Security with data structures

Make a Hash

What do all programs have in common? They store data at some point, usually in arrays – everything from command-line options to the input and output. But how is data actually stored by the program? Kurt explains. *By Kurt Seifried*

Many data types and structures exist – from simple strings and integers to multi-dimensional arrays. One of the most common and useful types is the simple growable array that can contain an arbitrary number of elements. You can basically put data in, look it up, and remove it as needed; this structure is also referred to as a one-dimensional array. Often, you can also insert an array as a data element into an existing array, giving you a multi-dimensional array that lets you do things like create a "customer" array with a number of elements containing a customer name, phone number, and so forth.

About Arrays

Arrays typically have three major operations: insertion (adding elements), lookups (reading a value), and removing elements. There are a number of ways to structure the data in memory, and various strategies can be used to optimize operations on the array. A simple linked list, for example, makes it easy to add an element to the beginning or end of an array but expensive to insert an item into the middle, because it must search the array for the appropriate location. In general, most programming languages that provide well-formed data array structures, such as Perl, Python, Ruby, and so forth, do a pretty good job.

The most common strategy is to create a key based on the data (essentially a hash value), the array is then ordered by these keys. A significant benefit here is that the keys will (usually) have a random and relatively uniform distribution (e.g., *01*, *23*, *45*, *67*, *89*). This approach is helpful when using things like binary trees, which benefit from having balanced key distribution. This is also referred to as a hash map.

Hash DoS

As is often true, there are ways for an attacker to abuse this setup. When the hash value is calculated, it is often done using a deterministic method (so that the same input will always result in the same hash value). If multiple inputs were processed that had the same hash value, most programming languages and applications would need several seconds and, in some cases, several minutes to insert all the values with the same hash value [1]. In this case, the system must examine not only the keys but also the actual values stored before it can insert a value. If there were several hundred or thousand overlapping keys, this would be equivalent to traversing a major portion of the array every time you tried to insert a new entry.

Simply put, you could send an HTTP POST request to a web server with 1,000 entries that would then take several minutes to process (basically bringing the web server to its knees). Two main strategies are available to deal with this problem. The first is to limit the number of inputs; this works well for things like HTTP POST requests, but it will not work for data arrays within a programming language.

The second method is to add some randomness to the hash functions, meaning they are still internally deterministic but, externally, are resistant to attacks. For example, if you add a random value between one and a million, your attacker would have to guess this value before finding a list of values that would then result in the same hash value. By changing the secret key every time an array is created, it basically becomes impossible for attackers to exploit this issue. Or does it?

Several applications use MurmurHash [2], which is pretty good and is in the public domain. But, it is not cryptographically secure. In short, it works most of the time until someone actively attacks it, which is exactly what Jean-Philippe Aumasson and Martin Bosslet demonstrated at the 2012 Chaos Computer Conference [3].

Another example of a weak hash is Btrfs. It uses CRC32 by default, for

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

which it is utterly trivial to find values that will cause collisions. Using pre-computed folder values, it is reported that creating several thousand folders (an operation that normally takes a tenth of a second or so) will take several seconds before timing out if you use malicious values [4].

SipHash

So, you need a hash that is cryptographically secure, which is basically a fancy way of saying that it will be highly resistant to attacks. Additionally, you'll want speed, and you'll want to be able to hash short values like *a*, *aaaa*, and *aaab* in a way that prevents highly similar inputs from having similar outputs. The output also needs to be relatively short so that you can use it in applications, such as memory arrays, or embed it within network packets. Finally, the hash needs to support the use of a secret key or value to permute the hash values so that attackers cannot pre-compute hash values to use in attacks. You also want it to be simple to implement.

SipHash [5], for example, was released on June 20, 2012, and eight days later it had 18 implementations, including Ruby, C, C#, Java, JavaScript, PHP; now, there is virtually complete coverage for all major languages.

Alternatives – Bloom Filters

As a rule, arrays are the data structure used when you have to store data for later lookups, but depending on your exact use case, you might have other options. If, for example, you don't actually need to look up the data but just need to know whether it exists, you might be able to use a Bloom filter. Through the magic of hashing and math, you can create a memory-efficient data structure that will store a set of data. The space you save comes at the cost of having possible false positives; however, the Bloom filter will never report something as being in the set when it is not. Depending on your tolerance, you can allow a higher number of false positives, which would allow you to save even more memory [6].

Alternatives – Binary Trees

Depending on your usage patterns, another option could be some form of binary tree. Typically, binary trees are fast, especially in memory, but they can be slow when loaded from the filesystem. A number of binary tree systems use various tricks like balancing the trees to order data efficiently only when the data is flushed to disk. In memory, having the data out of order won't matter much, but on disk, this will introduce significantly more delays. Red-black trees can, for example, offer worst-case guarantees, so although they might be slow, they will never become unusably slow or non-responsive – a critical feature for many operations.

Expiring Data

One of the most efficient ways to store data is to get rid of the data you don't need. Of course, you have to know which data you no longer need, and that can be tricky sometimes. Adding a timestamp field to the array entry for when the data was created or last used can allow you to search for and delete data that is old and unneeded.

Conclusion

The reality is that we live in a connected world and the main value of many programs is their ability to share, process, and distribute data. However, many people out there love to exploit and abuse these programs. So, it seems clear at this point that virtually all hash operations need to use cryptographically secure options for everything from secure encryption programs to the arrays that hold phone numbers. Better safe than sorry. ■■■

INFO

[1] Hash DoS: *http://www.ocert.org/advisories/ocert-2011-003.html*

[2] Murmur Hash: *https://sites.google.com/site/murmurhash/*

[3] Hash-Flooding DoS Reloaded: Attacks and Defenses: *http://events.ccc.de/congress/2012/Fahrplan/events/5152.en.html*

[4] Security Problem Discovered in Btrfs File-System: *http://www.phoronix.com/scan.php?page=news_item&px=MTI1MjU*

[5] SipHash: *https://131002.net/siphash/*

[6] Why Bloom filters work the way they do: *http://www.michaelnielsen.org/ddi/why-bloom-filters-work-the-way-they-do/*