New issue

Jump to bottom

For security, prevent `Function0` execution during `LazyList` deserialization #10118

Merged lrytz merged 1 commit into `scala:2.13.x` from `lrytz:deser` on Aug 31

lrytz commented on Aug 23 • edited

Member

This PR ensures that LazyList deserialization will not execute an arbitrary `Function0` when being passed a forged serialization stream.

Overview

The most important take-away is: **Deserializing untrusted data in a Scala (or Java) application is always a security risk.** This recommendation does **not** change after this PR is merged.

A forged serialization stream can be used to execute unrelated code, which can be exploited. The next section explains the case that is being addressed in this PR in detail. However, any class on an application's classpath with similar code can be exploited. Therefore, untrusted data should never be deserialized.

A presentation about the topic: https://www.youtube.com/watch?v=MTfE2OgUlKc

Technical details

The following details play together:

- `LazyList` has a `var lazyState: () => LazyList.State[A]` field, which is of type `Function0` in bytecode
- Serialization uses a "proxy" object with custom serialization / deserialization methods:
  - Evaluated elements of the lazy list are serialized first
  - Then the unevaluated tail of the lazy list is serialized using standard Java object serialization
- The deserialization method invokes `tail.prependedAll(init)` to reconstruct the lazy list (`init` are the evaluated elements)
- The `prependedAll` may invoke the `lazyState()` function of the `tail` object; this does not happen under normal circumstances, but a forged serialization stream can force the control flow into this situation

- The `lazyState` object is created when deserializing the `tail` object
  - On a typical application classpath, there are a lot of `Function0` subclasses; for example, every argument to a by-name parameter is encoded into a `Function0`
  - A forged serialization stream can contain any `Function0` instance for the `lazyState` field, for example one that interacts with the file system or the network

In this scenario, the invocation of the `lazyState` function happens before the deserialized `LazyList` is actually used (for example assigned to a local variable). So even if the `LazyList` object would lead to a `ClassCastException` later on, the custom `lazyState` function is already executed. In other words, a serialization stream containing a forged `LazyList` would lead to executing the `Function0` at any deserialization of the application, no matter what object type the application expects to read.

Change in this PR

In this PR, we change the `LazyList` deserialization method to ensure that this process cannot evaluate the `state` of the lazy `tail`.

The forged `Function0` can still be executed later when evaluating the tail of the lazy list. However, this is only possible if the application actually expects a serialized `LazyList` and evaluates its tail. This makes an exploit much less likely.

Credits

We would like to thank Marc Bohler for finding this issue and reporting it to us.

This issue is reported as CVE-2022-36944.

👍 3 💔 1

lrytz added the **library:collections** label on Aug 23

lrytz added this to the **2.13.9** milestone on Aug 23

lrytz requested a review from **NthPortal** as a code owner 3 months ago

NthPortal reviewed on Aug 23

View changes

src/library/scala/collection/immutable/LazyList.scala **Outdated**

...	...	@@ -1370,7 +1377,7 @@ object LazyList extends SeqFactory[LazyList] {
1370	1377	case a => init += a.asInstanceOf[A]

```
1371  1378          }
1372  1379          val tail = in.readObject().asInstanceOf[LazyList[A]]
1373        -       coll = init ++: tail
      1380  +       coll = tail.withNullLazyState(tail.prependedAll(init))
```

NthPortal on Aug 23 • edited — Contributor

why don't we use a `Builder[LazyList]` for `init` instead of an `ArrayBuffer`, and just call `lazyAppendedAll` on the result?

NthPortal on Aug 23 • edited — Contributor

in fact, because `tail` is also a `LazyList`, I believe we can use `LazyBuilder` and just call `addAll` with `tail` as an argument, and we don't even rebuild the `LazyList` ever (not that I think it has a huge performance impact)

NthPortal on Aug 23 • edited — Contributor

side note: there was a discussion at some point somewhere suggesting that no `Builder`s should be lazy (don't remember where), and that `LazyList.newBuilder` should return an eager `Builder` (probably for `List`) that calls `mapResult` to create a `LazyList`. On the off-chance that ever happens, we should explicitly create a `new LazyBuilder` rather than calling `LazyList.newBuilder` if we go with the change I suggested

NthPortal on Aug 23 • edited — Contributor

to put it differently: evaluating `tail` there does not represent a fundamental flaw, shortcoming or gap in the design of `LazyList`, it represents a careless programming error likely due to copy-pasting

lrytz on Aug 24 — Member, Author

Looking at `LazyBuilder`, I actually think we'd run into the same issue. If we change the code to

```
val tail = in.readObject().asInstanceOf[LazyList[A]]
coll = lazyBuilder.addAll(tail).result()
```

we have

```
override def addAll(xs: IterableOnce[A]): this.type = {
  if (xs.knownSize != 0) {
    ...

override def knownSize: Int = if (knownIsEmpty) 0 else -1
...
  private[this] def knownIsEmpty: Boolean = stateEvaluated && isEmpty
...
  override def isEmpty: Boolean = state eq State.Empty
```

So calling `addAll` could evaluate the `state` lazy val, which calls the `lazyState` function.

Load more...

lrytz on Aug 24 — Member — Author

Yeah, a forged serialization stream can have that field set to `true`

👍 1 👀 2

NthPortal on Aug 24 — Contributor

point. I guess we fall back to `lazyAppendedAll` then

NthPortal on Aug 25 • edited — Contributor

@lrytz I whipped up my own fix that uses `stateFromIteratorConcatSuffix` . I was going to push an alternate PR to consider, but I haven't figured out how to actually test that the fix works. Been having issues deserializing the lambda.

If you'd like to use my code, the only changes are to the `SerializationProxy` (below)

```
final class SerializationProxy[A](@transient protected var coll: LazyList[A]) extends Se

  private[this] def writeObject(out: ObjectOutputStream): Unit = {
    out.defaultWriteObject()
    var these = coll
    while(these.knownNonEmpty) {
      out.writeObject(these.head)
      these = these.tail
    }
    out.writeObject(SerializeEnd)
    out.writeObject(these)
  }

  private[this] def readObject(in: ObjectInputStream): Unit = {
    in.defaultReadObject()
    val init = new ListBuffer[A]
    var initRead = false
    while (!initRead) in.readObject match {
      case SerializeEnd => initRead = true
      case a => init += a.asInstanceOf[A]
    }
    val tail = in.readObject().asInstanceOf[LazyList[A]]
    coll = newLL(stateFromIteratorConcatSuffix(init.toList.iterator)(tail.state))
  }

  private[this] def readResolve(): Any = coll
}
```

and to import `ListBuffer` instead of `ArrayBuffer`.

Edit: I think the only change is to `readObject`?

lrytz on Aug 29 — Member — Author

Thank you, this looks good, nicer than my workaround 👍

lrytz on Aug 29 — Member — Author

Also added a test.

NthPortal reviewed on Aug 23

View changes

NthPortal left a comment — Contributor

it feels weird to me to add a new method used just for this when we already have lazy methods. I think the main issue was using an `ArrayBuffer` (which is on me)

NthPortal reviewed on Aug 23

View changes

`test/junit/scala/collection/immutable/LazyListTest.scala` Outdated — Show resolved

SethTisue added the `prio:blocker` label on Aug 23

NthPortal reviewed on Aug 24

View changes

`src/library/scala/collection/immutable/LazyList.scala` — Show resolved

lrytz force-pushed the `deser` branch from `0fb2d9d` to `064f5c9` 3 months ago — Compare

NthPortal reviewed on Aug 24

View changes

test/junit/scala/collection/immutable/LazyListTest.scala Outdated Show resolved

lrytz force-pushed the deser branch from 064f5c9 to 977c87f 3 months ago Compare

lrytz added the release-notes label on Aug 25

lrytz force-pushed the deser branch 3 times, most recently from a8aacb0 to a40c17a 3 months ago Compare

lrytz requested a review from SethTisue 3 months ago

NthPortal approved these changes on Aug 30

View changes

src/library/scala/collection/immutable/LazyList.scala Outdated Show resolved

test/junit/scala/collection/immutable/LazyListTest.scala Outdated Show resolved

test/junit/scala/collection/immutable/LazyListTest.scala Show resolved

Prevent Function0 execution during LazyList deserialization ... f24c226

lrytz force-pushed the deser branch from a40c17a to f24c226 3 months ago Compare

lrytz enabled auto-merge 3 months ago

lrytz merged commit 3a12413 into scala:2.13.x on Aug 31 View details

2 of 3 checks passed

SethTisue removed the prio:blocker label on Sep 1

SethTisue changed the title ~~Prevent Function0 execution during LazyList deserialization~~ For security, prevent Function0 execution during LazyList deserialization on Sep 1

SethTisue changed the title ~~For security, prevent Function0 execution during LazyList deserialization~~ For security, prevent `Function0` execution during `LazyList` deserialization on Sep 1

xuwei-k commented on Sep 16 — Contributor

Should we fix collection-compat?
https://github.com/scala/scala-collection-compat/blob/d6723b241ab943f8f0e7da21292c7994ce4d9f80/compat/src/main/scala-2.11_2.12/scala/collection/compat/immutable/LazyList.scala

lrytz commented on Sep 16 — Member, Author

👍 thank you for the pointer!

sideeffffect mentioned this pull request on Oct 9

For security, prevent `Function0` execution during `LazyList` deserialization (backport from 2.13) scala/scala-collection-compat#557

Closed

Reviewers

NthPortal ✓

SethTisue ●

Assignees

No one assigned

Labels

library:collections release-notes

Milestone

2.13.9

4 participants