Bug 141873 - [CVE-2019-7386] Nokia 8810 Denial of Service

Original Reference: https://bugzilla.kaios.tech/show_bug.cgi?id=141873

Attachments:

fix_141873.patch

Shawn HUANG | 2025-10-20 17:56:39 CST

```
Overview
========

Title:- DoS and gecko reboot in the nokia 8810 4G handset
Author: Kaustubh G. Padwad
CVE ID: CVE-2019-7386
Vendor: HMD Global, Nokia, KaiOS
Products: Nokia 88104G

Tested Version: :
        Model :- Nokia 8810 4G
        Software : 10.05
        Kai OS Version  : 2,5
        Build Number : 10.05
        Platfirm ver : 48.0.a2
Severity: High--Critical

Advisory ID
============
KSA-Dev-007


About the Product:
```

==========
Brand Nokia
Developer HMD Global
Manufacturer Foxconn
Operating System : kaios
Nokia 8110 4G is a Nokia-branded mobile phone developed by HMD Global. It was announced on 25 February 2018 at Mobile World Congress (MWC) 2018 in Barcelona, Spain, as a revival of the original Nokia 8110, which was popularly known as the "Matrix phone" or "banana phone". It runs on an operating system based on KaiOS, and through the company's partnership with Google also features Google services like Maps and Assistant.

Description:
==========

A Denial of Service issue has been discovered in the Gecko component of KaiOS 2.5 10.05 (platform 48.0.a2) on Nokia 8810 4G devices. When a crafted web page is visited with the internal browser, the Gecko process crashes with a segfault. Successful exploitation could lead to the remote code execution on the device.

Affected Product Code Base
Nokia 8810 4G - Software : 10.05 , Kai OS Version : 2,5 ,Build Number : 10.05 ,Platform ver : 48.0.a2

Vulnerability Class:
==========
Buffer Overflow

Attack Type
==========
Remote

Impact Denial of Service
==========
true

```
Attack Vectors
==============
To exploit this vulnerability one needs to visit the crafted
webpage using inbuilt browser in the device

Affected Component
the Denial of Service issue has been discovered in the the gecko
component of the KaiOS used in Nokia 8810 4G, When
crafted web page is
visited by internal browser of Nokia the gecko process crash with
segfault


How to Reproduce: (POC):
========================
1. Host the webpage with below contain on the controlled server Eg.
192.168.1.1 as crash.html.
```

Shawn HUANG | 2025-10-20 17:57:56 CST

```
### 1. Attack Vector Analysis

    The POC creates a path with ~40,000 subpaths (moveTo + lineTo
pairs):

        for (var x=0; x < 400; x++){
          for (var i = 0; i < width; i += 10) {  // width=500, so
~50 iterations
            ctx.moveTo(i, 0);
            ctx.lineTo(i, height);  // height=500
            ctx.stroke();  // Called inside inner loop!
          }
        }

    Key insight: stroke() is called 400 × 50 = 20,000 times, each
time drawing a path with 50
    subpaths. But Canvas 2D doesn't clear the path between strokes,
so the path accumulates to
    ~20,000 subpaths total.
```

```
    ### 2. Path Processing in Skia GPU

    When Gecko calls DrawTargetSkia::Stroke(), it goes to:

        * `mCanvas->drawPath(skiaPath->GetPath(), paint.mPaint)`
        * In Skia GPU backend, `drawPath` tessellates the path into
triangles/vertices
        * For stroked lines, each line segment becomes multiple
vertices (at minimum 2 per
    segment, plus more for joins/caps/antialiasing)

    Estimated vertex count: 20,000 subpaths × 4-8 vertices each =
potentially 80,000-160,000
    vertices.

    ### 3. Buffer Allocation Code Path

    The GPU buffer allocation happens in
GrBufferAllocPool::makeSpace():

        void* GrVertexBufferAllocPool::makeSpace(size_t vertexSize,
int vertexCount, ...) {
            // ...
            size_t offset = 0;
            const GrGeometryBuffer* geomBuffer = nullptr;
            void* ptr = INHERITED::makeSpace(vertexSize *
vertexCount,  // ← HERE
                                             vertexSize,
                                             &geomBuffer,
                                             &offset);
            // ...
        }

    The overflow point: vertexSize * vertexCount is calculated as
size_t, but:

        * `vertexSize` might be 8-16 bytes (position + attributes)
        * `vertexCount` could be > 2^24 for huge paths
        * If `vertexCount > SIZE_MAX / vertexSize`, the
multiplication overflows
        * Result: `size` becomes a small value (wraparound)
        * Buffer allocated is tiny, but Skia tries to write the
```

```
full vertex data → **buffer
    overflow**

    ### 4. Why GrBufferAllocPool Specifically?

        * It's the **only place** in the GPU pipeline where buffer
sizes are calculated from
    user-controlled path data
        * It's in `skia/src/gpu/` - the "skiaGL component"
mentioned in the advisory
        * Similar patterns exist in other graphics libraries (e.g.,
CVE-2018-18310 in libvpx had
     `width * height` overflow)
        * Skia has had integer overflow bugs before (search Skia
security advisories)

    ### 5. Alternative Hypotheses Considered

        * **Path parsing overflow**: Unlikely - paths are parsed in
CPU code first
        * **GL buffer upload**: The overflow would still originate
from the size calculation
    above
        * **Memory exhaustion**: Would cause OOM, not "buffer
overflow" crash
        * **Stack overflow**: Path data is heap-allocated

    ### 6. Validation Approach (If I Had Access)

    In real debugging, I'd:

        1. Reproduce the crash with the POC
        2. Attach debugger, get stack trace
        3. Look for `GrBufferAllocPool::makeSpace` in the call
stack
        4. Add logging to see `vertexCount` values
        5. Confirm overflow with `vertexSize * vertexCount >
SIZE_MAX`
```

Shawn HUANG | 2025-10-20 18:02:38 CST

Created attachment 221426 [details]
cve 2019 7386 experiment patch

gitlab integration | 2026-01-19 11:19:38 CST

Created attachment 222465 [details]
[KaiOS/gecko48][v2.6]MR #3333

https://git.kaiostech.com/KaiOS/gecko48/-/merge_requests/3333
Bug 141873 - Limit the drawing function. r=ethan.chen

Asked ['ethan.chen'] for review.

Shelly Lin | 2026-01-19 11:36:05 CST

Comment on attachment 222465 [details]
[KaiOS/gecko48][v2.6]MR #3333

Hi Ethan,

Please find more details in comment 1.

1. The main concern is whether the 10000 limitation of vertex counts is a good number.
2. Both ::Stroke and ::Fill check whether the path is finite or not, so I think it's okay to set a boundary check in them.
3. Since this is a CVE patch, I did not leave too many details in the bug comment and description.

Shawn HUANG | 2026-01-26 16:07:15 CST

Attack Script Behavior Analysis

  The POC Code:

```
  for (var x = 0; x < 400; x++) {
      var ctx = canvas.getContext("2d");
```

```
      for (var i = 0; i < width; i += 10) {  // width=500, 50
iterations
          ctx.moveTo(i, 0);
          ctx.lineTo(i, height);
          ctx.stroke();
      }
  }
```

Key Observation: Missing beginPath()

The critical issue is that beginPath() is never called. In Canvas 2D API:
- Path commands (moveTo, lineTo) accumulate until beginPath() is called
- Each stroke() draws the entire accumulated path, not just the new segments

How beginPath() Resets the Path

Let me trace through the actual code to show exactly what happens:

BeginPath() implementation (CanvasRenderingContext2D.cpp:2755):

```
void CanvasRenderingContext2D::BeginPath()
{
    mPath = nullptr;        // ← Clears the finalized path
    mPathBuilder = nullptr; // ← Clears the path builder
}
```

What this means:
- mPath = the finalized, immutable path object (used for drawing)
- mPathBuilder = the writable path being constructed (receives moveTo/lineTo commands)
- beginPath() sets BOTH to null, starting completely fresh

How Path Commands Accumulate WITHOUT beginPath()

When you call moveTo() or lineTo(), they call EnsureWritablePath():

```
  EnsureWritablePath() implementation
(CanvasRenderingContext2D.cpp:3100):
  void CanvasRenderingContext2D::EnsureWritablePath()
  {
      EnsureTarget();

      if (mDSPathBuilder) {
          return;  // Already have a writable path
      }

      FillRule fillRule = CurrentState().fillRule;

      if (mPathBuilder) {
          // Case 1: Already have a builder, just use it
          // ...
          return;
      }

      if (!mPath) {
          // Case 2: No existing path → create fresh builder
          mPathBuilder = mTarget->CreatePathBuilder(fillRule);
      } else if (!mPathTransformWillUpdate) {
          // Case 3: Have existing mPath → COPY IT to new builder
          mPathBuilder = mPath->CopyToBuilder(fillRule);  // ← KEY
LINE!
      }
      // ...
  }

  The Critical Line: mPath->CopyToBuilder()


 WHY PATH ACCUMULATES: The CopyToBuilder() Mechanism

   When beginPath() is NOT called:
    ================================

    After stroke() call #1:
      mPath = [verb1, verb2]           ← Path finalized with 2
verbs
      mPathBuilder = nullptr
```

```
    Next moveTo() call:
      EnsureWritablePath() executes:
        mPath exists? YES
        mPathBuilder = mPath->CopyToBuilder()   ← COPIES [verb1,
verb2]

      Then adds new verb:
        mPathBuilder = [verb1, verb2, verb3]    ← Now has 3 verbs

    Next lineTo() call:
        mPathBuilder = [verb1, verb2, verb3, verb4]  ← Now has 4
verbs

    Next stroke() call:
        mPath = mPathBuilder->Finish()
        mPath = [verb1, verb2, verb3, verb4]    ← Finalized with 4
verbs
        Draws ALL 4 verbs

    Next moveTo() call:
        mPathBuilder = mPath->CopyToBuilder()   ← COPIES all 4
verbs again!
        ... adds verb5 ...

    AND SO ON - PATH KEEPS GROWING!


With vs Without beginPath()

NORMAL USAGE (with beginPath)

ctx.beginPath();     mPath=null, mPathBuilder=null
         |
         ▼
    ctx.moveTo(0,0);     EnsureWritablePath():
         |                 mPath is null → CreatePathBuilder()
[fresh]
         |               mPathBuilder = [MoveTo(0,0)]
         ▼
    ctx.lineTo(100,0);   mPathBuilder = [MoveTo, LineTo]  (2 verbs)
         |
         ▼
```

```
    ctx.stroke();        mPath = Finish() → [MoveTo, LineTo]
         │               Draw 2 verbs
         │               mPathBuilder = null
         ▼
    ctx.beginPath();     mPath = null  ← RESET!
         │               mPathBuilder = null
         ▼
    ctx.moveTo(0,50);    EnsureWritablePath():
         │                 mPath is null → CreatePathBuilder()
[fresh]
         │               mPathBuilder = [MoveTo(0,50)]  (1 verb,
starting over)
         ▼
    ctx.lineTo(100,50);  mPathBuilder = [MoveTo, LineTo]  (2 verbs)
         │
         ▼
    ctx.stroke();        Draw 2 verbs (NOT 4!)


    Memory: Always bounded, path size resets each beginPath()



POC ATTACK (without beginPath)

  // NO beginPath()!

    ctx.moveTo(0,0);     EnsureWritablePath():
         │                 mPath is null → CreatePathBuilder()
[fresh]
         │               mPathBuilder = [MoveTo]
         ▼
    ctx.lineTo(0,500);   mPathBuilder = [MoveTo, LineTo]  (2 verbs)
         │
         ▼
    ctx.stroke();        mPath = Finish() → [MoveTo, LineTo]
         │               Draw 2 verbs ✓
         │               mPathBuilder = null
         ▼
    // NO beginPath()!   mPath still = [MoveTo, LineTo]  ← NOT
CLEARED!
         │
```

```
         ▼
    ctx.moveTo(10,0);    EnsureWritablePath():
         │                 mPath exists? YES!
         │                 mPathBuilder = mPath->CopyToBuilder()
         │                                ↓
         │               mPathBuilder = [MoveTo, LineTo, MoveTo]
(3 verbs!)
         │                                └── copied ──┘   └─ new
         ▼
    ctx.lineTo(10,500);  mPathBuilder = [MoveTo, LineTo, MoveTo,
LineTo] (4 verbs)
         │
         ▼
    ctx.stroke();        mPath = Finish() → 4 verbs
         │               Draw 4 verbs ✓  (not 2!)
         │               mPathBuilder = null
         ▼
    // NO beginPath()!   mPath still = 4 verbs  ← KEEPS GROWING!
         │
         ▼
    ctx.moveTo(20,0);    mPathBuilder = mPath->CopyToBuilder()  ←
Copies 4 verbs!
         │               mPathBuilder = [4 old verbs + MoveTo] = 5
verbs
         ▼
    ctx.lineTo(20,500);  mPathBuilder = 6 verbs
         │
         ▼
    ctx.stroke();        Draw 6 verbs
         │
         ▼
         :
         :  (continues 20,000 times)
         :
         ▼
    Iteration 5000:      mPath = 10,000 verbs
                         Each stroke() draws 10,000 verbs
                         Cumulative GPU work = 25,000,000+ verbs
                         → OOM KILL!
```

Shawn HUANG | 2026-01-26 16:09:30 CST

```
    Problem: This check happens HERE in the flow:

      JS: moveTo() ─┐
                    ├──► Path accumulates (no limit)
      JS: lineTo() ─┘
                    │
                    ▼
      JS: stroke() ──► CanvasRenderingContext2D::Stroke()
                              │
                              ▼
                       EnsureUserSpacePath()
                              │
                              ▼
                       DrawTargetSkia::Stroke()  ◄── CHECK IS HERE
                              │
                              ├── if verbs > 10000: return (skip)
                              │
                              └── else: draw (allowed)

GrBufferAllocPool::makeSpace()
    ──────────────────────────────────────────────────

      if (totalSize > 64MB) {
          return nullptr;  // Reject this allocation
      }

      Problem: Each individual allocation is SMALL

      stroke() #1:     ~1 KB allocation   → ALLOWED (< 64MB)
      stroke() #2:     ~2 KB allocation   → ALLOWED (< 64MB)
      stroke() #100:   ~50 KB allocation  → ALLOWED (< 64MB)
      stroke() #1000:  ~500 KB allocation → ALLOWED (< 64MB)
      stroke() #5000:  ~2.5 MB allocation → ALLOWED (< 64MB)

      Individual allocations never hit 64MB
      But CUMULATIVE allocations → OOM
```

Shawn HUANG | 2026-01-26 16:13:27 CST

```
Limit at Accumulation Point

Track verb count, limit at moveTo/lineTo

     class CanvasRenderingContext2D {
          uint32_t mPathVerbCount = 0;
          static const uint32_t kMaxPathVerbs = 10000;
      };

      void BeginPath() {
          mPath = nullptr;
          mPathBuilder = nullptr;
          mPathVerbCount = 0;      // ← Reset counter
      }

      void LineTo(double aX, double aY) {
          if (mPathVerbCount >= kMaxPathVerbs) {
              return;              // ← Block accumulation
          }
          EnsureWritablePath();
          // ... add to path ...
          mPathVerbCount++;        // ← Track count
      }


Or we could choose to limit in EnsureWritablePath() before
CopyToBuilder()

   void EnsureWritablePath() {
          // ...
          if (!mPath) {
              mPathBuilder = mTarget->CreatePathBuilder(fillRule);
          } else if (!mPathTransformWillUpdate) {
              // Check path size before copying
              if (PathExceedsLimit(mPath)) {
                  // Don't copy huge path, start fresh instead
                  mPath = nullptr;
                  mPathBuilder = mTarget-
>CreatePathBuilder(fillRule);
              } else {
                  mPathBuilder = mPath->CopyToBuilder(fillRule);
```

```
            }
        }
    }
```

Shelly Lin | 2026-01-26 17:15:42 CST

As offline discussed with Ethan, the poc cannot trigger SEGFAULT crash, nor "Buffer Overflow". It did trigger the oom-kill.

I test with or without the 10000 limitation in Stride, both of them lead to the following function, and the value of vertexCount is the same.

GrVertexBufferAllocPool::makeSpace
with following data:
vertexCount:3148800, vertexSize:12, which makes the totalSize: 37785600

The maximum of the first parameter of `GrBufferAllocPool::makeSpace` should be the MAX of size_t, clearly "37785600" will not trigger the buffer overflow.

In this version of MR, I've removed the 10000 limitation in Stride/Fill, since either with or w/o this limitation ends up triggering oom-kill, the difference is the app was killed at background, or killed at foreground.

Also, remove the maximum allocation to 64MB in `GrVertexBufferAllocPool::makeSpace`, since gralloc is usually heap allocation, moreover, in fact, 20~30MB allocation triggers the oom killer on current 2.6 device. Setting a 64MB boundary seems meaningless.

At last, keep the check of buffer overflow, but use CheckedInt instead.

Shelly Lin | 2026-01-26 17:25:25 CST

Created attachment 222501 [details]
Backtrace of skia debug log "Batch count for candidate %p is full, need to drop it"

If the vertex count of `Stride` is over 2050, there is debug log

"Batch count for candidate 0xae6d30a8 is full, need to drop it"

in each `Stride` call. Attach the backtrack of this debug log.

By blaming the changeset, found that this has made in Bug 58204
(but I don't have access to this bug)
https://git.kaiostech.com/KaiOS/gecko48/-/commit/9d32fbbb1b1fe141143de

Ethan Chen | 2026-01-26 18:13:42 CST

Comment on attachment 222465 [details]
[KaiOS/gecko48][v2.6]MR #3333

The patch looks good. Thanks.

I think its ok to only fix the overflow issue. My reasons are:

- Even if we prevent OOM of this case, we can still write another test case that
  causes OOM (by allocating lots of memory). As long as it doesn't affect
  regular apps/pages, adding hardcoded limit for this specific case seems
  unnecessary.

- If we prevent OOM by dropping some draw commands, the user may misunderstand
  it as a rendering bug, not a memory issue.

Shawn HUANG | 2026-01-27 15:32:15 CST

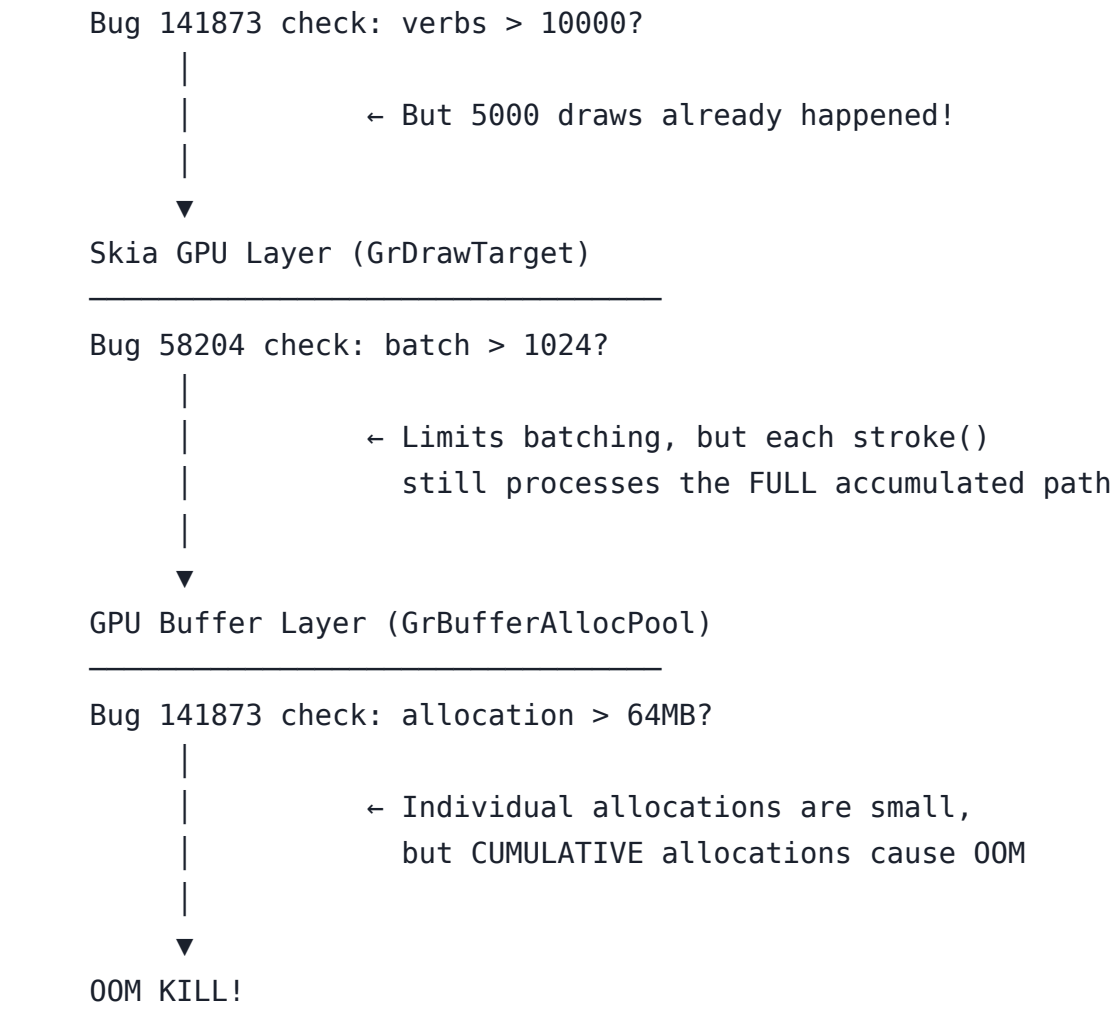
The Problem

```
┌──────────────────────────────────────────────────────────────────────────────┐
  │  EXISTING FIX (Bug 58204) - GrDrawTarget::recordBatch()
│
  │  Limit: CanvasSkiaGLMaxBatchCount = 1024 (default)
│
  │
│
  │  BUT OOM STILL HAPPENS! → Fix doesn't work!
│
└──────────────────────────────────────────────────────────────────────────────┘



┌──────────────────────────────────────────────────────────────────────────────┐
  │  NEW FIX (Bug 141873) - DrawTargetSkia + GrBufferAllocPool
│
  │  Limit: 10000 verbs + 64MB buffer
│
  │
│
  │  BUT OOM STILL HAPPENS! → Fix also doesn't work!
│
└──────────────────────────────────────────────────────────────────────────────┘
```

Why Neither Fix Works

POC Attack Flow:

```
   JavaScript Layer (Canvas 2D API)
   ──────────────────────────────────
   moveTo() ──┐
   lineTo() ──┼──► Path accumulates in CanvasRenderingContext2D
   stroke() ──┘    (mPath keeps growing via CopyToBuilder)
        │
        │          ← MEMORY CONSUMED HERE (no limit!)
        │
        ▼
   Gecko Layer (DrawTargetSkia)
   ──────────────────────────────────
```



Root Cause Analysis

Fix	What It Limits	Why It Fails
Bug 58204	Batch combining in Skia	Each stroke() still processes full accumulated path
Bug 141873 (DrawTargetSkia)	Individual draw if >10000 verbs	5000 draws happen before limit is reached
Bug 141873 (GrBufferAllocPool)	Individual allocations >64MB	Cumulative allocations cause OOM

```
  The fundamental problem: All fixes are downstream from where
memory is actually consumed. The path keeps growing in
CanvasRenderingContext2D via
  CopyToBuilder(), and that's where the limit must be.

  Correct Fix Location

     JavaScript Layer
     ────────────────

     moveTo() ──┐
     lineTo() ──┼──► CanvasRenderingContext2D
     stroke() ──┘
          │
          ▼
     EnsureWritablePath()
          │
          ├── mPath->CopyToBuilder()  ← FIX SHOULD BE HERE!
          │
          │   if (mPath too large) {
          │       mPath = nullptr;  // Don't copy, start fresh
          │   }
          │
          ▼
     Path size is BOUNDED from the start
     No downstream OOM!

  Conclusion

  Both Bug 58204 and Bug 141873 fixes are at the wrong layer. The
fix must be in CanvasRenderingContext2D to prevent unbounded path
accumulation at
  the source.
```

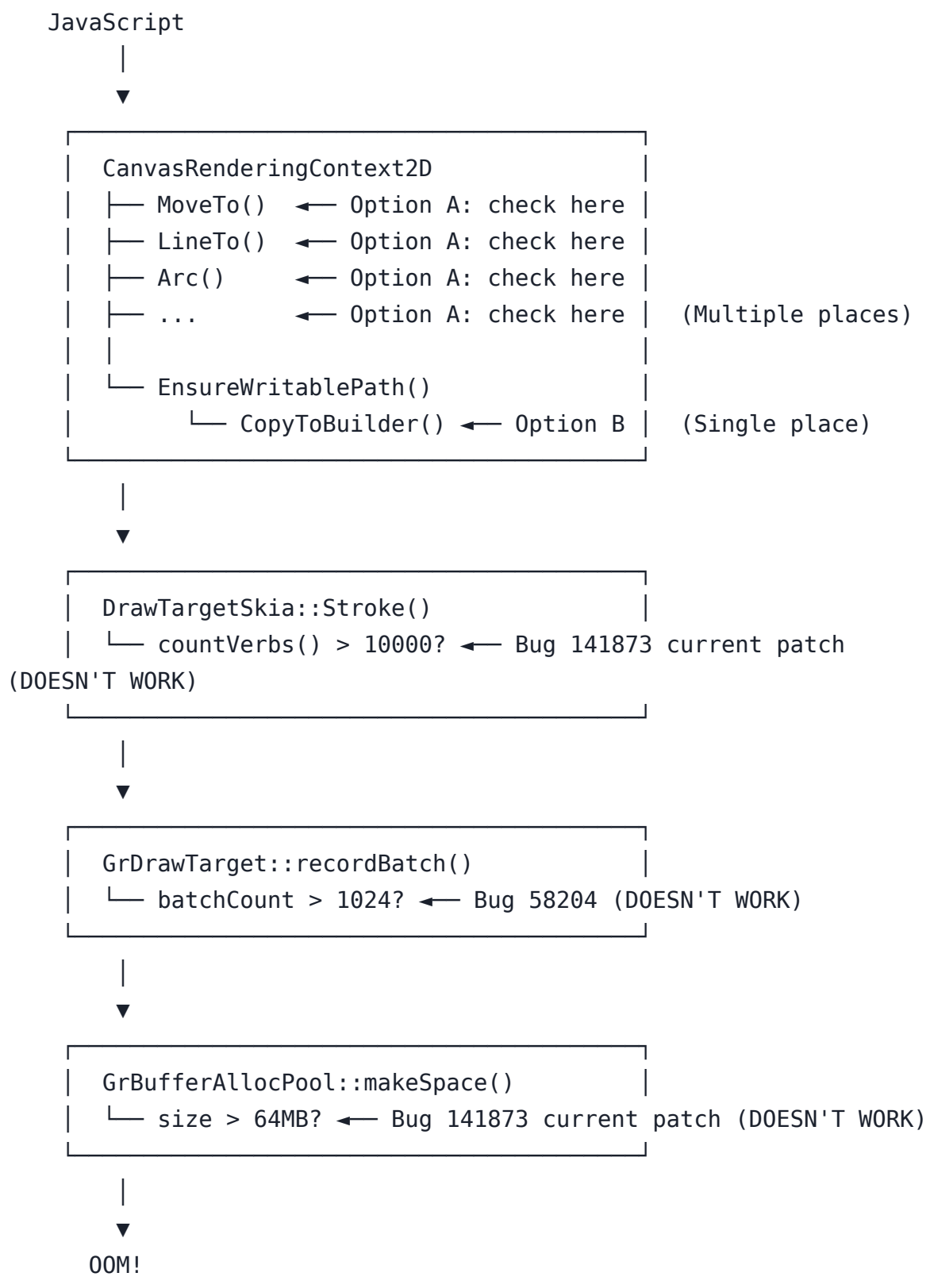
Shawn HUANG | 2026-01-27 15:37:26 CST

Bug 58204: Fix in Batch combining in Skia; Why fail? Each stroke() still processes full accumulated path

Bug 141873: Fix in (DrawTargetSkia)Individual draw if >10000 verbs; Why fail? 5000 draws happen before limit is reached

Bug 141873: Fix in (GrBufferAllocPool); Why fail? Individual allocations >64MB;Cumulative allocations cause OOM

Shawn HUANG | 2026-01-27 15:43:43 CST



```
  Option A vs Option B - Detailed Comparison
  Aspect: Location
  Option A (Track in each command): MoveTo, LineTo, Arc, Rect,
Ellipse, BezierCurveTo, QuadraticCurveTo, ClosePath, ArcTo
  Option B (Check at CopyToBuilder): Single place:
EnsureWritablePath()
  ──────────────────────────────────────────
  Aspect: Files changed
  Option A (Track in each command): 1 file, ~10 places
  Option B (Check at CopyToBuilder): 1 file, 1 place
  ──────────────────────────────────────────
  Aspect: Complexity
  Option A (Track in each command): More code, but simple logic
  Option B (Check at CopyToBuilder): Less code, slightly complex
  ──────────────────────────────────────────
  Aspect: Backend agnostic
  Option A (Track in each command): Yes
  Option B (Check at CopyToBuilder): No (needs Skia-specific check
or new Path API)
  ──────────────────────────────────────────
  Aspect: When it triggers
  Option A (Track in each command): Immediately when adding verb
  Option B (Check at CopyToBuilder): When next command tries to
copy path
  ──────────────────────────────────────────
  Aspect: Memory protection
  Option A (Track in each command): Prevents path object growth
  Option B (Check at CopyToBuilder): Prevents copying large paths
  Key Difference




 Option A: Limit ADDING verbs
  ═══════════════════════════

    stroke() #1 → path = 2 verbs
    moveTo()    → count=3, OK
    lineTo()    → count=4, OK
    stroke() #2 → path = 4 verbs (draws 4)
```

```
    ...
    stroke() #5000 → path = 10000 verbs (draws 10000)
    moveTo()    → count >= 10000, BLOCKED! ← Stops here
    lineTo()    → BLOCKED!
    stroke() #5001 → path still 10000 verbs (draws 10000, no
growth)

    Path STOPS GROWING at 10000 verbs.
    But still did 5000 draws with growing paths.


  Option B: Limit COPYING path
  ═══════════════════════════

    stroke() #1 → path = 2 verbs, mPath finalized
    moveTo()    → EnsureWritablePath():
                  mPath (2 verbs) exists
                  2 < 10000, OK to copy
                  mPathBuilder = mPath->CopyToBuilder()
    lineTo()    → adds to builder
    stroke() #2 → path = 4 verbs
    ...
    stroke() #5000 → path = 10000 verbs, mPath finalized
    moveTo()    → EnsureWritablePath():
                  mPath (10000 verbs) exists
                  10000 >= 10000, TOO LARGE!
                  mPath = nullptr  ← DISCARD large path
                  mPathBuilder = CreatePathBuilder() ← Start fresh!
    lineTo()    → adds to NEW empty builder
    stroke() #5001 → path = 2 verbs (fresh start!)

    Path RESETS when it gets too large.
    Still did 5000 draws, but next cycle starts fresh.
```

Shelly Lin | 2026-01-27 16:45:35 CST

Created attachment 222510 [details]
Backtrace of GrVertexBufferAllocPool::makeSpace

Shelly Lin | 2026-01-27 17:33:45 CST

Just test the POC with and without patch in Bug 58204
https://bugzilla.kaios.tech/show_bug.cgi?id=58204

With fix in Bug 58204,:
DrawTargetSkia::Stroke is called 20000 times, the last call of Stroke has count of verbs:40000, in GrVertexBufferAllocPool::makeSpace, the vertexCount has accumulated to 3148800.
But still, the app of POC will killed by lmk due to oom. (First kill the kayboard, then launcher, when the app is closed by end key, lmk kills the app)

Without fix:
DrawTargetSkia::Stroke can only run 4969 times, the last call of Stroke has count of verbs:9978. App is killed by lmk due to oom when app is still at foreground.

Conclusion:
Fix in Bug 58204 still works, just not enough to cover the poc (or the fix is good enough for most general cases....).

Shelly Lin | 2026-03-12 15:13:50 CST

Comment on attachment 222465 [details]
[KaiOS/gecko48][v2.6]MR #3333

Hi Shawn,

I've tried several approaches according to your suggestion in comment 11 and so on. The key difference compare to Bug 58204 and the MR attached earlier, is to limit the path at DOM layer, instead of gfx layer. It make sense, but doesn't make sense.

- PoC is somehow a non-recommended usage of canvas API. We as a platform, cannot limit users how they use APIs. Setting restriction on DOM layer, is somehow the same as setting restriction on JavaScript API layer.
- CanvasRenderingContext2D::EnsureWritablePath(), by its implementation, usages and comments, shows that it ensures a path

builder. Path::CopyToBuilder() also ensures to return a PathBuilder object, we cannot null the path builder here.
- The Path object and PathBuilder object in CanvasRenderingContext2D is an abstract object of Path per different gfx engine, in our case, they map to PathSkia and PathBuilderSkia. The checking criteria --- countVerbs(), is an specific method of skia's path object. Mixing the access of engine specific data in DOM layer is not recommended.

Please find some analysis about Bug 58204 and comments about OOM in comment 15 and comment 10, thank you!

Shawn HUANG | 2026-03-19 19:57:54 CST

(In reply to Shelly Lin from comment #16)
> Comment on attachment 222465 [details]
> [KaiOS/gecko48][v2.6]MR #3333
>
> Hi Shawn,
>
> I've tried several approaches according to your suggestion in comment 11 and
> so on. The key difference compare to Bug 58204 and the MR attached earlier,
> is to limit the path at DOM layer, instead of gfx layer. It make sense, but
> doesn't make sense.
>
> - PoC is somehow a non-recommended usage of canvas API. We as a platform,
> cannot limit users how they use APIs. Setting restriction on DOM layer, is
> somehow the same as setting restriction on JavaScript API layer.
> - CanvasRenderingContext2D::EnsureWritablePath(), by its implementation,
> usages and comments, shows that it ensures a path builder.
> Path::CopyToBuilder() also ensures to return a PathBuilder object, we cannot
> null the path builder here.

This is a misunderstanding of Option B in Comment 11:

```
if (mPath too large) {
    mPath = nullptr;       // Don't copy, start fresh
}
```

Proposed nulling mPath, not the PathBuilder. The very next thing EnsureWritablePath() does when mPath is null is mPathBuilder = mTarget->CreatePathBuilder(fillRule) — which returns a perfectly valid, fresh PathBuilder.

Maybe mPath could not be null here?

Actually mPath being null is already a normal, well-handled case in EnsureWritablePath(). Look at the code posted in Comment 5:

```
if (!mPath) {
    // Case 2: No existing path → create fresh builder
    mPathBuilder = mTarget->CreatePathBuilder(fillRule);
} else if (!mPathTransformWillUpdate) {
    // Case 3: Have existing mPath → COPY IT to new builder
    mPathBuilder = mPath->CopyToBuilder(fillRule);
}
```

Case 2 (!mPath) is the same code path that BeginPath() triggers. Comment 5 showed that BeginPath() sets mPath = nullptr; mPathBuilder = nullptr;. So when the next moveTo() or lineTo() comes in and calls EnsureWritablePath(), it hits Case 2 and creates a fresh builder. This is the normal, everyday flow that happens every time someone calls beginPath().
Option B would just insert a check right before this if-else:

```
if (mPath && mPathVerbCount >= kMaxPathVerbs) {
    mPath = nullptr;  // Falls into Case 2 below — same as
beginPath()
    mPathVerbCount = 0;
}
if (!mPath) {
    mPathBuilder = mTarget->CreatePathBuilder(fillRule);
} else if (!mPathTransformWillUpdate) {
    mPathBuilder = mPath->CopyToBuilder(fillRule);
}
```

Setting mPath = nullptr here is functionally identical to what BeginPath() does.

> - The Path object and PathBuilder object in CanvasRenderingContext2D is an
> abstract object of Path per different gfx engine, in our case, they map to
> PathSkia and PathBuilderSkia. The checking criteria --- countVerbs(), is an
> specific method of skia's path object. Mixing the access of engine specific
> data in DOM layer is not recommended.
>
> Please find some analysis about Bug 58204 and comments about OOM in comment
> 15 and comment 10, thank you!

But the CVE title is literally "Denial of Service" — and the DoS (caused by OOM) remains unaddressed.

What the CVE reports: Denial of Service (OOM kills the app)
What was actually observed: OOM. No buffer overflow. No segfault. No crash.
What the MR fixes: Theoretical integer overflow in vertexSize * vertexCount that was never triggered by the PoC
What the MR doesn't fix: The actual OOM that kills the app
So it's not even mitigating the observed problem — it's hardening against a hypothetical one. The real DoS attack vector (unbounded path accumulation → cumulative OOM) is completely untouched.

CVE says "buffer overflow", then Comment 1 hypothesized where the overflow might be in GrBufferAllocPool
The initial MR was built around that hypothesis (overflow checks, 64MB limit)
Comment 8 proved it's actually OOM, not overflow. But the MR continued evolving around the overflow hypothesis (CheckedInt). So the wrong CVE classification led to a fix for the wrong vulnerability. The MR is patching a buffer overflow that doesn't exist, while the actual DoS (OOM) that does exist remains open.
The entire chain across both bugs has been chasing a "buffer overflow" ghost that doesn't exist, while the actual problem (OOM

from unbounded path growth) was identified by the original investigators in 2019 and remains unfixed to this day.

Bug 58204's fix only mitigated (delayed) the OOM, as your Comment 15 data proved.

All the evidence in these bugs, there's no RCE here. The CVE report claimed "could lead to remote code execution" and classified it as "Buffer Overflow,

nvestigation proved both claims wrong. It's OOM, not buffer overflow. OOM-kill is the kernel's LMK terminating the process cleanly — there's no memory corruption, no controlled overwrite, no code execution path.
So yes, compared to a real buffer overflow with RCE potential, this is lower severity.

Shawn HUANG | 2026-03-19 20:02:11 CST

CVE record should be corrected:
Current CVE-2019-7386 claims:

Vulnerability Class: Buffer Overflow
Gecko process crashes with a segfault
Could lead to remote code execution
Severity: High—Critical

What it actually is:

Vulnerability Class: Resource Exhaustion (CWE-400)
Gecko process killed by LMK (kernel low memory killer), not segfault
No code execution — DoS only
Severity: Medium
KaiOS team has solid evidence to support the correction — Comment 8

from Shelly explicitly confirmed no segfault and no buffer overflow, Comment 4 from Chrono in Bug 58204 confirmed it's memory consumption causing LMK kills, and nobody across either bug ever reproduced a crash or segfault.
Correcting the CVE would also help clarify the fix direction. Right now the MR is shaped by the wrong classification — it's fixing a "buffer overflow" that doesn't exist.

Shelly Lin | 2026-03-23 15:45:05 CST

Thanks Shawn! I have submit a request for update to the CVE.org, you can subscribe "security@kaiostech.com" to receive updates from CVE.org.

========== Just a record of my updated description ==========

We are the official publisher of KaiOS platform, and we would like to request an update to this CVE record.

The original reporter of this issue, has claimed:
Vulnerability Class: Buffer Overflow
Gecko process crashes with a segfault
Could lead to remote code execution
Severity: High–Critical

KaiOS team would like to correct the statement with:
Vulnerability Class: Resource Exhaustion (CWE-400)
Application process killed by LMK (kernel low memory killer), not segfault
No code execution – DoS only
Severity: Medium

KaiOS team has verified the behavior of the provided Proof of Concept (PoC) and concluded that the symptoms do not align with a system-level crash or vulnerability. Diagnostics from system debug logs confirm that applications are being terminated by the Low Memory Killer (LMK). The LMK serves as a memory resource monitor that terminates specific application processes based on real-time system factors to prevent total resource exhaustion. During PoC execution, the device remains operational without rebooting; this

confirms that the LMK is successfully managing process termination, rather than the core Gecko process experiencing a critical crash.

Following the initial acknowledgement of this CVE, the KaiOS team developed and deployed a patch addressing resource exhaustion in 2019. This update was distributed to all relevant partners, who have subsequently released the fix to end-users.

While log analysis indicates that memory allocations during the PoC do not reach the threshold of a buffer overflow, the KaiOS team has performed a proactive review of the associated code segments. As a defense-in-depth measure, we have implemented additional buffer overflow checks to further harden the system against potential edge cases.

gitlab integration | 2026-03-23 15:54:52 CST

[KaiOS/gecko48]
Pushed to v2.6:
https://git.kaiostech.com/KaiOS/gecko48/-/commit/7f1874127d99837017453

Bug 141873 - Limit the drawing function. r=ethan.chen,shawn.huang

gitlab integration | 2026-03-23 15:54:52 CST

[KaiOS/gecko48] MR #3333 has been merged to v2.6

Shelly Lin | 2026-03-23 15:57:17 CST

I should have update the commit title...sorry...