NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 6 Issue 1, Pg. No. 60-64, AIJCST-V6I1P106, 2024
https://doi.org/10.63282/3117-5481/AIJCST-V6I1P106

Original Article

Improving Observability and Stability in Wayland-Based Compositors: Lifecycle Logging, Buffer Validation, and Crash Hardening in Production Display Stacks

***Rajasekhar, Sunkara**
Independent Researcher, USA.

Abstract:

Production display stacks on consumer streaming and smart display devices are commonly built on top of Wayland, Weston, and the Linux Direct Rendering Manager and Kernel Mode Setting subsystems. These platforms are mature and well-documented, but the integration of a Wayland-based stack into a shipping consumer device introduces specific failure modes around shared memory buffer handling, display connection lifecycle, and event ordering across the compositor, the shell, and the graphics abstraction layer that the device uses to expose graphics services to applications. This paper presents a set of engineering interventions applied to a shipping consumer device platform, organized around two themes. The first theme is stability hardening. Specific defects addressed include null pointer dereferences in the compositor shell, a screenshot subsystem crash caused by inadequate validation of shared memory buffers, and a display connection teardown segmentation fault. The second theme is observability. Specific interventions include the addition of monotonic sequence numbering to graphics event logging and the refactoring of buffer lifecycle logging to accelerate root cause analysis. The paper also describes the implementation of window management APIs covering remote window detection, screen constraints, and dual-interface pointer event handling, together with the C++ unit and integration test scaffolding that accompanied each change. The contribution is a set of patterns that transfer to other Wayland-based embedded display platforms, with emphasis on the practical operational concerns that frequently differentiate a shipping product from a reference implementation.

Keywords:

Wayland, Weston, Compositor, Display Server, Embedded Linux, Smart Display, Streaming Device, DRM, KMS, Shared Memory Buffer, Observability, Crash Hardening, C Plus Plus, Integration Testing.

Article History:

Received: 20.11.2023

Revised: 22.12.2023

Accepted: 05.01.2024

Published: 16.01.2024

1. Introduction

Wayland is a display server protocol developed at freedesktop.org. Weston is the reference compositor for the protocol. Together with the Linux Direct Rendering Manager and the Kernel Mode Setting subsystem, they form the foundation of a modern Linux display stack. The stack is widely used in embedded contexts including automotive in-vehicle infotainment, industrial control panels, kiosks, and consumer streaming and smart display devices.

When the stack is integrated into a shipping consumer device, additional layers are added on top of the reference. A product-specific compositor or compositor shell extends the reference compositor with the policy decisions and the user interface elements specific to the product. A graphics abstraction layer is added to expose graphics services to applications written against a higher-level

API than the raw Wayland protocol. A display manager handles the lifecycle of display connections in the device-specific way the product requires. These layers are where most of the defects relevant to a shipping product appear, because they are where the well-tested reference code meets the specifics of the product, the hardware, and the application ecosystem the product hosts.

This paper describes a set of engineering interventions applied to such a product-specific stack. The interventions are organized around two themes. The first is stability. The interventions in this theme address specific defects that caused process crashes during normal device operation. The second is observability. The interventions in this theme address the cost of diagnosing defects in the stack by making it cheaper to read and interpret the logs the stack produces. The paper also covers the implementation of a set of window management APIs and the test scaffolding that accompanied the changes.

The rest of the paper is organized as follows. Section 2 summarizes the Wayland-based stack architecture for readers who are not familiar with it. Section 3 covers the stability interventions. Section 4 covers the observability interventions. Section 5 covers the window management APIs and the test scaffolding. Section 6 discusses the patterns that transfer to other Wayland-based embedded platforms. Section 7 concludes.

2. Background: The Wayland-Based Display Stack

2.1. Wayland and Weston

Wayland is a protocol for communication between a display server and its clients. The display server is referred to as a compositor, because it composes the contents of the connected clients into the final image displayed on screen. Weston is the reference implementation of a Wayland compositor. It is intentionally minimal and is suitable for embedded and mobile use cases. In an embedded product, the product team often uses Weston or libweston as the foundation for a product-specific compositor that extends the reference with product-specific shells and policy.

2.2. DRM and KMS

The compositor talks to the display hardware through the Direct Rendering Manager and the Kernel Mode Setting subsystem in the Linux kernel. KMS handles mode setting on the display output. DRM provides the buffer management interface that the compositor uses to submit rendered frames for scanout. Rendering itself is commonly performed with OpenGL ES through EGL, with the resulting buffers shared with the compositor through the Linux dma-buf mechanism.

2.3. Buffers and Shared Memory

Clients send their rendered content to the compositor in the form of buffers. The buffer can be a shared memory buffer described in the Wayland protocol or a more efficient dma-buf shared between the client and the compositor without copy. The compositor is responsible for tracking the lifecycle of these buffers, releasing them back to the client when it is finished with them so that the client can reuse them, and refusing to access buffers that have been released or that are otherwise invalid. Buffer lifecycle bugs are a common source of defects in a Wayland-based stack.

2.4. The Product-Specific Layers

On top of this reference foundation, the product-specific stack adds at least three layers. The compositor shell implements the policy decisions about how windows are arranged, how input is routed, and how the user interface elements that are not client windows are rendered. The graphics abstraction layer exposes graphics services to applications written against a higher-level API. The display manager handles the lifecycle of display connections in the device-specific way the product requires. The interventions described in this paper are applied at these layers.

3. Stability Interventions

This section describes the stability interventions applied to the product-specific layers. Each intervention addresses a specific defect that caused process crashes in shipping devices or in pre-release testing.

3.1. Null Pointer Dereferences in the Compositor Shell

The compositor shell receives objects from a number of upstream sources, including the reference compositor library, the client connections, and the input subsystem. Some of these objects are nullable in the relevant interfaces and the shell code did not

consistently check for null before dereferencing them. A subset of the resulting null pointer dereferences caused crashes that were observed in production telemetry.

The intervention was to add explicit null checks at the points where the shell first receives the relevant object, and to log a structured event when a null is observed so that the rate of nulls can be tracked over time. The intervention is not a behavioral change for the cases where the object is non-null. It is a guard that prevents the dereference from being attempted when the object is null, and that surfaces the unexpected null in the logs in a form that is easy to query. The fix is shallow in code volume but high in stability impact, because each crash that is prevented is a process restart that is avoided and a customer-visible event that does not happen.

3.2. Screenshot Subsystem Crash on Unvalidated Shared Memory Buffers

The screenshot subsystem reads frame buffer contents through shared memory buffers exported from the rendering side. The subsystem assumed that the buffer attributes received from the producer were consistent with the buffer itself. Under specific timing conditions, this assumption was violated. The subsystem then attempted to read past the end of the actual mapped region, which caused a crash.

The intervention was to add explicit validation of the buffer attributes against the size and stride of the mapped region before any read is performed. If the attributes are inconsistent, the subsystem now logs a structured event and aborts the screenshot operation, rather than attempting the read. The intervention covers the case where the producer has changed the buffer between the time the attributes were posted and the time the consumer attempts to read it, which is the specific race that produced the crashes in production.

3.3. Display Connection Teardown Segmentation Fault

Display connections in the product-specific display manager are torn down both during expected events such as a user-initiated shutdown and during unexpected events such as a hot unplug of a display output. The teardown path had a defect in which a destructor was invoked on an object that had already been freed in a different teardown branch. The result was a segmentation fault that was difficult to reproduce because it required a specific ordering between two asynchronous events.

The intervention was to consolidate the ownership of the affected objects so that the teardown branches no longer competed for the right to destroy them. The fix is a refactoring rather than a guard. The general principle is that lifecycle defects of this shape are best addressed by making the ownership unambiguous, rather than by adding additional checks to each teardown path. The structured event logging introduced in the observability work described in Section 4 made the original defect tractable to diagnose, because the sequence of events leading up to the fault could be reconstructed from the logs.

4. Observability Interventions

Diagnosing defects in a display stack is hard because the events involved are concurrent, the time scales are short, and the relevant state is spread across multiple processes. This section describes two interventions that reduce the cost of diagnosis.

4.1. Monotonic Sequence Numbering

Graphics events were previously logged with timestamps but without a sequence number. When the same logger was used from multiple producers, the resulting log sequence could be ambiguous about the order in which the events occurred, especially when timestamps tied. The intervention was to add a monotonic sequence number to each logged event, assigned at log time. The sequence number is global within the process and is strictly increasing across all events from all producers. This removes the ambiguity. The cost is one atomic increment per logged event, which is negligible. The benefit is that any reader of the logs, whether human or automated, can reconstruct the exact order of events without having to reason about timestamp ties.

4.2. Refactored Buffer Lifecycle Logging

Buffer lifecycle defects are common in Wayland-based stacks. The reference protocol prescribes a contract between the client and the compositor about who owns the buffer at each point in time, and bugs in either side of this contract produce visual artifacts or crashes. The previous logging in the product-specific stack made it difficult to follow a single buffer through its lifecycle because the logs for a given buffer were scattered across different log levels and were not consistently keyed on a buffer identifier.

The intervention was to refactor the logging so that every event involving a buffer is logged at a consistent level, with a consistent key, and with the full lifecycle state of the buffer at the time of the event. The result is that a diagnostic engineer can extract the lifecycle of a single buffer from the logs by filtering on its identifier. This converts what was previously a manual cross-reference exercise across multiple log files into a simple filter. The change does not introduce new log volume in production because the affected log statements were already emitted. It re-targets the existing log statements so that they collectively tell a coherent story about each buffer.

5. Window Management APIs and Test Scaffolding

In parallel with the stability and observability work, a set of window management APIs were implemented in the product-specific layers. This section describes these APIs and the test scaffolding that accompanied them.

5.1. Remote Window Detection

Remote window detection allows the shell to recognize windows that are sourced from a remote rendering process rather than from a local client. This distinction matters for input routing and for the policy decisions about whether a window can be raised, lowered, or moved. The API exposes a query interface to other components of the shell that need to make these decisions, and an event interface for components that need to react to the appearance or disappearance of remote windows.

5.2. Screen Constraints

Screen constraints allow the shell to express limits on how a client window can be sized or positioned. The API exposes constraint primitives that the shell can apply to a client window. The compositor enforces the constraints when the client attempts to commit a size or position that violates them. The constraints are advisory in the sense that they describe what the shell wants to permit; the client is not required to know about them.

5.3. Pointer Event Handling with Dual-Interface Support

The shell supports two interfaces for receiving pointer events from clients. The intervention was to add handling code that accepts pointer events through both interfaces, normalizes them into a single internal representation, and routes them to the appropriate consumer. The dual-interface support is necessary because the application ecosystem on the platform includes clients written against the older interface and clients written against the newer interface, and the shell has to accommodate both.

5.4. C++ Unit Tests and Integration Test Applications

Each of the APIs described above was accompanied by C++ unit tests that exercised the API in isolation, and by integration test applications that exercised the API in the context of a running compositor. The integration test applications were also used as fixtures for the pixel validation work described in companion publications. The test binaries were consolidated so that the same binary could be invoked by the continuous integration pipeline for functional testing, by the pixel validation pipeline, and by engineers running local repro of defects. The consolidation reduced the maintenance burden on the test code, because a change to the test fixtures had to be made in only one place.

6. Patterns That Transfer to Other Platforms

This section summarizes the patterns that emerged from the work and that transfer to other Wayland-based embedded display platforms.

6.1 Guard the Layer Boundaries

Most of the null pointer crashes were at boundaries where the product-specific code received objects from the reference code or from the input subsystem. The pattern that emerged is to treat layer boundaries as the right place for defensive checks, rather than scattering checks throughout the product-specific code. A check at the boundary covers all of the downstream code that receives the object.

6.2. Validate Before Trust on Shared Memory

Shared memory between processes is a place where the trust model has to be explicit. The screenshot subsystem crash described in Section 3.2 happened because the consumer trusted the producer to keep the attributes consistent with the buffer. The pattern that emerged is to validate buffer attributes against the size and stride of the mapped region at the consumer, before any read, and to abort the operation if validation fails.

6.3. Consolidate Ownership for Lifecycle Bugs

Lifecycle bugs in C++ display code are often best addressed by consolidating ownership of the affected objects rather than by adding checks to each teardown path. The display connection teardown defect in Section 3.3 is an example. The pattern that emerged is to look for the smallest set of changes that make ownership unambiguous, and to prefer those over a larger set of changes that work around the ambiguity.

6.4. Make Logs Tell a Story per Object

The buffer lifecycle logging refactor in Section 4.2 is an example of a more general pattern. Logs are most useful when they tell a coherent story about a specific object as it moves through its lifecycle. Logs that are organized by component or by log level often do not tell that story, because the relevant events for a single object are scattered. The pattern that emerged is to key the relevant events on the object identifier and to ensure that the key appears in every event, so that a filter on the key reconstructs the lifecycle.

6.5. Sequence Numbers Are Cheap

The monotonic sequence numbering in Section 4.1 is the cheapest observability improvement described in this paper. The cost is one atomic increment per logged event. The benefit is unambiguous event ordering. The pattern that emerged is to add sequence numbering to any log stream that is read by humans or by automation to reconstruct event order, especially in concurrent systems where timestamps alone are not sufficient.

6.6. Test Scaffolding Is Part of the Stack

The C++ unit tests and integration test applications described in Section 5.4 are not auxiliary artifacts. They are part of the stack and they have their own maintenance cost. Consolidating the test binaries so that a single binary serves multiple downstream pipelines reduced this maintenance cost without sacrificing coverage. The pattern that emerged is to treat the test scaffolding as a first-class deliverable with the same code review discipline as the production code, rather than as a separate workstream that the production code team can ignore.

7. Conclusion

The Wayland reference stack is mature, but the product-specific layers added on top of it in a shipping consumer device are where most of the defects relevant to a shipping product appear. The interventions described in this paper, which focus on stability hardening at layer boundaries and on observability improvements through sequence numbering and object-keyed logging, are the kinds of interventions that improve the cost of operating such a stack without changing its fundamental architecture. The window management APIs and the consolidated test scaffolding round out the work by adding new capabilities to the shell with test coverage that supports continued development. Teams operating similar Wayland-based embedded display stacks can apply the same patterns with attention to their own product specifics.

Acknowledgments

This work was performed in the context of device operating system development for streaming and smart display devices. The author thanks the graphics engineering team and the device platform teams for their collaboration.

References

[1] Wayland project documentation, freedesktop.org.
[2] Weston compositor documentation, freedesktop.org.
[3] Linux Kernel Direct Rendering Manager and Kernel Mode Setting documentation, Linux kernel project.
[4] OpenGL ES specification, Khronos Group.
[5] EGL specification, Khronos Group.
[6] Linux dma-buf documentation, Linux kernel project.
[7] Bovet, D. P. and Cesati, M. Understanding the Linux Kernel, 3rd Edition. O'Reilly Media, 2005.
[8] Love, R. Linux Kernel Development, 3rd Edition. Addison-Wesley Professional, 2010.
[9] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, 2016.
[10] Majors, C., Fong-Jones, L., and Miranda, G. Observability Engineering. O'Reilly Media, 2022.
[11] Akenine-Moller, T., Haines, E., and Hoffman, N. Real-Time Rendering, 4th Edition. CRC Press, 2018.