SocketIO / EngineIO DoS

May 2020

Quite a while ago, I reported an application Denial of Service vulnerability in the Socket.IO / Engine.IO parser implementations in nodejs and python. A single HTTP POST request can cause extreme CPU and memory usage, but in nodejs, a single HTTP POST request can even kill the server with a `Javascript heap out of memory` fatal error.

I assume some of what I've written is incorrect as I'm not an expert on v8 internals, but I do really love getting to the bottom of edge-case performance issues.

Protocol

The engine.io protocol allows bi-directional communication between a server and client, abstracting away the actual transport. The transport can be WebSockets, but if that isn't supported then another transport such as HTTP long polling is possible.

When the WebSocket transport is used, packets are encapsulated by the engine.io protocol. First there is a number specifying the packet type. For instance, ping packets starting with `2` are sent as WebSocket data even though WebSocket has its own heartbeat mechanism. Sending the JSON data `{"a": 123}` requires prefixing with a `4`. The socket.io protocol on top of that, if used, will add a `2` prefix meaning EVENT, so a WebSocket listener will receive `"42{\"a\":123}"`.

Using the long-polling transport, a payload containing multiple packets can be sent. In version 3 of the protocol, the payload is encoded as:

```
<length1>:<packet1>[<length2>:<packet2>[...]]
```

e.g `6:42[{}]11:4abcdefghij1:2` contains 3 packets:

1. Socket.io packet of length 6: Message (4), Event (2, `[{}]`)
2. Packet of length 11: Message (4), Data (`abcdefghij`)
3. Packet of length 1: Ping (2)

With WebSockets, the 3 packets would be sent separately. With HTTP long polling, the payload would be POSTed to `http(s)://host/socket.io/?EIO=3&transport=polling&sid=$SESSIONID`.

The denial-of-service bug lies in:

- Inefficient parsing of packets from payloads
- Maximum HTTP body size of 100MB

Make your 100MB count

You can send a payload containing 1e8 bytes to the server. That's quite a huge message, but how can we cause the server the most pain and suffering? The main methods are:

- **Many tiny packets**: send 25,000,000 empty event packets `2:422:422:422:42`
- **One giant int**: send the largest possible packet with integer data `99999991:42222222222222222222...`
- **Many heartbeats**: send 33,333,333 ping packets `1:21:21:21:2...`

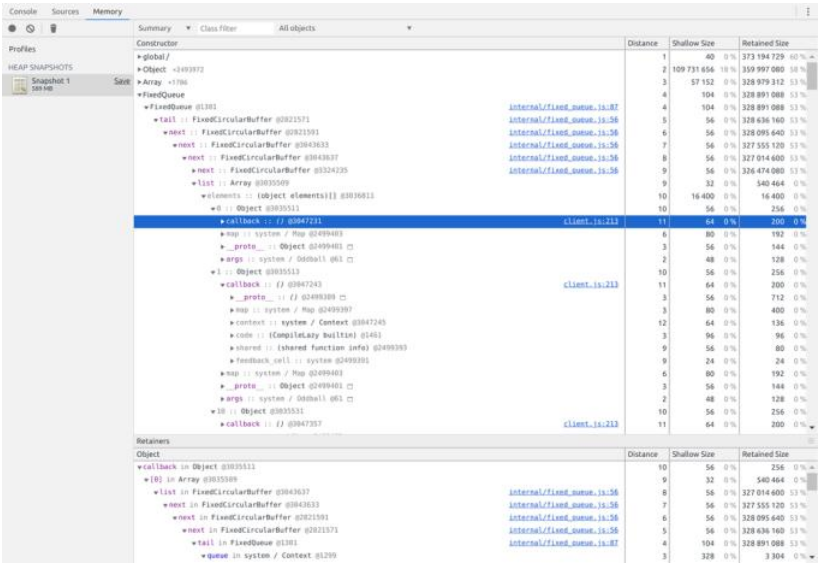
Loading the body string into memory automatically eats up 100MB as a starting point, but it gets a hell of a lot worse.

Nodejs

With NodeJS, if the ping timeout (default 30s) is exceeded then the processing appears to be cancelled. Therefore, sending a payload which is so large it doesn't reach the memory exhausting step within the ping timeout will not kill the process. It will just waste CPU for 30 seconds. Sending a slightly smaller payload instead may cause the process to exit.

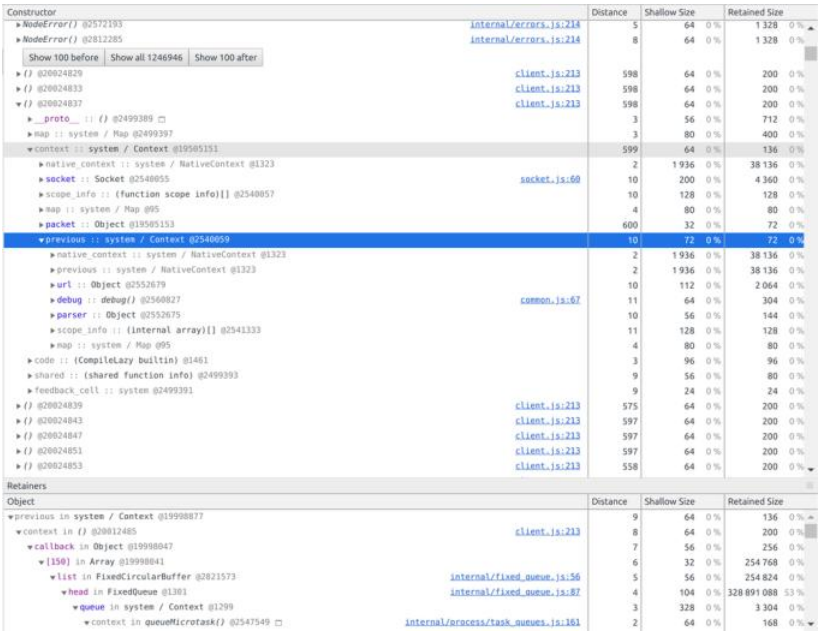
Many tiny packets

The bug here is due to this 2016 change. As the parser reads packets from the payload, it doesn't emit the `socket.onpacket` event immediately. Instead it queues up a new closure with `process.nextTick`. Since the next tick of the event loop doesn't come until all packets have been parsed, memory usage blows up.



FixedQueue retaining many closures to be executed before the next tick

`process.nextTick` stores the closures in FixedCircularBuffers inside a FixedQueue. Each of these closures retains 200 bytes of heap memory (retained means that if this closure could be garbage collected, it would free this amount of heap memory). Not a lot per closure (no giant objects retained), but it adds up to ~5gb.



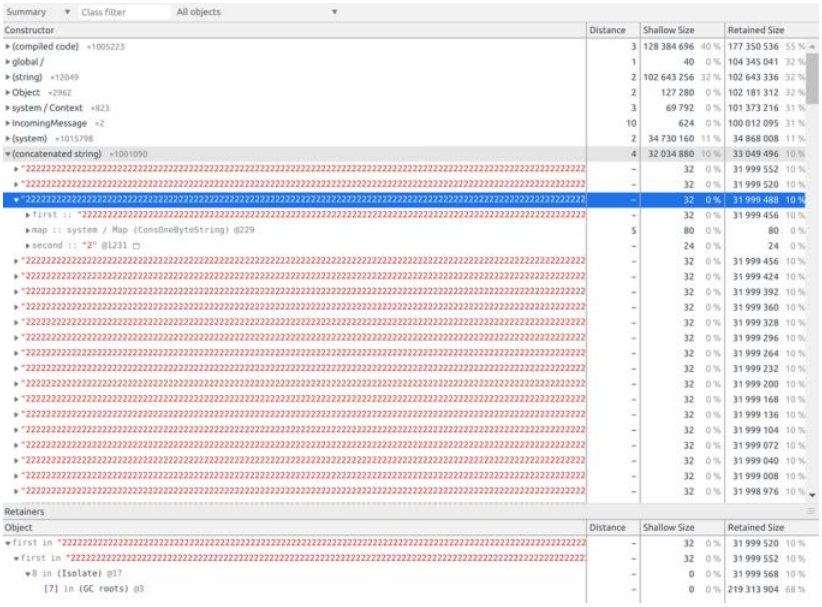
A huge number of closures. Closure code (96 bytes) + context (64 bytes) + packet (72 - 32 due to double counting) = 200 bytes

One giant integer

This is best explained by looking at my fix.

Luckily string concatenation in v8 doesn't create an entirely new string like in some languages where string builders are required. Instead, `a + b` becomes `ConsString { first = a, second = b }` pointing to the two smaller strings. There are even optimised versions `ConsOneByteString` and `ConsTwoByteString`.

Sending the "One giant int" packet can cause OOM via building up many many `ConsOneByteString` objects (32 bytes each) due to concatenation: 99999989 `ConsOneByteString`s and then converting the massive integer to a `Number`.



ConsOneByteString

```
==== JS stack trace =========================================

    0: ExitFrame [pc: 0x13c5b79]
Security context: 0x152fe7b808d1 <JSObject>
    1: decodeString [0x2dd385fb5d1] [/node_modules/socket.io-parser/index.js:~276] [pc=0xf59746881be]
    2: add [0x31fc2693da29] [/node_modules/socket.io-parser/index.js:242] [bytecode=0xa7ed6554889 off

FATAL ERROR: Ineffective mark-compacts near heap limit Allocation failed - JavaScript heap out of mem
 1: 0xa09830 node::Abort() [node]
 2: 0xa09c55 node::OnFatalError(char const*, char const*) [node]
 3: 0xb7d71e v8::Utils::ReportOOMFailure(v8::internal::Isolate*, char const*, bool) [node]
 4: 0xb7da99 v8::internal::V8::FatalProcessOutOfMemory(v8::internal::Isolate*, char const*, bool) [no
 5: 0xd2a1f5  [node]
 6: 0xd2a886 v8::internal::Heap::RecomputeLimits(v8::internal::GarbageCollector) [node]
 7: 0xd37105 v8::internal::Heap::PerformGarbageCollection(v8::internal::GarbageCollector, v8::GCCallb
 8: 0xd37fb5 v8::internal::Heap::CollectGarbage(v8::internal::AllocationSpace, v8::internal::GarbageC
 9: 0xd3965f v8::internal::Heap::HandleGCRequest() [node]
10: 0xce8395 v8::internal::StackGuard::HandleInterrupts() [node]
11: 0x1042cb6 v8::internal::Runtime_StackGuard(int, unsigned long*, v8::internal::Isolate*) [node]
12: 0x13c5b79  [node]
```

Many heartbeats

This causes OOM as many pongs are created to reply to all the pings.

Python

With eventlet, a single payload can DoS the entire server until processing completes due to the absence of `eventlet.sleep` calls. Without eventlet, the non-production server remains responsive until the thread pool is exhausted, so requires more than 1 concurrent request.

Many tiny packets (special)

Payload: `2:4½2:4½2:4½2:4½2:4½2:4¼...`

When non-ascii characters are present in the payload,

```
encoded_payload.decode('utf-8', errors='ignore')
```

is much slower:

```
export N=100000; python -m timeit -s "x=b'2:42' * $N" "x.decode('utf-8', errors='ignore')"; python -m
10000 loops, best of 3: 37.6 usec per loop
10 loops, best of 3: 29.3 msec per loop

export N=10000000; python -m timeit -s "x=b'2:42' * $N" "x.decode('utf-8', errors='ignore')"; python
100 loops, best of 3: 9.08 msec per loop
10 loops, best of 3: 2.95 sec per loop
```

As engineio reads a packet, it decodes the entire remaining payload and then advances the length of the packet. So for an N-packet payload, the decode function is applied to:

- (string of N packets)
- (string of N-1 packets)
- (string of N-2 packets)

so slowing down the decoding makes the DoS much more potent as it's $O(n^2)$!

This was fixed by the maintainer.

All others

The python code seems to generally run slower than the nodejs code. Large payloads cause DoS primarily by wasting CPU time since python doesn't have a max heap size in the same way as v8. One giant int is slow as `int("2" * int(1e7))` is incredibly slow in python, perhaps because it allows Unicode digits like ٢ as well.

Exploit

I made a repo bcaller/kill-engine-io containing test servers and code to trigger the DoS. Enjoy.

Servers with a lower max HTTP body size are less vulnerable. In fact, the default has been lowered in newer versions.

Tags: security | poc | research | nodejs | python

[blog by caller] Correspondence welcome at 𝒷log {@} cal𝓁e𝓇.ⓧⓨℤ