SipHash - a short input PRF

Author: Written by Jason A. Donenfeld <jason@zx2c4.com>

SipHash is a cryptographically secure PRF -- a keyed hash function -- that performs very well for short inputs, hence the name. It was designed by cryptographers Daniel J. Bernstein and Jean-Philippe Aumasson. It is intended as a replacement for some uses of: *jhash*, *md5_transform*, *sha1_transform*, and so forth.

SipHash takes a secret key filled with randomly generated numbers and either an input buffer or several input integers. It spits out an integer that is indistinguishable from random. You may then use that integer as part of secure sequence numbers, secure cookies, or mask it off for use in a hash table.

Generating a key

Keys should always be generated from a cryptographically secure source of random numbers, either using get_random_bytes or get_random_once:

```
siphash_key_t key;
get_random_bytes(&key, sizeof(key));
```

If you're not deriving your key from here, you're doing it wrong.

Using the functions

There are two variants of the function, one that takes a list of integers, and one that takes a buffer:

```
u64 siphash(const void *data, size_t len, const siphash_key_t *key);
```

And:

```
u64 siphash_1u64(u64, const siphash_key_t *key);
u64 siphash_2u64(u64, u64, const siphash_key_t *key);
u64 siphash_3u64(u64, u64, u64, const siphash_key_t *key);
u64 siphash_4u64(u64, u64, u64, u64, const siphash_key_t *key);
u64 siphash_1u32(u32, const siphash_key_t *key);
u64 siphash_2u32(u32, u32, const siphash_key_t *key);
u64 siphash_3u32(u32, u32, u32, const siphash_key_t *key);
u64 siphash_4u32(u32, u32, u32, u32, const siphash_key_t *key);
```

If you pass the generic siphash function something of a constant length, it will constant fold at compile-time and automatically choose one of the optimized functions.

Hashtable key function usage:

```
struct some_hashtable {
        DECLARE_HASHTABLE(hashtable, 8);
        siphash_key_t key;
};

void init_hashtable(struct some_hashtable *table)
{
        get_random_bytes(&table->key, sizeof(table->key));
}

static inline hlist_head *some_hashtable_bucket(struct some_hashtable *table, struct interesting_input *input
{
        return &table->hashtable[siphash(input, sizeof(*input), &table->key) & (HASH_SIZE(table->hashtable) -
}
```

You may then iterate like usual over the returned hash bucket.

Security

SipHash has a very high security margin, with its 128-bit key. So long as the key is kept secret, it is impossible for an attacker to guess the outputs of the function, even if being able to observe many outputs, since 2^128 outputs is significant.

Linux implements the "2-4" variant of SipHash.

Struct-passing Pitfalls

Often times the XuY functions will not be large enough, and instead you'll want to pass a pre-filled struct to siphash. When doing this, it's important to always ensure the struct has no padding holes. The easiest way to do this is to simply arrange the members of the struct in descending order of size, and to use offsetendof() instead of sizeof() for getting the size. For performance reasons, if possible, it's probably a good thing to align the struct to the right boundary. Here's an example:

```
const struct {
        struct in6_addr saddr;
        u32 counter;
        u16 dport;
} __aligned(SIPHASH_ALIGNMENT) combined = {
        .saddr = *(struct in6_addr *)saddr,
        .counter = counter,
        .dport = dport
};
u64 h = siphash(&combined, offsetofend(typeof(combined), dport), &secret);
```

Resources

Read the SipHash paper if you're interested in learning more: https://131002.net/siphash/siphash.pdf

---

HalfSipHash - SipHash's insecure younger cousin

Author: Written by Jason A. Donenfeld <jason@zx2c4.com>

On the off-chance that SipHash is not fast enough for your needs, you might be able to justify using HalfSipHash, a terrifying but potentially useful possibility. HalfSipHash cuts SipHash's rounds down from "2-4" to "1-3" and, even scarier, uses an easily brute-forcable 64-bit key (with a 32-bit output) instead of SipHash's 128-bit key. However, this may appeal to some high-performance *jhash* users.

Danger!

Do not ever use HalfSipHash except for as a hashtable key function, and only then when you can be absolutely certain that the outputs will never be transmitted out of the kernel. This is only remotely useful over *jhash* as a means of mitigating hashtable flooding denial of service attacks.

Generating a HalfSipHash key

Keys should always be generated from a cryptographically secure source of random numbers, either using get_random_bytes or get_random_once:

hsiphash_key_t key; get_random_bytes(&key, sizeof(key));

If you're not deriving your key from here, you're doing it wrong.

Using the HalfSipHash functions

There are two variants of the function, one that takes a list of integers, and one that takes a buffer:

```
u32 hsiphash(const void *data, size_t len, const hsiphash_key_t *key);
```

And:

```
u32 hsiphash_1u32(u32, const hsiphash_key_t *key);
u32 hsiphash_2u32(u32, u32, const hsiphash_key_t *key);
u32 hsiphash_3u32(u32, u32, u32, const hsiphash_key_t *key);
u32 hsiphash_4u32(u32, u32, u32, u32, const hsiphash_key_t *key);
```

If you pass the generic hsiphash function something of a constant length, it will constant fold at compile-time and automatically choose one of the optimized functions.

Hashtable key function usage

```
struct some_hashtable {
        DECLARE_HASHTABLE(hashtable, 8);
        hsiphash_key_t key;
};

void init_hashtable(struct some_hashtable *table)
{
        get_random_bytes(&table->key, sizeof(table->key));
}

static inline hlist_head *some_hashtable_bucket(struct some_hashtable *table, struct interesting_input *input
{
        return &table->hashtable[hsiphash(input, sizeof(*input), &table->key) & (HASH_SIZE(table->hashtable)
}
```

You may then iterate like usual over the returned hash bucket.

Performance

HalfSipHash is roughly 3 times slower than JenkinsHash. For many replacements, this will not be a problem, as the hashtable lookup isn't the bottleneck. And in general, this is probably a good sacrifice to make for the security and DoS resistance of HalfSipHash.