DRM Framebuffer Pixel Extractor:

Project Overview

This illustrates my complete development journey of a Linux kernel module designed to extract pixel data from DRM (Direct Rendering Manager) framebuffers for real-time screen capture. The project evolved through multiple technical challenges, from initial eBPF limitations to complex Intel GPU tiling issues.

Phase 1: Initial Approach with eBPF and Why I Switched

The eBPF Attempt

I initially started with an eBPF (Extended Berkeley Packet Filter) approach for intercepting DRM framebuffer operations. eBPF seemed attractive because:

- It provides a safer way to run kernel-level code (No kernel crash risk)
- It has built-in verifier for safety
- It's more maintainable and less risky than kernel modules

Critical Issues with eBPF

However, I encountered two major issues:

1. **Map Size Limitations**: eBPF has strict limits on map sizes (typically 1MB or less), which is insufficient for storing full framebuffer data. Modern displays (1920x1080x4 bytes = ~8MB for RGBA) far exceed these limits.
2. **Verifier Problems**: The eBPF verifier is extremely strict about memory access patterns and loop bounds. My dynamic framebuffer handling code couldn't pass verification due to:
    - Dynamic memory allocation requirements
    - Complex pointer arithmetic for pixel data access
    - Unbounded loops needed for processing variable-sized framebuffers

Decision to Switch to Kernel Module

Given these limitations, I made the decision to develop a kernel module instead, with increased complexity and security considerations in exchange for:

- Unlimited memory allocation capabilities
- Direct access to kernel APIs
- Freedom from verifier constraints
- Ability to use kprobes for function interception

Phase 2: Kernel Module Architecture and Initial Challenges

Core Architecture Design

My kernel module uses several key components:

```c
struct fb_pixel_data {
    struct drm_framebuffer *fb;
    struct drm_device *dev;
    void *pixel_buffer;
    size_t buffer_size;
    uint32_t width, height;
    uint32_t format;
    uint32_t pitch;
    uint64_t timestamp;
    bool valid;
    bool has_pixels;
};
```

This structure captures both metadata and actual pixel data from framebuffers.

Challenge 1: Finding the Right Kernel Buffer Access Point

The Problem: Initially, I struggled to understand how to access the actual pixel data from DRM framebuffers. The DRM subsystem is complex, with multiple layers of abstraction.

My Discovery Process:

1. I first tried intercepting high-level DRM functions but found they didn't provide direct access to pixel data
2. I examined the DRM framebuffer structure and discovered it contains references to GEM (Graphics Execution Manager) objects
3. I realized that the actual pixel data is stored in GEM objects, not directly in the framebuffer structure

The Solution: I focused on the `drm_framebuffer_init` function as tracepoint because:

- It's called when framebuffers are created/updated
- It provides access to both the framebuffer metadata and the underlying GEM objects
- It's a stable API point across different kernel versions

Challenge 2: GEM Object Access Methods

The Core Issue: GEM objects can store pixel data in various ways depending on the graphics driver and memory management strategy.

My Multi-Method Approach:

Method 1: SHMEM-Based Access

c

```c
if (gem_obj->filp && gem_obj->filp->f_mapping) {
    struct address_space *mapping = gem_obj->filp->f_mapping;
    // Access pages through the file mapping
    struct page *page = find_get_page(mapping, index);
    void *kaddr = kmap_atomic(page);
    memcpy((char*)capture->pixel_buffer + copied, kaddr, to_copy);
    kunmap_atomic(kaddr);
}
```

This method works for GEM objects backed by system memory (SHMEM).

Method 2: DMA-buf Access

c

```c
if (gem_obj->dma_buf && gem_obj->import_attach) {
    ret = dma_buf_vmap(gem_obj->dma_buf, &map);
    if (map.is_iomem) {
        memcpy_fromio(capture->pixel_buffer, map.vaddr_iomem, to_copy);
    } else {
        memcpy(capture->pixel_buffer, map.vaddr, to_copy);
    }
}
```

This handles imported DMA buffers, common in modern graphics stacks.

Challenge 3: Kprobe Implementation and Architecture-Specific Considerations

The Problem: Kprobes need to extract function parameters from CPU registers, and this varies by architecture. It is not like a kfunc probe or tracepoint where parameters are passed in a consistent way. So in order to access a kprobe function parameters we need to extract them from the CPU registers. So there is a possibility that this code will not work the same way on different architectures and different kernel versions.

c

```c
static int handler_drm_framebuffer_init(struct kprobe *p, struct pt_regs *regs)
{
#ifdef CONFIG_X86_64
    dev = (struct drm_device *)regs->di;
    fb = (struct drm_framebuffer *)regs->si;
#elif defined(CONFIG_ARM64)
    dev = (struct drm_device *)regs->regs[0];
    fb = (struct drm_framebuffer *)regs->regs[1];
#endif
}
```

I implemented architecture-specific register access to ensure compatibility across different CPU architectures.

Phase 3: The Intel GPU Tiling Challenge

The Stripping Problem Discovery

Initial Symptoms: After successfully extracting pixel data, I noticed the resulting images were heavily stripped or corrupted, appearing as horizontal bands of misaligned pixels.

My Investigation Process:

1. I initially suspected my memory copying logic was incorrect
2. I verified that I was reading the correct amount of data (width × height × bytes_per_pixel)
3. I confirmed the framebuffer metadata (width, height, pitch) was correct
4. I realized the issue wasn't with my code but with how Intel GPUs store framebuffer data

Understanding Intel GPU Memory Optimization

The Root Cause: Intel GPUs use tiling for memory optimization. Instead of storing pixels linearly (row by row), they store them in tiles to improve cache efficiency and memory bandwidth utilization.

Types of Tiling:

- **Linear**: Traditional row-by-row storage (what I expected)
- **X-tiling**: 512-byte tiles arranged in a specific pattern
- **Y-tiling**: More complex tiling with different dimensions and patterns

Attempting to Force Linear Storage

My First Approach: I investigated whether I could force the Intel driver to use linear storage instead of tiled storage.

Why This Failed:

- Tiling is deeply integrated into the Intel graphics driver for performance reasons
- Forcing linear storage would require significant driver modifications
- Modern Intel GPUs are optimized for tiled access patterns

The Pitch Investigation

My Theory: I initially thought the "pitch" (bytes per row) value might be incorrect, causing me to read data with wrong stride values.

What I Discovered:

- The pitch value was correct for the *tiled* format
- The pitch represents the stride in the tiled format, not the linear format
- This explained why my linear interpretation was producing corrupted images

Multi-Monitor Display Confusion

The Symptom: Even with two monitors connected, I was seeing what appeared to be a single monitor's worth of data in my captures.

My Investigation:

- I initially thought I was only capturing one monitor's framebuffer
- I tried to identify separate framebuffers for each monitor

The Reality:

- Modern display systems often use a single large framebuffer containing multiple monitor regions
- The single monitor appearance was actually due to the tiling corruption making the second monitor's data unrecognizable
- The data was there, but the tiling made it appear as noise

Python Pillow Experimentation

My Experiment: I used Python's Pillow library to try different image reconstruction approaches:

python

```python
# Attempting to reconstruct the image with different parameters
from PIL import Image
import numpy as np

# Try different width/height combinations
img = Image.frombuffer('RGBA', (width, height), raw_data, 'raw', 'RGBA', 0, 1)
```

Results:

- Pillow gave me "less stripped" images, confirming that the data was present but improperly arranged
- This validated my theory that the issue was tiling-related, not data corruption
- I could see recognizable image content, but still with significant artifacts

Intel Detiling Script Discovery

The Breakthrough: I discovered that Intel provides official detiling utilities for their GPUs.

Initial Challenges:

- The GitHub repository was not readily available or had access issues
- I had to find alternative sources for the Intel detiling algorithms
- The documentation was sparse and targeted at Intel's internal tools

Understanding the Modifier Field

The Key Discovery: I found that the DRM framebuffer structure contains a "modifier" field that indicates the tiling format:

c

```c
struct drm_framebuffer {
    // ... other fields ...
    uint64_t modifier;  // This tells us the tiling format!
};
```

Modifier Values:

- `0x0`: Linear (no tiling)
- `0x1`: X-tiling
- `0x2`: Y-tiling
- Other values for more complex formats

My Detection Logic:

c

```c
// Check the modifier to determine tiling format
if (fb->modifier == DRM_FORMAT_MOD_LINEAR) {
    // Linear format - no detiling needed
} else if (fb->modifier & DRM_FORMAT_MOD_INTEL_X_TILED) {
    // X-tiling detected
    use_x_tiling_detile = true;
} else if (fb->modifier & DRM_FORMAT_MOD_INTEL_Y_TILED) {
    // Y-tiling detected
    use_y_tiling_detile = true;
}
```

Working Solution with FFmpeg

The Process:

1. Extract raw tiled framebuffer data using my kernel module
2. Apply Intel's X-tiling detiling algorithm (since modifier was 0x1)
3. Use FFmpeg to convert the detiled raw data to a proper image format

FFmpeg Command:

bash

```bash
ffmpeg -f rawvideo -pixel_format bgr0 -video_size 3840x1080 -i /proc/drm_fb_raw -frames:v 1 output.png
```

Success: This pipeline finally produced correct, undistorted images showing both monitors properly.

Phase 4: Integrated Detiling Solution

Integration Strategy

Instead of relying on external tools, I integrated the detiling logic directly into my kernel module:

Benefits of Integration:

- Single-step process from kernel to usable image data
- Reduced latency by eliminating external processing steps
- Better error handling and debugging capabilities
- Simplified deployment (single kernel module)

Phase 5: Performance Analysis and Latency Issues

Latency Measurement Setup

I implemented timing measurement to understand the performance characteristics:

The 150ms Latency Problem

Measurement Results: My tests revealed a consistent ~150ms latency for screen capture, which is too high for real-time applications.

Potential Causes and Analysis:

1. **Memory Copy Operations**:
    - Multiple large memory copies (GEM → temporary buffer → detiled buffer)
    - Each copy operation for 8MB+ data takes significant time
    - Solution: Implement in-place detiling or streaming processing
2. **Cache Efficiency**:
    - Large memory operations can cause cache misses
    - Detiling algorithm accesses memory in non-linear patterns
    - Solution: Optimize memory access patterns, use cache-friendly algorithms
3. **Framebuffer Update Frequency**:
    - I might be capturing redundant or intermediate framebuffer states
    - Graphics drivers may update framebuffers more frequently than the actual display refresh rate
    - Solution: Implement throttling or change detection

Code Architecture Deep Dive

Module Structure and Components

1. Data Structures

c

```c
struct fb_pixel_data {
    struct drm_framebuffer *fb;  // Reference to DRM framebuffer
    struct drm_device *dev;      // Graphics device
    void *pixel_buffer;          // Actual pixel data
    size_t buffer_size;          // Size of pixel buffer
    uint32_t width, height;      // Dimensions
    uint32_t format;             // Pixel format (RGBA, etc.)
    uint32_t pitch;              // Bytes per row
    uint64_t timestamp;          // Capture timestamp
    bool valid;                  // Structure validity flag
    bool has_pixels;             // Whether pixel data was extracted
};
```

This structure serves as my primary data container, holding both metadata and pixel data.

2. Circular Buffer Management

c

```c
static struct fb_pixel_data captured_fbs[MAX_FB_CAPTURE];
static int capture_count = 0;
static int current_index = 0;
static DEFINE_MUTEX(capture_mutex);
```

I use a circular buffer to store multiple captures, allowing for:

- Historical data access
- Comparison between frames
- Buffering during high-frequency updates

3. Memory Management Strategy

Allocation:

c

```c
capture->pixel_buffer = vmalloc(capture->buffer_size);
```

I use `vmalloc()` for large allocations because:

- Framebuffers can be very large (>8MB)
- `kmalloc()` has size limitations
- `vmalloc()` provides virtually contiguous memory

Deallocation:

```c
if (capture->pixel_buffer) {
    vfree(capture->pixel_buffer);
    capture->pixel_buffer = NULL;
}
```

4. Proc Interface Implementation

Information Interface (`/proc/drm_fb_pixels`):

- Displays capture metadata
- Shows pixel format information

Raw Data Interface (`/proc/drm_fb_raw`):

- Provides direct access to raw framebuffer pixel data
- Implements proper bounds checking
- Uses `copy_to_user()` for safe data transfer

Error Handling and Safety

1. Parameter Validation

```c
if (!fb || !fb->obj[0]) {
    pr_warn("Invalid framebuffer or missing GEM object\n");
    return -EINVAL;
}
```

2. Memory Safety

```c
if (offset >= capture->buffer_size) {
    mutex_unlock(&capture_mutex);
    return 0; // EOF
}

to_copy = min_t(size_t, count, capture->buffer_size - offset);
```

3. Mutex Protection

All critical sections are protected by mutexes to prevent race conditions in multi-threaded environments.

Current Status and Next Steps

What's Working

- Successful framebuffer interception via kprobes

- Multi-method GEM object access
- Intel X-tiling detection and detiling
- Integrated kernel module solution
- Proc interface for data access
- Framebuffer extraction from kernel space

Known Issues

- 150ms latency

Next Steps

- Inevstigate and minimize latency

Steps to reproduce the project:

1. **Clone the Repository**: Get the source code from my GitHub repository.
2. **Build the Kernel Module**: Use the provided Makefile to compile the module against your kernel headers.
3. **Load the Module**: Use `sudo insmod *.ko` to load the module into the kernel.
4. **Check `/proc` Interfaces**: Verify that the `/proc/drm_fb_pixels` and `/proc/drm_fb_raw` interfaces are available.
5. **Check dmesg Logs**: Use `dmesg` to view debug output and ensure the module is functioning correctly.
6. **Use the Command** `ffmpeg -f rawvideo -pixel_format bgr0 -video_size 3840x1080 -i /proc/drm_fb_raw -frames:v 1 output.png` to capture the framebuffer data and convert it to an image.
7. **Unload the Module**: Use `sudo rmmod <module_name>` to safely remove the module when done.

Fixing the latency issue:

I changed the way I tested the latency, initially I was using a shellscript that was running the commands `ffmpeg -f rawvideo -pixel_format bgr0 -video_size 3840x1080 -i /proc/drm_fb_raw -frames:v 1 output.png & python3 script.py` and then checking the time recorded by the stopwatch on the screenshots. But then I learned that running the shellscript and using `&` bash command introduces additional latency due to process scheduling and context switching. So, I changed to using the `parallel` command to run both commands in parallel without the `&` operator, which allows for better CPU core utilization and reduces the overhead of process management.

This is the command I used to run both commands in parallel:

```bash
parallel --lb ::: \
'taskset -c 2 /bin/python3 /home/carbon/Documents/WashU/bpf/ebpf/test/test2.py' \
'taskset -c 3 ffmpeg -f rawvideo -pixel_format bgr0 -video_size 3840x1080 -i /proc/drm_fb_raw -frames:v 1 frar
```

Doing this reduced the latency to around 60ms, which is significantly better than the initial 150ms.