SPOTCODEREVIEWS

Debug Review

2018-157

Client

Long-Running Process Errors After Update

horst@spotcodereviews.com

2018-11-17

Contents

1 Report Confidentiality (Mutually Voided)

Unless mutually voided, information provided in this report is confidential and must not be disclosed to third parties.

2 Executive Summary

2.1 Review Subject

The client operates a system where the communication between individual components is based on SOAP messaging over an HTTP transport. After a software update involving custom code and open source software, the client experienced HTTP request errors in the context of some long-running processes in a production environment.

2.2 Recommendations

In the course of the debug review, the root cause was identified as a `pthread_t` resource leak that lead to hitting the glibc-internal `pthread_t` limit which in turn triggered errors during thread-based asynchronous cURL DNS resolutions. By addressing the resource leak, the issue can be resolved.

3 Timeboxes

The effort required for a code review is inherently variable. To minimize risk, most reviews are limited by means of a timebox - a fixed time period within which the code review takes place. The client agreed to the initial timebox during the briefing and can request additional timeboxes thereafter (subject to availability). Communication and report compilation do not require a timebox, they are covered by timebox fees unless other agreements exist.

Timeboxes				
Date	Contact	Timebox (h)	*P*(*SubstantialInput*)	Fee Type
2018-11-14	Client	4	Medium	Fixed
• Performed initial analysis • Reviewed cURL code • Ruled out DNS as root cause • Identified resource leak as root cause				
2018-11-15	Client	4	High	Fixed
• Reviewed custom code • Identified resource leak in custom code • Recommended resource leak fix • Recommended proactive prevention based on load test resource monitoring				

4 Briefing

4.1 Environment

Environment	
Architecture	x86_64
Platform	Linux
Languages	C
Open Source Software	RHEL 7.5/7.6 (Kernel 3.10.0), cURL (7.62.0)

4.2 Context

The HTTP request errors produced the following error message:

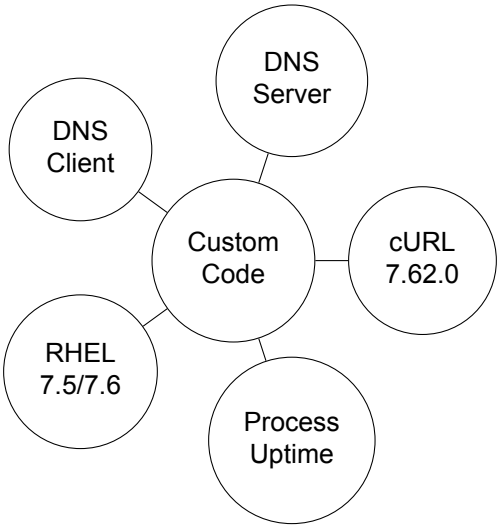
```
HTTP request failed (Error Code: 6)
```

Based on the fact that cURL is used to communicate via HTTP, the error code must be interpreted using the cURL documentation and/or source code:

```
CURLE_COULDNT_RESOLVE_HOST
```

The custom code uses the cURL easy interface to interact with the library. More detailed error information is unaviable because `CURLOPT_ERRORBUFFER` is not used for error handling. Consequently, the review must be based on the assumption that the root cause is indeed DNS-related.

The error scenario affecting custom code has the following known dependencies:



Custom Code. Updated, cURL-related code hasn't changed. The full diff is too large for manual inspection.

cURL. Unchanged (distributed separately).

DNS Server. Unchanged.

DNS Client. Unchanged. The client does not use nscd to cache DNS requests.

RHEL. Updated from 7.5 to 7.6 (including the Linux Kernel).

Process Uptime. The error has never occurred before and not all long-running processes are affected. Log analysis with the client revealed that affected long-running processes have a considerably longer uptime than unaffected processes. Before the update, those long-running processes completed processing well before the current uptime at the point-in-time of the error. The reason for the longer execution time is unknown. However, that fact might be related to the root cause.

As of the briefing, the error occurred several times. In all cases, a longer than usual process uptime could be identified. The assumption is that long-running processes exceeding a specific process uptime threshold (more or less consistent across current occurrences) are affected by the error. The time of day varied and seems to be unrelated. It is unknown whether or not the error occurs permanently for a given process because those processes treat the error as fatal.

Generally speaking, errors triggered by a longer process uptime may be related to resource exhaustion, particularly if the process uptime is similar for affected processes. Combined with the fact that neither the cURL-related code nor cURL itself were updated, the DNS-related components haven't changed and RHEL updates are unlikely to affect this scenario, resource usage in relation to cURL and call sites setting `CURLE_COULDNT_RESOLVE_HOST` must be investigated.

The issue cannot be reproduced in a test environment, so the analysis must be based on the production environment, must not affect performance significantly, and should not require process restarts.

5 Analysis

5.1 Action Item: Eliminate DNS Dependency

Risk-Reward Analysis	
Risk	Minimal (human error resulting in incorrect/missing entries)
Mitigation Potential	Minimal (potential, yet unlikely cause can be ruled out)

Facts provided during the briefing do not point to an actual DNS issue. A DNS error occurring after a faily consistent process uptime is highly unlikely. To rule out the DNS route, I recommend bypassing the DNS server by adding required hosts to `/etc/hosts`.

Theoretical scenarios affecting DNS stability:

- Excessive DNS server response times leading to timeouts
- Packet loss leading to timeouts (UDP/TCP)
- Timeouts could cause excessive resource usage / resource exhaustion for multithreaded code triggering DNS errors (depends on asynchronous request handling)

5.1.1 Results

The change had no effect on the error. Thus, the DNS server can be ruled out. The DNS client would have to fail in a manner that affects both, DNS- and hosts-based resolution, which is highly unlikely.

5.2 cURL Error Call Sites

To identify a list of possible circumstances leading to `CURLE_COULDNT_RESOLVE_HOST` besides actual DNS erros, I inspected the respective cURL error call sites.

cURL creates a new thread for a DNS request to enable asynchronous processing and timeouts.


```
685   if(init_resolve_thread(conn, hostname, port, &hints)) {
686     *waitp = 1; /* expect asynchronous response */
687     return NULL;
688   }
689
690   failf(data, "getaddrinfo() thread failed to start\n");
691   return NULL;
```

curl-7.62.0/lib/asyn-thread.c

`init_resolve_thread` essentially calls `Curl_thread_create`, a thin `pthread_create` wrapper.

```
62  curl_thread_t Curl_thread_create(unsigned int (*func) (void *), void *arg)
63  {
64    curl_thread_t t = malloc(sizeof(pthread_t));
65    struct curl_actual_call *ac = malloc(sizeof(struct curl_actual_call));
66    if(!(ac && t))
67      goto err;
68
69    ac->func = func;
70    ac->arg = arg;
71
72    if(pthread_create(t, NULL, curl_thread_create_thunk, ac) != 0)
73      goto err;
74
75    return t;
76
77  err:
78    free(t);
79    free(ac);
80    return curl_thread_t_null;
81  }
```

curl-7.62.0/lib/curl_threads.c

Besides potential out-of-memory situations, which are highly unlikely, the most likely error scenario is an error returned by `pthread_create`. Two return values are particularly relevant:

```
       EAGAIN Insufficient resources to create another thread.

       EAGAIN A system-imposed limit on the number of threads was
              encountered.  There are a number of limits that may
              trigger this error: the RLIMIT_NPROC soft resource limit
              (set via setrlimit(2)), which limits the number of
              processes and threads for a real user ID, was reached; the
              kernel's system-wide limit on the number of processes and
              threads, /proc/sys/kernel/threads-max, was reached (see
              proc(5)); or the maximum number of PIDs,
              /proc/sys/kernel/pid_max, was reached (see proc(5)).
```

`EAGAIN` indicates that

- `RLIMIT_PROC` was reached
- `/proc/sys/kernel/threads-max` was reached
- `/proc/sys/kernel/pid_max` was reached
- A glibc-internal implementation-specific limit was reached

- An out-of-memory situation was reached

The long-running processes host 100-200 threads at any given time. On the affected systems, `RLIMIT_PROC` is 4096, `/proc/sys/kernel/threads-max` is 131072, `/proc/sys/kernel/pid_max` is 128040. Since the last restart, none of the systems have had less then 128 GB of available memory (based on `free`).

Reaching glibc-internal limits is not possible under normal circumstances because other limits are more likely to be reached earlier. One particular situation that makes this likely is a resource leak caused by creating new threads but not detaching them directly or indirectly. The glibc-internal limit can be determined empirically with the following sample:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *threadmain(void *args)
5  {
6    return NULL;
7  }
8
9  int main(int argc, char **args)
10 {
11   unsigned count = 0;
12   pthread_t thread;
13   while (1) {
14     int ret = pthread_create(&thread, NULL, threadmain, NULL);
15     if (ret != 0) {
16       printf("error %d count %u\n", ret, count);
17       break;
18     }
19     count++;
20   }
21   return 0;
22 }
```

maxpthreads.c

The sample returns a limit of 32747 and, in the current glibc implementation, is primarily influenced by `/proc/sys/vm/max_map_count`. `pthread_create` allocates the thread stack via `mmap` and installs a guard page at the end of the stack for stack overflow detection using `protect`. This result in two memory mappings per thread creation:

```
7fffeefd1000-7fffef7d1000 rw-p 00000000 00:00 0
7fffef7d1000-7fffef7d2000 ---p 00000000 00:00 0
```

$65530 - 32747 \cdot 2 = 36$ remaining mappings, the number of mappings required by the sample for library initialization. The affected systems use the `/proc/sys/vm/max_map_count` default. If the long-running process were to leak one `pthread_t` per minute, the limit would be reached in 22.75 days, 0.38 days for one per second.

5.3 Action Item: Monitor Thread Counts

Risk-Reward Analysis	
Risk	Minimal (human error resulting in manual command execution)
Mitigation Potential	None (input for further action)

According to the cURL code review, an excessive amount of running threads can trigger the error in question. To rule this out, the thread counts of affected processes should be monitored. One simple way to accomplish this if the PID is known:


```
ls -1 /proc/self/task | tail -n +2 | wc -l
```

An increasing number of threads could indicate hangs, deadlocks or excessive run times of threads in any area of the application.

5.3.1 Results

The number of threads ranges between 100 and 200 with no apparent spikes. As a result, the thread count can be ruled out.

5.4 Action Item: Determine cURL Error Location

Risk-Reward Analysis	
Risk	Low (process termination resulting from gdb session, human error resulting in manual command execution)
Mitigation Potential	None (input for further action)

Long-running processes should not be restarted or slowed down by introducing significant overhead. So, the method of identifying the cURL error location must be minimally invasive.

Using GDB and debug symbols, identifying the error location can be attempted. The goal is find out if `pthread_create` actually fails and, if it does, whether or not `EAGAIN` is returned. Based on the cURL code analysis and assembly, I identified three instruction offsets that should support that goal. The build is a production build, so branches in the code do not necessary correspond to branches at the instruction level. Therefore, the addresses of breakpoints were chosen manually to guarantee that the break location has value.

```
gdb \
  -p <pid> \
  -batch \
  -ex "set breakpoint pending on" \
  -ex "b curl_easy_init" \
  -ex "cont" \
  -ex "disable 1" \
  -ex "b *(Curl_thread_create+132)" \
  -ex "b *(Curl_resolver_getaddrinfo+893)" \
  -ex "b *(Curl_resolver_getaddrinfo+967)" \
  -ex "cont" \
  -ex "generate" \
  -ex "print \$eax" \
  -ex "info registers" \
  -ex "bt" \
  -ex "info threads" \
  -ex "thread apply all bt" \
  -ex "info proc all"
```

After attaching to the target process, an initial breakpoint is set to an entrypoint of the cURL easy interface to ensure that the library is loaded and a request was triggered. Then, the initial breakpoint is disabled and the actual cURL breakpoints are set. A triggered breakpoints generates a dump and writes all available context information to stdout.

5.4.1 Results

The second breakpoint was triggered and the return value register `eax` contained `11` - corresponding to `EAGAIN`.

```
Thread 142 "HTTP IO" hit Breakpoint 2, Curl_thread_create (func=func@entry
   =0x7f8dc9ed8180 <getaddrinfo_thread>, arg=arg@entry=0x7f8cf2f38a88) at
   curl_threads.c:78
```



```
$1 = 11
```

Additionally, `info threads` returned a thread count within the expected range. `info proc all` showed a significant number of memory mappings (in the thousands) that matched the pattern of thread stack mappings (stack, guard page):

```
      0x7f52a3174000     0x7f52a3974000   0x800000        0x0
      0x7f52a3974000     0x7f52a3975000     0x1000        0x0
```

The large gap between the actual number of threads and allocated stacks confirmed a `pthread_t` leak.

The root cause in terms of the actual origin of the resouce leak remains to be identified. The generated output, `info threads`, contained a hint that clearly identifies the origin:

```
  173  Thread 0x7f8c5defb700 (LWP 14553) "Worker" 0x00007f8dcd7419f5 in
     pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
  255481 Thread 0x7f4b2420e700 (LWP 31237) "Check Processor" 0
     x00007f8dcd741da2 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /
     lib64/libpthread.so.0
```

The output revealed a considerable thread ID gap resulting from non-detached threads that were not running anymore. Based on that indication, Threads of the process seem to be mostly static and long-running, otherweise the gap would present itself differently. Assuming a high-frequency leak (which is expected to reach the glibc-internal limit), the last thread named ”Check Processor” seems to be short-lived and represents a promising resource leak origin candidate.

5.5 Check Processor

The code creating the ”Check Processor” thread is completely unrelated to the HTTP infrastructure and was introduced with the update. The purpose of the thread is to run code asynchronously at regular intervals. The code is as follows:

```
385  #define CHECK_INTERVAL 5
386  while (!shutdown()) {
387    pthread_t t;
388    int result = pthread_create(&t, NULL, check, NULL);
389    if (result != 0) {
390      vlog("Check thread creation failed");
391      break;
392    }
393    sleep(CHECK_INTERVAL);
394  }
```

checks/timer.c

Neither the caller, nor the callee (`check`) release the thread handle by either calling `pthread_detach` or `pthread_join`. With an interval of 5 seconds, thread handles are exhausted after 1.895 days. This figure matches the time to failure experience by the client. By addressing this resource leak, the subject of this code review can be fully addressed.

6 Recommendations

6.1 Check Processor Resource Handling

To address the identified resource leak, a non-detached thread, adding `pthread_detach` would suffice. However, the check processor logic does not account for the fact that the last thread might still be running when a new thread is about to be started. Usually, it is desirable to delay actions triggered at

regular intervals if the previous action is still running to eliminate a potential overlap that might cause unexpected behavior and/or a potentially unbounded number of thread creations if those threads share resources that require mutual exclusion for access. Therefore, my recommendation is to wait for the previous thread to finish using `pthread_join` and only then create a new thread. Alternatively, the thread could be reused.

```
385   #define CHECK_INTERVAL 5
386   pthread_t t;
387   int started = 0;
388   while (!shutdown()) {
389     int result = 0;
390     if (started) {
391       result = pthread_join(t, NULL);
392       if (result != 0) {
393         vlog("Joining check thread failed");
394         break;
395       }
396     }
397     result = pthread_create(&t, NULL, check, NULL);
398     if (result != 0) {
399       vlog("Check thread creation failed");
400       break;
401     }
402     started = 1;
403     sleep(CHECK_INTERVAL);
404   }
```

checks/timer.c

6.2 Load Test Resource Monitoring

To ensure that this or similar issues related to resource usage can be found earlier, resources should be monitored during load tests. Relevant resources include but are not limited to:

- Process file descriptors
- Process memory mappings
- Process memory usage
- Process thread count
- System memory usage (e.g. to cover shared memory)
- Disk space

For monitoring `pthread_t` instances, the thread count must be compared with the thread stack count. Assuming the thread stack size is known, a simple mechanism count be:

```
#!/bin/bash
pid=$1
threshold=20
t=$(ls -1 /proc/${pid}/task | tail -n +2 | wc -l)
ts=$(pmap ${pid} | grep '8192K rw---   \[ anon \]' |wc -l)
diff=$((${t} >= ${ts} ? ${t} - ${ts} : ${ts} - ${t}))
if [ ${diff} -ge ${threshold} ]; then
  echo "Potential pthread_t leak" 1>&2
fi
```


6.3 cURL Error Handling

The error handling of the cURL integration should be improved to maximize the context in case of cURL errors. Currently, only the cURL error status is taken into account:

```
HTTP request failed (Error Code: 6)
```

The underlying code is as follows:

```
1423   res = curl_easy_perform(curl);
1424   if (res != CURLE_OK) {
1425     fprintf(stderr, "HTTP request failed (Error Code: %d)\n", res);
```

io/httprequest.c

Error handling can be improved by enabling `CURLOPT_ERRORBUFFER`. Also, the error status should be converted to a text representation using `curl_easy_strerror()` to ease error interpretation without having to consult the cURL documentation.

```
1423   char errbuf[CURL_ERROR_SIZE];
1424   curl_easy_setopt(curl, CURLOPT_ERRORBUFFER, errbuf);
1425   errbuf[0] = '\0';
1426   res = curl_easy_perform(curl);
1427   if (res != CURLE_OK) {
1428     size_t len = strlen(errbuf);
1429     if (len) {
1430       fprintf(stderr, "HTTP request failed (%s: %s)\n",
1431         curl_easy_strerror(res),
1432         errbuf);
1433     }
1434     else {
1435       fprintf(stderr, "HTTP request failed (%s)\n",
1436         curl_easy_strerror(res));
1437     }
1438   }
```

io/httprequest.c

