OSS Fuzzing Improvements for Envoy Proxy

Final Report and Management Summary

2023-08-10

X41 D-SEC GmbH
Krefelder Str. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de

Funded by Google LLC
Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Editor*
1	2023-07-13	Final Report and Management Summary	R. Femmer, A. Vehreschild
2	2023-08-16	Public Report	L. Gommans

Contents

1 Executive Summary

X41 was engaged to asses and improve the fuzzing efforts of Envoy Proxy. In the first phase of this project, numerous issues discovered by the existing fuzzers were addressed and fixed. The insight revealed that the fuzzers found lots of non-issues (noise) and suffered from low performance.

The aim of the second phase is to identify and mitigate the reasons for the low performance for a selected set of fuzzers deemed high priority. Guarding the Envoy code base from erroneous code additions is a key objective. This report describes the efforts of the second phase.

We attribute the low performance of current fuzzers to the necessity of validating input due to how the fuzzers were designed, leading to a low utilization of processing time. The current design also leads to inefficiencies regarding achievable coverage, which result in a high number of issues with no or low impact. In the following chapters, we detail the design deficiencies and describe how they were remedied.

In the course of the second phase, *four* new fuzzers and an improved method for input generation were added. Existing fuzzers were altered to benefit from the approach.

Chapter 2 introduces concepts of software testing and fuzzing. The definitions given are not authoritative and only aim to give the reader the ability to follow our chosen terminology.

Chapter 3 describes the state of fuzz testing in Envoy and describes the issues identified by X41 as well as the methods employed to remedy these issues.

The report concludes with recommendations for future work in chapter 4.

The Appendix gives a short summary of the results of the first phase and lists some issues discovered in the second phase.

2 Terminology

2.1 Unit And Integration Testing

The goal of unit and integration testing is to exercise a selected path of code with a set of chosen input data and assert that the data is mutated or interpreted correctly. Unit tests usually refer to tests of small units of code, whereas integration tests integrate multiple units to test their collective behavior. These tests can ensure that the intended functionality has been achieved and executing them after committing changes to the code base can ensure that the functionality is not lost due to the changes.

The test behavior, input, and output of these tests are tailored to exercise one specific path through the code.

2.2 Fuzz Testing

In fuzz testing, an attack surface is defined and exercised by feeding semi-random data to it and observing the program for consistency. The program is modeled as a state graph with a defined entry state. The state directly translates to the state of CPU and memory. It is assumed that the entry state can be partially controlled by a potential attacker, who is able to control the contents of a buffer in memory. By letting the CPU execute the code the state graph is traversed. By observing edges that are traversed and reporting clearly erroneous conditions, the security auditor can identify problematic inputs which allow for unintended state changes of the program under test. These edges are often what allow exploit authors to deviate the program behavior from the intended state to a state of their choosing[1].

[1] Dullien, 2017. Weird Machines, Exploitability, and Provable Unexploitability

In this report, we call the fuzzer the compiled unit of code including the harness, but excluding the driving library, which may either be Hongfuzz [2] or libFuzzer[3].

[2] `https://github.com/google/honggfuzz`
[3] `https://llvm.org/docs/LibFuzzer.html`

3 Fuzz Testing In Envoy

3.1 Starting Point And Previous Efforts

Envoy implements more than 65 fuzzers where 19 are considered high priority. The high priority fuzzers exercise code paths that are exposed to potentially malicious input by being exposed to Internet traffic. As Envoy is offered as a service, the configuration interface is also considered exposed. Within the first phase, circa 60 issues found by fuzzers were analyzed and evaluated for their security impact and nearly 30 were fixed by X41. We learned that the fuzzers

1. validate the input generated by the mutator and valid input is generated rarely
2. cover configuration and data plane simultaneously
3. use debug assertions that may end runs prematurely
4. are big in size and part of their low performance can be attributed to this
5. are sometimes designed as randomized unit tests.

The following sections describe the issues and their consequences in detail.

3.1.1 Validating Input

Envoy is configured using Protobuf messages. Fuzzers that also target the configuration interface receive input generated by libprotobuf-mutator [1]. The input is extended by mutated input for the target data plane. The Protobuf message descriptions in Envoy are extended by validation rules, which generate additional code that can be used to check that a given message adheres to these rules. This code is used to validate configuration options at runtime but is also employed in the fuzzers. As a consequence most randomly generated input is rejected before the fuzz run,

[1] `https://github.com/google/libprotobuf-mutator`

since it is unlikely that random input satisfies a set of validation rules.

3.1.2 Fuzzing Control Plane And Data Plane

Envoy is heavily configurable and parses configuration files to Protobuf messages, which configure the individual functional parts of Envoy. Whereas for traditional server infrastructures the configuration is conducted by a trusted administrator, modern cloud-based infrastructures often expose the configuration to their customers. This necessitates to expand the traditional threat model: Risks arise not only from exposing the data plane of a software to untrusted input from the internet, but also from exposing the configuration plane to a (potentially malicious) customer. In addition to exploitable vulnerabilities, any crashes that may impact business metrics agreed to with the customer become risk factors. This poses a challenge to the security auditor, as the inclusion of the configuration plane as attack surface opens up another dimension in the state space of the program under testing: Fuzz testing the data plane in isolation attempts to explore all reachable memory states of the program for all possible CPU states. For complex software like Envoy this alone is impossible due to the large range of possible states. Taking into account the configuration plane multiplies the amount of possible states with the amount of configurations that enable new code paths or memory states.

Fuzzing the configuration plane poses further challenges, as the range of possible configurations is usually limited due to validation, but is still too vast to enumerate. To explore the set of possible (valid) configurations, a generator is necessary that is able to produce valid configurations efficiently. However, at the time no such technology exists for Protobuf messages. As a result, a randomly generated configuration is likely to be rejected and no attempt at fuzzing the data plane for the generated configuration is made. In fact, fuzzers that randomly generate configurations and are forced to abort when they fail to validate, suffer from poor performance and therefore stand no chance in finding any issues in the code under testing.

Fuzzing configuration and data simultaneously reduces the coverage of the code examined, because the fuzzer may steer away from a configuration that it should explore more deeply.

X41 proposed the following process:

1. Devise an efficient method to generate valid configuration files by introspecting the validator.
2. Set up a two-stepped fuzzing process, generating a valid configuration file first which is then re-used for fuzzing the data plane for a chosen number of iterations.

The results of these efforts are described in sec. 3.2.

3.1.3 Fuzzing Debug Builds

One of the issues X41 identified was that fuzzers are compiled with debug flags in place. This means that assertions that are violated trigger an ***abort()*** preventing the fuzzer to proceed. The debug assertion being violated may hint at a possible problem with the code. However, this also may not be the case or the impact of a violated assertion is negligible. The software project may decide to fix the bug leading to the violated assert or fix the assertion acknowleding that the assertion actually does not hold in all cases. However, even if all debug asserts in the project would only protect against serious problems, this may not be the case for libraries in use and compiled into the project. Changing an assertion in a 3rd party library for the benefit of a fuzzer is a problematic proposal.

The greater issue with debug assertions however is, that it stops the fuzzer from exploring deeper parts of the code unless the assertion is removed or the bug is fixed. Until then, issues that may arise after the assertion are being shadowed. An example could be found in the http2 codec implementation. In a getter for the stream handle an assert enforced that the result is non-null. However, at most call-sites the stream was checked for being non-null and then execution would branch appropriately. I.e., the call-site was able to a handle getting no stream at all. By introducing a new getter ***getStreamUnchecked()*** that omitted the assert the fuzzer is now able to continue investigating the code paths previously blocked.

Fuzzers set up this way may repeatedly ***abort()*** on assertions that are not issues with high impact. This can hurt the performance of the fuzzer and blocks improvement of the fuzzer on low priority bugs. Lastly, having to work around asserts while developing a fuzzer may lead to arbitrarily validating parts of the input, therefore encouraging to fix the issue in the fuzzer, rather than in the program.

X41 recommends to disable debug assertions as fuzzers with assertions tend to produce noise and shadow real issues.

3.1.4 Large executable size

Envoy generates executable files with very large sizes stemming from the approach of linking all kinds of components, like filters or extensions, to the executable instead of only the ones that are addressed by the test. This resulted in two peculiarities. Sometimes the compiler was not even able to use relative jumps anymore, i.e. the destination of a jump was more then $2^{31} - 1$ bytes away from the jump instruction. The other issue was that due to the target address being farther away than what would fit into the cache lines, a lot of cache misses occurred, which of course impacted the overall performance.

Large executable sizes have further consequences in the context of fuzzing. These have been identified previously and described elsewhere[2].

3.1.5 Integration Testing and Fuzzing

Various fuzzers that were designed to take structured input modeled to the state of some class. The fuzzer then validates that the structured input can be encoded to the state of the class under test and the state is then used to instantiate the class according to the input. The target code unit proceeds then to encode the state to e.g. bytes to be sent over the wire (e.g. HTTP or base64). The result is then subsequently decoded by the counterpart code unit, effectively testing one narrow code path with various inputs.

This approach has multiple issues:

1. The fuzzer needs to validate if the structured input can be encoded to the target unit or class. Input that fails this validation has to be rejected resulting in low performance as only a fraction of generated inputs can be used.
2. The fuzzer is prone to generating noise due to the fact that the state of some instance of a complex data structure is artificially set by the fuzzer and not by the program under test itself. The fuzzer may not set the state consistently for all values of the input or changes in the code may break the fuzzer in various ways.
3. The fuzzer misses the often more important attack surface of the decoder, as the decoder is only exercised with correctly encoded data.

X41 has addressed these issues by adding new dedicated fuzzers for high priority fuzzers, specifically for the HTTP codec libraries, c.f. sec. 3.6.

[2] Ada Logics, 2021. "Envoy fuzzing improvements"

3.2 Generation of Valid Input

The input to fuzz tests in Envoy usually is represented as structured text. YAML[3] is used to encode it. The structure of the YAML is defined using Google's protobuf[4] library. The structure definition is augmented by validation rules that can be interpreted by the protoc-gen-validate[5] package.

The validation rules may for example add constraints on the values of a member of a message, its cardinality, or whether a member is required:

```
1  message AdvanceTime {
2    // member is valid only when between 0 < milliseconds < 86400000
3    uint32 milliseconds = 1 [(validate.rules).uint32 = {gt: 0 lt: 86400000}];
4  }
5
6  message Action {
7    oneof action_selector {
8      // Ensure that exactly one of the alternatives is initialized
9      option (validate.required) = true;
10     OnData on_data = 1;
11     AdvanceTime advance_time = 2;
12   }
13 }
```

Listing 3.1: Example of validation rules

Listing 3.1 shows an example of two validation rules. In line 3 member `milliseconds` is declared to be only valid when greater than 0 and less than $86.400.000$ ms. The second validation rule in line 9 formulates the condition, that at least one of the alternatives in the `oneof action_selection` needs to be used.

The libprotobuf-mutator being used for creating new input during the fuzzer's mutation phase has only very limited knowledge about the input's structure. Libprotobuf-mutator makes no use of any validation rules present in the protobuf definitions. I.e. the mutator would generate an input having an `Action` message with no alternative of the `oneof` set. The validator then later on rejects this input, rendering the fuzz run of minor use. The fuzzer/mutator will only seldom compute a valid input this way.

The fuzzer allows to add a post processor to each fuzz test. The post processor is supplied a proposed new input for the next fuzz run. In this post processor not only checks can be done,

[3] YAML Ain't Markup Language
[4] `https://protobuf.dev/`
[5] `https://github.com/bufbuild/protoc-gen-validate`

but the input can be manipulated, too. Adding a new component to this post processor which examines the input and corrects it according to the validation rules is promising in reducing the number of inputs rejected by the validation step.

A prototype is proposed by X41 to handle the most often used validation rules. The validation rules are protobuf messages themselves. They can be examined using the reflection mechanisms provided by protobuf messages. A validation rule is added as an extension to a message's or member's type and can be retrieved using protobuf's access functions. Given the example in listing 3.1 on the milliseconds, the component would set the member to lower bound $+1$, if no value in the valid range is set.

The required option in the `oneof` of the `Action` message is implemented by ensuring that at least one alternative is set. If none has been set yet, one is chosen with uniform distribution over the available alternatives to allow the fuzzer to examine as many code paths as possible. This unfortunately had a caveat when a message referencing itself in multiple `oneof` alternatives was endeavored. These self-referencing messages are used to model conditions and have dis- and conjunctive alternatives (and/or operators). Choosing uniformly distributed from them leads to deeply nested conditions that most of the time produce a stack overflow and are rather seldom used in the wild. In these cases the uniform distribution is discarded and the and- and or-alternatives are chosen only with a probability of 0.05.

Another interesting construct in the protobuf messages used is a `typed_config`, where the type of the data in a `value` member is defined by a string given as additional input. I.e. the data in the `value` member resembles a valid protobuf message. But the type of this message is not defined when declaring the protobuf message that contains the `typed_config`. This implements context-aware polymorphism. For example, the configuration of a header matcher allows the and- and or-operations mentioned above to logically combine predicates that either match a header name using an exact match string comparison or a regular expression. Choosing one or the other is done using a `typed_config`. There are of course more matchers available. Reducing the example to these two is just for brevity. The `typed_config` can also be marked as required in the validation rules. To support this the component proposed uses a table to choose from. The table supplies the type for the `typed_config` and a default constructed message for the value member based on limited context, i.e. its parent type and the member that mandates its use. After that the component is applied to the default constructed member to ensure its validity.

3.3 Improved Fuzzing Strategy

An improved fuzzing strategy was devised to increase the performance of fuzzers that attempt to fuzz the configuration and data plane simultaneously: ***network_readfilter_fuzz_test*** and ***network_writefilter_fuzz_test*** have been modified to separately mutate the configuration and the data used for fuzzing. The configuration then was write protected for some number of iterations to allow the fuzzer to dive deeper into the data plane. That is the fuzzer's main loop now resembles something like in listing 3.2.

```
struct FuzzInput {
    Configuration config;
    Data data;
} input;
int count = 0;
for(;;) {
    input.data.mutate();
    test(input);
    if (count > SOME_LIMIT) {
        input.config.mutate();
        count = 0;
    }
    ++count;
}
```

Listing 3.2: New fuzzer main loop

Due to the use of callbacks in the fuzzer driver, the loop in the real code looks completely different. But listing 3.2 gives the general idea. The benefit of this separation is not only that the fuzzer attacks and therefore tests the data plane more, but also the number of executions per seconds done increases significantly. The configuration part in the input most of the time is more complicated and checking and correcting it is therefore expensive. The data on the other hand usually is just some binary or a command. Interpreting the data in the fuzzer is rather a no-op or really cheap, so that most of the time is spend in the code under test and no longer in figuring validity of the input.

3.4 Reducing Binary Size

Keeping the set of dependencies an artifact has on components or libraries small seems natural. Unfortunately this can get tedious in large projects. Therefore those tend to go for an all inclusive approach and define build dependencies that comprise all possible components and libraries. Although understandable this does have impacts on the binary size, when the compiler (and linker) is not clever enough or even told to remove unreachable code. Furthermore it is hard for the build tools to decide whether some code can never be reached.

Having a binary that contains more code than is needed on first glance might not be a problem. But when taking cache lines and followed by that cache misses into account a large binary runs slower when it has to perform far jumps, that invalidate several layers of caches. Remember that the farther away from the CPU data has to be fetched from the longer it takes.

The fuzzers ***network_readfilter_fuzz_test*** and ***network_writefilter_fuzz_test*** already had a white-list of the filters to test in them. These lists have been moved into the build system to only add the filters under test into the binary. The fuzzing routine retrieves a list of the filters to fuzz from an already present registration mechanism in the code. The changes therefore were quite minimal. The impact of the change is notable but not huge, i.e. the number of executions per seconds increased by a factor of 0.1 to 0.5 depending on the filter the fuzzer has chosen to execute.

3.5 Network Readfilter Fuzzer

The network read filter fuzzer targets extensions which filter particular data streams, for instance database protocols. The filter may be implemented to block requests if they meet certain criteria. For this purpose untrusted data is interpreted (hence "read") and therefore the network read filters constitute a high priority attack surface. Currently, there are 17 filter extensions present in Envoy, however only a small subset of 7 filter extensions are being fuzzed. The reason for that is, that some filter extension have dedicated fuzzers written for them or they need more context or external services running to function properly. Furthermore, some filters are deemed to be unstable according to the formulated threat model.

The fuzzer for network readfilter fuzzing targets the configuration as well as the data plane of the filter as described above. In the input of the fuzzer the first part gives a configuration, including the type of the filter to test, and the second part selects one of three possible actions. Those comprise a new connection, given data to process and an advancement of time. The latter is for example used to check expiration of sessions.

This fuzzer also suffers from the bipartite input, where the configuration is changed too quickly for the fuzzer being able to explore an arbitrary number of code paths for the specific filter. Furthermore any data being valid for one filter will probably not be applicable to the next filter. X41 therefore proposed to stick to one configuration for a given set of fuzz runs before generating a new configuration. This fuzzer was also the first candidate for the valid input generator as described in sec. 3.2. This accelerated the fuzzing speed by a factor of 4 to 20 depending on the configuration in use.

3.6 HTTP Codec Fuzzers

Envoy supports all three major HTTP versions and uses external codec libraries for parsing. At the time of writing, these are Balsa and a so-called legacy codec for HTTP/1, oghttp, and nghttp for HTTP/2 and Quiche for HTTP/3. Envoy wraps these libraries using thin wrappers, which provide an abstract interface generalizing the HTTP version to a set of data structures (headers, data, trailers). Fuzz tests exist in the Envoy project to target the HTTP/1 and HTTP/2 codecs in the following way:

1. A set of headers, data, and trailers are generated by a protobuf mutator
2. The set is deserialized into Envoy's generic data structures
3. The data structures are encoded by the codec to on-the-wire bytes
4. The bytes are decoded by the same codec back to Envoy's representation.

A fuzzer for the QUIC and HTTP/3 protocol was missing. The structure of the existing fuzzers resembles randomized unit tests rather than a fuzzer. Since the decoder is only exposed to input that was generated by an encoder, the fuzzer is artificially constrained and will not be able to maximize coverage of the decoder. However, it is the decoder that is most exposed to untrusted input. Targeting both the encoder and decoder this way will not expose critical security vulnerabilities and increase the runtime of the fuzzer for no added benefit.

In the scope of this project new fuzzers were contributed changing the approach to let the mutator generate on-the-wire bytes that may be random or represent a structured QUIC or HTTP message. The fuzzers target the decoder of the library directly and omit the encoder completely. The new fuzzers triggered several debug assertions that the previous fuzzers were not able to expose.

3.7 HTTP Connection Manager and Router Fuzzer

In Envoy, the HTTP Connection Manager manages one connection between a server and a client where exactly one (HTTP/1.0) or multiple HTTP streams are being processed. Simplified, a codec interprets the raw data sent and passes it to the connection manager, which passes it through a filter chain. The last filter in this filter chain is the router filter, which decides if the request is to be sent upstream or if an error or direct response can be generated immediately. The connection manager has the responsibility to manage the lifetime of these streams. There have been bugs, including CVE-2022-29227, in the past, which highlighted a need for coverage of these code paths. Indeed, a fuzzer that would simulated the generation of various streams with a number of differently behaving clusters was lacking. For this reason a fuzzer was devised that would be able to model more complex scenarios around the handling of HTTP streams and exercise some of Envoy's features like direct HTTP responses (cached responses) or internal redirects. Covering the decoding of HTTP has been omitted as this is covered elsewhere.

4 Conclusions and future Work

In conclusion X41 was able to contribute fuzzers that exercise the largest attack surface more efficiently and more importantly a method to fuzz both the configuration plane and data plane separately. An input generator which is able to generate valid configuration files more efficiently has significantly impacted fuzzing performance. Possible future work may directly be deducted from this report; Making more fuzzers reap the benefits of the new input generator and rewriting fuzzers to employ the method of fuzzing configuration and data plane is a natural continuation.

Special fuzzers may be contributed to exercise (possibly still unstable) filters in the read and write filter fuzzers. There are small details omitted from the current Envoy QUIC and HTTP/3 fuzzers like the handshake protocol and HTTP/3 header packing. Some of these may be covered already in their respective project.

X41 would like to thank the corresponding teams at Google, OSTIF, and Envoy for their collaboration and support.

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Review of the Mozilla Firefox updater[1]
- X41 Browser Security White Paper[2]
- Review of Cryptographic Protocols (Wire)[3]
- Identification of flaws in Fax Machines[4,5]
- SmartCard Stack Fuzzing[6]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[2] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[3] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[4] `https://www.x41-dsec.de/lab/blog/fax/`
[5] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[6] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

YAML	YAML Ain't Markup Language	10

A Appendix

First phase

X41 was asked to support in reducing the number of open fuzz issues in the envoy project. The project already has about 65 fuzzers implemented. Those ran on a regular basis and detected several bugs already. The maintainers of the project were somewhat overwhelmed by the number. That is were X41 stepped in providing expertise in fuzzers and development power to resolve bugs in fuzzers that were rated critical. The overall goal was to:

- reduce the number of open bugs,
- improve the coverage so fuzzers can explore code paths that were blocked, and
- reduce the signal-to-noise ratio to find actual security issues.

At the start of the project in March 2022 more than 150 issues were detected by the fuzzers and stored in a bug-tracker. The project partners identified high risk areas where the most impact on fixing bugs and improving coverage could be gained.

The diagram in figure A.1 shows the number of fuzz issues open at the end of project phase 1 as a blue bar. About 70 issues have been addressed (green bar) and only a small number of high risk issues were left open that have been present on project start. Note, that the red and the green bars summed up do not give the blue bars height, because the blue bar also comprises low priority fuzz issues.

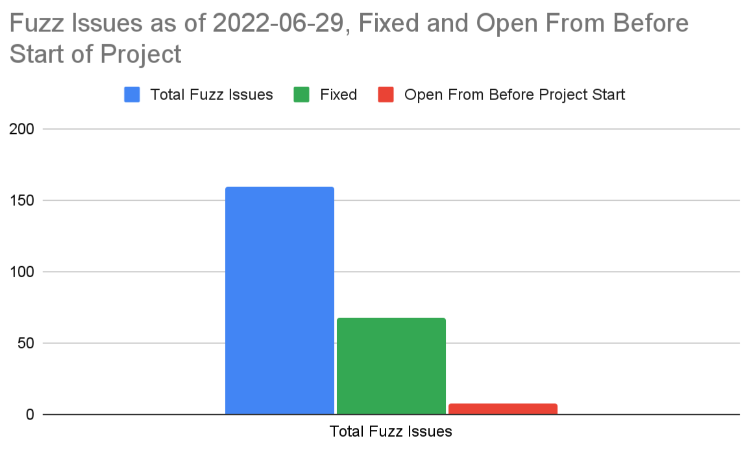

Figure A.1: Statistics on fuzzing issues before and after phase 1 of the project.

Selected Issues

One of the first issues tackled was about valid or rather invalid characters in header-names or -values. The fuzzer had generated these invalid headers and the tested routines in envoy then aborted although the invalid headers in a production environment would never get there. The fuzzer bypassed a step of the input parsing, that would have rejected the invalid headers ensuring that the input is fine for the inner components of envoy. The solution to this kind and some similar ones, was to add Protobuf validation rules to the input specification and therefore ensuring, that only valid headers were generated. Adding Protobuf validation rules was in about one third of all issues the fix.

In another issue an event loop was involved. The loop was not always quick enough to respond in time. Sometime it was not even configured to be triggered by the configuration chosen by the fuzzer. This was resolved by removing that filter from the available ones to the generic fuzzer. Instead the support for the specific fuzzer for this filter was extended.

A rather big class of fuzz issues were due to ***assert()***s being triggered by the fuzzer. The most likely cause was that due to mocking and simulation of an environment the conditions for that code containing the ***assert()*** were not met, which could not happen in production. This could only be resolved, by either completing the environment in the fuzzers or blocking the fuzzer from doing certain things, like writing to an already closed stream, by tracking state in the fuzzer better.

Escaping of rare special character and line ending in yaml-cpp

In the yaml-cpp library not all variations of line endings were recognized. A single carriage return, Mac-style line ending, was not detected as line break character. Only the combination of carriage return and newline, Windows style, was parsed correctly.

Furthermore was the ampersand character (&), which denotes an anchor in yaml, not correctly escaped when emitting yaml. X41 proposed fixed for both issues, which have been appreciated.

Second phase

During the second phase new fuzzers were added and these uncovered a number of issues. A selection is given below.

Integer underflow in oghttp2 codec

An integer underflow was discovered in the oghttp2 decoder library in quiche. Triggering the issue will crash a debug build of envoy on the assertion that *remaining_data_* is greater or equal than *padding_length*.

```
bool CallbackVisitor::OnDataPaddingLength(Http2StreamId /*stream_id*/,
                                          size_t padding_length) {
  QUICHE_DCHECK_GE(remaining_data_, padding_length);
  current_frame_.data.padlen = padding_length;
  remaining_data_ -= padding_length;
  if (remaining_data_ == 0 &&
      (current_frame_.hd.flags & NGHTTP2_FLAG_END_STREAM) == 0 &&
      callbacks_->on_frame_recv_callback != nullptr) {
    const int result = callbacks_->on_frame_recv_callback(
        nullptr, &current_frame_, user_data_);
    return result == 0;
  }
  return true;
}
```

Listing A.1: Integer underflow

The issue seems to be harmless, as *remaining_data_*, even when underflowed does not control critical behaviour and is a private variable to *class CallbackVistor* and does not get passed to the session callback interface.

Type confusion in a debug assertion in the oghttp2 codec

A debug assertion could be triggered that hints at a possible type confusion issue in the oghttp2 decoder library in quiche. The assertion assumes that the flags of the current http frame must not equal *kMetadataEndFlag*, which can be forced.

```
bool CallbackVisitor::OnMetadataEndForStream(Http2StreamId stream_id) {
  QUICHE_LOG_IF(DFATAL, current_frame_.hd.flags != kMetadataEndFlag);
  QUICHE_VLOG(1) << "OnMetadataEndForStream(stream_id=" << stream_id << ")";
  if (callbacks_->unpack_extension_callback) {
    void* payload;
    int result = callbacks_->unpack_extension_callback(
        nullptr, &payload, &current_frame_.hd, user_data_);
    if (result == 0 && callbacks_->on_frame_recv_callback) {
      current_frame_.ext.payload = payload;
      result = callbacks_->on_frame_recv_callback(nullptr, &current_frame_, user_data_);
    }
    return (result == 0);
  }
  return true;
}
```

Listing A.2: Possible type confusion

The severity of the issue may depend on the implementor of the *callbacks_* class, which may not be part of the quiche library or envoy. Each user of the library must conduct proper input validation, however, consumers of this API may be tempted to assume that the assertion holds and rely on it, which may have any number of adverse effects.

Type confusion in a debug assertion in the oghttp2 session handler

A debug assertion could be triggered that hints at a possible type confusion issue in the oghttp2 session handler in quiche. The assertion assumes that the current frame header type is of *HeaderType::REQUEST*, which may not hold. The severity of the issue is likely low.

```
HeaderType OgHttp2Session::NextHeaderType(
    absl::optional<HeaderType> current_type) {
  if (IsServerSession()) {
    if (!current_type) {
      return HeaderType::REQUEST;
    } else {
      QUICHE_DCHECK(current_type == HeaderType::REQUEST);
      return HeaderType::REQUEST_TRAILER;
    }
  } else
    /* ... */
}
```

Listing A.3: Possible type confusion