015/08/effectively-bypassing-kptrrestrict-on.html

On Genode, a regular component is not able to observe kernel output and the kernel image. Still the ELF image of core, other components of the initial static system, or the NOVA kernel's hypervisor information page are accessible via core's ROM service. For this reason, ROM session requests from arbitrary components should not unconditionally be routed to core's ROM service. When deploying Genode, ROM sessions should be routed on a per-label basis

3.11 Diminishing the attack surface

The mechanisms compiled in this section are Linux-specific precautions that are recommended to avoid unnecessary attack vectors. They are largely unrelated to Genode.

3.11.1 Hardlink restrictions

Genode has no notion of hardlinks.

3.11.2 ptrace scope

Ptrace is a system call that allows a process to manipulate or inspect another process of the same user. Its designated use case are debuggers. However, a compromised user program may ptrace to spy on sensitive processes of the user that are currently running - such as SSH sessions or GPG agent. For this reason, recent GNU/Linux distributions restrict the use of ptrace.

On Genode, there is no debugging facility comparable to ptrace. In order to debug a program, the program must to be executed under the control of the debugger in the first place. Since the debugger is the parent (the owner) of the debugging target, it naturally has the authority to inspect or manipulate the debugging target. There is no way to debug an unrelated component.

3.11.3 /dev/mem protection

The /dev/mem pseudo device allows user-level programs to directly map portions of the physical memory into their virtual address spaces. The most prominent user is the X server.

On Genode, access to physical memory is arbitrated via core's RAM and IO_MEM services. Those services ensure that no physical RAM page is handed out twice. The access to IO_MEM is preserved to user-level device drivers. Actually, regular user-level device drivers don't access this service directly but use the platform driver instead. The platform driver adds the notion of devices to the system and enforces access control at device granularity.

3.11.4 Disabling /dev/kmem

Genode provides no way to let user programs access any kernel memory.

3.11.5 Block module loading and kexec

The microkernel is not expandable by loadable modules.

Kexec is a feature to update the kernel at runtime without the need to reboot the machine.

There is no mechanism to replace the microkernel at runtime.

3.11.6 Blacklisting of rare protocols

There are no rare protocols or similar functionality dynamically loaded into the TCB of Genode.

3.12 Further mitigation mechanisms on non-Linux OSes

This section reviews approaches that we deem as noteworthy for their novelty or as source of inspiration.

3.12.1 Pledges (OpenBSD)

In contrast to most mitigation techniques that are focused on subverting different stages of privilege-escalation attacks 2.3, OpenBSDs "pledges" try to limit the reach of the actual exploitation phase of an attack (Section 2.4). The mechanism can be applied to applications that are trusted but may be vulnerable. Examples are the OpenBSD user-land tools and daemons that ship with OpenBSD.

pledge() a new mitigation mechanism (2015)

http://www.openbsd.org/papers/hackfest2015-pledge/mgp00001.html

The developer annotates program code with assumptions about the future behavior of the program. The program "pledges" that it will behave in a certain way in the future. If it happens to violate its pledge, it may have been compromised. Hence, pledges express a high-level model of the application's own behavior in advance of its execution. It therefore represents a form of privilege dropping. It makes it harder to

exploit a vulnerability of a program that is treated with pledges because the attacker's won't be able to extend the model of the program's behavior.

The pledge approach is closely tied to the semantics of the OpenBSD POSIX system. It is not directly transferable to Genode. Genode's approach would be to use short-living sandboxes to execute code that is complicated. The sandbox can be subjected to a rigid policy similar to the different types of "pledges". E.g., a sub-component for decoding a PNG images would receive the PNG image presented as a ROM session, it can write the pixels into a framebuffer session, but is not able to obtain a file-system or NIC session or any other session to an unrelated service. This approach corresponds to OpenBSD's privilege separation as implemented for a few selected daemons like OpenSSH. But in contrast to Unix-based operating systems where sandboxing requires additional effort and considerations, Genode's component architecture leverages sandboxing inherently.

3.12.2 Host-based intrusion detection (HIDS)

Host-based intrusion detection systems monitor the behavior of the system at runtime:

Host-based intrusion-detection system

https://en.wikipedia.org/wiki/Host-based_intrusion_detection_system

They detect persistent changes in the system, e. g., caused by an attacker installing a backdoor as payload of a successful exploit. Furthermore, the integrity of certain data structures (like the system-call table of the kernel) is monitored. If an anomaly is detected, the HIDS triggers an alarm, similarly to how AntiVirus software responds to detected malware. A HIDS puts itself in the shoe of a cautious user that monitors the operation of his machine and the files stored on the file system.

3.12.3 Microsoft EMET defense against ROP attacks

Microsoft's Enhanced Mitigation Experience Toolkit (EMET) provides a number of hardening "features" that can be selectively applied according to a policy defined by an educated user.

Microsoft Enhanced Mitigation Experience Toolkit (EMET)

https://adsecurity.org/?p=157

Among the features is an interesting counter measure against ROP on Windows:

"EMET makes sure that when a critical function is reached, it is reached via a CALL instruction rather than a RET instruction. This is a very useful mitigation and breaks many ROP gadgets. This mitigation may be incompatible with some applications. This mitigation is available for 32 bit processes."

However mitigations with a similar line of thinking (i. e., let the potentially compromised code check its own integrity) had been repeatedly implemented and subsequently defeated. While they raise the barrier for the current generation of attacks, the next generation will take this mitigation into account and work around it. In the longer term, the effect of such a cat-and-mouse game is an inflated complexity of the to-be-defended program.

4 Review of recent CVEs

This section reviews recently published Common Vulnerabilities and Exposures (CVEs) from Genode's perspective. Even though Genode's scope goes beyond that of an OS kernel, the review focuses of the CVEs related to Xen and Linux as the predominant open-source OS foundations.

During the review, certain repetitive patters become apparent. Section 4.1 discusses those patterns. It is followed by Sections 4.2 and 4.3 that contain the annotated summaries of the CVEs referring to the Xen hypervisor and the Linux kernel in between January and August of 2016. The annotations discuss their potential relation to Genode. Within these sections, the short descriptions given by the CVS appear *emphasized*. Our annotations are given in a regular typeface. Important Genode-related considerations are highlighted as **bold** text. Section 4.4 summarizes the lessons learned from reviewing the CVEs.

4.1 Typical kinds of vulnerabilities

4.1.1 Double-fetch issues

Double-fetch issues are related to time-of-test time-of-use issues. A service (like a device driver) receives a request for an operation along with parameters that are provided in a buffer shared with the client. If a value is repeatedly fetched from the buffer, the driver code expects the value to remain constant. However, the client may modify it at any point in time, corrupting the program flow or offset calculations that depend on the fetched values.

On Genode, services usually offer their operations via RPC interfaces. Genode's RPC mechanism is immune to double-fetch issues because client and server never operate on the same buffer. Arguments are copied by the microkernel via the IPC operation.

Double-fetch issues may still occur in situations where two components communicate over shared memory. The access to such shared memory buffers is typically safeguarded in C++ classes that facilitate save patterns, e. g., the so-called packet-stream interface takes care of copying-out packet descriptors.

4.1.2 Kernel-information leaks via parameter structures

Kernel-internal information can leak in several ways. But by far the most prominent vulnerabilities are information leaks via incompletely initialized parameter structures as explained in Section 2.2.3.

In principle, this problem could be mitigated by clearing the kernel stack on each kernel entry. Grsecurity's "STACKLEAK" plugin implements this idea. However, this technique seems to be generally regarded as impractical due to its performance costs.

4.1.3 Dereferenced null pointers or dangling pointers

- memory corruption

- fatal for the entire system

4.2 Xen hypervisor

http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=xen+2016

CVE-2016-6259 denial of service (missing SMAP whitelisting)

Xen 4.5.x through 4.7.x do not implement Supervisor Mode Access Prevention (SMAP) whitelisting in 32-bit exception and event delivery, which allows local 32-bit PV guest OS kernels to cause a denial of service (hypervisor and VM crash) by triggering a safety check.

The Xen hypervisor delivers exception information to paravirtualized 32-bit guest VMs by directly writing to user-space, which triggers a SMAP violation. The guest can deliberately trigger this condition. This issue would not occur on Genode because the microkernel never legitimately accesses the user space.

CVE-2016-6258 privilege escalation

The PV pagetable code in arch/x86/mm.c in Xen 4.7.x and earlier allows local 32-bit PV guest OS administrators to gain host OS privileges by leveraging fast-paths for updating pagetable entries.

An optimization in the hypervisor missed to validate input from the untrusted PV guest. Genode does not provide a PV mechanism as present in Xen. Hence the complexity of paravirtualized page-table manipulations is avoided by using the EPT mechanism of the hardware.

CVE-2016-5242 denial of service

The p2m_teardown function in arch/arm/p2m.c in Xen 4.4.x through 4.6.x allows local guest OS users with access to the driver domain to cause a denial of service (NULL pointer dereference and host OS crash) by creating concurrent domains and holding references to them, related to VMID exhaustion.

The allocation of VMIDs (similar to TLB tags) failed to consider the situation where the physically-limited ID space gets exhausted. In this condition, a null pointer was dereferenced, crashing the hypervisor. The condition could be triggered by guests that could deliberately trigger the allocation of VMIDs.

There is no simple-to-apply mitigation technique for these kind of bugs. Systematic tests of corner cases like this should be in place. The problem is related to the overflowing of reference counters. All places where refcounts of server-side objects can be manipulated by untrusted clients deserve special attention.

In Genode, similar issues to the exhaustion of physically-limited resources exist. E.g., the number of capabilities is globally limited. A component can allocate capabilities in an infinite loop to exhaust them. This problem should be solved by subjecting capability allocation to the same accounting scheme that Genode uses for physical memory.

Genode capabilities are reference counted. This lifetime management scheme could be changed to circular linked smart pointers such that no upper limit of the number of references exists.

CVE-2016-4963 denial of service

The libxl device-handling in Xen through 4.6.x allows local guest OS users with access to the driver domain to cause a denial of service (management tool confusion) by manipulating information in the backend directories in xenstore.

This is an input-validation problem where a central service interprets directory structures that are under control of untrusted driver domains. There is no equivalent of a global Xen store on Genode. Still, management components may interpret information reported by untrusted components (via Genode's report session). It goes without saying that such information must be parsed defensively.

CVE-2016-4962 denial of service, potential privilege escalation

The libxl device-handling in Xen 4.6.x and earlier allows local OS guest administrators to cause a denial of service (resource consumption or management facility confusion) or gain host OS privileges by manipulating information in guest controlled areas of xenstore.

Central management tools use to rely on meta data (kept at the Xenstore) that is under control of untrusted device VMs. In particular, a device VM may remove meta data that is needed to properly release resources on device VM destruction. In short, the integrity of the meta data operated on by central management components is unprotected.

The issue does not apply to Genode as there is no equivalent to a Xenstore.

CVE-2016-4480 privilege escalation

The guest_walk_tables function in arch/x86/mm/guest_walk.c in Xen 4.6.x and earlier does not properly handle the Page Size (PS) page table entry bit at the L4 and L3 page table levels, which might allow local guest OS users to gain privileges via a crafted mapping of memory.

The software page-table walker in the hypervisor misinterpreted page-table structures (by not interpreting the so-called "page size bit"). On Genode, there is not page-table walker interpreting guest page tables in the kernel or core. When hosting a guest VM on top of Genode, the page-table-walker is encapsulated in the VM-specific user-level virtual-machine monitor, which is untrusted. Hence, the flaw - if present - would have no consequences beyond the particular virtual machine.

CVE-2016-3960 denial of service, potential privilege escalation

Integer overflow in the x86 shadow pagetable code in Xen allows local guest OS users to cause a denial of service (host crash) or possibly gain privileges by shadowing a superpage mapping.

The aliasing of superpages with page-table structures was not implemented correctly, leading to a later de-reference of an undefined pointer. Similar issues could occur in a microkernel implementation. In the specific CVE the undefined pointer was fortunately a null pointer. If it was a dangling pointer, kernel ASLR would have helped to mitigate attacks exploiting this bug.

CVE-2016-3159/3158 information leak across VMs

The xrstor function in arch/x86/xstate.c in Xen 4.x does not properly handle writes to the hardware FSW.ES bit when running on AMD64 processors, which allows local guest OS users to obtain sensitive register content information from another guest by leveraging pending exception and mask bits. NOTE: this vulnerability exists because of an incorrect fix for CVE-2013-2076.

The fpu_fxrstor function in arch/x86/i387.c in Xen 4.x does not properly handle writes to the hardware FSW.ES bit when running on AMD64 processors, which allows local guest OS users to obtain sensitive register content information from another guest by leveraging pending exception and mask bits. NOTE: this vulnerability exists because of an incorrect fix for CVE-2013-2076.

The FPU emulation on AMD CPUs was not implemented correctly. In Genode/NOVA, instructions are not emulated in the kernel/hypervisor but solely by the untrusted user-level VMM.

However, in general, leaks via register contents may be present in a microkernel implementation of the context switching between threads. If the kernel misses to copy a register, one thread could read the register content previously written by another thread. There is no general mitigation measure that would counter such a hole in the implementation.

CVE-2016-2271 denial of service

VMX in Xen 4.6.x and earlier, when using an Intel or Cyrix CPU, allows local HVM guest users to cause a denial of service (guest crash) via vectors related to a non-canonical RIP.

A guest user program may deliberately modify the instruction pointer in a way that a consecutive VM entry will fail, leading to a crash of the entire VM. The issue has no consequences outside the single guest. In principle, this issue may be present in user-level VMMs on top of Genode. Genode's architecture would not help, but no mitigation measure would help either.

CVE-2016-2270 denial of service

Xen 4.6.x and earlier allows local guest administrators to cause a denial of service (host reboot) via vectors related to multiple mappings of MMIO pages with different cachability settings.

Guest OSes are able to deliberately produce inconsistencies in the cache attributes for memory pages. According to the Intel specification, such inconsistencies may yield undefined behavior such as a machine reboot.

Genode satisfies the invariant mandated by the specification by fixing cache attributes to physical memory objects (dataspaces). The attributes are cached/uncached (for RAM) or write-combined/in-order (for memory-mapped I/O resources). All memory mappings consistently inherit the attributes of the underlying dataspace.

CVE-2016-1571 denial of service

The paging_invlpg function in include/asm-x86/paging.h in Xen 3.3.x through 4.6.x, when using shadow mode paging or nested virtualization is enabled, allows local HVM guest users to cause a denial of service (host crash) via a non-canonical guest address in an INVVPID instruction, which triggers a hypervisor bug check.

This is an instruction-emulator issue. On Genode, this issue would be contained in the untrusted user-level VMM. The Genode system outside the VM would remain unaffected.

CVE-2016-1570 privilege escalation

The PV superpage functionality in arch/x86/mm.c in Xen 3.4.0, 3.4.1, and 4.1.x through 4.6.x allows local PV guests to obtain sensitive information, cause a denial of service, gain privileges, or have unspecified other impact via a crafted page identifier (MFN) to the (1) MMUEXT_MARK_SUPER or (2) MMUEXT_UNMARK_SUPER sub-op in the HYPERVISOR_mmuext_op hypercall or (3) unknown vectors related to page table updates.

Lacking validation of untrusted input allows PV guests to manipulate page-table structures. On Genode, there is no equivalent to Xen's PV guests.

4.3 Linux kernel

http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+kernel+2016

The CVEs for the Linux kernel refer to various parts of the kernel. In the following, we distinguish device drivers, protocol stacks, low-level kernel mechanisms, and security features.

4.3.1 Bugs in device drivers

Device drivers amount for the biggest share of kernel code. Hence, the fact that most CVEs refer to device drivers is not surprising. We expect the dark figure to be even much higher since device drivers receive relatively little attention from reviewers compared to generic parts of the kernel. The relevance of drivers is limited to the respective devices. So there is less incentive to review driver code compared to parts that are used by a big share of users. In cases where drivers undergo an actual review, large batches of CVEs are the result as illustrated by the 11 CVEs for Qualcomm device drivers.

The attack vectors on device drivers fall in two categories:

Vulnerabilities triggered by devices

Many device drivers expect their devices to be well-behaving because malicious devices as an attack model are relatively uncommon. However, given the fact that low-cost programmable USB dongles can mimic any kind of USB device, attacks via pluggable devices are no fiction. Granted, an attacker needs proximity to the machine. But there are many scenarios where this is the case such as using a laptop in a public environment.

Most CVEs of this kind result in dereferenced null pointers, effectively crashing the kernel. But the consequences of others are circumscribed as "other unspecified impact", in particular in the presence of a buffer overflow or memory corruption.

Vulnerabilities triggered via the user-level device API

Most device drivers expose the functionality of their device in the form of a device node to the user land. User-space applications that need to interact with the device open the corresponding pseudo file and access the device's functionality via `ioctl` operations on the file handle. The actual `ioctl` operations and their parameters are implemented specifically for each device driver.

The ioctl mechanism for the interaction of user-level software with devices is inherently prone to kernel-information leaks via parameter structures and double-fetch issues.

User-level access control policies effectively limit this attack vector to a few legitimate applications unless those policies can be circumvented. Some of the CVEs discussed in 4.3.4 would potentially open the door for illegitimate users of devices.

In contrast to Linux where device drivers offer hugely diverse `ioctl` interfaces, driver components on Genode solely implement generic interfaces that are defined per device class (block, framebuffer, input). Most driver-specific operations that inflate the `ioctl` interfaces of Linux drivers are concerned with device configuration rather than device operation. On Genode, the configuration of devices is covered by an out-of-band component configuration mechanism, which is not exposed to the clients of the device.

Most of the vulnerabilities listed below have the potential to crash the kernel. Several may be exploitable or compromise the machine ("unspecified impact"). Such an exploit may affect any part of the system. Kernel ASLR helps to counter such exploitation attempts.

On a Genode system, none of the vulnerabilities would have such drastic consequences because each driver lives in a dedicated user-level component. If a driver crashes, it becomes unavailable to its client but the reach of the problem is limited by the functional scope of the driver. E.g., a crashing or compromised USB driver won't put any data stored on a SATA disk at risk.

CVE-2016-6480 kernel crash caused by double-fetch issue

Race condition in the ioctl_send_fib function in drivers/scsi/aacraid/commctrl.c in the Linux kernel through 4.7 allows local users to cause a denial of service (out-of-bounds access or system crash) by changing a certain size value, aka a "double fetch" vulnerability.

CVE-2016-6130 kernel-information leak caused by double-fetch issue

Race condition in the sclp_ctl_ioctl_sccb function in drivers/s390/char/sclp_ctl.c in the Linux kernel before 4.6 allows local users to obtain sensitive information from kernel memory by changing a certain length value, aka a "double fetch" vulnerability.

CVE-2016-6156 double fetch in Chrome OS embedded controller

Race condition in the ec_device_ioctl_xcmd function in drivers/platform/chrome/cros_ec_dev.c in the Linux kernel before 4.7 allows local users to cause a denial of service (out-of-bounds array access) by changing a certain size value, aka a "double fetch" vulnerability.

Ssimilar to CVE-2016-6480.

CVE-2016-5829 buffer overflow in USB stack (HID)

Multiple heap-based buffer overflows in the hiddev_ioctl_usage function in drivers/hid/usbhid/hiddev.c in the Linux kernel through 4.6.3 allow local users to cause a denial of service or possibly have unspecified other impact via a crafted (1) HIDIOCGUSAGES or (2) HIDIOCSUSAGES ioctl call.

Buffer overflows in the USB stack. Genode uses the Linux USB stack as user-level component. However, there is no way for any client to issue ioctl operations, which would trigger the issue. The reach of the problem would be restricted to the USB driver component regardless.

CVE-2016-5728 memory corruption in Intel manycore platform stack

Race condition in the vop_ioctl function in drivers/misc/mic/vop/vop_vringh.c in the MIC VOP driver in the Linux kernel before 4.6.1 allows local users to obtain sensitive information from kernel memory or cause a denial of service (memory corruption and system crash) by changing a certain header, aka a "double fetch" vulnerability.

CVE-2016-5400 memory leak in USB SDR device driver

Memory leak in the airspy_probe function in drivers/media/usb/airspy/airspy.c in the airspy USB driver in the Linux kernel before 4.7 allows local users to cause a denial of service (memory consumption) via a crafted USB device that emulates many VFL_TYPE_SDR or VFL_TYPE_SUBDEV devices and performs many connect and disconnect operations.

Bug in a USB device driver for a software-defined radio device. On Genode, such a driver would run as dedicated component using the USB-device session of the USB driver component. The leaking memory would be accounted to the faulty USB-device driver. Once its assigned quota is exhausted, it would block on a resource request. The remaining system including other USB device drivers would remain unaffected.

CVE-2016-5342 heap overflow in wireless driver

Heap-based buffer overflow in the wcnss_wlan_write function in drivers/net/wireless/wcnss/wcnss_wlan.c in the wcnss_wlan device driver for the Linux kernel 3.x, as used in Qualcomm Innovation Center (QuIC) Android contributions for MSM devices and other products, allows attackers to cause a denial of service or possibly have unspecified other impact by writing to /dev/wcnss_wlan with an unexpected amount of data.

Heap overflow in wireless driver, triggered by an `ioctl` operation.

CVE-2016-4482 kernel-information leak in the USB core

The proc_connectinfo function in drivers/usb/core/devio.c in the Linux kernel through 4.6 does not initialize a certain data structure, which allows local users to obtain sensitive information from kernel stack memory via a crafted USBDEVFS_CONNECTINFO ioctl call.

Genode uses the Linux USB stack in a user-level component. The information leaks over the `ioctl` interface, which is not exposed to clients of the USB stack.

CVE-2016-3955 buffer overflow in USB net

The usbip_recv_xbuff function in drivers/usb/usbip/usbip_common.c in the Linux kernel before 4.5.3 allows remote attackers to cause a denial of service (out-of-bounds write) or possibly have unspecified other impact via a crafted length value in a USB/IP packet.

Input validation issue in USB-networking code, resulting in a possible buffer overflow.

CVE-2016-3951 unspecified impact via to double free in USB net

Double free vulnerability in drivers/net/usb/cdc_ncm.c in the Linux kernel before 4.5 allows physically proximate attackers to cause a denial of service (system crash) or possibly have unspecified other impact by inserting a USB device with an invalid USB descriptor.

The following CVEs show the same pattern where a null pointer is de-referenced in a USB device driver, eventually resulting in a kernel crash.

CVE-2016-3689 null pointer in USB input (ims-pcu) driver

CVE-2016-3140 null pointer in USB serial driver

CVE-2016-3139 null pointer in USB Wacom tablet driver

CVE-2016-3138 null pointer in USB driver

CVE-2016-3137 null pointer in USB-serial driver

CVE-2016-3136 null pointer in USB-serial driver

CVE-2016-2782 null pointer in USB-serial driver

CVE-2016-2188 null pointer in USB iowarrior driver

CVE-2016-2187 null pointer in USB tablet driver

CVE-2016-2186 null pointer in USB powermate driver

CVE-2016-2185 null pointer in USB ATI remote driver

CVE-2016-2184 null pointer in USB sound driver

CVE-2016-2384 double free in USB midi driver

Double free vulnerability in the snd_usbmidi_create function in sound/usb/midi.c in the Linux kernel before 4.5 allows physically proximate attackers to cause a denial of service (panic) or possibly have unspecified other impact via vectors involving an invalid USB descriptor.

CVE-2016-2117 kernel-information leak in atheros network driver

The atl2_probe function in drivers/net/ethernet/atheros/atlx/atl2.c in the Linux kernel through 4.5.2 incorrectly enables scatter/gather I/O, which allows remote attackers to obtain sensitive information from kernel memory by reading packet data.

Allows an network-remote attacker to obtain kernel-internal data, caused by wrong scatter/gather I/O handling. On Genode, the encapsulation of each driver in a separate address space limits the leak to network data.

There is a huge batch of CVEs related to Qualcomm device drivers. Most of the vulnerabilities may lead to privilege escalation.

CVE-2016-5344 memory corruption in Qualcomm video driver

CVE-2016-2068 buffer overflow in Qualcomm audio driver

CVE-2016-2067 buffer overflow in Qualcomm GPU driver

CVE-2016-2066 memory corruption in Qualcomm audio driver

CVE-2016-2065 kernel write by Qualcomm MSM driver

CVE-2016-2064 buffer over-read in Qualcomm MSM driver

CVE-2016-2063 buffer overflow in Qualcomm thermal driver

CVE-2016-2062 heap overflow in Qualcomm GPU driver

CVE-2016-2061 array overflow in Qualcomm Video4Linux support

CVE-2016-2059 missing permission check in MSM IPC router

CVE-2016-0723 use-after-free in TTY ioctl

Race condition in the tty_ioctl function in drivers/tty/tty_io.c in the Linux kernel through 4.4.1 allows local users to obtain sensitive information from kernel memory or cause a denial of service (use-after-free and system crash) by making a TIOCGETD ioctl call during processing of a TIOCSETD ioctl call.

4.3.2 Logical errors and bugs in protocol stacks (networking, file systems, audio)

Linux supports a large number of complex protocol stacks in the kernel. With protocol stacks, we refer to code that translates software interfaces between different levels of abstraction. E.g., a file system translates the notion of files and directories to the reading and writing of blocks on a block device. Or a network stack translates the socket API to network packets.

In contrast to device drivers that are usually exposed via a single pseudo device, protocol stacks are exposed to the user space via diverse kernel interfaces that are much less tangible than a single pseudo device. The interactions includes the handling of network-protocol families or file-system operations (e. g., the complicated semantics of hardlinks).

Protocol stacks may be exposed to the attackers from the "outside". E.g., a USB stick with a manipulated ISO 9660 file system may trigger bugs while parsing file-system meta data (CVE-2016-4913), or the network stack may misinterpret packets (CVE-2016-3707).

Within Genode, a protocol stack usually has the form of a library that is linked to the component that uses the protocol stack. E.g., a TCP/IP stack is linked directly to a networking application. If a network interface is shared by multiple applications, a dedicated component multiplexes the network interface. The multiplexing can be realized at different levels, e. g., below TCP/IP at network-packet level or above a TCP/IP stack at the socket API level. Thereby the component-based architecture helps to prevent complex protocol stacks to become central points of failure or prone to information leakage.

File systems

CVE-2016-6516 denial of service, possible privilege escalation

> *Race condition in the ioctl_file_dedupe_range function in fs/ioctl.c in the Linux kernel through 4.7 allows local users to cause a denial of service (heap-based buffer overflow) or possibly gain privileges by changing a certain count value, aka a "double fetch" vulnerability.*
>
> This is a potential heap overflow in the generic file-system code. On Genode, there is no global VFS. The bug - if present - would remain local to the VFS-using component. It would not put the kernel or the Genode base system at risk.
>
> Kernel ASLR would be an effective countermeasure against the exploitation of the bug on Linux.

CVE-2016-6197/6198 denial of service

> *The filesystem layer in the Linux kernel before 4.5.5 proceeds with post-rename operations after an OverlayFS file is renamed to a self-hardlink, which allows local users to cause a denial of service (system crash) via a rename system call, related to fs/namei.c and fs/open.c.*

fs/overlayfs/dir.c in the OverlayFS filesystem implementation in the Linux kernel before 4.6 does not properly verify the upper dentry before proceeding with unlink and rename system-call processing, which allows local users to cause a denial of service (system crash) via a rename system call that specifies a self-hardlink.

The CVE describes a bug in the VFS implementation with the interplay of hardlinks and overlay fs. This problem does not exist on Genode. There is no global VFS. The component-local VFS does not support hardlinks.

CVE-2016-4913 kernel-information leak

The get_rock_ridge_filename function in fs/isofs/rock.c in the Linux kernel before 4.5.5 mishandles NM (aka alternate name) entries containing 0 characters, which allows local users to obtain sensitive information from kernel memory or possibly have unspecified other impact via a crafted isofs filesystem.

The in-kernel ISO-file-system driver mishandles file-system meta data.

CVE-2016-4581 denial of service

fs/pnode.c in the Linux kernel before 4.5.4 does not properly traverse a mount propagation tree in a certain case involving a slave mount, which allows local users to cause a denial of service (NULL pointer dereference and OOPS) via a crafted series of mount system calls.

Bug in the traversal of file-system structures.

cve-2016-1575/1576/2853/2854 privilege escalation in overlayfs/aufs (union mounts)

The overlayfs implementation in the Linux kernel through 4.5.2 does not properly restrict the mount namespace, which allows local users to gain privileges by mounting an overlayfs filesystem on top of a FUSE filesystem, and then executing a crafted setuid program.

The overlayfs implementation in the Linux kernel through 4.5.2 does not properly maintain POSIX ACL xattr data, which allows local users to gain privileges by leveraging a group-writable setgid directory.

The aufs module for the Linux kernel 3.x and 4.x does not properly restrict the mount namespace, which allows local users to gain privileges by mounting an aufs filesystem on top of a FUSE filesystem, and then executing a crafted setuid program.

The aufs module for the Linux kernel 3.x and 4.x does not properly maintain POSIX ACL xattr data, which allows local users to gain privileges by leveraging a group-writable setgid directory.

Exploit works by combining FUSE with user name spaces. Aufs propagates extended file attributes from the mounted FUSE file system as is, instead of sanitizing the attributes.

Genode's VFS library has a stacked file-system concept similar to aufs. But it does neither support nor use extended file attributes. In contrast to mandatory access control frameworks on Linux, access control is not managed on a per-file (or per inode) basis.

Communication stacks

CVE-2016-6162

net/core/skbuff.c in the Linux kernel 4.7-rc6 allows local users to cause a denial of service (panic) or possibly have unspecified other impact via certain IPv6 socket operations.

Details are missing in the CVE. The report describes a triggered assertion in the network-handling code, which effectively halts the system.

CVE-2016-5696

net/ipv4/tcp_input.c in the Linux kernel before 4.7 does not properly determine the rate of challenge ACK segments, which makes it easier for man-in-the-middle attackers to hijack TCP sessions via a blind in-window attack.

This is a problem in the network-protocol implementation. Not affecting the kernel, and unrelated to mitigation techniques.

CVE-2016-5243/5244 kernel-information leak

The tipc_nl_compat_link_dump function in net/tipc/netlink_compat.c in the Linux kernel through 4.6.3 does not properly copy a certain string, which allows local users to obtain sensitive information from kernel stack memory by reading a Netlink message.

The rds_inc_info_copy function in net/rds/recv.c in the Linux kernel through 4.6.3 does not initialize a certain structure member, which allows remote attackers to obtain sensitive information from kernel stack memory by reading an RDS message.

CVE-2016-4951 denial of service, unspecified other impact

The tipc_nl_publ_dump function in net/tipc/socket.c in the Linux kernel through 4.6 does not verify socket existence, which allows local users to cause a denial of service (NULL pointer dereference and system crash) or possibly have unspecified other impact via a dumpit operation.

A user application can trigger a kernel crash caused by an unchecked de-referenced pointer in the kernel's "Transparent Inter-Process Communication protocol".

CVE-2016-4805 denial of service, unspecified other impact

Use-after-free vulnerability in drivers/net/ppp/ppp_generic.c in the Linux kernel before 4.5.2 allows local users to cause a denial of service (memory corruption and system crash, or spinlock) or possibly have unspecified other impact by removing a network namespace, related to the ppp_register_net_channel and ppp_unregister_channel functions.

Dangling pointer issue, related to network name spaces.

CVE-2016-4580 kernel-information leak in X.25

The x25_negotiate_facilities function in net/x25/x25_facilities.c in the Linux kernel before 4.5.5 does not properly initialize a certain data structure, which allows attackers to obtain sensitive information from kernel stack memory via an X.25 Call Request.

The vulnerability has the common pattern of a missing initialization of a parameter structure.

CVE-2016-4565 possible kernel-memory write via InfiniBand stack

The InfiniBand (aka IB) stack in the Linux kernel before 4.5.3 incorrectly relies on the write system call, which allows local users to cause a denial of service (kernel memory write operation) or possibly have unspecified other impact via a uAPI interface.

CVE-2016-4485/4486 kernel-information leak in llc/rtnetlink subsystem

The llc_cmsg_rcv function in net/llc/af_llc.c in the Linux kernel before 4.5.5 does not initialize a certain data structure, which allows attackers to obtain sensitive information from kernel stack memory by reading a message.

The rtnl_fill_link_ifmap function in net/core/rtnetlink.c in the Linux kernel before 4.5.5 does not initialize a certain data structure, which allows local users to obtain sensitive information from kernel stack memory by reading a Netlink message.

Improperly initialized parameter structures.

CVE-2016-3841 use-after-free in IPv6 stack

The IPv6 stack in the Linux kernel before 4.3.3 mishandles options data, which allows local users to gain privileges or cause a denial of service (use-after-free and system crash) via a crafted sendmsg system call.

CVE-2016-3707 privilege escalation

The icmp_check_sysrq function in net/ipv4/icmp.c in the kernel.org projects/rt patches for the Linux kernel, as used in the kernel-rt package before 3.10.0-327.22.1 in Red Hat Enterprise Linux for Real Time 7 and other products, allows remote attackers to execute SysRq commands via crafted ICMP Echo Request packets, as demonstrated by a brute-force attack to discover a cookie, or an attack that occurs after reading the local icmp_echo_sysrq file.

Genode's user-level network stacks have no side effects like the sysrq operations on Linux. Even when using the Linux-based LxIP stack on Genode, the vulnerability would remain without any system-global effect.

CVE-2016-3156 denial of service in the networking stack

The IPv4 implementation in the Linux kernel before 4.5.2 mishandles destruction of device objects, which allows guest OS users to cause a denial of service (host OS networking outage) by arranging for a large number of IP addresses.

CVE-2016-2070 network-remote denial of service (div by 0 in TCP/IP stack)

The tcp_cwnd_reduction function in net/ipv4/tcp_input.c in the Linux kernel before 4.3.5 allows remote attackers to cause a denial of service (divide-by-zero error and system crash) via crafted TCP traffic.

CVE-2016-0758 privilege escalation (buffer overflow in ASN.1 protocol stack)

Integer overflow in lib/asn1_decoder.c in the Linux kernel before 4.6 allows local users to gain privileges via crafted ASN.1 data.

Sound and video

CVE-2016-4568 possible kernel-memory write in Video4Linux subsystem

drivers/media/v4l2-core/videobuf2-v4l2.c in the Linux kernel before 4.5.3 allows local users to cause a denial of service (kernel memory write operation) or possibly have unspecified other impact via a crafted number of planes in a VIDIOC_DQBUF ioctl call.

There is a large batch of CVEs related to the sound system (ALSA). The vulnerabilities can be triggered via ALSA's `ioctl` interface. The potential damages are mostly described as denial-of-service. However, use-after-free issues may have further reaching effects.

CVE-2016-4569/4578 kernel-information leak in ALSA timer

CVE-2016-2549 deadlock in ALSA high-resolution timer

CVE-2016-2548 race during list manipulation in ALSA

CVE-2016-2547 use-after-free due to race in ALSA

CVE-2016-2546 use-after-free due to race in ALSA timer

CVE-2016-2545 race during list manipulation in ALSA timer

CVE-2016-2544 use-after-free caused race in ALSA sequencer

CVE-2016-2543 null pointer caused by race in ALSA sequencer

4.3.3 Bugs in the low-level parts of the kernel

The CVEs of this category are especially relevant for Genode because they are potentially related to functionality that lies in the scope of the microkernel or Genode's core component.

CVE-2016-5828 denial of service, or unspecified other impact

The start_thread function in arch/powerpc/kernel/process.c in the Linux kernel through 4.6.3 on powerpc platforms mishandles transactional state, which allows local users to cause a denial of service (invalid process state or TM Bad Thing exception, and system crash) or possibly have unspecified other impact by starting and suspending a transaction before an exec system call.

Power-PC-specific issue about the interplay of execve with transactional memory - not further studied.

CVE-2016-5412 denial of service

arch/powerpc/kvm/book3s_hv_rmhandlers.S in the Linux kernel through 4.7 on PowerPC platforms, when CONFIG_KVM_BOOK3S_64_HV is enabled, allows guest OS users to cause a denial of service (host OS infinite loop) by making a H_CEDE hypercall during the existence of a suspended transaction.

Power-PC issue related to the interplay of virtualization and transactional memory - not further studied.

CVE-2016-5340 privilege escalation

The is_ashmem_file function in drivers/staging/android/ashmem.c in a certain Qualcomm Innovation Center (QuIC) Android patch for the Linux kernel 3.x mishandles pointer validation within the KGSL Linux Graphics Module, which allows attackers to bypass intended access restrictions by using the /ashmem string as the dentry name.

Input validation problem in "Android shared memory" implementation, comparing pointed-to strings instead of pointer values. Mitigation measures would remain without effect.

CVE-2016-4794 use-after-free with unspecified other impact

Use-after-free vulnerability in mm/percpu.c in the Linux kernel through 4.6 allows local users to cause a denial of service (BUG) or possibly have unspecified other impact via crafted use of the mmap and bpf system calls.

Apparently this is a potential memory corruption caused by a race condition (if the "extension" of memory mappings races with the destruction of chunks). The CVE is too light on details to get a good picture.

This bug is in the inner part of the kernel's memory management. In Genode, the corresponding functionality resides within Genode's core component. Core has to deal with the problem of synchronizing the destruction of memory objects or memory mappings with the page-fault handling, which - depending on the used kernel - may have similar issues.

CVE-2016-4440 privilege escalation in KVM

arch/x86/kvm/vmx.c in the Linux kernel through 4.6.3 mishandles the APICv on/off state, which allows guest OS users to obtain direct APIC MSR access on the host OS, and consequently cause a denial of service (host OS crash) or possibly execute arbitrary code on the host OS, via x2APIC mode.

Currently, VMs on Genode/NOVA do not support the virtualization of x2APIC. After a brief review, it remains unclear whether the vulnerable code would reside in the untrusted user-level VMM or in the NOVA hypervisor.

CVE-2016-3961 denial of service

Xen and the Linux kernel through 4.5.x do not properly suppress hugetlbfs support in x86 PV guests, which allows local PV guest OS users to cause a denial of service (guest OS crash) by attempting to access a hugetlbfs mapped area.

Linux as a guest OSes crashes because it expects the presence of a feature (2 MiB mappings) that is unexpectedly not supported by the Xen hypervisor. The problem remains local to the guest.

CVE-2016-3713 denial of service, information leak in KVM subsystem

The msr_mtrr_valid function in arch/x86/kvm/mtrr.c in the Linux kernel before 4.6.1 supports MSR 0x2f8, which allows guest OS users to read or write to the kvm_arch_vcpu data structure, and consequently obtain sensitive information or cause a denial of service (system crash), via a crafted ioctl call.

CVE-2016-3157 denial of service

The __switch_to function in arch/x86/kernel/process_64.c in the Linux kernel does not properly context-switch IOPL on 64-bit PV Xen guests, which allows local guest OS users to gain privileges, cause a denial of service (guest OS crash), or obtain sensitive information by leveraging I/O port access.

This is a bug in the context-switching code in Linux that is specific to Xen PV guests. User applications can crash their underlying kernel. Other Xen guests remain unaffected.

CVE-2016-3070 denial of service (null pointer in memory management)

The trace_writeback_dirty_page implementation in include/trace/events/writeback.h in the Linux kernel before 4.4 improperly interacts with mm/migrate.c, which allows local users to cause a denial of service (NULL pointer dereference and system crash) or possibly have unspecified other impact by triggering a certain page move.

CVE-2016-2847 resource denial of service by using pipes

fs/pipe.c in the Linux kernel before 4.5 does not limit the amount of unread data in pipes, which allows local users to cause a denial of service (memory consumption) by creating many pipes with non-default sizes.

On Genode, there is no pipe-like inter-process mechanism that consumes unaccounted kernel resources on behalf of user programs.

CVE-2016-2550 resource denial-of-service via file descriptor allocation

The Linux kernel before 4.5 allows local users to bypass file-descriptor limits and cause a denial of service (memory consumption) by leveraging incorrect tracking of descriptor ownership and sending each descriptor over a UNIX socket before closing it. NOTE: this vulnerability exists because of an incorrect fix for CVE-2013-4312.

CVE-2016-2143 unspecified impact in page-table handling on s390 platforms

The fork implementation in the Linux kernel before 4.5 on s390 platforms mishandles the case of four page-table levels, which allows local users to cause a denial of service (system crash) or possibly have unspecified other impact via a crafted application, related to arch/s390/include/asm/mmu_context.h and arch/s390/include/asm/pgalloc.h.

CVE-2016-2069 possible privilege escalation (race in TLB flush)

Race condition in arch/x86/mm/tlb.c in the Linux kernel before 4.4.1 allows local users to gain privileges by triggering access to a paging structure by a different CPU.

The result of this race could be a stale mapping present in the TLB, for which no page-table entry exist any longer.

It might be worthwhile to check how the scenario explained in the CVE would relate to the TLB shoot-down protocols implemented by the base-hw and NOVA kernels.

CVE-2016-1583 privileged escalation (ecryptfs causing recursive page faults)

The ecryptfs_privileged_open function in fs/ecryptfs/kthread.c in the Linux kernel before 4.6.3 allows local users to gain privileges or cause a denial of service (stack memory consumption) via vectors involving crafted mmap calls for /proc pathnames, leading to recursive pagefault handling.

An attacker can deliberately trigger recursive page-fault resolution in the kernel, eventually overflowing the kernel stack. The stack flows into an adjacent page, which may be under user control.

This problem does not exist on Genode. Even though NOVA triggers page-faults in the kernel as an optimization to implement a sparse data structure (capability space) such in-kernel page faults are never triggered recursively. Genode's custom base-hw kernel never triggers any page faults in the kernel.

CVE-2016-0823 kernel-information leak by reading pagemap file

The pagemap_open function in fs/proc/task_mmu.c in the Linux kernel before 3.19.3, as used in Android 6.0.1 before 2016-03-01, allows local users to obtain sensitive physical-address information by reading a pagemap file, aka Android internal bug 25739721.

There is no such mechanism on Genode.

CVE-2016-0774 privilege escalation (pipe read/write)

The (1) pipe_read and (2) pipe_write implementations in fs/pipe.c in a certain Linux kernel backport in the linux package before 3.2.73-2+deb7u3 on Debian wheezy and the kernel package before 3.10.0-229.26.2 on Red Hat Enterprise Linux (RHEL) 7.1 do not properly consider the side effects of failed __copy_to_user_inatomic and __copy_from_user_inatomic calls, which allows local users to cause a denial of service (system crash) or possibly gain privileges via a crafted application, aka an "I/O vector array overrun." NOTE: this vulnerability exists because of an incorrect fix for CVE-2015-1805.

The problem lies in the iterative copying of data via "iov" structures where the number of already copied bytes was not tracked correctly.

4.3.4 Vulnerabilities in security-related functions

This category of CVEs highlights the problem that security "features" may introduce critical vulnerabilities due to their complex implementation. In particular the introduction of user name spaces and network name spaces as motivated by the use of containers stand out because they are repeatedly mentioned in several CVEs. There are 3 CVEs regarding the Berkeley Packet Filter (BPF), which is the base mechanism of the seccomp-bpf sandboxing mechanism.

CVE-2016-6187 privilege escalation

The apparmor_setprocattr function in security/apparmor/lsm.c in the Linux kernel before 4.6.5 does not validate the buffer size, which allows local users to gain privileges by triggering an AppArmor setprocattr hook.

Buffer overflow in AppArmor API function.

CVE-2016-6136 audit-log integrity issue

Race condition in the audit_log_single_execve_arg function in kernel/auditsc.c in the Linux kernel through 4.7 allows local users to bypass intended character-set restrictions or disrupt system-call auditing by changing a certain string, aka a "double fetch" vulnerability.

This is a double-fetch issue during process creation (execve), possibly affecting the content of audit logs.

CVE-2016-4997/4998 denial of service, kernel information leak

The compat IPT_SO_SET_REPLACE setsockopt implementation in the netfilter subsystem in the Linux kernel before 4.6.3 allows local users to gain privileges or cause a denial of service (memory corruption) by leveraging in-container root access to provide a crafted offset value that triggers an unintended decrement.

The IPT_SO_SET_REPLACE setsockopt implementation in the netfilter subsystem in the Linux kernel before 4.6 allows local users to cause a denial of service (out-of-bounds read) or possibly obtain sensitive information from kernel heap memory by leveraging in-container root access to provide a crafted offset value that leads to crossing a ruleset blob boundary.

Vulnerability was introduced with network namespaces, may be triggered from the inside of containers.

CVE-2016-4558 refcount overflow in Berkeley Packet Filter

The BPF subsystem in the Linux kernel before 4.5.5 mishandles reference counts, which allows local users to cause a denial of service (use-after-free) or possibly have unspecified other impact via a crafted application on (1) a system with more than 32 Gb of memory, related to the program reference count or (2) a 1 Tb system, related to the map reference count.

CVE-2016-4557 privilege escalation via use-after-free bug in BPF

The replace_map_fd_with_map_ptr function in kernel/bpf/verifier.c in the Linux kernel before 4.5.5 does not properly maintain an fd data structure, which allows local users to gain privileges or cause a denial of service (use-after-free) via crafted BPF INSTRUCTIons that reference an incorrect file descriptor.

Related to https://bugs.chromium.org/p/project-zero/issues/detail?id=808

CVE-2016-4470 use-after-free issue in Kernel keyring handling

The key_reject_and_link function in security/keys/key.c in the Linux kernel through 4.6.3 does not ensure that a certain data structure is initialized, which allows local users to cause a denial of service (system crash) via vectors involving a crafted keyctl request2 command.

CVE-2016-3672 information leak bypassing ASLR

The arch_pick_mmap_layout function in arch/x86/mm/mmap.c in the Linux kernel through 4.5.2 does not properly randomize the legacy base address, which makes it easier for local users to defeat the intended restrictions on the ADDR_NO_RANDOMIZE flag, and bypass the ASLR protection mechanism for a setuid or setgid program, by disabling stack-consumption resource limits.

CVE-2016-3134/3135 privilege escalation (kernel write)

The netfilter subsystem in the Linux kernel through 4.5.2 does not validate certain offset fields, which allows local users to gain privileges or cause a denial of service (heap memory corruption) via an IPT_SO_SET_REPLACE setsockopt call.

Integer overflow in the xt_alloc_table_info function in net/netfilter/x_tables.c in the Linux kernel through 4.5.2 on 32-bit platforms allows local users to gain privileges or cause a denial of service (heap memory corruption) via an IPT_SO_SET_REPLACE setsockopt call.

The vulnerability is present only when network namespaces are enabled.

CVE-2016-2383 kernel-information leak via Berkeley Packet Filter

The adjust_branches function in kernel/bpf/verifier.c in the Linux kernel before 4.5 does not consider the delta in the backward-jump case, which allows local users to obtain sensitive information from kernel memory by creating a packet filter and then loading crafted BPF instructions.

CVE-2016-2085 timing side channel

The evm_verify_hmac function in security/integrity/evm/evm_main.c in the Linux kernel before 4.5 does not properly copy data, which makes it easier for local users to forge MAC values via a timing side-channel attack.

CVE-2016-2053 denial of service (missing input validation in crypto function)

The asn1_ber_decoder function in lib/asn1_decoder.c in the Linux kernel before 4.3 allows attackers to cause a denial of service (panic) via an ASN.1 BER file that lacks a public key, leading to mishandling by the public_key_verify_signature function in crypto/asymmetric_keys/public_key.c.

CVE-2016-0821 ineffective list-pointer poisoning

The LIST_POISON feature in include/linux/poison.h in the Linux kernel before 4.3, as used in Android 6.0.1 before 2016-03-01, does not properly consider the relationship to the mmap_min_addr value, which makes it easier for attackers to bypass a poison-pointer protection mechanism by triggering the use of an uninitialized list entry, aka Android internal bug 26186802, a different vulnerability than CVE-2015-3636.

Poisoning is the approach to overwrite freed pointers with a pattern that yields an unresolvable page fault at a well-known address such that the de-referencing of dangling pointers can easily be detected (in contrast of overwriting such pointers with a null pointer).

CVE-2016-0728 privilege escalation (integer overflow in kernel keyring)

The join_session_keyring function in security/keys/process_keys.c in the Linux kernel before 4.4.1 mishandles object references in a certain error case, which allows local users to gain privileges or cause a denial of service (integer overflow and use-after-free) via crafted keyctl commands.

4.4 Lessons learned from the reviewed CVEs

Device drivers are most vulnerable

The biggest share of vulnerabilities is present in device drivers, which highlights the benefit of component-based OS architectures. On Linux, each of those vulnerabilities affect the entire system (i. e., via a kernel crash) whereas the same code running encapsulated in a Genode component has far less dramatic consequences.

Most device-driver vulnerabilities are triggered by ioctl operations or a certain device behavior. The ioctl attack surface can be effectively reduced by restricting user-level access to devices via mandatory access control frameworks or seccomp.

TCB reduction of Genode's virtualization architecture is effective

As seen in the Xen-related CVEs, bugs in the complex instruction emulator compromise not merely a single VM but the entire system. Thanks to the microkernel approach to virtualization as facilitated by running VirtualBox on Genode/NOVA, critical virtualization-related bugs remain isolated in the untrusted user-level VMM component. The underlying microkernel that establishes the isolation boundaries between components (such as VMs) is relieved from such complexities.

Recurring patterns of information leakage

Most information leaks have the same underlying cause, which lies in improperly initialized parameter structures. The same principle problem exists on Genode, where components communicate structured data as RPC arguments or via shared memory.

Complexity defeats security

Due to their complexity, some mitigation techniques like seccomp-bpf introduce new vulnerabilities as illustrated by the critical bugs found in the Berkeley packet filter. The same observation holds for user-level security features such as network name spaces or user name spaces, which actually introduced new privilege-escalation problems. Xen's support for paravirtualized guests also nicely illustrate this point. The complexities that come with the PV page-table handling and the related optimizations obviously have a price in terms of security.

Mitigation measures turn security disasters into denial-of-service issues

There exists a large number of write-to-kernel, use-after-free, double-free, or buffer-overflow issues. In most cases, the reported consequences are denial-of-service problems. Mitigation measures like kernel ASLR seemingly raise the bar for exploiting those vulnerabilities.

Even time-tested functionalities are not immune

- Bugs in the kernel's memory management
- Buffer overflow in the implementation of pipes
- Vulnerabilities present in the generic VFS code

Estimating the reach of a vulnerability often remains impossible

Many CVEs state "unspecified other impact" because the reach of vulnerabilities is impossible to predict in a monolithic architecture. E.g., bugs in the handling of ICMP network packets may result in a remote attacker issuing sysrq operations. In contrast, a component-based system eases the assessment of the consequences: By assuming a component to be under total control of an attacker, all interactions of the given component with other components can be examined and discussed in detail.

5 Improving the resilience of Genode

This section proposes possible steps to improve the resilience of Genode against attacks. For most of the proposed improvements, automated tests should be able to confirm their effectiveness.

5.1 Address known limitations / uncover unknown limitations

The primary focus should be the elimination of known problems and the systematic analysis of the existing design and code with respect to possible vulnerabilities. An example of a known problem is the unrestricted capability allocation as described in Section 2.1.

Supplementing the current suite of functional tests by systematic bad-case API tests would reveal unknown deficiencies that could be subsequently addressed. The design of bad-case tests could be aided by reviewing the code, paying special attention to bug-prone patterns like the use of pointers, dynamic memory allocations, locking, the use of variable-sized buffers, or `for` loops with indices.

5.2 Infrastructure for random-based mitigation techniques

Several mitigation techniques rely on the availability of an entropy source. In the context of Genode, this raises the following problems:

- We need a sufficiently good but low-complexity random-number generator (PRNG) that can be easily embedded into various parts of the Genode system.
- We need to find a seed for the PRNG in Genode's core component from a random source at boot time (and possibly later).
- We need to find a good way to propagate entropy throughout Genode's component tree. This entropy may be used to seed component-local PRNGs.
- We need to equip low-level data structures like a AVL-based best-fit allocator, the bit allocator, and the heap with new interfaces for incorporating randomness.

5.3 Tool-chain-based protections

Protection mechanisms provided by the current version of Genode's tool chain (based on GCC 4.9.2) could be enabled by default.

- The stack-smashing protection mechanism as discussed in Section 3.1 would obtain its random canary value via the infrastructure proposed in the previous section.
- Fortify source could be enabled for the compilation of libc-based components.

5.4 MMU-based protection mechanisms

The MMU protections explained in Section 3.6 should be implemented:

Data-execution prevention (DEP)

- Support for NX-bit handling in core, NOVA, and the base-hw kernel
- Distinguishing RX and RO segments in Genode's ELF binaries (right now, RO sections and code are located in a single segment)

Separating attached dataspaces by guard pages

As discussed in Section 3.3.5, guard pages help to mitigate heap-buffer overflow attacks.

Improvement of RAM vs. ROM dataspace handling

- Sealing RAM dataspaces to become read-only as proposed in Section 3.6.1.
- Possibly sealing a PD to disallow further executable mappings, thereby preventing the smuggling of new code into the running program,
- Allow RAM dataspace to be defensively attached as read-only mappings.

Dynamic linker

- Use eager binding (aka "bind-now") of the PLT and read-only jump-slot relocations by default,
- Lazy binding: using a random-located r/w shadow mapping for editing the PLT, while using a r/o mapping for the actual use by the application

5.5 Mitigating cold-boot attacks

By applying the clearing of dataspaces and heap allocations as discussed in Section 2.2.4, Genode would simplify the defense against cold-boot attacks. E.g., a developer of a component that processes sensitive information would not need to manually clear credentials from memory but would simply destroy the component or free the buffer that was tainted with sensitive information.

5.6 Address-space randomization

The various ideas presented in Section 3.4 could be implemented based on the infrastructure outlined in Section 5.2.

5.7 ELF-binary randomization

The implementation of the ELF-binary randomization idea as presented in Section 3.4.8 would effectively eliminate ROP-based attacks. The approach could be applied at two stages:

- At the bootstrapping of the kernel/core, the bootstrapping code could shuffle all ELF binaries that are present as boot modules.
- A ROM service component could transparently apply the shuffling of ELF images that are obtained by other (higher-level components) as ROM modules.

Note that this approach is not a proven path. So there are practical risks such as the complexity of the architecture-specific instruction decoding and the rewriting of jump targets (considering the constraints of PC-relative addressing). However, if successful, this technique would advance Genode's resilience far beyond the state of the art of mitigation techniques.

5.8 Heap protection

Implementation of the ideas presented in Section 3.3.5.

5.9 Tools for hardening the implementation

To further reduce the likelihood for the first stage of privilege escalation attacks and thereby to support the presumption that the kernel/core is not attackable, we would devise the creation of new tools that statically analyse Genode's code base. The analysis will aid the hardening of the implementation with respect to typical bugs. The tools should become a regular part of Genode's continuous test-and-integration infrastructure to make sure that the quality of the code does not degrade in the future. In the longer term, the effort could be cultivated towards the selective application of formal methods and model-checking techniques to the framework's most central code.

Examples of properties to maintain via static analysing tools are:

- Compile-time prevention of information leaks via RPC arguments as described in Section 3.10.2,
- Revealing undocumented object ownership issues by enforcing annotations of reference arguments,
- Uncovering memory leaks by enforcing an annotation at the allocation site that explains where/how the object is freed,
- Detecting possible integer overflows (e. g., asserting the absence of shift operators from the code base)
- Banning of pointer arithmetics except for a few well-documented places,
- Detection of possible out-of-bounds array accesses (possibly banning C-style arrays from the code of regular components),
- Detecting unbounded recursion,
- Spotting argument-validation issues