Simple Byte Hacking

A Programming Article by Efron Licht

Jan 2023

Many junior & intermediate programmers can be a little skittish around around byte-level hacking. You shouldn't be. It's not as hard as it's made out to be, and getting comfortable with low-level programming can make for simpler, more efficient code, In this article, we'll take an example of a real-world problem with a solution that was best discovered and implemented with byte-level programming.

We'll build a few command-line tools and benchmark results along the way.

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

Intro

At a previous job, we had a `mongoDB` database that was giving us some trouble by behaving differently between C# and Golang. We would serialize a struct containing a UUID on the C# end, but get a different UUID on the Golang end.

(If you don't know what a UUID is, all you need to know for this is that it's a 128-bit long number that serves as a unique ID in many databases, and that it has a human-readable format that looks like this: "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx", where `x` is one of 0-9 or A-F.)

We'd insert something like this on the C# side:

```
0a3ec7aa-60e3-457b-aca0-9fb9665339bf
```

And get something like this:

```
aac73e0a-e360-7b45-aca0-9fb9665339bf
```

Doing some reading, it turned out that this was a "legacy UUID encoding" of some kind, but none of the documentation I read explained *how they were different*. One of my co-workers, a clever junior engineer, found a javascript solution that seemed to work, but involved some rather nasty string manipulation:

```
var hex = uuid.replace(/[{}-]/g, ""); // remove extra characters
var a =
  hex.substr(6, 2) + hex.substr(4, 2) + hex.substr(2, 2) + hex.substr(0, 2);
var b = hex.substr(10, 2) + hex.substr(8, 2);
var c = hex.substr(14, 2) + hex.substr(12, 2);
var d = hex.substr(16, 16);
hex = a + b + c + d;
var base64 = HexToBase64(hex);
return new BinData(3, base64);
```

Testing the code in Javascript, it seemed to work. So my practical co-worker proposed we simply port the solution to Golang and move on. I was a little nervous: I don't like putting something in my code I don't understand. I wanted to take a look under the hood and see if I could find out what was happening, and if so, write a simple, efficient solution. We'd meet back up in 45 minutes. If I couldn't get anywhere, we'd just go with his solution and move on with our lives.

Using Our Eyes

> Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.
>
> **Fred Brooks**

The first step to decoding any kind of messaging protocol is just to *look at the data*. As mentioned before, UUIDs have a canonical form. I generated a few UUIDs on the C# end, printed them out in that form, stuck them in the database, pulled them out on the Go end, and printed *those* out, side-by-side.

c#	golang
`0a3ec7aa-60e3-457b-aca0-9fb9665339bf`	`aac73e0a-e360-7b45-aca0-9fb9665339bf`
`8fc4f550-aeae-4736-8ac2-a16a4d90cf6c`	`50f5c48f-aeae-3647-8ac2-a16a4d90cf6c`

c#	golang
`93e32e64-e78b-4a45-872c-fda57e5c4bea`	`642ee393-8be7-454a-872c-fda57e5c4bea`
`e973437d-0b5b-42ae-923d-cab553eb5aee`	`7d4373e9-5b0b-ae42-923d-cab553eb5aee`

These UUIDs are split up into five sections. Let's label them 0-1-2-3-4 and examine them one at a time.

- section 0 & 1: always contain the same characters, just in a different order.
- section 2: appears to be mirrored: if we have `457b` in c#, we have `7b45` on the go end.
- section 3 & 4: identical between C# and Golang.

This leads to a hypothesis: the legacy C# encoding is the same as the usual one, **but it has the bytes in a different order**. Maybe if we decompose the UUID into individual bytes and look at those, we can find out if that's true, and if so, what the ordering is.

Using Our Eyes (on the bytes)

A UUID is a 128-bit number, or 16 bytes. The latter is how the go library `github.com/google/uuid` handles them: as a `[16]byte` array. Let's write a small command-line program to parse a pair of UUIDS in their canonical form, decompose them into individual bytes, then print THOSE out side-by-side. (I like to write small command-line programs for tasks like this: you never know when you might need them later). For convenience, let's format the results as a markdown table, so we can insert them directly into, say, a blog post.

printpair.go

```
 1  package main // printpair.go
 2  import "os"
 3  import "fmt"
 4  import "strings"
 5  import "github.com/google/uuid"
 6  func main() {
 7      chsharp, golang := uuid.MustParse(os.Args[1]),
    uuuid.MustParse(os.Args[2])
 8      fmt.Printf("|i|c#|golang|\n")
 9      for i := 0; i < 16; i++ {
10          // the %x formatting verb means "format this as
    hexadecimal"
11          // the %02x means "and pad it with zeroes to length
    2, if it's shorter than that."
12          fmt.Printf("|%x|%02x|%02x|\n", i, csharp[i],
    golang[i])
13      }
```

```
14
15  }
```

Let's try running it:

```
1  printpair "0a3ec7aa-60e3-457b-aca0-9fb9665339bf" "aac73e0a-
   e360-7b45-aca0-9fb9665339bf"
```

i	c#	golang
0	93	64
1	e3	2e
2	2e	e3
3	64	93
4	e7	8b
5	8b	e7
6	4a	45
7	45	4a
8	87	87
9	2c	2c
a	fd	fd
b	a5	a5
c	7e	7e
d	5c	5c
e	4b	4b
f	ea	ea

I ran it through the rest of our pairs and confirmed our theory:

- bytes 8-F are unchanged
- bytes 6 & 7 are swapped
- bytes 5 & 6 are swapped
- bytes 0-4 seem to be permuted

Finding The Solution

But how exactly are they permuted? Let's build a program that makes a lookup table and run a bunch of results through it. As before, let's format the output as a markdown table so we can insert it directly into the article.

buildtable.go

```
1  package main // buildtable.go
2
3  import (
```

```
 4    "fmt"
 5    "os"
 6
 7    "github.com/google/uuid"
 8   )
 9   func main() {
10    csharp, golang := uuid.MustParse(os.Args[1]),
     uuid.MustParse(os.Args[2])
11    var c2g, g2c [16]byte
12    for i := range csharp {
13     for j := range golang {
14      if csharp[i] == golang[j] {
15       c2g[i] = byte(j)
16       g2c[j] = byte(i)
17      }
18     }
19    }
20    fmt.Printf("|i|c2g|g2c|\n")
21    fmt.Printf("|---|---|---|\n")
22
23    for i := 0; i < 16; i++ {
24     fmt.Printf("|%x|%02x|%02x|\n", i, c2g[i], g2c[i])
25    }
26   }
```

```
1   buildtable "93e32e64-e78b-4a45-872c-fda57e5c4bea"
    "642ee393-8be7-454a-872c-fda57e5c4bea"
```

i	c2g	g2c
0	03	03
1	02	02
2	01	01
3	00	00
4	05	05
5	04	04
6	07	07
7	06	06
8	08	08
9	09	09
a	0a	0a
b	0b	0b
c	0c	0c
d	0d	0d
e	0e	0e
f	0f	0f

This leads me to a few theories:

- c2g is the same as g2c.
- It's also it's own inverse: c2g(c2g(n)) == n.

Let's run it on a few more UUIDs to see if we get the same results:

```
buildtable "8fc4f550-aeae-4736-8ac2-a16a4d90cf6c" "50f5c48f-aeae-3647-8ac2-a16a4d90cf6c"
```

This gives an identical result. Promising!

But this:

```
buildtable "8fc4f550-aeae-4736-8ac2-a16a4d90cf6c" "50f5c48f-aeae-3647-8ac2-a16a4d90cf6c"
```

doesn't: we get the following

i	c#	golang
0	03	03
1	02	02
2	01	01
3	00	00
4	05	05
5	05	05
6	07	07
7	06	06
8	08	08
9	09	09
a	0a	0a
b	0b	0b
c	0c	0c
d	0d	0d
e	0e	0e
f	0f	0f

This is a different lookup table! Let's use `printpair` again to see if we can explain this result:

```
printpair "8fc4f550-aeae-4736-8ac2-a16a4d90cf6c" "50f5c48f-aeae-3647-8ac2-a16a4d90cf6c"
```

i	c#	golang
0	8f	50
1	c4	f5

i	c#	golang
2	f5	c4
3	50	8f
4	ae	ae
5	ae	ae
6	47	36
7	36	47
8	8a	8a
9	c2	c2
a	a1	a1
b	6a	6a
c	4d	4d
d	90	90
e	cf	cf
f	6c	6c

In this UUID, Bytes 4 and 5 are *both* `0xae`, so this mapping *is* correct; it's just not complete.

We now have a theory for the mapping function: it's just the table `c2g`.

That is, the items are mapped as follows:

before	after
0, 1, 2, 3	3, 2, 1, 0
4, 5	5, 4
6, 7	7, 6
8..=15	8..=15

First golang solution: `swap16`

Let's convert this to Go code: since we'll use a 16-byte lookup table, let's call it `swap16`.

```
1   var tbl [16]byte{0x3, 0x2,  0x1, 0x0,  0x5, 0x4,  0x7, 0x6,
    0x8, 0x9, 0xA, 0xB, 0xC,  0xD, 0xE, 0xF}
2   // convert a mongoDB c# legacy UUID to a standard one, or
    visa-versa.
3   func swap16(src uuid.UUID) uuid.UUID {
4       dst := uuid.UUID{}
5       for i, j := range tbl {
6           dst[i] = src[j]
7       }
8   }
```

second golang solution: `swap8`

This seems slightly inefficient to me: after all, we never use the back half of the table! Let's omit that, saving some iteration and a whole 8 bytes of memory.

```
1  var tbl [8]byte{3, 2, 1, 0, 5, 4, 7, 6}
2  // create a mongoDB c# legacy UUID from a standard one, or
   visa-versa.
3  func swap8(src uuid.UUID) uuid.UUID {
4      dst := src // copy the entire thing
5      for i, j := range tbl { // and permute the first half
   according to the swap table.
6          dst[i] = src[j]
7      }
8  }
```

Just as I was ready to benchmark `swap8`, my co-worker came to me with a Go version of the string-manipulation based Javascript solution. Let's call it `swapJS`.

the javascript-style solution

```
1   var removeExtras = strings.NewReplacer("{", "", "}", "",
    "-", "")
2   import "encoding/hex"
3   func swapJS(id string) uuid.UUID {
4    b16 := removeExtras.Replace(id)
5    a := b16[6:6+2] + b16[4:4+2] + b16[2:2+2] + b16[0:0+2]
6    b := b16[10:10+2] + b16[8:8+2]
7    c := b16[14:14+2] + b16[12:12+2]
8    d := b16[16:]
9    src, _ := hex.DecodeString(a + b + c + d)
10   var dst uuid.UUID
11   copy(dst[:], src)
12   return dst
13  }
14
```

With our new insights, we can understand `swapJS`: it swaps the bytes around, just like we do, only it does so by manipulating the *string representation of the bytes* rather than the bytes themselves.

To spell it out:

- Disassemble the canonical form into the four sections in our mapping, where each two-character pair is a hexadecimal-encoded byte.
    - a: `(0, 1, 2, 3) <-> (3, 2, 1, 0)`
    - b: `(4, 5) <-> (5, 4)`
    - c: `(5, 6) <-> (6, 5)`
    - d: `(8..=15) <-> (8..=15)`
- Glue them together into a 32-character string representing 16 hexadecimal bytes.

- Parse the string as hexadecimal.
- Copy it into the underlying [16]byte of a uuid.UUID. (In the original javascript solution, they transformed it from Hex to Base64, but that has to do with MongoDB internals).

one last micro-optimization

The string-manipulation solution has a good idea: it *omits the lookup table entirely*. Maybe we can do that?

```
1  func swapDirect(u uuid.UUID) uuid.UUID {
2   u[0], u[1], u[2], u[3] = u[3], u[2], u[1], u[0]
3   u[4], u[5] = u[5], u[4]
4   u[6], u[7] = u[7], u[6]
5   return u
6  }
```

Sanity check: basic tests

We have four solutions that seem equivalent. Let's quickly write some tests that make sure they are. In this case, we have a well-trusted function, `uuid.New()`, that will generate random test data. Let's lean on that to generate 20K or so tests:

```
 1  func TestSwapEquivalence(t *testing.T) {
 2   rand.Seed(time.Now().UnixNano())
 3   for i := 0; i < 20_000; i++ {
 4    u := uuid.New()
 5    res8, res16, resDirect, resJS := swap8(u), swap16(u),
    swapDirect(u), swapJS(u.String())
 6    if res8 != res16 || res8 != resDirect || res8 != resJS {
 7     t.Errorf("expected all 4 to be equivalent: %s %s %s
    %s", res8, res16, resDirect, resJS)
 8    }
 9    if swap8(res8) != u || swap16(res16) != u ||
    swapDirect(resDirect) != u || (swapJS(resJS.String())) !=
    u {
10
    t.Errorf("expected swap to be it's own inverse: %s %s %s
    %s", res8, res16, resDirect, resJS)
11    }
12   }
13  }
```

benchmarking: the code

We now have four equivalent solutions to the same problem: `swap8`, `swap16`, `swapDirect`, and `swapJS`. I assume that `swap8` or `swapDirect` will be the fastest solution, but I'm not sure by how much. Let's compare their performance under two

different situations: when we start with a string, and when we start with the `[16]byte` `uuid.UUID`

```
 1  package bench_test // bench/bench_test.go
 2
 3  import (
 4   "encoding/hex"
 5   "os"
 6   "strings"
 7   "testing"
 8
 9   "github.com/google/uuid"
10  )
11
12  // go's compiler will try and optimize out functions that
    don't do anything,
13  // so we attempt to work around that by assigning to this
    global.
14  var _uid uuid.UUID
15
16  // in TestMain, we generate 128 random UUIDs and alternate
    which one we use for each test,
17  // just in case UUIDs take different amounts of time to
    format or parse
18  // depending on their format. this may be overly clever.
19  var uuids [128]uuid.UUID
20  var uuidStrings [128]string
21
22  func TestMain(m *testing.M) {
23
24   for i := range uuids {
25    uuids[i] = uuid.New()
26   }
27   // preformat the UUIDs as strings for the "string"
    scenario
28   for i := range uuidStrings {
29    uuidStrings[i] = uuids[i].String()
30   }
31   os.Exit(m.Run())
32  }
33  func BenchmarkSwap8(b *testing.B) {
34   for i := 0; i < b.N; i++ {
35    _uid = swap8(uuids[b.N%128])
36   }
37  }
38  func BenchmarkSwap8String(b *testing.B) {
39   for i := 0; i < b.N; i++ {
40    _uid = swap8(uuid.MustParse(uuidStrings[b.N%128]))
41   }
42  }
43
44  func BenchmarkSwap16(b *testing.B) {
45   for i := 0; i < b.N; i++ {
```

```
46      _uid = swap16(uuids[b.N%128])
47     }
48    }
49    func BenchmarkSwap16String(b *testing.B) {
50     for i := 0; i < b.N; i++ {
51      _uid = swap16(uuid.MustParse(uuidStrings[b.N%128]))
52     }
53    }
54
55    func BenchmarkSwapDirect(b *testing.B) {
56     for i := 0; i < b.N; i++ {
57      _uid = swapDirect((uuids[b.N%128]))
58     }
59    }
60
61    func BenchmarkSwapDirectString(b *testing.B) {
62     for i := 0; i < b.N; i++ {
63      _uid = swapDirect(uuid.MustParse(uuidStrings[b.N%128]))
64     }
65    }
66
67    func BenchmarkSwapJS(b *testing.B) {
68     for i := 0; i < b.N; i++ {
69      _uid = swapJS((uuids[b.N%128].String()))
70     }
71    }
72
73    func BenchmarkSwapJSString(b *testing.B) {
74     for i := 0; i < b.N; i++ {
75      _uid = swapJS((uuidStrings[b.N%128]))
76     }
77    }
```

sidenote: formatting the output

We're going to look at a lot of benchmarks on this blog, so let's take a moment to write a program to format the benchmarks for us into a nice markdown table.

That is, given output from `go test -bench=. -benchmem` that looks like this:

```
1   BenchmarkSwap8-8                148829586
    7.814 ns/op           0 B/op          0 allocs/op
2   BenchmarkSwap8String-8          23469300
    50.81 ns/op            0 B/op          0 allocs/op
3   // some omitted
```

We'd like output like this:

name	runs	ns/op	%/max	bytes	%/max	allocs	%/max
Swap8	1.54e+08	7.77	2.25	0	0	0	0
Swap8String	2.36e+07	50.3	14.6	0	0	0	0

```
// fmtbench
package main // fmtbench/main.go

import (
 "flag"
 "fmt"
 "io"
 "log"
 "math"
 "os"
 "regexp"
 "sort"
 "strconv"
 "strings"
)

var sortBy = flag.String("sort-by", "none",
"sort criteria: options 'none' 'allocs' 'name', 'runtime'")

func main() {
 flag.Parse()
 switch strings.ToLower(*sortBy) {
 case "none", "allocs", "name", "runtime":
  // nop
 default:
  flag.Usage()
  log.Printf("unexpected value %q for flag -sortby",
*sortBy)
 }
 input :=
strings.TrimSpace(string(must(io.ReadAll(os.Stdin))))
 lines := strings.Split(input, "\n")
 goos := strings.TrimPrefix(lines[1], "goarch: ")
 goarch := strings.TrimPrefix(lines[0], "goos: ")
 pkg := strings.TrimPrefix(lines[2], "pkg: ")
 fmt.Printf("## benchmarks %s: %s/%s\n", pkg, goos, goarch)
 fmt.Println(`|name|runs|ns/op|%/max|bytes|%/max|allocs|%/
max|`)
 fmt.Println(`|---|---|---|---|---|---|---|---|`)
 // get results and min/max
 type result struct {
  name                    string
  runs, ns, bytes, allocs float64
 }
 var results []result
 var maxNS, maxBytes, maxAllocs float64
 {
  re := regexp.MustCompile(`Benchmark(.+)\s+(\d+)\s+(.+)ns/
op\s+(\d+) B/op\s+(\d+)`)

  for _, line := range lines {
   match := re.FindStringSubmatch(line)
   if match == nil {
    continue
```

```
50      }
51      atof := func(i int) float64 { return
    must(strconv.ParseFloat(strings.TrimSpace(match[i]), 64)) }
52      res := result{name: match[1], runs: atof(2), ns:
    atof(3), bytes: atof(4), allocs: atof(5)}
53      results = append(results, res)
54      maxNS, maxBytes, maxAllocs = math.Max(maxNS, res.ns),
    math.Max(maxBytes, res.bytes), math.Max(maxAllocs,
    res.allocs)
55     }
56    }
57
58    { // sort results
59     var less func(i, j int) bool
60     switch *sortBy {
61     case "none":
62      goto PRINT
63     case "allocs":
64      less = func(i, j int) bool { return results[i].allocs <
    results[j].allocs }
65     case "name":
66      less = func(i, j int) bool { return results[i].name <
    results[j].name }
67     case "runtime":
68      less = func(i, j int) bool { return results[i].ns <
    results[j].ns }
69     }
70     sort.Slice(results, less)
71    }
72   PRINT:
73    for _, res := range results {
74     fmt.Printf("|%s|%.3g|%.3g|%0.3g|%.3g|%0.3g|%.3g|%0.3g|
    \n", res.name, res.runs, res.ns, (res.ns/maxNS)*100,
    res.bytes, (res.bytes/maxBytes)*100, res.allocs,
    (res.allocs/maxAllocs)*100)
75    }
76   }
77
78   func must[T any](t T, err error) T {
79    if err != nil {
80     log.Print("unexpected error")
81     log.Print("USAGE:  go test -bench=. -benchmem DIR |
    bench")
82     log.Fatal(err)
83    }
84    return t
85   }
86
```

comparing benchmarks

Let's run the benchmarks and feed them to `fmtbench`:

```
1  go test -bench=. -benchmem ./bench | go run ./fmtbench
```

Before we look at the results, let's make some guesses:

- `swap8` or `swapDirect` will be the fastest solution, depending on the generated code, since none of them involve allocating any memory and are simple. I'm not sure which will be faster: this could depend on the compiler.
- `swap16` will be close in performance: probably within 50%.
- In the case where we have to parse strings into UUIDs, they probably all have comparable performance.

benchmark gitlab.com/efronlicht/uuidswap/bench: goarch: arm64/goos: darwin

function name	runs	ns/op	%/max	bytes	%/max	allocs	%/max
SwapDirect	2.87e+08	4.39	1.28	0	0	0	0
Swap8	1.54e+08	7.79	2.26	0	0	0	0
Swap16	8.91e+07	13.3	3.88	0	0	0	0
SwapDirectString	2.54e+07	49.6	14.4	0	0	0	0
Swap8String	2.36e+07	51	14.8	0	0	0	0
Swap16String	2.09e+07	57.1	16.6	0	0	0	0
SwapJSString	4.61e+06	258	75	64	57.1	2	66.7
SwapJS	3.48e+06	344	100	112	100	3	100

How pretty!~~This definitely didn't take me longer than all of the other parts~~.

We note the following conclusions:

- `swapDirect` is fastest, and `SwapJS` is slowest.
- Unsurprisingly, simply moving around the bytes is much faster than moving around and parsing a lot of strings. Our fastest solution is `swapDirect`.
- The difference between `swapDirect` and `swap16` *is* significant: it's over 3 times as fast!
- But even when we have to parse UUIDs, the byte-manipulation approaches are five times faster.
- The best-case scenario (`swapDirect` fed a UUID) is 80 times faster than the worst-case (`swapJS` fed a UUID).

So, the question remains, *was all this work worth it?* In general, synthetic benchmarks don't tell you much about the performance of your overall program. As long as a function isn't glacially slow, if it's not called often, it doesn't really impact the performance of your program. If you want to know how your program needs to behave in practice, you need to *profile*, not *benchmark*: benchmarking is for narrowing in on the best solution to a localized problem.

Still, regardless of whether this has a meaningful performance impact, I'm happy with our results:

- Finding the solution took me under an hour (Writing this article took longer, of course.)
- it gave us some insight into the underlying problem: that the legacy C# encoding is the *same bytes in a different order*.
- It was fun! You rarely get to mess with bytes directly.
- Unmarshaling a UUID is an extremely common operation. Nearly every interaction with `mongoDB` is likely to involve one or many UUIDs. While the runtimes individually are very small, they can add up.
- We made a number of handy command-line programs that may be useful in the future. In my case, `fmtbench` seems like it might be handy for a number of further articles.
- If the string-manipulation solution ever broke in production, and *we didn't understand how it worked*, it could have taken us well over an hour to fix.
- Best of all, I think that `swapDirect` is probably as fast as you can get without writing platform-dependent assembly. This probably doesn't matter, but I , for one, get a warm fuzzy feeling from writing the *best possible solution*.

With luck, you've learned something about benchmarking, bytes, or UUIDs. And remember: if you're not sure where to start, try *just looking at it*.

MORE ARTICLES