Safety Requirements for a Generic Linux System

1. Index

1. Index	**2**
2. Documentation Control	**3**
3. Terms and Abbreviations	**4**
4. References	**4**
5. Purpose of the document	**4**
6. Structure of the document	**4**
7. Requirements on the Generation of Binary Artefacts	**5**
8. Components Safety Integrity Level	**6**
9. Components FFI	**11**
10. Spatial FFI requirements for Linux components	**18**
11. License: CC BY-SA 4.0	**23**

2. Documentation Control

Item	**Description**
Title	Generic Safety Requirements for a Linux System
Author(s)	Igor Stoppa
Reviewers List	Aingara Paramakuru Deepak Nibade Eli Gurvitz Hadi Zibaeenejad Lauri Ora Michal Szczepankiewicz Raghavendra Vishnu Kumar Raman Kishore Sanjay Trivedi Shrife Mohamed Vito Magnanimo
Revision	1.0
Date	Jan 17, 2024

3. Terms and Abbreviations

See the matching section in Interference Scenarios for an ARM64 Linux System

4. References

4.1. Interference Scenarios for an ARM64 Linux System

4.2. CC BY-SA 4.0 Deed | Attribution-ShareAlike 4.0 International | Creative Commons - https://creativecommons.org/licenses/by-sa/4.0/ License

5. Purpose of the document

This document lists Safety Requirements for a computer system, on the assumption that the Operating System is Linux.

The reference document Interference Scenarios for an ARM64 Linux System, explores some of the possible failure modes and their implications.

However, this document will not address safety requirements related to failure modes, because the mandated behaviour in presence of a failure is tightly coupled with a specific system design, its use cases and the related requirements.

Instead, the present document revolves around the more generic aspects of safety-related requirements, for a Linux system.

These requirements might not even need to apply, if the Linux system is part of a larger one that employs, for example, redundancy, to address some of the safety issues.

However, even in that case, the requirements mentioned here should be considered and explicitly addressed. Even if only to say that they do not apply, because of redundancy.

6. Structure of the document

- The next three sections refer to high level requirements for the use of an operating system in a safety scenario.

- They are followed by a section which is specific to Linux and its architecture.

7. Requirements on the Generation of Binary Artefacts

This section covers the non-source-code aspect of the analysis and the requirements. In general the requirements are applied to source code, however in practice the source code needs to be converted into machine executable instructions and here there is a possibility for introducing unsafe behaviours, due to the multitude of parameters available when performing the conversion from source to binary.

Modern toolchains can go even beyond that and inject whole new functionality that is completely invisible, should the analysis stop at sources.

Anti ROP/JOP mechanisms are very common examples of code injection happening during compilation.

7.1. Requirements:

7.1.1. The toolchain must be qualified according to its specific set of requirements and qualification criteria.

These toolchain-oriented requirements are meant to ensure that the toolchain will generate binary artefacts aligned with the safety integrity level requirements.

The safety standards define criteria for both artefacts and toolchain.

7.1.2. The parameter values passed to the toolchain must be compatible with the configuration that was used for the qualification.

7.1.3. Provide evidence of the qualification of the toolchain and any other tool involved in the generation of the binary artefacts.

7.1.4. Provide evidence that the configuration used for the toolchain is compatible with the one used for its qualification.

7.1.5. In case it is required to use the toolchain outside of the parameters space compatible with the existing qualification, the qualification must be updated accordingly.
This is quite undesirable, but if it is the only viable solution, then the toolchain qualification must be updated.

8. Components Safety Integrity Level

8.1. Scope

Qualification at safety integrity level "X", for selected (parts of) user processes and selected (parts of) kernel drivers.

8.2. Assumptions:

8.2.1. Components can belong to any of the following classes:

8.2.1.1. User-space processes

8.2.1.2. Kernel device drivers

8.2.1.3. Kernel threads

The selection criteria for assigning safety classification to various components is determined when tailoring the requirement template to a specific system, with its related use-cases.

As a result of the safety analysis and decomposition, certain requirements might be assigned only to parts of a component; (e.g. a specific kernel or user-space thread that is in charge of a safety relevant aspect).

However, it makes sense to consider sub components if and only if it can be proven that they are not exposed to interference from other parts of the same component (e.g. all user-space threads belonging to a certain process share the same address space and data).

8.3. Primary Requirement:

Provide evidence that the selected components are meeting the criteria associated with the specified safety integrity level.

8.4. Derived Requirements:

8.4.1. Safety requirements for a user-space process extend to both user-mode and kernel-mode execution.

A user-space process typically executes in user-mode, however it needs sometimes to invoke the kernel, through syscalls, for performing activity at a higher level of privilege.

The safety requirements assigned to the user-space process extend also to this condition, while it is executing a syscall in kernel mode.

When addressing requirements allocated to a certain user-space process, also its kernel-mode execution must be accounted for.

8.4.2. Safety requirements assigned to kernel drivers can extend to both kernel threads and work queues utilised: a device driver with safety requirements might need to execute work either through a kernel thread or a workqueue or similar means of work deferral.

Safety requirements assigned to such a device driver shall extend by default to any other forked execution path, should the decomposition determine that it is required.

For example, the device driver might rely on a kernel thread to interact asynchronously with a hardware component that would otherwise introduce too much latency, if it was accessed synchronously.

In case any safety requirements apply to said interaction, they will extend also to the kernel thread involved.

Should the driver perform additional non safety-relevant activity (e.g. logging), the safety requirements will not need to apply to such activity, synchronous or not. In these cases, it is intended that said activity is not part of any safety mechanism or argumentation, for example because there are other means in place that are responsible for detecting and advertising safety-relevant anomalies.

8.4.3. If/when considering the **“Proven in Use”** argumentation, it is important to substantiate the claims of equivalence between the system already in use, which is considered qualified as safe, and the system under analysis, which must be qualified.

The burden of proof for any and all conditions associated with this sort of claim rests exclusively on the subject making the claim.

Various perspectives need to be considered:

8.4.3.1. Generation of Binary Artefacts

The current toolchain must match the qualifications of the old one. Ideally it would be the same, but equivalence, if proven, is sufficient.

Furthermore, it must be used with parameters that yield equivalent binary artefacts (changing optimisation level between reference and target builds would significantly reduce the confidence of binary equivalence). Here too, the equivalence of the parameters used, between the reference system and the system under analysis, must be proven.

8.4.3.2. Hardware Configuration

The “proven in use” argumentation relies on historical data. Such data must have been obtained from a set of systems based on homogenous hardware, and there must be evidence of this. Similarly, the system under analysis must be sufficiently similar to the abovementioned fleet. This too must be proven. Lacking evidence for this uniformity, it is not possible to utilise whatever historical data might have been extracted from the flet during its operations.

8.4.3.3. Source Code

Just like there is a need for hardware homogeneity across the reference fleet and between the fleet and the system under analysis, the same need is extended to the software employed. Likewise, evidence must be provided for the equivalence.

8.4.3.4. Timing Perspective

The need for homogeneity mentioned in the two previous points about hardware and source code extends to activation patterns and requirements as well: it must be proven that hardware and software of the system under analysis shall be excited in the same way as the reference configuration was. Or in a way that is proven to be equivalent.

The equivalence refers to pattern, frequency, duration and overlapping of events. Not just events relevant to safety aspects, but any event that can affect the system.

Furthermore, latency requirements must be equivalent.

8.4.3.5. Memory Layout

The following observations are particularly important in case there are no hard mechanisms to ensure either memory integrity or detection of memory corruption for safe components.

The fact that the fleet of reference devices didn't seem to exhibit memory corruption, even after extensive use, doesn't automatically mean that it was free from corruption.

Corruption might have happened even in the fleet of devices used as reference, but the specific memory layout of various allocations might have acted as a buffer toward components with safety requirements, *thus making the interference irrelevant to that specific scenario.*

Lacking hard barriers, either it is proven that the memory layout of the new system is equivalent, or the **"proven in use argumentation" won't be directly applicable**.

8.4.4. Testing must adequately represent the use cases, as they are described.

Any deviation from the use cases scenarios must be documented and justified.

This is particularly relevant for the "Proven in use" argumentation, where the test cases of the reference implementation must be proven to be valid, representative and exhaustive of the new scenario.

Any gap must be addressed, until the required test coverage is reached.

Any intentional omission or deviation must be justified.

8.4.4.1. An extensive amount of focused testing, designed according to safety cases, and intended as another way of obtaining a large body of empirical data, is another type of argumentation that requires special attention, similarly to "Proven in use".

While it can make more robust claims due to the likelihood of relying on data that is more aligned with present needs, it still shares some of the inherent deficiencies:

8.4.4.1.1. Most likely data is not collected on exactly the same hardware/software across all the testing units, especially if the testing campaign happens in parallel with development.

It needs to be proven and documented that the data used for the argumentation is from a setup sufficiently similar to the final one.

8.4.4.1.2. Testing conditions must meet the same level of variance that is expected to be found in the field, including variance of input parameters and permutations of their activation sequence and timing.

Lacking this, it is impossible to rule out that once deployed, the system will not encounter unforeseen operating conditions, under which it can fail and compromise the safety claims.

8.4.4.1.3. Since the testing data needs to be collected, and inputs must be automated, it must be proven that the testing harness(es) responsible for these activities do not anyhow alter the test conditions, to the point that the system being tested is not representative anymore of the system being released.

This point might seem obvious, but it is necessary to analyse thoroughly various operating parameters.

For example, extensive monitoring and logging can completely alter the inner timing of the system, preventing it from entering lower power states of idleness, also associated with longer latency and possibly the ignoring of certain inputs.

But this is just one example, and the analysis must cover the whole range of testing-induced alterations in the regular behaviour.

9. Components FFI

9.1. Scope

Handling of spatial and / or temporal interference at safety integrity level "X", for selected (parts of) user processes and selected (parts of) kernel drivers.

The same considerations made in the previous chapter, about decomposition, allocation of requirements to sub-components and their freedom from interference apply here as well.

9.2. Assumptions:

9.2.1. The "selected components" mentioned are expected to be individually qualified at a certain safety integrity level "X".

9.2.2. Components can belong to any of the following classes:

9.2.2.1. User-space processes

9.2.2.2. Kernel device drivers

9.2.2.3. Kernel threads

9.2.3. The selection criteria for assigning safety classification to various components is determined when tailoring the template to a specific system, with related use-cases.

9.2.4. The system also comprises components that are considered to be at a lower safety integrity level (including non safety-qualified ones), and that is accepted, because they are not involved in the above mentioned use cases, and over-qualifying them would be both more expensive and more cumbersome to manage (longer qualification times, etc).

9.2.5. Components at a safety integrity level X are theoretically exposed to interference, both spatially and temporally, from other components at a lower safety integrity level.

9.2.6. The interference considered can originate from any other component classified at a lower safety, including those which are not formally qualified, but are simply quality-managed.

9.3. **Primary Requirements:**

9.3.1. Provide evidence that, for the selected components at safety integrity level “x”, any interference, both spatial and temporal, originating from components at a lower safety integrity level shall always be either detected or prevented.

9.3.2. Depending on the system or application, the requirement of coping with a failure can translate into either prevention or detection:

9.3.2.1. **Prevention:** the system shall not allow the failure to happen, which means that there won’t be undetected interference, because the interference will be avoided.

Here the associated latency requirement in responding to the failure might be relaxed, with less strict demands on the reaction time, because the safety of said components is not compromised.

Additional actions might be specified, like notification of the attempted failure.

Any mechanism that provides prevention from interference, when itself subject to interference, must either not cascade it, even if affected, or it must be immune (i.e. it must be itself free from interference).

9.3.2.2. **Detection:** the failure **can** happen, but it shall be detected.

Since the failure is not prevented, it must be assumed that the component stops operating in a safe way right when the failure occurs.

It therefore becomes very important to comply with the timing requirements associated with the detection, that will require performing some specified action, that is expected to take the system back to a safe state - whatever might be the definition of safe state in the specific context.

It is expected that the detection mechanism won’t be affected by the very same failure that it must reveal.

The choice between prevention, detection, recovery is driven by both availability requirements and intrinsic latency, specific to the system at hand.

It is therefore not possible to say upfront whether hanging or any other form of unresponsiveness can be considered as a safe behaviour (provided that there is some form of recovery mechanism that will take the system to a safe state).

But, whatever the safety classification of hanging in a specific use case, it must be justified.

Availability itself has its own safety integrity level requirements.

Such requirements are conditioned to the safety integrity level requirements assigned to freedom from interference, since it would be impossible to ensure availability at a safety integrity level higher than the one assigned to freedom from interference.

.

9.4. Derived Requirements:

9.4.1. Provide evidence that the detection mechanism itself is qualified at the same-or-higher safety integrity level that it is expected to support in other components.

9.4.2. Refer also to the requirements associated with the previous chapter **“Components Safety Integrity Level”.**

9.4.3. In case multiple safety integrity levels are part of the scenario, provide evidence that mixed-criticality is supported, detecting for each safety integrity level any interference that might originate from lower safety integrity levels.

9.4.4. The exact set of targets for this requirement must be defined through analysis and decomposition of the various use cases.

9.4.5. Where detection is not feasible, it can be replaced by prevention, even if only detection was required, but the prevention must be performed at the same-or-better safety integrity level than what was initially required for the detection.

Depending on the specific use case or finer grained requirement, it might be either impossible, unfeasible or impractical to prove detection of certain types of interference.

In such cases it is also acceptable to substitute detection with prevention, as long as it happens at the same or higher safety integrity level specified by the requirement at hand.

It is thus acceptable to implement the prevention to a certain safety integrity level "X", and then claim that anything that is at a lower safety integrity level, it simply cannot cause interference, because that interference is already prevented.

Of course, proof is required that the prevention is indeed happening at said safety integrity level "X".

9.4.6. In case interference is mitigated through countermeasures focused on protecting safety-relevant context, it must be detailed which classes of interference are covered by such countermeasures, and how the mitigation happens.

9.4.7. In case interference is mitigated through countermeasures focused on addressing potential sources of interference, **each** source must be listed and exhaustively detailed, together with its countermeasure.

Should a source of interference affect multiple targets in different ways, requiring different countermeasures, the description process must be iterated **for each** different countermeasure.

9.4.8. Functional/safety decomposition for spatial partitioning must be backed by a mechanism that can ensure the partitioning will not be violated silently, and such mechanism must have a sufficient safety qualification.

For example, architectural decomposition alone is **not** sufficient for claiming prevention from interference between the subsystems identified. It needs to be backed by some form of safety-compliant way of either detecting or outright denying any attempted violation of the partitioning.

9.4.9. Temporal partitioning (e.g. checkpointing) is a valid alternative, where applicable, but it must be verifiable that certain potential sources of interference can be neglected, because they cease to exert any relevant influence before they can actually interfere.

9.4.10. All (classes of) sources of interference considered must be listed, then described in terms of:

9.4.10.1. Possibility of actually generating any interference: if the triggering event cannot take place at all during a certain use case, then the interference is not relevant for said use case.

This aspect must be considered and made explicit.

If it is not possible to determine the triggering event, then the interference must be taken for granted.

9.4.10.2. Potential/probable targets of the interference, based on exposure.

For example, considering a uniform probability of interference across the whole address range, a large data set has higher chances of being hit by an interference, compared to a single variable, because it has a larger exposure.

The criticality of the target needs to be addressed as well.

9.4.11. The previous point should lead to considerations about tradeoffs in using kernel components with low safety integrity level to attempt improving the safety of user-space components.

9.4.11.1. For example, what is the safety integrity level of security mechanisms used frequently in the kernel, and what is the trade-off, if they turn out to be a source of interference?

They can easily generate cascaded interference, because of the frequent complex operations they have to perform (e.g. SELinux growing/shrinking its AVC cache.)

9.4.11.2. Similarly, what is the safety integrity level of frequently used memory managers, and how can they generate interference?

9.4.12. It’s worth mentioning the case of software components with lower safety integrity level, which must not be allowed to interfere, either directly or indirectly, with other components at higher level, through shared hardware components.

This case is particularly important, because it can be quite common to have a shared hardware bus (e.g. I2C, SPI, PCI, etc.) which is populated by hardware components with different safety allocations.

While the individual components might be at different safety integrity levels, the bus itself must inherit the strictest safety requirements from the peripherals it connects.

9.4.13. The kernel sources include also a set of stock library functions (e.g. list management, locking, maths, etc.) which are used ubiquitously, and are therefore certain to appear in any safe context.

At the very minimum, those library functions which are used in a safe context must be qualified accordingly.

These library functions are typically stateless, so this point is not directly referring to the integrity of kernel data structures (which deserves a separate discussion), but rather the qualification of common code that must be compatible with the stricter safety requirements associated with any of its users.

9.4.14. Certain hardware components have the ability of acting as masters on the memory bus, completely sidestepping the mechanisms that the MMU employs to ensure hardware isolation (e.g. a video codec with DMA capability).

These components constitute a remarkable hazard, if they are not qualified to the highest safety integrity level required by the use cases being considered.

Therefore, the requirements shall encompass both the device drivers of said memory bus masters, and their respective configuration mechanisms.

Typically, but not always, the hazard is mitigated through the presence of an IO/MMU, which can limit the reach of memory bus master peripherals.

In such cases, the requirements do not need to be extended to the peripheral device drivers (unless they are involved for some other reason with safety use cases).

However, it is now the IO/MMU device driver and control mechanisms that inherit safety requirements, because they affect how effective the protection from memory bus master peripherals shall be.

Should any device have the capability of bypassing the IO/MMU, it falls back to the previous scenario, because the device is an unbound bus master.

10. Spatial FFI requirements for Linux components

10.1. Initialisation phase

10.1.1. Assumptions

10.1.1.1. Linux has a very complex initialisation sequence, which comprises a wide range of features, functions and components.

10.1.1.2. Because of its high level of optimisation, it is possible that some late stages of the kernel initialisation happen in parallel with the initialisation of user processes.

10.1.1.3. Unless stated otherwise, the initialisation mentioned below is system initialisation, not just kernel.

10.1.1.4. From a safety perspective, the initialisation is equally exposed to interference as the regular execution; perhaps even more, because during initialisation there is a peak of activity that is unlikely to be found during regular operations.

10.1.1.5. It is assumed that the system is unable to cause safety hazards until the initialisation is complete, by design, as safety functions are not active. This is a reasonable assumption in the vast majority of applications. Special cases can be dealt with as exceptions. The goal is that even those can fall back on the assumption previously described.

10.1.2. Primary Requirements

10.1.2.1. Post initialisation, all those components with allocated safety requirements must be proven to be in a safe state. Ideally it's an operational safe state, reached either by not having incurred any safety violations (e.g. data corruption or misconfiguration) or by having recovered from them. Depending on specific requirements about availability, it might or might not be allowed to resort to having either an idle or inactive fallback safe state.

10.1.3. Derived Requirements

10.1.3.1. Ideally, it would be possible to assert FFI through the entire initialisation phase, however it is also acceptable to assess, when the system boots, that the system is now operating within the boundaries of the safety requirements.

Assuming the validation is sufficiently exhaustive, it is acceptable to assume that, if the validation clears, whatever other interference might have taken place prior to the initialisation is neglectable.

10.2. Safety classification of individual Linux components

10.2.1. Assumptions

10.2.1.1. Linux is a monolithic kernel, architecturally composed by a large variety of components and subsystems.

10.2.1.2. In terms of exposure to mutual interference, each and every component may interfere with each other, where there is no safeguard in place (e.g. MMU barriers) that can ensure either detection or protection, whatever might be specified by the requirements for the actual system and use cases.

10.2.1.3. As discussed in the document Interference Scenarios for an ARM64 Linux System, certain components / features are particularly critical and their integrity is instrumental in supporting claims to safety, especially when it is not immediately obvious when they might be corrupted.

10.2.2. Primary Requirements

10.2.2.1. Evidence must be provided that each and every component within the Linux kernel can be considered QM.

10.2.2.2. Components with allocation of safety requirements have therefore constraints of freedom from temporal interference, to which they must adhere.

10.2.2.3. Non - exhaustive list of components and features for which it is necessary to assess safety integrity level and compatibility with safety requirements allocated to the overall system:

10.2.2.3.1. Call stacks used during syscalls of safe contexts.

10.2.2.3.2. Call stacks used during interrupts related to safe contexts.

10.2.2.3.3. Call stacks of exceptions related to safe contexts.

10.2.2.3.4. Linear mappings: memory used by either kernel threads or user-space processes with safety requirements.

10.2.2.3.5. IPC mechanisms (e.g. shared memory, pipes, etc) used by processes with allocation of safety requirements.

10.2.2.3.6. Buddy allocator

10.2.2.3.6.1. Memory allocations provided by the buddy allocator must be compatible with the safety integrity level of the requestors.

Given a requestor with requirements allocated for a specific safety integrity level and for a specific availability, the memory allocations it receives must satisfy either the same requirements or stricter ones.

10.2.2.3.6.2. The buddy allocator must not cause cascaded interference, for example by lending memory that is already allocated, as an effect of an interference affecting its internal metadata, that it uses to keep track of existing allocations.

10.2.2.3.7. Slab allocator
The previous requirements on the buddy allocator must be applied also to the slub allocator.

10.2.2.3.8. Vmalloc allocator
The previous requirements on the buddy allocator must be applied also to the vmalloc allocator.

10.2.2.3.9. Static allocations

Link-time memory allocations must be compatible with the safety integrity level of their respective users.

10.2.2.3.10. Other allocators that might be used as part of a safety relevant use case and thus might have allocated safety requirements (e.g. genalloc, memblock, cma_alloc).

10.2.2.3.11. Page tables

10.2.2.3.11.1. Page tables in use by the Linux kernel.

Since the kernel page tables serve a multitude of components with different levels of safety requirements, they must conform with the highest safety integrity level present.

10.2.2.3.11.2. Page tables used by processes with allocation of safety requirements.

10.2.2.4. The previous list must be completed by the entity responsible for supplying the Linux kernel involved in the related safety-relevant use cases.

10.2.3. Derived Requirements

10.2.3.1. Taking as example the call stack of the syscall performed by a user-space process with allocation of safety requirements, it is not sufficient to rely on architectural decomposition for making claims of freedom from interference from other components at lower safety integrity level.

It needs to be backed by a mechanism that can actually either detect or prevent the interference.

10.2.3.2. Since the mentioned allocators are by default QM, and exposed to interference from other QM components within the kernel, it needs to be proven how they can support claims to higher safety integrity levels made on users of the memory allocations provided.

10.2.3.3. Both the buddy allocator and the vmalloc allocator are involved in memory allocation for both kernel-side and user-space needs (for user-space, it’s the backing of malloc).

Therefore, they have even more direct opportunities to cause interference. This too must be addressed.

11. License: CC BY-SA 4.0