PENETRATION TEST REPORT

for

Secure Open Source (Mozilla)

V1.0
Amsterdam
November 9th, 2017

Document Properties

Client	Secure Open Source (Mozilla)
Title	PENETRATION TEST REPORT
Target	Graphite font system
Version	1.0
Pentesters	Stefan Marsiske, Pierre Pronchery
Authors	Stefan Marsiske, Pierre Pronchery, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	October 12th, 2017	Stefan Marsiske	Initial draft
0.2	October 20th, 2017	Pierre Pronchery	Imported more findings
0.3	November 7th, 2017	Marcus Bointon	Review
1.0	November 9th, 2017	Marcus Bointon	Final version

Contact

For more information about this Document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Overdiemerweg 28 1111 PP Diemen The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between August 28, 2017 and October 12, 2017, Radically Open Security B.V. carried out a code audit for Secure Open Source (Mozilla).

This report contains our findings as well as detailed explanations of exactly how ROS performed the code audit.

1.2 Scope of work

The scope of the penetration test was limited to the following target:

- Graphite font system

Some parts of the source code that are not part of release builds (debugging or tracing code), or are considered deprecated (ALL_TTFUTILS, GRAPHITE2_NSEGCACHE) were explicitly out of scope.

1.3 Project objectives

The objective of the project was to conduct a thorough code review, with particular focus on identifying issues that might be difficult to find by fuzzing.

1.4 Timeline

The Security Audit took place between August and October 2017.

1.5 Results In A Nutshell

Of all the issues found and reported during this audit, only one was rated with an elevated severity, MGR-001 (page 10). It could only possibly be exploited in combination with another issue (allowing it to reach this error condition) but no such issue could be identified.

Further integer overflow conditions were identified in MGR-005 (page 14), MGR-015 (page 24), and MGR-021 (page 29). This was the most represented class of bugs after NULL pointer dereferencing, as in MGR-003 (page 12), MGR-004 (page 13), MGR-008 (page 16), MGR-010 (page 18), MGR-011 (page 19), MGR-017 (page 25), and MGR-019 (page 27). Then, two possible floating point exceptions were found in MGR-007 (page 15) and MGR-009 (page 17). There again, the impact is normally limited to harmless crashes (Denial of Service).

A common security mitigation was found to be explicitly disabled in MGR-002 (page 11). The only other issue with a moderate impact is MGR-006 (page 15), an out-of-bounds read operation, while the remaining issues were rated Low.

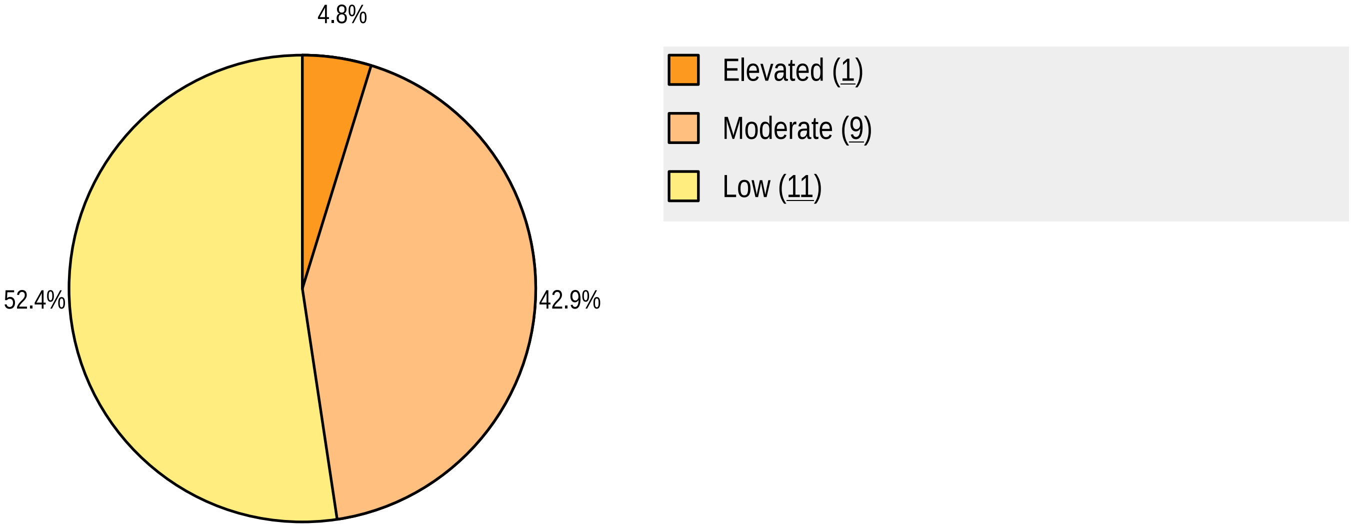
Generally it can be said that the code seems quite robust. The extensive fuzzing conducted previously certainly helped improve the overall level of security.

1.6 Summary of Findings

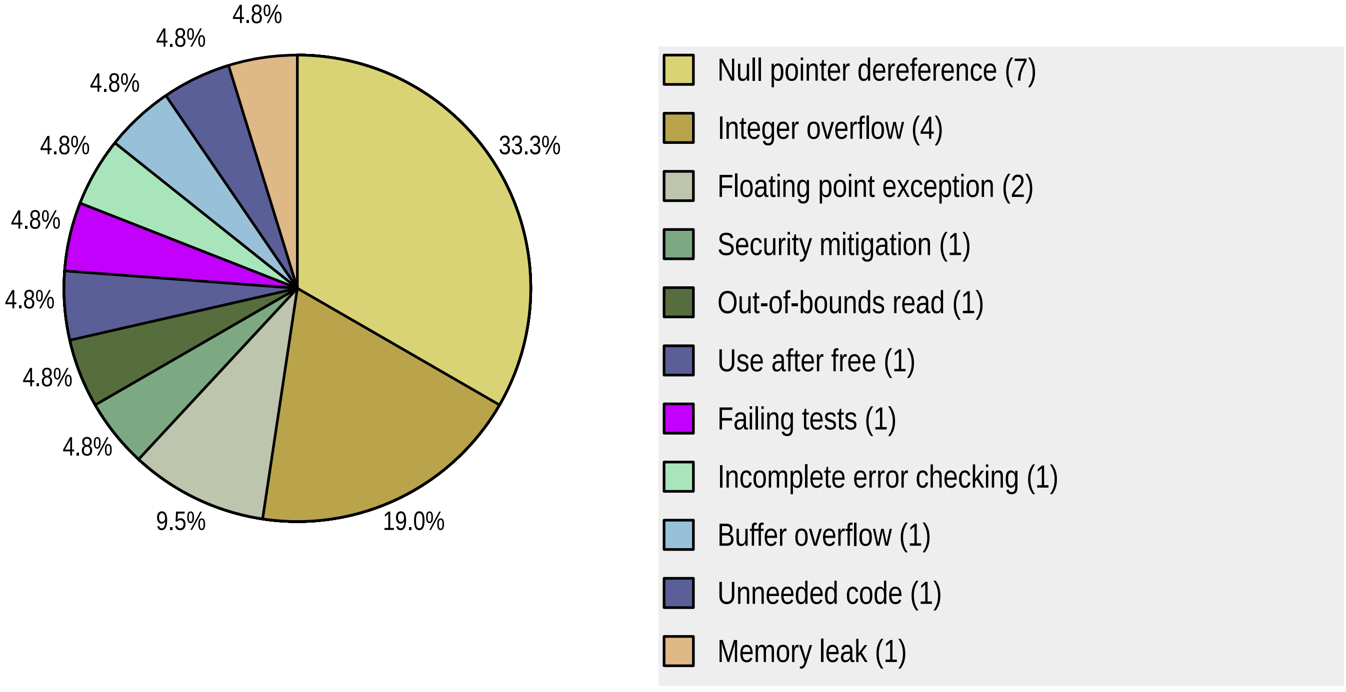
ID	Type	Description	Threat level
MGR-001	Integer overflow	A generic memory allocation routine, gralloc(), wraps the malloc() heap allocator from libc, but with the extra ability to allocate arrays of the type desired. This multiplication is not checked for overflows, thereby possibly allocating less memory than actually intended and without reporting errors. This may result in memory corruption.	Elevated
MGR-002	Security mitigation	Stack Smashing Protection (SSP) is a technology initially developed by IBM (originally called "ProPolice"), and included in the GCC compiler since 2006. It mitigates a number of Buffer Overflow conditions by modifying the layout of variables on the stack and checking canary values when returning from function calls. Although now in use by most software distributions, support for SSP was found to be explicitly disabled in Graphite.	Moderate
MGR-003	NULL Pointer Dereference	Segment::newSlot() has multiple branches which return NULL, this can be passed directly to Segment::freeSlot() which is not prepared to handle such pointers.	Moderate
MGR-004	NULL Pointer Dereference	In Pass::collisionShift() the check for start being non-NULL is confusing, as the loop around it makes sure start is not NULL. However, the variable c might be NULL, as seg->collisionInfo() could return NULL.	Moderate
MGR-005	Integer overflow	The implementation of the List class uses a distance() function, which returns a signed integer. This value is then passed to memory handling functions which expect an unsigned integer; this can lead to various problems.	Moderate
MGR-006	Out-of-bounds Read	Despite being aware of bounds (via m_size), the operator implementation does not check for them.	Moderate
MGR-007	Floating Point Exception	The constructor of the Font class calculates a division with input supplied by the user, which may trigger a floating point exception.	Moderate
MGR-008	NULL Pointer Dereference	The constructor of Font initializes a pointer which might be NULL that is later dereferenced without checking.	Moderate

MGR-009	Floating Point Exception	It is possible to trigger a floating point exception (FPE) with user-supplied input in a Fonts action or constraint code.	Moderate
MGR-010	NULL Pointer Dereference	A NULL pointer might be dereferenced in ShiftCollider::mergeSlot().	Moderate
MGR-011	NULL Pointer Dereference	There is no error check when allocating memory temporarily on the heap in FileFace::get_table_fn(), just before reading data from a file. This may result in an uncontrolled crash in circumstances where the program is unable to allocate more memory.	Low
MGR-012	Use after free	Some code related to logging does not adequately clear a pointer after freeing it. As a result, if this code is called again, an invalid pointer will be dereferenced, possibly allowing code execution.	Low
MGR-013	Failing tests	The LZ4 decompression routine from Graphite was tested with the test suite from the original liblz4 library. The routine did not pass the "fuzzer" test from liblz4, with a number of different errors.	Low
MGR-014	Incomplete error checking	The return value for a "loca table" lookup function is negative on error, whereas the underlying type is size_t (unsigned). Consumers of this function claim to check for errors but may fail to do so in some cases.	Low
MGR-015	Integer overflow	An integer overflow can be found in overrun_copy() of the LZ4 implementation.	Low
MGR-016	Buffer overflow	The overrun_copy function can write out-of-bounds, by at most word-size-1 bytes.	Low
MGR-017	NULL Pointer Dereference	Segment::linkClusters() contains a possible NULL pointer dereference.	Low
MGR-018	Unneeded code	free() is called on a pointer if the pointer is NULL. This is more of a code smell than a vulnerability, however it might hint at deeper issues.	Low
MGR-019	NULL Pointer Dereference	A NULL pointer dereference issue might be found in the constructor of NameAndFeatureRef, although difficult to trigger.	Low
MGR-020	Memory Leak	A memory leak condition occurs when realloc() fails; some memory resources are wasted.	Low
MGR-021	Integer overflow	The list implementation does not handle integer overflows when calling realloc(). This is probably not a problem in the current implementation of Graphite, but it would be safer to check.	Low

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
MGR-001	Integer overflow	Check the multiplication for possible overflows.

MGR-002	Security mitigation	Enable Stack Smashing Protection (SSP) again.
MGR-003	NULL Pointer Dereference	Add appropriate error handling for the pointer returned being NULL.
MGR-004	NULL Pointer Dereference	Handle NULL pointers and integer overflows.
MGR-005	Integer overflow	Handle overflows and negative values before passing the distance to the memory handling functions.
MGR-006	Out-of-bounds Read	Check bounds and handle erroneous indexes accordingly.
MGR-007	Floating Point Exception	Handle also parameters INT_MIN/-1 and divisor being 0.
MGR-008	NULL Pointer Dereference	Handle failure of memory allocation.
MGR-009	Floating Point Exception	Handle also INT_MIN/-1 case in the VM.
MGR-010	NULL Pointer Dereference	Check if `exclSlot` is NULL, and abort the operation if it is.
MGR-011	NULL Pointer Dereference	Handle failure of memory allocation.
MGR-012	Use after free	• Clear the global variable after freeing it. • Check for the variable to be valid before using it.
MGR-013	Failing tests	Investigate the failure cases.
MGR-014	Incomplete error checking	• Improve the internal API when checking for errors. • Review these sanity checks.
MGR-015	Integer overflow	Check for integer overflow and report errors.
MGR-016	Buffer overflow	Possibly ignore, as the LZ4 decompressor prohibits exploitable conditions to trigger.
MGR-017	NULL Pointer Dereference	Introduce a check for NULL and handle accordingly.
MGR-018	Unneeded code	Review and refactor this code.
MGR-019	NULL Pointer Dereference	Review the handling of NULL pointers.
MGR-020	Memory Leak	realloc() into a temporary variable, and free the former variable if realloc() fails to allocate memory.
MGR-021	Integer overflow	Handle overflows.

2 Methodology

2.1 Planning

Our general approach during this code audit was as follows:

1. **Historical Vulnerabilities**
   We looked at previously identified security vulnerabilities to identify possible areas of interest.
2. **Grepping**
   We attempted to identify areas of interest by grepping for memory operations: new, malloc, calloc, realloc, gralloc, free.
3. **Static checks**
   We also used two automated tools, flawfinder and cppcheck, to look for issues. Besides lots of false positives we caught one memory leak.
4. **Concolic analysis**
   Using the Angr framework we isolated and ran a concolic analysis on the LZ4 decompressor, unfortunately the Z3 solver was overwhelmed when confronted with inputs of about 128 MB, which is the upper limit for the decompressor when invoked from Graphite. Smaller input variables provided no exploitable results.
5. **Ignored sources**
   We ignored two conditional compilation directives: GRAPHITE2_NSEGCACHE and ALL_TTFUTILS, as we were told these are being deprecated.

2.2 Risk Classification

Throughout the document, each vulnerability or risk identified has been labeled and categorized as:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/ reputational losses occurring as a result.
- **High**
  High risk of security controls being compromised with the potential for significant financial/ reputational losses occurring as a result.
- **Elevated**
  Elevated risk of security controls being compromised with the potential for material financial/ reputational losses occurring as a result.
- **Moderate**
  Moderate risk of security controls being compromised with the potential for limited financial/ reputational losses occurring as a result.
- **Low**

Low risk of security controls being compromised with measurable negative impacts as a result.

Please note that this risk rating system was taken from the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting.

3 Reconnaissance and Fingerprinting

Through automated scans we were able to find some useful information about the software. We generated many false-positives, mostly because some functions were mistaken for POSIX functions, however we found one memory leak using these tools.

3.1 Automated Scans

As part of our code audit we used the following automated scans:

- flawfinder – https://www.dwheeler.com/flawfinder/
- cppcheck – http://cppcheck.sourceforge.net/

4 Pentest Technical Summary

4.1 Findings

We identified the following issues:

4.1.1 MGR-001 — Potential Integer Overflow in Memory Allocator

Vulnerability ID: MGR-001

Vulnerability type: Integer overflow

Threat level: Elevated

Description:

A generic memory allocation routine, `gralloc()`, wraps the `malloc()` heap allocator from libc, but with the extra ability to allocate arrays of the type desired. This multiplication is not checked for overflows, thereby possibly allocating less memory than actually intended and without reporting errors. This may result in memory corruption.

Technical description:

In file `src/inc/Main.h`, template `gralloc()`, line 83:

```
83 template <typename T> T * gralloc(size_t n)
84 {
85 #ifdef GRAPHITE2_TELEMETRY
86     telemetry::count_bytes(sizeof(T) * n);
87 #endif
88     return static_cast<T*>(malloc(sizeof(T) * n));
89 }
```

The multiplication line 88 can overflow for high values of `n`. The value of `sizeof(T)` is unlikely to be big enough to cause overflows on its own.

The `grzeroalloc()` routine in the same file is not subject to this issue, as it uses `calloc()` instead of `malloc()`, which normally implements a check for overflows.

See also https://www.fefe.de/intof.html and http://undeadly.org/cgi?action=article&sid=20060330071917.

Impact:

Elevated (Some issues can become exploitable when combined with this one)

Recommendation:

Check the multiplication for possible overflows.

4.1.2 MGR-002 — Graphite Builds With the Stack Protector Disabled

Vulnerability ID: MGR-002

Vulnerability type: Security mitigation

Threat level: Moderate

Description:

Stack Smashing Protection (SSP) is a technology initially developed by IBM (originally called "ProPolice"), and included in the GCC compiler since 2006. It mitigates a number of Buffer Overflow conditions by modifying the layout of variables on the stack and checking canary values when returning from function calls. Although now in use by most software distributions, support for SSP was found to be explicitly disabled in Graphite.

Technical description:

On both Linux and Mac OS X, the build system disables SSP explicitly.

In `sources/graphite/src/CMakeLists.txt`:

```
113 if (${CMAKE_SYSTEM_NAME} STREQUAL "Linux") 114     set_target_properties(graphite2 PROPERTIES
  115         COMPILE_FLAGS   "-Wall -Wextra -Wno-unknown-pragmas -Wendif-labels -Wshadow -Wctor-
dtor-privacy -Wnon-virtual-dtor -fno-rtti -fno-exceptions -fvisibility=hidden -fvisibility-inlines-
hidden -fno-stack-protector"
```

```
145 if (${CMAKE_SYSTEM_NAME} STREQUAL "Darwin") 146     set_target_properties(graphite2 PROPERTIES
147         COMPILE_FLAGS   "-Wall -Wextra -Wno-unknown-pragmas -Wimplicit-fallthrough -
Wendif-labels -Wshadow -Wno-ctor-dtor-privacy -Wno-non-virtual-dtor -fno-rtti -fno-exceptions -
fvisibility=hidden -fvisibility-inlines-hidden -fno-stack-protector -mfpmath=sse -msse2"
```

Martin Hosken, from the Graphite project, explained the rationale:

"The -fno-stack-protector went in because it was causing problems (or performance issues)"

Impact:

Moderate (A common mitigation technique is not applied)

Recommendation:

Enable Stack Smashing Protection (SSP) again.

4.1.3 MGR-003 — Graphite/src/Segment.cpp Constructor Possible Null Pointer Dereference

Vulnerability ID: MGR-003

Vulnerability type: NULL Pointer Dereference

Threat level: Moderate

Description:

Segment::newSlot() has multiple branches which return NULL, this can be passed directly to Segment::freeSlot() which is not prepared to handle such pointers.

Technical description:

The constructor of the `Segment` class contains the following snippet:

```
Segment::Segment(unsigned int numchars, const Face* face, uint32 script, int textDir)
...
{
    freeSlot(newSlot());
```

`newSlot()` can return NULL, but `freeSlot` does not handle this.

Impact:

Moderate (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Add appropriate error handling for the pointer returned being NULL.

4.1.4 MGR-004 — Graphite/src/Pass.cpp CollisionShift NULL Pointer Dereference & Integer Overflow

Vulnerability ID: MGR-004

Vulnerability type: NULL Pointer Dereference

Threat level: Moderate

Description:

In `Pass::collisionShift()` the check for `start` being non-NULL is confusing, as the loop around it makes sure `start` is not NULL. However, the variable `c` might be NULL, as `seg->collisionInfo()` could return NULL.

Technical description:

```
const SlotCollision * c = seg->collisionInfo(s);
if (start && (c->flags() & (SlotCollision::COLL_FIX | SlotCollision::COLL_KERN)) ==
 SlotCollision::COLL_FIX
```

`start` is always non-NULL here, and `seg->collisionInfo(s)` could return NULL:

```
SlotCollision *collisionInfo(const Slot *s) const { return m_collisions ? m_collisions + s-
>index() : 0; }
```

This code can lead to a possible NULL pointer dereference. Also, there might be an integer overflow on 32-bit systems, as `s->index()` returns a `uint32`.

The Segment class member-variable `m_collisions` gets initialized from `Face::runGraphite()` using `seg->initCollisions()`, which properly reports if `m_collisions` has been allocated correctly. So it should not be NULL, however TOCTOU possibilities arise if this pointer can be zeroed somehow.

Furthermore it might also be interesting to check `Pass::collisionKern()`, where `seg->collisionInfo(s)` is also invoked and can return NULL, but is not checked for this possibility.

Impact:

Moderate

Recommendation:

Handle NULL pointers and integer overflows.

4.1.5 MGR-005 — Graphite/src/inc/List.h Possible Integer Overflow

Vulnerability ID: MGR-005

Vulnerability type: Integer overflow

Threat level: Moderate

Description:

The implementation of the `List` class uses a `distance()` function, which returns a signed integer. This value is then passed to memory handling functions which expect an unsigned integer; this can lead to various problems.

Technical description:

`distance()` returns a signed integer, which is then passed as a parameter to various functions where a negative value might be dangerous. These issues are probably very difficult to trigger within the current implementation of Graphite, but it would make sense to add checks for overflows after the use of `distance()` to be sure. One example is listed below; more can be found in the same file.

```
typename Vector<T>::iterator Vector<T>::_insert_default(iterator p, size_t n) {
    // Move tail if there is one
    if (p != end()) memmove(p + n, p, distance(p,end())*sizeof(T));
```

Impact:

Moderate

Recommendation:

Handle overflows and negative values before passing the distance to the memory handling functions.

4.1.6 MGR-006 — Graphite/src/inc/Rule.h Slotmap::operator[] Does Not Check Bounds

Vulnerability ID: MGR-006

Vulnerability type: Out-of-bounds Read

Threat level: Moderate

Description:

Despite being aware of bounds (via `m_size`), the operator implementation does not check for them.

Technical description:

Though this issue does not seem to be exploitable at first glance, further TOCTOU issues (Time Of Check, Time Of Use) could make it actually dangerous.

Impact:

Moderate

Recommendation:

Check bounds and handle erroneous indexes accordingly.

4.1.7 MGR-007 — Graphite/src/Font.cpp Font::Font() Division-related FPE

Vulnerability ID: MGR-007

Vulnerability type: Floating Point Exception

Threat level: Moderate

Description:

The constructor of the Font class calculates a division with input supplied by the user, which may trigger a floating point exception.

Technical description:

```
Font::Font(float ppm, const Face & f, const void * appFontHandle, const gr_font_ops * ops)
: m_appFontHandle(appFontHandle ? appFontHandle : this),
  m_face(f),
  m_scale(ppm / f.glyphs().unitsPerEm()),
```

`f.glyphs().unitsPerEm()` can be 0 (or -1 and ppm INT_MIN). `glyps().unitsPerEM()` might be 0, as it comes from the user supplied font constructor in glyphcache.cpp:

```
_glyphs ? _glyph_loader->units_per_em() : 0
```

and further down the call-graph ttfutil.cpp in DesignUnits:

```
const Sfnt::FontHeader * pTable = reinterpret_cast<const Sfnt::FontHeader *>(pHead);
return be::swap(pTable->units_per_em);
```

But later in the call-graph `load_face` (from `gr_make_face_with_ops`) calls `Face::readGlyphs()` which verifies if `units_per_em > 0`, this happens earlier than the font instantiation and is a precondition for it:

```
gr_face *face = gr_make_file_face(argv[1], 0);                          /*<1>*/
if (!face) return 1;
font = gr_make_font(pointsize * dpi / 72.0f, face);                     /*<2>*/
```

So triggering this FPE is very difficult, but could be prone to TOCTOU problems. To reduce these chances it would make sense to include checks for the `divisor!=0` and also the division not being of the `(INT_MIN/-1)` kind.

Impact:

Moderate (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Handle also parameters INT_MIN/-1 and divisor being 0.

4.1.8 MGR-008 — Graphite/src/Font.cpp M_advances NULL Pointer Dereferences

Vulnerability ID: MGR-008

Vulnerability type: NULL Pointer Dereference

Threat level: Moderate

Description:

The constructor of `Font` initializes a pointer which might be NULL that is later dereferenced without checking.

Technical description:

```
Font::Font(float ppm, const Face & f, const void * appFontHandle, const gr_font_ops * ops) {
    ...
    m_advances = gralloc<float>(nGlyphs);
    if (m_advances)
    {
        for (float *advp = m_advances; nGlyphs; --nGlyphs, ++advp)
            *advp = INVALID_ADVANCE;
    }
}
```

The member variable of the `Font` class `m_advances` is dereferenced from `Slot::finalize()` and `gr_slot_advance_X()`.

Impact:

Moderate

Recommendation:

Handle failure of memory allocation.

4.1.9 MGR-009 — Floating Point Exception in VM

Vulnerability ID: MGR-009

Vulnerability type: Floating Point Exception

Threat level: Moderate

Description:

It is possible to trigger a floating point exception (FPE) with user-supplied input in a Fonts action or constraint code.

Technical description:

By compiling a custom SILF action using ttx (from fonttools>=3.16.0) it is possible to generate a floating point exception. To reproduce insert the following snippet into the ttx generated from `tests/fonts/MagyarLinLibertineG.ttf` into the beginning of action rule `rule index="39" precontext="1" sortkey="2"`

```
PUSH_LONG(2147483648)
PUSH_LONG(4294967295)
```

```
DIV
```

Also replace the `ret_zero` with a `pop_ret` at the end of this action, then compile it back into a ttf, and invoke `./tests/examples/simpletests/fonts/MagyarLinLibertineG\#1.ttf 'ŰÁ'` to trigger a FPE.

This is because if the dividend is INT_MIN (0x80000000) and the divisor is -1, then the result of the division is undefined behaviour in C/C++.

Impact:

Moderate (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Handle also INT_MIN/-1 case in the VM.

4.1.10 MGR-010 — Possible NULL Pointer Dereference in ShiftCollider::mergeSlot()

Vulnerability ID: MGR-010

Vulnerability type: NULL Pointer Dereference

Threat level: Moderate

Description:

A NULL pointer might be dereferenced in `ShiftCollider::mergeSlot()`.

Technical description:

In `ShiftCollider::mergeSlot()` the following snippet:

```
...
if (cslot->exclGlyph() > 0 && gc.check(cslot->exclGlyph()) && !isExclusion)
{
    // Set up the bogus slot representing the exclusion glyph.
    Slot *exclSlot = seg->newSlot();
    exclSlot->setGlyph(seg, cslot->exclGlyph());
```

`seg->newSlot()` could return a NULL pointer, but this is not checked.

Impact:

Moderate (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Check if `exclSlot` is NULL, and abort the operation if it is.

4.1.11 MGR-011 — Potential Crash in FileFace::get_table_fn()

Vulnerability ID: MGR-011

Vulnerability type: NULL Pointer Dereference

Threat level: Low

Description:

There is no error check when allocating memory temporarily on the heap in `FileFace::get_table_fn()`, just before reading data from a file. This may result in an uncontrolled crash in circumstances where the program is unable to allocate more memory.

Technical description:

In file `src/FileFace.cpp`, method `FileFace::get_table_fn()`, line 95:

```
80 const void *FileFace::get_table_fn(const void* appFaceHandle, unsigned int name, size_t *len)
81 { [...]
94     tbl = malloc(tbl_len);
95     if (fread(tbl, 1, tbl_len, file_face._file) != tbl_len)
96     {
97         free(tbl);
98         return 0;
99     }
```

The `tbl` variable may be `NULL` if `malloc()` fails to allocate memory, in which case `fread()` will likely try to write data at this address. This will typically result in a crash.

Impact:

Low (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Handle failure of memory allocation.

4.1.12 MGR-012 — Potential Use After Free When Logging

Vulnerability ID: MGR-012

Vulnerability type: Use after free

Threat level: Low

Description:

Some code related to logging does not adequately clear a pointer after freeing it. As a result, if this code is called again, an invalid pointer will be dereferenced, possibly allowing code execution.

Technical description:

In file `src/gr_logging.cpp`, function `gr_stop_logging()`, line 110:

```
110 void gr_stop_logging(GR_MAYBE_UNUSED gr_face * face)
111 {
112 #if !defined GRAPHITE2_NTRACING
113     if (face && face->logger())
114     {
115         FILE * log = face->logger()->stream();
116         face->setLogger(0);
117         fclose(log);
118     }
119     else if (!face && global_log)
120     {
121         FILE * log = global_log->stream();
122         delete global_log;
123         fclose(log);
124     }
125 #endif
126 }
```

If this code is compiled in and `gr_stop_logging()` is called twice (with `face` set to `NULL`), then `global_log` will be used (line 121) after being free'd (line 122), as it is not cleared as it should be. An attacker able to reach this condition and able to control the memory pointed at by `global_log` will be able to execute code.

Impact:

Low (Apparently not used in release builds)

Recommendation:

- Clear the global variable after freeing it.
- Check for the variable to be valid before using it.

4.1.13 MGR-013 — The LZ4 Parser Does Not Pass the Tests From Liblz4

Vulnerability ID: MGR-013

Vulnerability type: Failing tests

Threat level: Low

Description:

The LZ4 decompression routine from Graphite was tested with the test suite from the original liblz4 library. The routine did not pass the "fuzzer" test from liblz4, with a number of different errors.

Technical description:

The following is sample output from the wrapper written for liblz4's `fuzzer` test:

```
Test 17 : LZ4_decompress_safe should have failed, due to input size being too large (seed 3183,
 cycle 0)
Test 2 : LZ4_decompress_safe() failed on data compressed by LZ4_compress_destSize (seed 3183, cycle
 26)
Test 4 : LZ4_decompress_safe() failed on data compressed by LZ4_compressHC_destSize (seed 3183,
 cycle 26)
Test 12 : LZ4_decompress_safe failed despite sufficient space (seed 3183, cycle 26)
Test 13 : LZ4_decompress_safe failed despite amply sufficient space (seed 3183, cycle 26)
Test 2 : LZ4_decompress_safe() corrupted decoded data (seed 3183, cycle 189)
Test 2 : LZ4_decompress_safe() failed : did not fully decompressed data (seed 3183, cycle 211)
Test 13 : LZ4_decompress_safe did not regenerate original data (seed 3183, cycle 214)
```

This LZ4 code is used in `src/Face.cpp`, method `Face::Table::decompress()`, line 313:

```
313 Error Face::Table::decompress()
314 {
315     Error e;
316     if (e.test(_sz < 5 * sizeof(uint32), E_BADSIZE))
317         return e;
318     byte * uncompressed_table = 0;
319     size_t uncompressed_size = 0;
320
321     const byte * p = _p;
322     const uint32 version = be::read<uint32>(p);    // Table version number.
323
324     // The scheme is in the top 5 bits of the 1st uint32.
325     const uint32 hdr = be::read<uint32>(p);
326     switch(compression(hdr >> 27))
327     {
328     case NONE: return e;
329
330     case LZ4:
331     {
332         uncompressed_size  = hdr & 0x07ffffff;
333         uncompressed_table = gralloc<byte>(uncompressed_size);
334         if (!e.test(!uncompressed_table || uncompressed_size < 4, E_OUTOFMEM))
335         {
336             memset(uncompressed_table, 0, 4);   // make sure version number is initialised
337             // coverity[forward_null : FALSE] - uncompressed_table has been checked so can't be
 null
338             // coverity[checked_return : FALSE] - we test e later
339             e.test(lz4::decompress(p, _sz - 2*sizeof(uint32), uncompressed_table,
 uncompressed_size) != signed(uncompressed_size), E_SHRINKERFAILED);
340         }
341         break;
342     }
```

Impact:

Low (Some valid fonts may fail to decompress)

Recommendation:

Investigate the failure cases.

4.1.14 MGR-014 — Incomplete Sanity Check When Looking up Glyphs

Vulnerability ID: MGR-014

Vulnerability type: Incomplete error checking

Threat level: Low

Description:

The return value for a "loca table" lookup function is negative on error, whereas the underlying type is `size_t` (unsigned). Consumers of this function claim to check for errors but may fail to do so in some cases.

Technical description:

In file `src/TtfUtil.cpp`, function `LocaLookup()`, line :

```
1202 /*-----------------------------------------------------------------------------------------------
1203     Return the offset stored in the loca table for the given Glyph ID.
1204     (This offset is into the glyf table.)
1205     Return -1 if the lookup failed.
1206     Technically this method should return an unsigned long but it is unlikely the offset will
1207         exceed 2^31.
1208 ----------------------------------------------------------------------------------------------*/
1209 size_t LocaLookup(gid16 nGlyphId,
1210         const void * pLoca, size_t lLocaSize,
1211         const void * pHead) // throw (std::out_of_range)
1212 {
1213     const Sfnt::FontHeader * pTable = reinterpret_cast<const Sfnt::FontHeader *>(pHead);
1214     size_t res = -2; [...]
1222             res = be::peek<uint16>(pShortTable + nGlyphId) << 1;
1223             if (res == static_cast<size_t>(be::peek<uint16>(pShortTable + nGlyphId + 1) << 1))
1224                 return -1; [...]
1232             res = be::peek<uint32>(pLongTable + nGlyphId);
1233             if (res == static_cast<size_t>(be::peek<uint32>(pLongTable + nGlyphId + 1)))
1234                 return -1; [...]
1238     // only get here if glyph id was bad
1239     return res;
```

As a result, it seems that this function may:

- return -1 if the lookup failed;
- return -2 if nothing was recognized;
- or return any value for `res` according to what was read.

On 32-bit platforms (or anywhere where `size_t` is the same size as `uint32`) the value read into `res` may also be `-1`, even on error, as it will be cast to a signed value later. This confuses consumers of this function.

Moreover, some consumers may be confused by further corner-cases, such as in `GlyfLookup()`:

```
1637 /*-----------------------------------------------------------------------------------------
1638     Return a pointer into the glyf table based on the given tables and Glyph ID
1639     Since this method doesn't check for spaces, it is good to call IsSpace before using it.
1640     Return NULL on error.
1641 ---------------------------------------------------------------------------------------*/
1642 void * GlyfLookup(gid16 nGlyphId, const void * pGlyf, const void * pLoca,
1643                            size_t lGlyfSize, size_t lLocaSize, const void * pHead)
1644 { [...]
1668     long lGlyfOffset = LocaLookup(nGlyphId, pLoca, lLocaSize, pHead);
1669     void * pSimpleGlyf = GlyfLookup(pGlyf, lGlyfOffset, lGlyfSize); // invalid loca offset
 returns null
1670     return pSimpleGlyf;
1671 }
```

According to the comment on line 1669, invalid offsets should be caught by `GlyfLookup()`, in which case it is expected to return `NULL`. This may not be the case:

```
/*-----------------------------------------------------------------------------------------
    Return a pointer into the glyf table based on the given offset (from LocaLookup).
    Return NULL on error.
---------------------------------------------------------------------------------------*/
void * GlyfLookup(const void * pGlyf, size_t nGlyfOffset, size_t nTableLen)
{
    const uint8 * pByte = reinterpret_cast<const uint8 *>(pGlyf);
        if (nGlyfOffset + pByte < pByte || nGlyfOffset + sizeof(Sfnt::Glyph) >= nTableLen)
            return NULL;
    return const_cast<uint8 *>(pByte + nGlyfOffset);
}
```

This function performs two checks:

- `nGlyfOffset + pByte < pByte` will always fail if `pByte` (so really `pGlyf`) is `NULL`, or if `nGlyfOffset` represents success as `-1`;
- `nGlyfOffset + sizeof(Sfnt::Glyph) >= nTableLen` will wrap around for any negative value close to 0 (e.g. `-1` or `-2`) and therefore almost always be smaller than `nTableLen`.

In the unlikely case that the address for `pByte` is situated lower than `sizeof(Sfnt::Glyph)` (about 10 to 20 depending on alignment), a value of `nGlyfOffset` between `-pByte` and `-sizeof(Sfnt::Glyph)` will therefore bypass this test on 32-bit platforms.

Impact:

Low (Some sanity checks may be bypassed in unlikely conditions)

Recommendation:

- Improve the internal API when checking for errors.

- Review these sanity checks.

4.1.15 MGR-015 — Graphite/src/inc/Compression.h::overrun_copy Integer Overflow Leads to Uninitialized Buffer

Vulnerability ID: MGR-015

Vulnerability type: Integer overflow

Threat level: Low

Description:

An integer overflow can be found in `overrun_copy()` of the LZ4 implementation.

Technical description:

```
u8 * overrun_copy(u8 * d, u8 const * s, size_t n) {
    size_t const WS = sizeof(unsigned long);
    u8 const * e = s + n;
    do
    {
        unaligned_copy<WS>(d, s);
        d += WS;
        s += WS;
    }
    while (s < e);
    d-=(s-e);

    return d;
}
```

If `e` is overflowed, then only one word will be copied, but `d` will be updated as if all `n` bytes were copied.

Impact:

Low

Recommendation:

Check for integer overflow and report errors.

4.1.16 MGR-016 — Graphite/src/inc/Compression.h::overrun_copy Possible Buffer Overflow

Vulnerability ID: MGR-016

Vulnerability type: Buffer overflow

Threat level: Low

Description:

The `overrun_copy` function can write out-of-bounds, by at most word-size-1 bytes.

Technical description:

The following snippet shows the relevant code:

```
u8 * overrun_copy(u8 * d, u8 const * s, size_t n) {
    size_t const WS = sizeof(unsigned long);
    u8 const * e = s + n;
    do
    {
        unaligned_copy<WS>(d, s);
        d += WS;
        s += WS;
    }
    while (s < e);
    d-=(s-e);

    return d;
}
```

Since `overrun_copy` only copies word-sized chunks and `n % wordsize != 0`, there is a chance for a write operation outside of the corresponding buffer. It seems the two invocations of this function in `src/Decompressor.cpp` do make sure there is always more than wordsize bytes at the end of the buffer. This makes this issue only potential, in the case of another TOCTOU issue.

Impact:

Low

Recommendation:

Possibly ignore, as the LZ4 decompressor prohibits exploitable conditions to trigger.

4.1.17 MGR-017 — Graphite/src/Segment.cpp linkClusters Null Pointer Dereference

Vulnerability ID: MGR-017

Vulnerability type: NULL Pointer Dereference

Threat level: Low

Description:

Segment::linkClusters() contains a possible NULL pointer dereference.

Technical description:

The following snippet shows the function of interest:

```
void Segment::linkClusters(Slot *s, Slot * end)
{
    end = end->next();

    for (; s != end && !s->isBase(); s = s->next());
    Slot * ls = s;
```

This function is only called from `Segment::finalise()`, which guards the `s` parameter against being NULL. The `end` parameter is not guarded however. Although it seems natural that if there is is a start slot, there must also be a last slot, this is not clear immediately from the code.

Impact:

Low (Availability can be restricted due to Denial of Service attacks)

Recommendation:

Introduce a check for NULL and handle accordingly.

4.1.18 MGR-018 — Graphite/src/inc/Sparse.h Sparse(x,y) Code Smell

Vulnerability ID: MGR-018

Vulnerability type: Unneeded code

Threat level: Low

Description:

`free()` is called on a pointer if the pointer is NULL. This is more of a code smell than a vulnerability, however it might hint at deeper issues.

Technical description:

In the constructor of the `Sparse` class:

```
    m_array.values = grzeroalloc<mapped_type>((m_nchunks*sizeof(chunk) + sizeof(mapped_type)-1)
                                             / sizeof(mapped_type)
                                             + n_values);

    if (m_array.values == 0)
    {
        free(m_array.values); m_array.map=0;
        return;
    }
```

It is unclear why `free()` is called here. It is probably not a security bug, but it would be interesting to check the developer's intent here.

Impact:

Low

Recommendation:

Review and refactor this code.

4.1.19 MGR-019 — Graphite/src/inc/FeatureMap.h Possible NULL Pointer Dereference

Vulnerability ID: MGR-019

Vulnerability type: NULL Pointer Dereference

Threat level: Low

Description:

A NULL pointer dereference issue might be found in the constructor of `NameAndFeatureRef`, although difficult to trigger.

Technical description:

```
NameAndFeatureRef(const FeatureRef* p/*not NULL*/) : m_name(p->getId()), m_pFRef(p) {}
```

A possible NULL pointer dereference, maybe impossible to reach with a NULL in the current implementation. TOCTOU style abuse might make this exploitable though.

Impact:

Low

Recommendation:

Review the handling of NULL pointers.

4.1.20 MGR-020 — Graphite/src/Code.cpp Machine::Code::Code() Constructor Possible Memory Leak

Vulnerability ID: MGR-020

Vulnerability type: Memory Leak

Threat level: Low

Description:

A memory leak condition occurs when `realloc()` fails; some memory resources are wasted.

Technical description:

`realloc()` returns NULL in case of error, but does not free the original segment. This leads to a memory leak in the following code:

```
    else
        _code = static_cast<instr *>(realloc(_code, total_sz));
    _data = reinterpret_cast<byte *>(_code + (_instr_count+1));

    if (!_code)
    {
        failure(alloc_failed);
        return;
    }
```

The memory leak itself is quite benign, but it should be fixed nonetheless.

Impact:

Low

Recommendation:

realloc() into a temporary variable, and free the former variable if realloc() fails to allocate memory.

4.1.21 MGR-021 — Graphite/src/inc/List.h Possible Integer/memory Overflow

Vulnerability ID: MGR-021

Vulnerability type: Integer overflow

Threat level: Low

Description:

The list implementation does not handle integer overflows when calling `realloc()`. This is probably not a problem in the current implementation of Graphite, but it would be safer to check.

Technical description:

With carefully crafted input it might be possible to cause an overflow in the `List` implementation:

```
void Vector<T>::reserve(size_t n) {
...
m_first = static_cast<T*>(realloc(m_first, n*sizeof(T)));
```

Impact:

Low

Recommendation:

Handle overflows.

5 Future Work

- **Additional fuzzing on the VM**
  Given the considerable amount of effort put into fuzzing before this project, we were specifically tasked to look for flaws typically not covered by fuzzers. It still seems possible to find more complicated issues by fuzzing though. To that effect, a specific setup could be created, fuzzing only a particular action or constraint in the Silf table. The surrounding sanity checks, addresses and checksums could then be left untouched in the fuzzed font, without affecting the results of the fuzzer.
- **Deeper focus on the state engine**
  We did not look for logic errors in the state engine. We suspect a Denial of Service condition could be reached, where the state engine would be tricked to loop infinitely. A path to this situation could also be found while fuzzing the Silf table, as suggested above.

6 Conclusion

Two particular classes of bugs were uncovered during the audit: integer overflows and NULL pointer dereferences. While the former may be exploitable in some conditions, no real danger was identified. Most of the remaining issues should only trigger controlled, harmless crashes in normal conditions.

As a consequence the code looks generally robust, even though some TOCTOU conditions might still be lurking. It is however apparent that there has been effort put into fuzzing and hardening Graphite over the past year.

Finally we want to emphasize that security is a continuous process; this penetration test is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end. Do not hesitate to let us know if you have any further questions or need further clarification of anything in this report.

Appendix 1 Testing team

Stefan Marsiske	Stefan runs workshops on radare2, embedded hardware, lock-picking, soldering, gnuradio/SDR, reverse-engineering, and crypto topics. In 2015 he scored in the top 10 of the Conference on Cryptographic Hardware and Embedded Systems Challenge. He has run training courses on OPSEC for journalists and NGOs.
Pierre Pronchery	Pierre Pronchery is a Senior IT-Security Consultant and an accomplished developer. Freelancing for about a decade now, he could be found auditing major companies in the Telecommunications and Finance sectors, or supporting the Open Source Software and Hardware movements. He is a developer for the NetBSD Foundation since 2012, and more recently, a co-founder of Defora Networks GbR in Germany.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.