Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report gRPC 09.-10.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. D. Weißer, J. Larsson,
BSc. J. Hector, MSc. N. Krein, Dipl.-Ing. A. Inführ

Index

Introduction

> *“gRPC is a modern open source high performance RPC framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.”*
>
> From https://grpc.io/about/

This report documents the findings of a security assessment targeting the gRPC software. Carried out by Cure53 in autumn 2019, this project specifically entailed a penetration test and a source code audit. Featuring primarily the C++ implementation from the v1.2.4.x branch of gRPC maintained by Google, the tests were generously sponsored by CNCF.

As for the resources, seven senior testers from the Cure53 were tasked with completing this project. After being commissioned by CNCF to execute the assessment, the Cure53 team worked with a budget of eighteen person-days, all spent on the scope and

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

documentation in late September and early-to-mid October of 2019. The focus was placed on the aspects linked to the HTTP2 stack, gRPC compression features and buffering mechanisms.

The Cure53 team followed the white-box methodology, which is a typical approach for CNCF projects and signifies access to the codebase which is actually available as open source. In addition, a GCP environment which was initially set up by Cure53 and later supplanted with two additional environments furnished by Google, served as a close approximation of what could be found in a production environment. Cure53 further got briefed by Google about the key focus areas for this audit noted above.

In order to best address the three main arenas, three Work Packages (WPs) have been delineated. While WP1 specifically focused on the HTTP2 Protocol Stack, the tests performed for WP2 entailed investigating encryption and authentication mechanisms and deployments. Finally, in WP3 Cure53 honed in on the compression and buffering features. The two-pronged strategy, which again is typical for CNCF-funded projects, was also deployed here and meant that work has been divided into dedicated penetration testing and the phase of code auditing. Specific tasks are further elaborated on in the *Coverage* section of this document.

The project started as scheduled and progressed quickly. During the assessment, Cure53 communicated with the Google team in a jointly used Slack channel, enabling real-time exchanges. In the interest of time efficiency, findings were live-reported to the gRPC team, so that the fixes could be discussed by the involved parties. Among three issues spotted in the codebase, one has been categorized as a security vulnerability with a risk level set to *“Medium”.* The remaining flaws were considered to only signify general weaknesses without much exploitation potential. This outcome is quite impressive, especially given the thorough and focused penetration testing, fuzzing and code auditing approaches. Consequently, this Cure53 assessment points towards a rather positive result and a decent level of code maturity at gRPC.

The report will now shed more light on the scope used for this assessment and then elaborate on the testing methodology and test coverage. It then moves on to discussing all spotted findings in chronological order, as well as with sufficient technical depth and detail. Finally, the report will close with a conclusion in which the Cure53 team elaborates on the impressions gathered during the assessment. Both broad and more granular recommendations regarding the security properties of the tested gRPC software ensue.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Scope

- **gRPC - the C++ based RPC library and framework**
  - WP1: HTTP/2 Protocol Stack
  - WP2: Encryption and Authentication
  - WP3: Compression and Buffering
  - **Sources were available via GitHub**
    - https://github.com/grpc/grpc/tree/v1.24.x
  - **Environment and Instructions**
    - A test environment on GCP was available for the Cure53 team and co-maintained by Google to foster testing
    - Cure53 further received detailed instructions about key focus areas and software setup from Google

Test Methodology

The following paragraphs describe the testing methodology used during the audit of the gRPC codebase. The test was driven by two approaches over three Work Packages. Each strategy - i.e. code auditing and pentesting - fulfilling different goals. In particular, the manual source code reviews centered on spotting insecure code patterns. Usually issues around memory corruption issues, race conditions, information leakage or similar flaws can be found in this context. During the second phase, it was evaluated whether the stated security goals and premise can, in fact, withstand real-life attack scenarios.

Part 1: Manual Code Auditing

This section lists the steps that were undertaken during the first phase of the audit against the gRPC software compound. Since no major issues were spotted, the list portrays the thoroughness of the audit and attests to the impressively high quality of the project.

- After familiarizing themselves with the documentation and codebase, Cure53 team continued checking of string functions and memory allocation wrappers spotted in the codebase in scope. Auditing of the usual C-language-relevant dangerous sinks, like *memcpy*, *strcpy*, *sprintf* etc. took place in the early phase of the code analysis.
- Further attention was given to how Base64-encoded input is being treated and the Cure53 team attempted to cause the decoder to stumble by providing malformed Base64 sequences using training bytes, malformed padding sequences and illegal characters. It was demonstrated that no issues could be spotted in this area and that the handling of Base64 by gRPC was generally well-implemented.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- Attention was also dedicated to the implementation of the *GZIP* and *DEFLATE* compression and decompression. It was quickly found that gRPC makes use of standard libraries such as *zlib* and that the integration of those was done properly. The tests attempting to cause a Denial-of-Service or alike were unsuccessful and the implementation made a good impression.
- Further attention was given to the *HPACK* parser implementation - especially avenues where indices are being utilized or where different type and length values could be used - as specified in RFC 7540[1]. One area in the audited code appeared to be affected by an Integer Underflow but a closer analysis showed that indeed all necessary checks were put in place by the maintainers to keep this issue from having any effect.
- A code audit against the implementation of the header element construction was performed to check if the results of the parsing processes are handled well. No significant findings could be identified.
- In addition, the code handling message frames, i.e. in *core/ext/transport/chttp2/transport/parsing.cc,* were inspected and checked for errors potentially allowing for DoS, memory corruption or alike. No implementation flaws could be observed and the code makes a clean impression.

Part 2: Code-Assisted Penetration Testing

The following list documents the distinguishable steps taken during the second part of the test. A code-assisted penetration test was executed against the pre-configured server instance on GCP running gRPC and provided by the development team. Since only a few miscellaneous issues were found during the first part of the audit, this additional approach was used to ensure maximum coverage of the originally defined attack surface.

- All previously discovered gRPC vulnerabilities were closely inspected to gain a general overview of repeating patterns and possible avenues to be explored.
- The test-cases for all encryption and authentication aspects were evaluated, in particular in the realm of handling broken certificates. Items interfacing with *BoringSSL* were found to be done properly and no errors could be identified.
- A locally modified version of *greeter_server_tls* was used to inspect the TLS protocol exchange alongside verifying the validity of simultaneously recorded *tcpdump* output.
- The execution flow and source code between *greeter_client_tls* and *greeter_server_tls* was cross-referenced to obtain an overview of the exposed components.

[1] https://tools.ietf.org/html/rfc7540#page-12

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- The application of oversized and malformed key and certificate files was attempted to probe the handling of such errors. The code was found to be handling all cases terminally but gracefully.
- A working fuzzer setup was created and tested. Once successful, the fuzzer was run over the course of the penetration test and audit. It managed to in fact produce a couple of useful results, for instance GRP-01-001.
- Extensive binary analysis of the fuzzer-generated backtraces in combination with the source code was undertaken to unveil additional problems but success was limited due to time constraints.
- Further penetration testing and fuzzing attention was given to the compression- and decompression-related implementations for channels and messages in the tested codebase. It was checked if any undocumented compression algorithms were supported but this proved not to be the case.
- Tests making use of compression bombs were performed by creating a modified gRPC client capable of sending those. No findings were spotted in this realm; after a certain threshold (i.e. 7MB of payload) value, the server started to ignore compressed payload and fell back to the uncompressed text.
- Additionally, sending of varying slice counts to the server was attempted to see if any unexpected behavior could be provoked. Cure53 gave up on this since no problematic reactions could be observed.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *GRP-01-001*) for the purpose of facilitating any future follow-up correspondence.

GRP-01-001 Server: DoS through uninitialized pointer dereference *(Medium)*

While fuzzing the communication between the gRPC *greeter* example client, a segmentation fault was observed due to dereferencing a *null* pointer. This led to a Denial-of-Service (DoS). Upon further investigation of the crash, it was discovered that an empty *path* in the request headers causes this issue.

The following code excerpt highlights the relevant parts.

Affected File:
src/core/lib/surface/server.cc

Affected Code:

```
static void start_new_rpc(grpc_call_element* elem) {
  channel_data* chand = static_cast<channel_data*>(elem->channel_data);
  call_data* calld = static_cast<call_data*>(elem->call_data);
[...]
  if (chand->registered_methods && calld->path_set && calld->host_set) {
[...]
    /* check for a wildcard method definition (no host set) */
    hash = GRPC_MDSTR_KV_HASH(0, grpc_slice_hash_internal(calld->path));
    for (i = 0; i <= chand->registered_method_max_probes; i++) {
      rm = &chand->registered_methods[(hash + i) %
                                      chand->registered_method_slots];
      if (!rm) break;
      if (rm->has_host) continue;
      if (!grpc_slice_eq(rm->method, calld->path)) continue;
      if ((rm->flags & GRPC_INITIAL_METADATA_IDEMPOTENT_REQUEST) &&
          0 == (calld->recv_initial_metadata_flags &
                GRPC_INITIAL_METADATA_IDEMPOTENT_REQUEST)) {
        continue;
      }
      finish_start_new_rpc(server, elem, &rm->server_registered_method->matcher,
                           rm->server_registered_method->payload_handling);
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

When the empty *path* is used to compute the hash value for indexing the *registered_methods* array, it is possible that the resulting index points to an entry that has not been initialized. Since the allocated space for the given array is filled with *null* values, the *server_registered_method* pointer will be *null.*

It should be noted that the crash does not trigger reliably and depends on the global *seed* value used for computing the hashes. This *seed* value changes with every restart of the application. Therefore, when trying to reproduce the issue, it may be necessary to restart the server application multiple times.

Steps to reproduce:

- Download the gRPC *helloworld* client
- Replace the *path* in *Greeter_method_names* with an empty string
- Run the client against a gRPC server

It is recommended to ensure that no empty value for *path* can be given.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

GRP-01-002 General: Refs to freed memory not automatically nulled *(Low)*

While auditing the gRPC's underlying memory allocation functionalities and wrappers around the *libc malloc* function family, it was noticed that pointers to freed memory are not automatically nulled. Instead, coding patterns like the following were observed.

Examples:

- ```
  gpr_free(service_config);
  service_config = nullptr;
  ```
- ```
  gpr_free(tbl->ents);
  tbl->ents = nullptr;
  ```
- ```
  gpr_free(s->header_array.headers);
  s->header_array.headers = nullptr;
  ```
- *etc...*

This pattern of freeing memory and manually setting the corresponding pointer to *nullptr* stems from the fact that the wrapper around libc's *free* is implemented without automatically setting this pointer to *null* as a mandatory operation.
  ```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected File:
grpc/src/core/lib/gpr/alloc.cc

Affected Code:
```
void gpr_free(void* p) {
  GPR_TIMER_SCOPE("gpr_free", 0);
  g_alloc_functions.free_fn(p);
}
```

Considering the fact that it is easy to forget to reset the freed pointer to *null* when necessary, it might make sense to implement this in *gpr_free* itself. It should be considered to rewrite *gpr_free* so that it accepts the address of the pointer as a parameter and resets the pointer to *null* after the final call to *free*. Automatically setting unused pointers to *nullptr* is not only a defensive style that protects against dangling pointer bugs or use after frees, it also makes sure that developers cannot omit important *nullptr* assignments.

GRP-01-003 General: Calls to *malloc* suffer from potential integer overflows *(Low)*

Another more general weakness was found in the usage of allocation functions in gRPC's *libc* wrappers around *malloc*. As one can see in the following code, *gpr_malloc* is not implemented in a way that takes care of allocating memory for an array of elements with the same size for each element.

Affected File:
grpc/src/core/lib/gpr/alloc.cc

Affected Code:
```
void* gpr_malloc(size_t size) {
  GPR_TIMER_SCOPE("gpr_malloc", 0);
  void* p;
  if (size == 0) return nullptr;
  p = g_alloc_functions.malloc_fn(size);
  if (!p) {
    abort();
  }
  return p;
}
```

Nevertheless, code like in the following snippets is used to make room for multiple elements in one single call to *gpr_malloc*.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Examples:

- *uri_parser.cc*

```
uri->query_parts_values =
      static_cast<char**>(gpr_malloc(uri->num_query_parts *
sizeof(char**)));
```

- *http_connect_handshaker.cc*
-

```
headers = static_cast<grpc_http_header*>(
    gpr_malloc(sizeof(grpc_http_header) * num_header_strings));
```

- *etc …*

Although no concrete scenario of this vulnerability being in effect was spotted, this coding pattern is oftentimes prone to integer overflows when multiplying array sizes with the number of the array's elements.

It is recommended to create a *calloc(size_t nmemb, size_t size)* style wrapper that takes care of allocating *nmemb* elements of *size* for each item. Internally, it should be checked for integer overflow when multiplying both values inside the function. This not only represents a more defensive coding pattern that tries to prevent certain bug classes, but also takes the burden away from developers who otherwise would need to check for integer overflows.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

This Cure53 report clearly demonstrates a strong security posture of the investigated gRPC software. After spending eighten days on the scope in September and October of 2019, seven members of the Cure53 team can conclude that the project complies with its security promises.

During this CNCF-funded project, an explicit focus was placed on analyzing the HTTP2 protocol flow, where especially the parsing and unpacking of different headers was evaluated. Considering the small time-frame, only rudimentary checks could be performed. In parallel to auditing of the source code, the provided *helloworld* server/client was fuzzed with a modified version of AFL. This yielded a DoS vulnerability in the server. However, no other bugs were found using this methodology, pointing to the gRPC's strengths.

The HTTP data compressions pertinent to *GZIP* and *DEFLATE* use standard libraries and were therefore excluded from the code audit. The compression features were nevertheless manually pentested and stood strong to Cure53's scrutiny. As part of the analysis connected to the crash described in GRP-01-001, a more in-depth look into the *HPACK* parsing code was taken. Overall, the code made a good impression in that it follows good coding practices and displays proper checks in critical areas, for example the *uint32* number parsing paid special attention to detection of integer overflows. A closer look at the frame similarly revealed it to be well-written, indicating a coding guideline being strictly followed by the developers.

The TLS examples were used as a basis for analyzing the TLS implementation in gRPC. Here the malformed strings and certificate constructs were evaluated in the pursuit to find logical flaws in the implementation. Crashes were observed but none of them were deemed to be of security value, since they required patching the example binaries with out-of-bounds strings and values. No bugs were discovered during this phase and, similarly, no leaks were observed during the analysis of TLS traffic, handshakes, etc. The protocol sequences displayed no anomalies.

The existence and mandatory inclusion of a regression test, several sanitizer components and the integration of a fuzzing infrastructure to the protocol aspects further translates to building on the notion of a responsibly maintained software system. Setting up a test harness proved to be straightforward and reliable. The development team gave detailed instructions on how to build all of the necessary components, again confirming that having their project tested poses no issue. It must be emphasized that, considering the large codebase and the existing constraints of the audit, a significant but far from complete code coverage has been achieved.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

In sum, judging from the somewhat limited coverage achieved during this code audit and code-assisted penetration test, the Cure53 team can only attest to a very high quality of the examined gRPC system. This autumn 2019 project ascertained that the gRPC team is fully capable of delivering excellent results in terms of security and maintainability.

Cure53 would like to thank Srini Polavarapu, Hope Casey-Allen, Nicolas Noble and April Kyle Nassi of Google, as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.