Protocol chaos

Jan Macháček, Miguel Lopez, Matthew Squire

February 25, 2018

Abstract

Chaos engineering is a way to introduce unexpected failures in order to discover a system's failure modes. The chaos engineering tools usually introduce faults that focus on the connections between the system's components by disrupting the expected operation of the network (by introducing latency, packet loss, all the way to complete network partitions) or by disrupting the operation of the nodes hosting the components (by reducing the available CPU or memory resources). Our work on chaos testing showed that this is not only insufficient, but that more "damage" can be achieved by focusing first on how a system handles unexpected input–trivially structurally invalid messages; or structurally valid messages that contain invalid values; or strucurally valid messages with valid, yet damaging or malicious values. Once the system can reliably handle invalid and malicious inputs, the infrastructure chaos engineering can be used to tease out further failure scenarios.

1 Protocol chaos

Validation of inputs is acklowledged to be important, yet it is very often not implemented as thoroughly as it should be; not for the lack of understanding its importance, but for the effort it requires. Consider a component that accepts a message to announce a greeting n numer of times; it exposes a REST endpoint at *POST /greeting* and expects a message with the *greeting* and *count* values. The description of the service and the field names in the input message provide intuition about the expected values: the *greeting* can be one of *{ "hello", "hi", "top␣of␣the␣afternoon␣to␣you,␣sir!"}* and similar; the value for *count* is an integer in the range of $(1; 5)$. The first level of validation is to check that the values posted indeed represent valid "types" (the *greeting* is text; the *count* is an integer). This is "included for free" in strongly-typed languages if the input message uses appropriate types for its members; and it is simple to do in weakly-typed languages. The next level of validation should focus on acceptable range of values. For the *count* property, the validation code should verify that it is indeed in the range of (say) $(1; 5)$. The validation code for the *greeting* property can be more complex: it is difficult to precisely define valid or invalid *string* . The possible invalid examples include empty string, very long string, etc; though it remains easy to find an invalid string that nevertheless passes the validation code, and it is just as easy to find valid string that fails. (Consider "Good morning, Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch!": it is perfectly valid greeting in a small village in North Wales.) If discovering the validation rules for *greeting* is difficult, it is even more difficult to exhaustively test those rules.

It is possible to implement a test that verifies that the system under test processes the message as expected. However, this is a very broad requirement, so it is usually split into multiple *unit* tests. But even with this split, the initial problem of imagining and then coding the possible messages still remains. In order to allow valid messages conforming to some protocol to be generated, there needs to be a machine-readable description of the protocol. This description needs to include the elements and their types. The types should include at least integer numbers, floating point numbers, strings, enumerations, arrays, maps, and other messages; it is useful if the protocol description can describe sum types. The protocol tooling should also be able to take the protocol descriptions and generate source code for the target language and framework combination. Protocol Buffers [7] is an example of the protocol definition language and rich tooling that satisfies these requirements. The code in Listing 1 shows how to define a message X with two fields in the Protocol Buffers syntax.

```
message X {
    int32 count = 1;
    string greeting = 2;
}
```

Listing 1: Trivial protocol definition

Given this definition, the Protocol Buffers tooling can be used to generate the source code for Scala, Swift, C++, and many other languages. The generated code for each message includes definition of the protocol; the definition is shown in pseudo-code in Listing 2.

The message X contains two fields, both optional: *count*, of type 32 bit signed integer

```
field {
  name: "count"
  number: 1
  label: LABEL_OPTIONAL
  type: TYPE_INT32
  json_name: "count"
}
```

and *greeting*, of type UTF-8 string

```
field {
  name: "greeting"
```

```
    number: 2
    label: LABEL_OPTIONAL
    type: TYPE_STRING
    json_name: "greeting"
}
```

Listing 2: Trivial protocol definition

Using this definition, it is possible to devise a generator that examines the types of the fields, and emits possible values that conform to the field's type: the range of values for *int32* includes $\{0, 1, 64, 2^{31} - 1, -2^{31} + 1, -100, -1\}$; the range of values for *string* includes empty, short alphanumeric, long alphanumeric, and even more values can be found on The Big List of Naughty Strings [12]. Generators for other types follow similar pattern in the samples they provide. It is useful to add heuristics to the generator, making it consider the field name in addition to the field type. This way, a generator for field *url* can generate (malicious) URLs in addition to simply one of the *naughty strings*. The range of values that the generator can emit for each type is very large, the generator needs to take–in addition to the field definition–a random number generator and emits a structure that can be used to combine with other generators. See Listing 3 for an example in ScalaCheck.

```
object MessageGenerator {
  Constructs a generator for a type of type A, with optionally
  provided hints for the fields
  def msg[A <: GeneratedMessage with Message[A]]
      (implicit cmp: GeneratedMessageCompanion[A],
       hint: Hint[A] = noHint): Gen[A] = ...

  The default hints that can be imported into scope where
  msg is used
  object hints {
    implicit def default[A]: Hint[A] = ...
  }

  Hint for type A is able to provide specific generator for the given fd
  trait Hint[+A] {
    def hint(fd: FieldDescriptor): Option[Gen[Any]]
  }

}

To use it, we prepare the parameters and seed for the application of
the generators
val parameters = Gen.Parameters
                   .default
                   .withSize(1999)
def seed = Seed(System.currentTimeMillis())
import MessageGenerator._
import hints._

Generate one "chaotic" instance of message X, using the in-scope
Hints[X]
val x = msg[X](parameters, seed)
println(x)
```

Listing 3: Generator

Several executions of the code above yielded usefully-damaged instances of message X (shown in Listing 4); these instances were indeed valid according to the protocol definition, but some of the values can be considered unusual.

```
X(count: -4959664 greeting: "dlnz6hbbfeb6ofu...")
X(count: 0 greeting: "dlnz6hbbfeb6ofu...")
X(count: -1 greeting: "dlnz6hbbfeb6ofu...")
X(count: 1 greeting: "dlnz6hbbfeb6ofu...")
...
X(count: 0 greeting: "sqjsY8y3bvhg3az...")
```

Listing 4: Generator examples

Even though this code is specific to ScalaCheck[9] in Scala, there are similar *property-based* testing frameworks available for other languages (such as RapidCheck[8] for C++, SwiftCheck[11] for Swift, and similar).

2 Security and integrity

It is important to consider the attack vectors on the protocol (how easy is it to construct a malicious message) that causes unexpected behavior in the protocol implementations; it is just as important to understand the resilience of the protocol's wire format against transmission errors. This should inform the design of security and integrity measures, which can range from simple CRC checksums to cryptographic signatures and encryption[1].

2.1 Integrity

Message integrity mechanisms detect errors in the layers that follow the construction of the message. These layers include not only the network transport, but also the protocol codecs, networking libraries, etc. For example, Kafka[2] defines maximum message size. The wire format plays an important role in being able to detect failures in the messages' integrity. The proportion of useful data to the protocol wire format control structures defines the probability of a bit error damaging the message so that it cannot be decoded. Take the message in Listing 5, which shows a message in the Protocol Buffer language.

```
message X {
    int32 count = 1;
    string greeting = 2;
}
```

Listing 5: Trivial protocol definition

A wire format of a message conforming to this protocol with *count* → *42; greeting* → *"Hello,␣world"* is shown in Figure 1 and Figure 2.

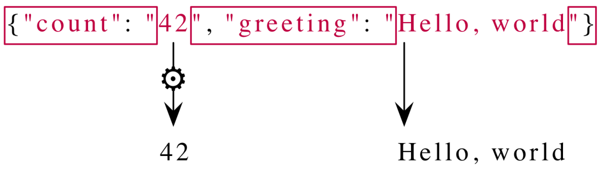


Figure 1: JSON wire representation

[1] Doing nothing is dangerous, particularly for systems that are exposed to other systems or even the Internet.

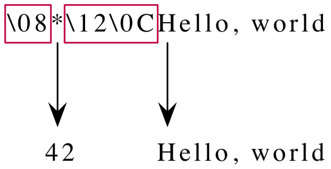


Figure 2: Protocol Buffers binary wire representation

The binary representation directly maps to the bytes that make up the 4 B *int32* value and the *string* with length *0x0c*. There are only 3 B that appear on the wire in addition to the useful information. The JSON representation contains 28 B that are included on the wire in addition to the useful information; moreover, the value *42* has to be translated from a textual representation to the *int32* value. Consider a situation of a bit error on the wire in 1 random byte. The probability of hitting the bytes that form the wire format is 3/16 in the Protocol Buffers wire format; and 7/4 in the JSON wire format. The damage to the message is therefore more likely to be detectable by parsing errors in the less dense JSON representation (see Figure 3).

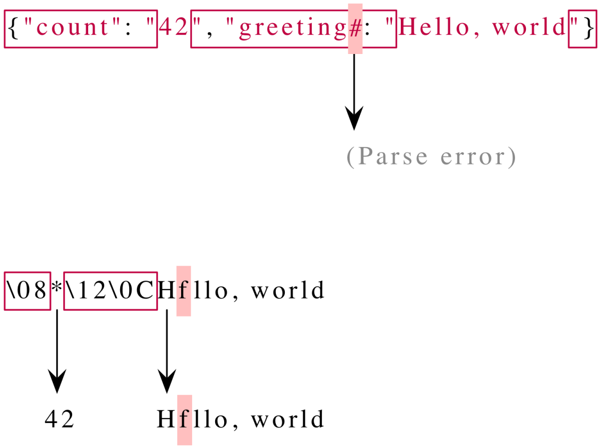


Figure 3: Bit error impact

The more dense the wire representation is, the more important it is to include bit error detection. A simple checksum using CRC32 [5] is sufficient for systems with low to moderate message rates. A parallel (though CPU-bound) implementation of algorithm 1, capable of brute-forcing 1 200 000 messages/s per second on i7-7920HQ CPU took 1564.6 s (averaged over 10 runs) to find one instance of conflicting message[2].

An example of such conflicting message of type *X* from Listing 5 is shown in Table 1. The messages have the same *count* property (set to 583145568), differ in the *greeting* property by 19 bits, yet share the same CRC32 value (1540204925). Consider a system that includes a component that "reliably" introduces random bit errors and handles 120 000 messages/s. The interval between undetectable damage to messages that rely on CRC32 to detect errors will be only 4 h!

```
Data: generator $g$ of instances of $X$ with sufficiently
      large number of bits
Result: message $x$, its wire representation $b$ and
        damaged wire representation $b'$ such that $b'$
        is valid according to the protocol format:
        $b \neq b' \wedge deser(b) \neq$
        $deser(b') \wedge checksum(b) = checksum(b')$
while $true$ do
    $x \leftarrow g()$
    $b \leftarrow ser(x)$
    $c \leftarrow checksum(b)$
    repeat
        $b' \leftarrow damage(b)$
        $c' \leftarrow checksum(b')$
        if $c = c' \wedge deser(b) \neq deser(b')$ then
            return $x, b, b'$
        end
    until $damaged\text{-}bits(b') < threshold$
end
```

Algorithm 1. Bit damage

idx	492	493	521	680	689	1052	1760	2203	2544
greeting[*idx*]	U	]	E	?	z	|		%	v
greeting'[*idx*]	E	L	\01	*	2	\32	D	!	p

Table 1. Bit damage

In a system that processes large number of messages, or if the possibility of encountering undetectable damaged messages is unacceptable (say in systems that deal with sensitive personal or financial information) use cryptographic hashes to verify the integrity of the messages. Unfortunately, this comes with computational overhead which is summarized in Table 2 (with CRC32 on 1 KiB as the baseline).

	Message size				
Algorithm	100 B	1 KiB	10 KiB	100 KiB	1 MiB
CRC32	3	1	7	606	9574
SHA-256[a]	9	26	118	11860	120595
SHA-256[b]	50	150	681	68787	699450
SHA-512[b]	94	156	475	48619	487513

Table 2. Integrity overhead

[a] Intel SHA[3] extensions

[b] Plain x86_64 implementation

[2] Even though it is easy to find *different* messages that have the same CRC32 value, it is difficult to find messages that share the same CRC32 value, but differ only by a few bits *and* are valid wire format representations.

The results show that using robust cryptographic digest (SHA-256 or SHA-512) add significant overhead when compared to the simple CRC32 checksum, though on CPUs with the SHA extensions and matching implementation[3] the cost of the overhead is significantly lower.

JWT [4] provides value semantics for authorisation, but also provides mechanisms to verify the integrity of the messages. JWT combines the message itself (called payload in JWT) and *claims* that can be used for authorisation.

2.2 Malicious messages

A system can be attacked by sending it malicious input, aiming to either make it unavailable, or to make it perform operation that the attacker is not privileged to do.

The denial-of-service style attacks aim to completely crash the nodes running the services; for example by causing out-of-memory or stack-overflow errors that the system should not catch, or causing high CPU usage, or by making the system under attack read too much data from an I/O device. An example of a message that causes stack-overflow exception in Protocol Buffers using *"com.google.protobuf" % "protobuf−java" % "3.4.0"* and *"com.trueaccord.scalapb" %% "compilerplugin" % "0.6.6"* [10] is shown in Listing 6.

Construct a malicious message that starts with the valid serialized bytes *b* and adds 6000 bytes with the value 99.

```
val b  = Array[Byte](8, 1, 18, 3)
val mb = Array.fill(6000)(99.toByte)
Array.copy(b, 0, mb, 0, b.length)
```

Decode an instance of *X* the input *mb*

```
X.validate(mb)
↑
Exception in thread "main" java.lang.StackOverflowError
 at ...$StreamDecoder.readTag(...:2051)
 at ...$StreamDecoder.skipMessage(...:2158)
 at ...$StreamDecoder.skipField(...:2090)
 ...
```

Listing 6: Malicious message

Other wire formats are just as susceptible to this type of attacks: sending too-deeply nested JSON causes the same behaviour in eager JSON parsers, for example *"org.json4s" %% "jackson" % "3.5.1"* (viz Listing 7).

Construct malicious JSON document

```
val mj = """{"x":""" * 2000
```

Decode an instance of *X* from the input *mj*

```
JsonFormat.fromJsonString[X](mj)
↑
Exception in thread "main" java.lang.StackOverflowError
 at ...JsonStreamContext.<init>(...:43)
 at ...JsonReadContext.<init>(...:58)
 at ...JsonReadContext.createChildObjectContext(...:128)
 at ...ReaderBasedJsonParser._nextAfterName(...:773)
 at ...ReaderBasedJsonParser.nextToken(...:636)
 at ...JValueDeserializer.deserialize(...:45)
 ...
```

Listing 7: Malicious message

The thread that throws the *StackOverflowError* arising from *JsonFormat.fromJsonString[X](mj)* and *X.validate(mb)* without handling it in a *try* / *catch* block terminates. This attack can be mitigated and the impact of the too-deep recursion causing the *StackOverflowError* s on other parts of the system contained by moving the decoding to a thread from a *dedicated* pool[4]. a suitable language & framework makes this a fairly easy task. An example of safe, though very inelegant (because of the use of *Await.result*) code in Scala is shown in Listing 8. The *decode* function can be used as a direct replacement in your code (though production implementation should pay extra attention to properly configure the *ExecutionContext* with sufficient number of threads and pinning behavior). Non-blocking toolkits (such as Akka[1] and Play[6] in Scala) use non-blocking style throughout, which means that there is no need to block for the result of the asynchronous operation; the code in Listing 8 and the performance measurements represent the worst-case scenarios.

The decoding function takes the bytes *b* to decode and schedules the decoding in the *ec*. It returns the result of the decoding, but will survive a *StackOverflowError*.

```
def decode(b: Array[Byte])
  (implicit ex: ExecutionContext): Try[X] = {
  import scala.concurrent.duration._
  val timeout = 100.milliseconds
  Try(Await.result(Future(X.parseFrom(b)), to))
}
```

Construct "good" payload

```
val b  = Array[Byte](8, 1, 12, 3, 65, 66, 67)
```

Construct malicious payload

```
val mb = Array[Byte](8, 1, 12, 3, 65, 66, 67,
                     99, ..., 99)
```

With an implicit *ExecutionContext* in scope, applying *decode* to the good payload *b* returns *Success*(*X*(...)) in roughly 17 ms.

```
decode(b)
```

With an implicit *ExecutionContext* in scope, applying *decode* to the good payload *b* returns *Failure*(*TimeoutException*) after the timeout of 100 ms elapses.

```
decode(mb)
```

Listing 8: Stack Overflow Mitigation

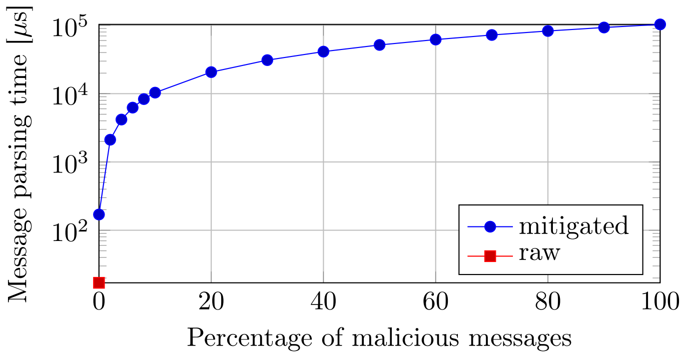
While the code to implement the mitigation against this type of attack is straightforward, moving the decoding to another thread adds significant dispatch overhead (roughly 17 times the raw) to the decoding (see Figure 4).

Another attack is one that sends messages to the system that are just under the *maximum−size* limit. With sufficient, though still fairly low number of messages, this will cause the system to fail with out-of-memory error; and this cannot be contained in a separate thread pool; a proof-of-concept code is shown in Listing 9.

[3] For example, on x86_64 CPUs it will be necessary to verify that the compiler / runtime emits the *SHA256RNDS2, SHA256MSG1, SHA256MSG2* instructions

[4] A side benefit of moving the decoding to a dedicated thread pool provides bulk-heading of the decoding code from the rest of the application code

Figure 4: Mitigation overhead



```
Construct a malicious message that specifies the greeting property to
be a string of 10485759 characters. The eager parser makes a copy of
the entire buffer, the streaming parser reads & allocates the memory
by chunks.
val b  = Array[Byte](8, 1, 18, −1, −1, −1, 4)
val mb = Array.fill(b.length + 10485759)(65.toByte)
Array.copy(b, 0, mb, 0, b.length)

Simulate in flight processing, even though we count to 200 the OOM
happens around 165
val inFlight: ArrayBuffer[X] = ArrayBuffer()
for (_ ← 0 to 200) {
  val x = X.parseFrom(mb)
  ↑
  Exception in thread "main" java.lang.OutOfMemoryError:¶
   Java heap space
   at ...StringCoding.decode(...:215)
   at ...String.<init>(...:463)
   at ...ArrayDecoder.readString(...773)
   ...
  inFlight += x
}
```

Listing 9: Malicious message

It is particularly dangerous if the system is designed to read messages from a journal in a transactional-like manner: because the message causes a crash that cannot be handled, the *read* operation is never confirmed; upon restart, the system will read the malicious message again. While the implementation of the protocol decoding code should be as robust as possible, it is necessary to consider other mechanisms to skip a message that causes fatal errors. A useful testing approach is to provide a generator that can yield malicious payloads for any message type; see Listing 10.

```
object MessageGenerator {
  Constructs a malicious generator for a type of type A,
  with optionally provided hints for the fields
  def msgM[A <: GeneratedMessage with Message[A]]
      (implicit cmp: GeneratedMessageCompanion[A],
       hint: Hint[A] = noHint): Gen[A] = ...
}
```

Listing 10: Malicious Generator

2.3 Practical application

In the first application of the chaos approaches & tooling we found three distinct classes of problems: the deserialization code, the tracing and logging code, and the error recovery and retry code. Once discovered, all were simple to fix, but the consequence of missing the bugs would have resulted in complete loss of service had similar messages reached our system in production.

The mitigation for first class of issues focused on setting strict message size limits, and wrapping the decoding code in a separate thread pool, which provided the necessary isolation. The next issue found was in the logging machinery: too-deeply-nested messages resulted in the failures in the structured logging code. Both issues cause the services to react in the same way, the system as a whole didn't crash because it is using Akka supervision, instead individual services or actors crashed consuming a given message from Kafka either due to the deserializer or the logger, the message processing wasn't acknowledged. After restart the services consumed the same last batch of acknowledged messages creating an endless loop of restarts, meaning the messages before the malicious one are reprocessed and no messages after the culprit can be consumed.

References

[1] *Akka*. `https://akka.io`.

[2] *Apache Kafka*. `https://kafka.apache.org`.

[3] Guilford Jim, Kirk Yap, and Vinodh Gopal. *Fast SHA-256 Implementations on Intel Architecture Processors*. Tech. rep. Intel.

[4] *JSON Web Tokens*. `https://jwt.io`.

[5] W Kowalk. *CRC Cyclic Redundancy Check Analysing and Correcting Errors*. `http://einstein.informatik.uni-oldenburg.de/papers/CRC-BitfilterEng.pdf`. Universität Oldenburg.

[6] *Play Framework*. `https://playframework.com`.

[7] *Protocol Buffers*. `https://developers.google.com/protocol-buffers/`.

[8] *RapidCheck*. `https://github.com/emil-e/rapidcheck`.

[9] *ScalaCheck*. `https://scalacheck.org`.

[10] *ScalaPB*. `https://scalapb.github.io`.

[11] *SwiftCheck*. `https://github.com/typelift/SwiftCheck`.

[12] *The Big List of Naughty Strings*. `https://github.com/minimaxir/big-list-of-naughty-strings`.