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

Pentest-Report Envoy Proxy 02.2018

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, J. Larsson, Dipl.-Ing. A. Inführ, BSc. J. Hector, G. Kopf, M. Münch

Index

Introduction

> *“Originally built at Lyft, Envoy is a high performance C++ distributed proxy designed for single services and applications, as well as a communication bus and “universal data plane” designed for large microservice “service mesh” architectures.*
>
> *Built on the learnings of solutions such as NGINX, HAProxy, hardware load balancers, and cloud load balancers, Envoy runs alongside every application and abstracts the network by providing common features in a platform-agnostic manner. When all service traffic in an infrastructure flows via an Envoy mesh, it becomes easy to visualize problem areas via consistent observability, tune overall performance, and add substrate features in a single place.”*

From https://www.envoyproxy.io

This report documents the results of a penetration test and source code audit of the Envoy proxy software. The project was carried out by Cure53 in February 2018 and yielded eight security-relevant findings.

In terms of resources, the testing team comprised six members of the Cure53 and two invited experts from Secfault Security GmbH. This overall eight-member team was granted a time budget of twenty days to complete the assessment, with a caveat that an adequate budget fraction should be preserved for the task of fix verifications and communications to happen post-assessment. Moving on to the approach, it has to be underlined that all relevant sources of the Envoy compound are openly available on Github. This naturally means that the security evaluation was rooted in a white-box premise since the envisioned attackers would have unlimited access to Envoy public data.

It should be stated that the core scope items concerned exposition to rogue user-input, with particular foci pointing to the HTTP parser, the header sanitizer and other components of similar notoriety. What is more, the Envoy maintainers specifically requested that the testers examine the TLS configuration, XFF and the generally *slowloris*-style DoS attacks. While all this was inspected by Cure53, the focus on DoS attacks also translated to looking at decompression of malicious HTTP content and necessitated coverage with tests targeting the HTTP2 features. The Envoy Admin web interface was placed under scrutiny of the testers as well, yet it was granted a much lower priority because it is not expected to become a viable target. Upon a later request from the maintainers, this component of the scope was actually ignored at sequent stages of the project.

In terms of the execution of this assessment, the project was split into two phases, with the first centered on the source code audit and a search for code bugs, and the second encompassing a full-blown penetration test. The latter tried to attack an actually running instance of Envoy and, quite clearly, investigated more complex attack scenarios, DoS attacks, spoofing and bypasses. Under this premise, there was a clearly defined attacker-model, specifically signifying a rogue and external attacker seeking to abuse bugs and vulnerabilities in the Envoy entities. The presumed goals of the adversaries would be to escalate their findings and make the Envoy project either slow or completely shut down.

Among the already noted eight findings, there was an even mid-way split between four actual security vulnerabilities and four general weaknesses. It is vital to emphasize that no issues were marked as “Critical” in terms of security impact, severity or scope. This absence of high-risk problems is a very good indicator for the broader state of security matters at the Envoy compound.

In the following paragraphs, this report will briefly focus on the technical outline of the scope and illuminates the overall coverage. It then moves on to the findings which are discussed on a case-by-case basis and presented together with mitigation advice. The closing section allows for some general notes on the robustness of the Envoy project in light of this Cure53 and Secfault Security GmbH's joint assessment.

Scope

- **Envoy Proxy**
    - https://www.envoyproxy.io/
    - Exact snapshot that was used for testing
        - https://github.com/envoyproxy/envoy/tree/master
        - Commit c31077b28e4f8a7db17895d5d2570e806e9e2a3e

Test Coverage

This section comments on the test coverage reached by the testing team in the given timeframe.

To start with the key item of the scope, the HTTP header manipulation was extensively examined. The handling was investigated with a special focus on sanitization and performed also with edge-case values. It was attempted to bypass header stripping, inject duplicate headers and break the HTTP transcoding by message splitting. The HTTP2 header compression aspect was covered in relation to the Denial-of-Service attack vectors.

The Admin Web Interface was checked for typical security problems. The findings in this realm were concerning as the item was admittedly inadequate in terms of no Authentication, lacking Header Security and the absence of the critical CSRF token.

In relation with the JSON parser, state machine was inspected for malformed input handling and related memory management defects. An existing problem with incomplete JSON input was found but ultimately deemed unexploitable. A particular focus was placed here on the intrinsic types like numbers and strings.

Next on the list was the gRPC implementation which was checked for interface interference issues. Also in this context there was a preselected item of top-priority, notably concerning the potential threading model incompatibilities. Buffering and decoding problems were considered in the HTTP and JSON filters, wherein an existing general data starvation case was unveiled but discarded as insignificant.

The Lua implementation was investigated for common C interface misuse. The coroutine state machine logic was verified. General integer overflows and null pointer dereferences could not be identified in this realm.

For the event handling code in the Ratelimit component, extra scrutiny did not yield any issues and found it adequate. Another aspect that generated no substantial findings was the Redis protocol parser which was verified with special attention to logic issues like treatment of invalid messages and illegal field lengths. Denial-of-service scenarios and out-of-bound situations could not be found in this realm. This positive evaluation extended to MongoDB parser code which was subjected to similar procedures as the Redis parser implementation and held strong.

Further, the Network component was considered for potential logic and DoS bugs but found to be rather clean and strong against the non-standard problem areas. Here the explication could be derived from the lack of the application logic's implementation.

Finally the SSL implementation was checked for typical TLS-related issues, configuration malpractice and common certificate validation problems and the core Router request handling was investigated for potential DoS scenarios. This last item was examined in regard to the brokerage code of the component.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a 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. *EP-01-001*) for the purpose of facilitating any future follow-up correspondence.

EP-01-001 HTTP: Lacking Admin-Interface Security allows CSRF and DOS (*High*)

It was noticed quite early in the assessment that the Envoy's Admin Interface, normally reachable under *http://localhost:9901*, is highly insecure. This is because it completely fails to follow modern security standards. There is actual authentication (although per default the instance is only reachable locally), no HTTP security headers are in place and CSRF tokens are not deployed. The last point is especially dangerous because all administrative operations can be carried out by simple GET requests. Therefore luring an internal administrator onto websites that embed the code like the following will signify changes to the entire global configuration.

Example HTML *(disable logging)***:**

```
<img src="http://localhost:9901/logging?admin=off">
```

Example HTML *(shutdown the server)***:**

```
<img src="http://localhost:9901/quitquitquit">
```

Besides having a mass-impact in terms of modifications, the attacker may even turn off the server. Both these high-impact action may occur without the administrator noticing for certain. Although the Envoy team has expressed their awareness of the bad security premises of its Web Interface, it is still necessary to highlight the importance of relevant fixes. It is recommended to offer additional authentication, e.g. similar to the Apache's *digest-auth* mechanism. Furthermore, CSRF tokens should be urgently implemented, for example by utilizing *X-CSRF-TOKEN* headers on each authenticated request.

EP-01-002 HTTP: Potential BoF with HeaderStrings and Inline Buffers (*Medium*)

While analyzing how Envoy implements HTTP headers inside their *HeaderMap,* it was noticed that there were three possible implementations, depending on the use-case at hand. One of them concerned inline buffers that were mostly used for local variable type declarations where simple C-style *char* arrays were used for storage. This can be seen in the following definition of a buffer.

Affected File:
envoy/include/envoy/include/envoy/http/header_map.h

Affected Code:

```
union {
  char inline_buffer_[128];
  uint32_t dynamic_capacity_;
};
```

An actual problem was later discovered when checking how a header of this composition is being created in the respective constructor inside the *HeaderString* class. The following code depicts said faulty implementation.

Affected File:
envoy/include/envoy/source/common/http/header_map_impl.cc

Affected Code:

```
HeaderString::HeaderString(HeaderString&& move_value) {
  type_ = move_value.type_;
  string_length_ = move_value.string_length_;
[...]
  case Type::Inline: {
    buffer_.dynamic_ = inline_buffer_;
    memcpy(inline_buffer_, move_value.inline_buffer_, string_length_ + 1);
    move_value.string_length_ = 0;
    move_value.inline_buffer_[0] = 0;
    break;
  }
  }
}
```

One can observe the later used *string_length* as being formed from the passed parameter and later used as a direct value for *memcpy*. This practically creates a call such as *memcpy(dest, src, strlen(src))*. In this call the length of the source *operand* is left unchecked and can be bigger than the space reserved at destination. Since this function can be an actual pitfall for potential buffer overflows, it is recommended to explicitly limit the length of the source *operand* to the reserved room at the destination, namely 128 bytes.

EP-01-003 HTTP: Potential UaF with HeaderStrings and Ref. Buffers *(Medium)*

While analyzing the *HeaderMap* implementation of Envoy, it was noticed that the code makes use of the *c_str()* method for obtaining references to the underlying character buffers of the STL strings. This pattern is not generally problematic but issues can arise when the string object is invalidated (e.g. deleted). In case of deletion or similar state, it has to be ensured that the reference to the underlying memory buffer is also invalidated. Otherwise, the so-called *use after free* or similar vulnerability patterns can occur. One particular example is shown below.

Affected File:
envoy/include/envoy/source/common/http/header_map_impl.cc

Affected Code:

```
HeaderString::HeaderString(const std::string& ref_value) :
type_(Type::Reference) {
  buffer_.ref_ = ref_value.c_str();
  string_length_ = ref_value.size();
}
```

After calling the above *copy* constructor, *ref_value* could go out of scope or be deleted. In such cases a reference to an invalid buffer would be stored in the *HeaderString* object. It should be noted that this is only one example whilst saving the *return* value of the *c_str()* method appears to be common throughout the analyzed code base. Using alternative approaches such as *std::move* should be evaluated in order to address this finding in a more general manner.

EP-01-004 HTTP: Potential Integer Overflow during Header Encoding *(Medium)*

Another potential buffer overflow resulting from an integer overflow has been spotted inside the HTTP/1 codec. This is due to the fact that two unsigned 32-bit integers were used inside an addition before being assigned to a 64-bit unsigned integer. The following code shows the prototype of the *reserveBuffer* function which accepts a *uint64_t* as a size parameter for the later allocation.

Affected File:
envoy/source/common/http/http1/codec_impl.cc

Affected Source:

```
void ConnectionImpl::reserveBuffer(uint64_t size) {
[...]
```

However, the *encodeHeader* function uses two *uint32_t* types and adds them before passing them to *reserveBuffer*.

Affected Source:

```
void StreamEncoderImpl::encodeHeader(const char* key, uint32_t key_size, const
char* value, uint32_t value_size) {

  connection_.reserveBuffer(key_size + value_size + 4);
  ASSERT(key_size > 0);

  connection_.copyToBuffer(key, key_size);

[...]
```

Since the addition is based on 32-bit integers, an addition of the values *0x7fffffff*, *0x7fffffff*, and *4* (for instance) will lead to the value *2* being passed to *reserveBuffer*. Thus, it will cause an allocation of a significantly smaller size. This was also checked inside the assembly where the addition is carried out and stored into the 32-bit large *esi* register before being passed to *reserveBuffer*:

Assembly snippet:

```
   0x0000000000684049 <+25>:    lea    0x4(%r13,%r12,1),%esi
   0x000000000068404e <+30>:    sub    $0x18,%rsp
   0x0000000000684052 <+34>:    mov    0x38(%rdi),%rdi
   0x0000000000684056 <+38>:    mov    %rcx,-0x38(%rbp)
   0x000000000068405a <+42>:    callq  0x683f80
<Envoy::Http::Http1::ConnectionImpl::reserveBuffer(unsigned long)>
```

Later on, *copyToBuffer* is called with *key_size* of *0x7fffffff*, despite there being room for only *2* bytes. This would eventually be caught by the *ASSERT* statement inside *copyToBuffer* where the remaining size is checked against the passed length, as illustrated below.

Affected Source:

```
void ConnectionImpl::copyToBuffer(const char* data, uint64_t length) {
  ASSERT(bufferRemainingSize() >= length);
  memcpy(reserved_current_, data, length);
  reserved_current_ += length;
}
```

However, since this is not explicitly declared as an *RELEASE_ASSERT,* there is no guarantee that this check will eventually land in the compiled version, thus opening the door for another potential overflow. It is recommended to catch the mentioned integer overflow, for example by relying on the built-in functionalities of GCC’[1], specifically *__builtin_uadd_overflow*.

---

[1] https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

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.

EP-01-005 Common: strlcpy does not check for zero-sized Parameters *(Low)*

Another potentially flawed function was identified inside the Envoy's utility library where all sorts of useful wrappers are defined. One of the items is a *strlcpy* implementation which is essentially just a wrapper around *strncpy,* used to make sure that the destination is always *null*-terminated.

Affected File:
envoy/source/common/common/utility.cc

Affected Source:
```
size_t StringUtil::strlcpy(char* dst, const char* src, size_t size) {
  strncpy(dst, src, size - 1);
  dst[size - 1] = '\0';
  return strlen(src);
}
```

The dangers associated with wrappers like these one stems from the fact that the programmer always needs to make sure that the passed size is actually greater than zero. Otherwise the *size - 1* will eventually wrap around to *0xffffffff* and cause a copy operation that will continue to result in an overflow as long as no terminating *null* byte is found in the source *operand*. A useful hardening addition here would be to make the wrapper itself check whether *size* is greater than 0. The proposed strategy will always prevent incorrect usage of the *strlcpy* function.

EP-01-006 Redis: User-Controlled Allocation leads to DoS *(Medium)*

During a cursory inspection of the *Redis* codec it was found that the implementation relied on a user-supplied *length* arguments to perform memory allocations. Please consider the code excerpt below to understand the corresponding flaw.

Affected File:
envoy/source/common/redis/codec_impl.cc

Affected Source:
```
} else {
  std::vector<RespValue> values(pending_integer_.integer_);
  current_value.value_->asArray().swap(values);
  pending_value_stack_.push_front({&current_value.value_->asArray()[0], 0});
  state_ = State::ValueStart;
}
```

It can be observed that the code uses the previously parsed value of *pending_integer_.integer_* for allocating an array. Supplying an overly large array *length* might hence provoke an out-of-memory condition, resulting in a DoS issue. It is generally recommended to perform strict checks on all user-provided *length* arguments before using them to perform allocations.

EP-01-007 MongoDB: Stack Exhaustion via unbounded Recursion *(Medium)*

A cursory inspection of the MongoDB implementation revealed that the BSON parsing code allows for unbounded recursion depths when nested documents are being handled. The following code excerpt can shed light on why this is a bad practice.

Affected File:
envoy/source/common/mongo/bson_impl.cc

Affected Source:

```
void DocumentImpl::fromBuffer(Buffer::Instance& data) {
[...]
case Field::Type::DOCUMENT: {
  ENVOY_LOG(trace, "BSON document");
  addDocument(key, DocumentImpl::create(data));
  break;
}
```

Affected File:
envoy/source/common/mongo/bson_impl.h

Affected Source:

```
static DocumentSharedPtr create(Buffer::Instance& data) {
  std::shared_ptr<DocumentImpl> new_doc{new DocumentImpl()};
  new_doc->fromBuffer(data);
  return new_doc;
}
```

It can be observed that the code recursively calls the *fromBuffer()* method for building new *DocumentImpl* instances. A deeply-nested BSON document might hence lead to a stack memory exhaustion, resulting in a DoS condition. It is recommended to generally restrict the number of possible nesting levels to an appropriate value.

EP-01-008 MongoDB: Lax Parsing when processing malformed Messages *(Low)*

Another issue found during a cursory inspection of the MongoDB implementation was linked to the parsing of code, which was relatively permissive when it came to malformed messages. A code excerpt supplied next can illustrate the issue at hand.

Affected File:

envoy/source/common/mongo/bson_impl.cc

Affected Source:

```
void ReplyMessageImpl::fromBuffer(uint32_t, Buffer::Instance& data) {
  ENVOY_LOG(trace, "decoding reply message");
  flags_ = Bson::BufferHelper::removeInt32(data);
  cursor_id_ = Bson::BufferHelper::removeInt64(data);
  starting_from_ = Bson::BufferHelper::removeInt32(data);
  number_returned_ = Bson::BufferHelper::removeInt32(data);
  for (int32_t i = 0; i < number_returned_; i++) {
    documents_.emplace_back(Bson::DocumentImpl::create(data));
  }

  ENVOY_LOG(trace, "{}", toString(true));
}
[...]

bool DecoderImpl::decode(Buffer::Instance& data) {
[...]
  uint32_t message_length = Bson::BufferHelper::peakInt32(data);
  ENVOY_LOG(trace, "message is {} bytes", message_length);
  if (data.length() < message_length) {
    return false;
  }
[...]
  switch (op_code) {
  case Message::OpCode::OP_REPLY: {
    std::unique_ptr<ReplyMessageImpl> message(new ReplyMessageImpl(request_id, 
response_to));
    message->fromBuffer(message_length, data);
    callbacks_.decodeReply(std::move(message));
    break;
  }
```

It can be observed that the code makes use of the *decode()* method in order to parse a MongoDB message from a buffer. In a first step, the message *length* field is extracted. After performing several other operations, the actual message is decoded using *message->fromBuffer(message_length, data)*.

However, as the definition of *ReplyMessageImpl::fromBuffer()* shows, the previously extracted *length* field is not actually considered. The *fromBuffer()* method could hence consume more bytes than the message's *length* field actually indicates. Clearly, in order to provoke such a situation, a malformed message would have to be sent in the first place. To sum up this flaw and mitigation, it might be advisable to add stricter checks regarding message sizes to the parsing code. A revised strategy would signify a defense-in-depth approach.

Conclusion

In spite of eight security-relevant findings, the team responsible for this assessment of the Envoy compound can attest to the overall good state of security matters at the tested project. Tasking an eight-member testing team from Cure53 and Secfault Security GmbH entities has clearly demonstrated the robustness of the Envoy scope, while also enabled for some minor flaws to be pointed out and - ideally - remediated. After spending twenty days on the Envoy test target in February 2018, the penetration testers concluded that the software was appropriately built and deployed. Similarly positive impression concerned the Envoy code, which the auditors found to be well-written.

To comment on some specifics of this white-box assessment, it should be noted that the maintainers requested a specific threat model to be covered. This encompassed a data-plane compromise and Denial-of-Service, which were addressed in depth by Cure53. Because of this focus, the less concerning areas were subsequently treated with lower priority. From the test it quickly became apparent that the software boasted a modern architecture and security was integrally interwoven into the deployment actions and design decisions taken by the project's development team.

Further noticeable was the fact that the team run various code quality assurance tools and scanners to make sure that the software was not affected by the bugs and memory corruptions commonly affecting similar projects, especially those surfacing during a shared timeframe. Another particularly positive aspect to mention was the careful selection and integration of high-quality and well-vetted external components. Despite through checks, not a single item bearing a noteworthy “Critical” risk was spotted.

On a less positive note, the above highly commendable attention to security did not extend to the web frontend for Admin users. This component was then taken out of scope. The decision was explained by the fact that having this item considered as part of the attack surface does not hold in terms of the frontend carrying a minimal risk.

For the sake of completeness it should be noted that a full coverage of the entire spectrum of the software in scope was an impossible task. By acknowledging the sheer volume, a considerable size and the myriad of implementation details on Envoy, it should be clarified that the goal was to account for security on the focal areas and most sensitive, potentially risk-laden or threatened items. The actually reached level of coverage was considered good, especially as the robustness of the audited code and implementation of good practices appeared to have carried through different layers and arenas. This can be read as an indicator of consistently good patterns and high-quality security being present across the entire scope rather than a finding only applicable to the explicitly prioritized scope.

In sum, the testers found the software to be quite mature and well-structured despite its inherent complexity. It can be expected that the discovery of the flaws will only increase the overall good direction that the Envoy team is heading in from a security standpoint.

Cure53 would like to thank Harvey Tuch, Matt Klein and Joshua Marantz from the Envoy proxy team as well as Chris Aniszczyk of The Linux Foundation for their excellent project coordination, support and assistance, both before and during this assignment. Special thanks need to be extended as well to The Linux Foundation for funding this project.