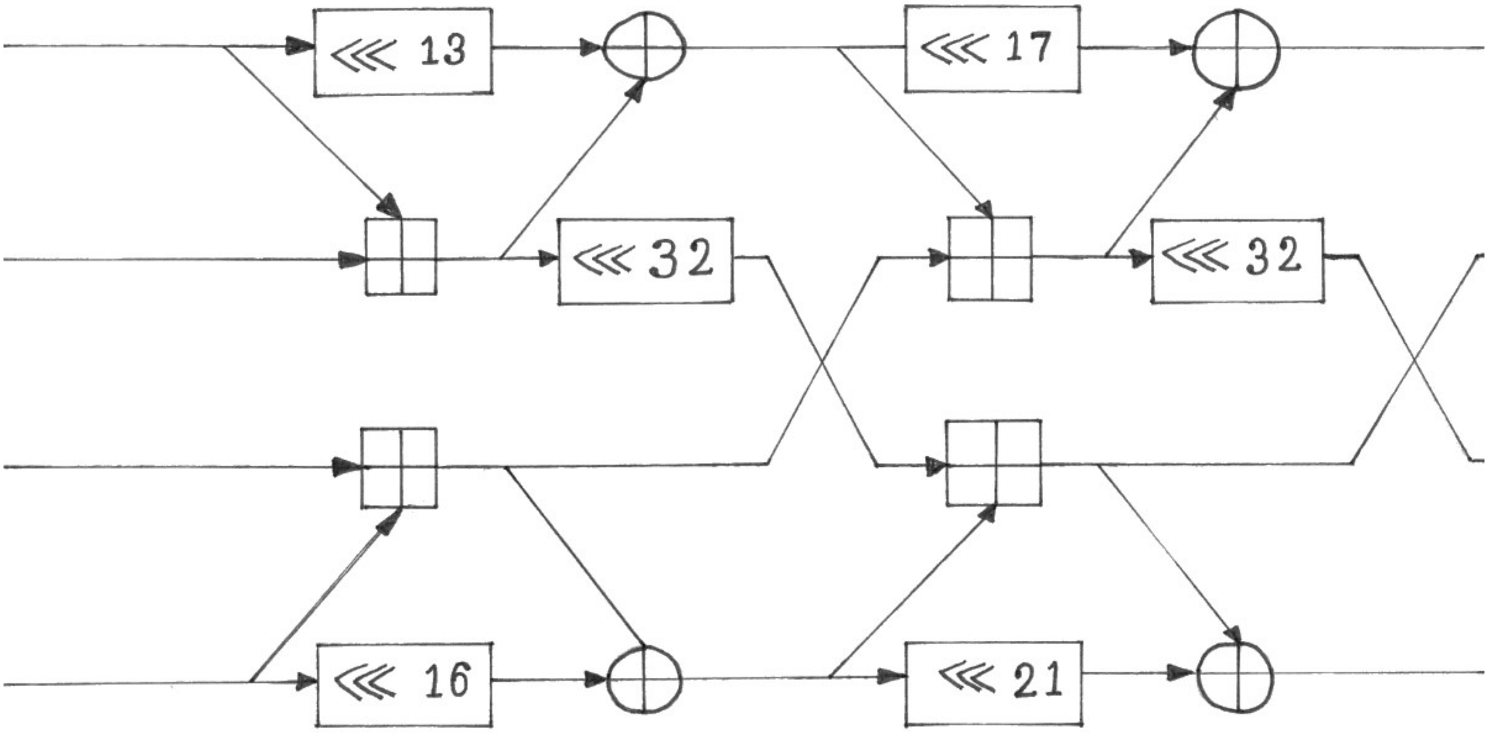


SipRound, the building block of SipHash. Illustration by Siobhán K Cronin

The moment when you realize every server in the world is vulnerable

Carl Tashian Aug 24, 2016 · 7 min read

Hash tables. Dictionaries. Associative arrays. Whatever you like to call them, they are *everywhere* in software. They are core. And when someone finds a vulnerability in such a low-level data structure, almost all software is implicated.

This is a story of one of those core vulnerabilities, and how it took a decade to uncover and resolve. The story is pretty amazing. But for context, let's review what hash tables

Hash Tables 101

Hash tables are incredibly convenient and fast. They let you put labels on things and throw them into memory buckets, and later on you can pull them back out and use them for whatever you want. They were invented in the 1950s and their underlying mechanics haven't changed much over the years.

Let's create a hash table and put some stuff in it:

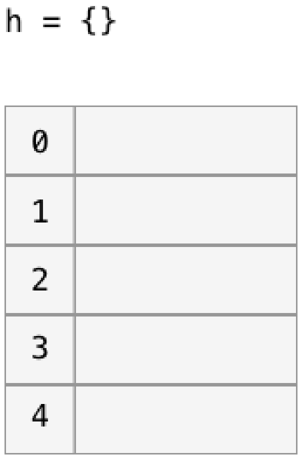
```
h = {}

h['a'] = 6
h['b'] = 3
h['f'] = 9

print h['a']

>>> 6
```

Each key and value will be stored in a bucket in memory. Lets say we start out with 5 empty buckets.

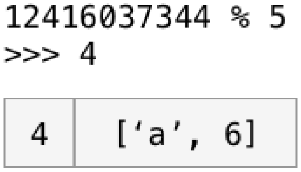


```
h = {}
```

0	
1	
2	
3	
4	

When we add the key 'a', which bucket should it go into? We want to be able to find it easily later. This is where the *hashing function* comes in. Every hash table is backed by a deterministic hashing function that turns any key into to a large, fixed-length number, which we call the hash. So the hash for 'a' might be 12416037344.

Because it's deterministic, if we run the hash function on 'a' again sometime later, we'll still get 12416037344. Now that we have the hash for our key, we need to reduce that hash into a bucket number (0–4 in our case). The simplest way to do this is to modulo by the number of buckets:

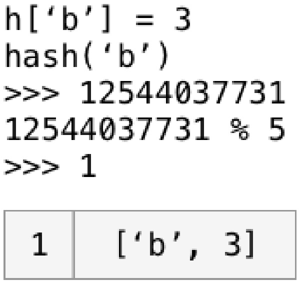


```
12416037344 % 5
>>> 4
```

4	[‘a’, 6]

Great. So ‘a’ goes into bucket #4.

Now, if we keep going and hash ‘b’, we get 12544037731. And (12544037731 % 5) is 1. So ‘b’ goes into bucket #1.



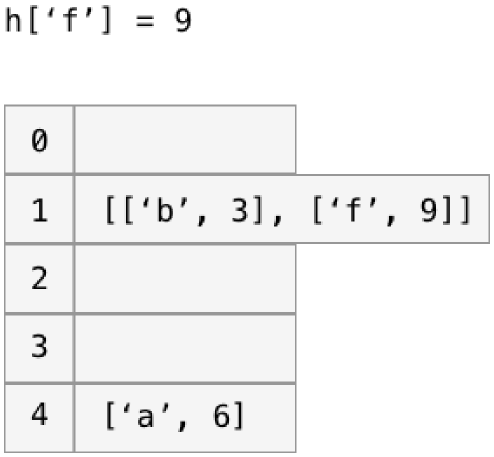
```
h[‘b’] = 3
hash(‘b’)
>>> 12544037731
12544037731 % 5
>>> 1
```

1	[‘b’, 3]

Now let’s add ‘f’ to the table.

Hashing ‘f’ yields 13056039271. And (13056039271 % 5) is 1. But we already have something in bucket #1! What now?

We have a collision. Hash table collisions happen pretty often. One of the simplest ways to resolve collisions is to set each bucket up as a list, and just keep adding to that list whenever a collision occurs. This is known as a chained hash table. Here’s what it might look like:



```
h[‘f’] = 9
```

0	
1	[[‘b’, 3], [‘f’, 9]]
2	
3	
4	[‘a’, 6]

Why not just add more buckets to the table? Well, eventually we will need to do that, but at that point the entire table has to be rehashed, so it should only be done occasionally. Adding to a list is much faster.

Usually.

Unfortunately, collisions open the door to the biggest weakness in hash tables. As soon as we have collisions, the time required for accessing an element starts gradually creeping up because we have to loop through the list within the bucket.

When hash tables were invented, the selection of a great hash function came down to two things:

- Performance. It must be fast as hell. Of course.
- Uniform density output. A great hash function should consistently, uniformly distribute arbitrary keys nicely across the hash table, because we want to avoid collisions as much as possible.

And that was it. Security was not in the picture. Some very simple, very fast general purpose hash functions were developed over the years, and they worked well for several decades.

A vulnerability is discovered

An international community of mathematicians and computer scientists is constantly hunting for the next vulnerability. We don't hear much about them outside of Internet security circles, but the people who find and fix these vulnerabilities are heroes and, as much as any military, they are keeping us safe.

In 2003, a paper by Scott A. Crosby and Dan S. Wallach of Rice University described such a vulnerability. They found a theoretical class of attacks called algorithmic complexity attacks.

The idea was simple: If we have an algorithm that is normally a superfast O(1) in time complexity, but that has an obscure, unlikely corner case where a huge $O(n^2)$ nested loop can happen, our algorithm might be vulnerable to attack. Specifically, if an attacker

The paper describes hash tables as a possible attack vector and it even reproduces the attack on several open source projects. But no one seemed to notice that this was a widespread problem affecting nearly all hash table implementations until 2011 when, at the 28th Chaos Communication Congress, Alexander Klink and Julian Wälde demoed an attack on a much broader set of programming languages and servers.

Attacking the complexity of the hash table algorithm is known as a hash-flooding attack, and Klink and Wälde showed that nearly every web application written in PHP, Ruby, Python, Java, ASP.NET, and v8 was implicated. The attackers used a common web application platform feature that converts HTTP POST parameters into a hash. As Klink and Wälde point out in the demo, this feature would be present even in the most simple “Hello World” PHP program.

```
<html>
  <body>
    <?php echo ‘<p>Hello World</p>’; ?>
  </body>
</html>
```

This was bad news. Because it attacked the PHP platform code, this attack could take down a server that was running even the most basic “Hello World” script!

The attack works by figuring out in advance which hash keys could trigger collisions. With the hash functions in use at the time, it was easy enough to invert the hash function and generate a ton of keys, then use them for an attack. For most web applications, the attack involved sending a large (1MB) HTTP POST request that would build a hash table on the server with 10,000 or even 100,000 collisions. And as more collisions happen, the insertion time for each additional element begins to rise to $O(n^2)$ and the CPU goes to 99% utilization.

The response

That was 2011. In response to Klink and Wälde’s findings, most languages switched to a hashing function that incorporates a secret randomized seed value that is reset every time a new program is run. This way, an attacker couldn’t just invert the same generic

And that worked.

For a year or so, anyway.

Then in 2012, at 29c3, this new method was exploited using a more complex collision attack that used differential cryptanalysis. Without going into too much detail, it turned out that the random seed value didn't really introduce quite enough of a difference into the output of the hash function, and it could again be attacked by using some specially crafted values.

The 29c3 talk, given by Daniel J. Bernstein, Jean-Philippe Aumasson, and Martin Boßlet, includes a demo of the attack.

All of the common general purpose hash functions were now broken.

To understand what happened next, we have to talk about cryptographic vs. non-cryptographic hash functions. A cryptographic hash function is a kind of hash function that is computationally very difficult to invert. Like a trap door, as compared to a regular door. This "one-way" nature is how it differs from general-purpose hash functions. It's commonly used to store encrypted passwords in such a way that cannot be read by an attacker (or anyone) but that can be tested against the input from a user who is trying to log in: Just encrypt whatever the user types using the one-way function, and compare it to the encrypted value we stored when we originally set the password. This is technically why the bank can't tell anyone their password — they don't know it!

The issue with cryptographic hash functions is that they tend to be very slow. Much slower than general-purpose functions. So programming languages traditionally avoided using them in performance-sensitive scenarios — like hash tables.

As part of their 29c3 talk, Bernstein and Aumasson introduced SipHash, a cryptographic hash function they developed that's a lot faster than previous algorithms. While still slower than non-cryptographic hashes, it's fast enough to back hash tables. It hits all the right tradeoffs between security, performance, and uniformity of output values. In fact, there are even different versions of SipHash that allow developers to find their own sweet spot between performance and security. (Because, after all, security is just a matter of degrees of difficulty of an attack. Nothing is truly secure.)

Not only did Bernstein and Aumasson find and exploit the vulnerability, they designed and built an elegant fix for it. And by 2013, many programming languages had adopted it as their primary hash function.

SipHash is safe. At least, so far. While no one has found a way to quickly generate lots of collisions with SipHash, it has also not been proven impossible to do so.

The next Chaos Communication Congress, 33c3, is coming up this winter and who knows? Maybe someone will announce a clever new exploit, and a safer or faster hash function. And the cycle continues.

If you made it this far, you should join my mailing list about technology and humanity.

Security Data Structures Programming Tech Software Development