Hash-flooding DoS reloaded: attacks and defenses

Jean-Philippe Aumasson, Kudelski Group
Daniel J. Bernstein, University of Illinois at Chicago
Martin Boßlet, freelancer
Hash-flooding DoS reloaded: attacks and defenses

Jean-Philippe Aumasson, Kudelski Group
Daniel J. Bernstein, University of Illinois at Chicago
Martin Boßlet, freelancer
Jean-Philippe
Cryptography expert at the Kudelski Group
Applied crypto researcher
https://131002.net  @aumasson

Martin
Independent SW engineer and security expert
Ruby core dev team member
http://www.martinbosslet.de  @_emboss_
Hash-flooding DoS reloaded: attacks and defenses
Denial-of-Service (DoS) attacks

“Attempt to make a machine or network resource unavailable to its intended users.” — Wikipedia
Popular DoS techniques are distributed HTTP or TCP SYN flood... (DDoS)
More subtle techniques exploit properties of TCP-congestion-avoidance algorithms...
Hash-flooding DoS reloaded: attacks and defenses

```c
nextpos = prevpos ^ get4(pos);
prevpos = pos;
pos = nextpos;
if (++loop > 100) return 0; /* to protect against hash flooding */
}

return 0;
```
Hash tables used in many applications to maintain an association between objects

Example: Python dictionaries

d={}
# empty table

d[12345]=0xc
# insertion

d[‘astring’]=‘foo’
# insertion

d[‘a’, ‘tuple’]=0
# insertion

print d[‘a string’]
# lookup
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes \(O(n)\) operations on average.
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes O(n) operations on average.

\[
\begin{array}{ccc}
0 & 1 & 2 \\
\end{array}
\]

12345: 0xc

d[12345]=0xc, hash(12345)=1
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes $O(n)$ operations on average.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'astring': 'foo'</td>
<td>12345: 0xc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`d['astring'] = 'foo', hash('astring') = 0`
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes \(O(n)\) operations on average.

\[
\begin{array}{ccc}
0 & 1 & 2 \\
\text{‘astring’: ‘foo’} & 12345: 0xc & (‘a’,’tuple’): 0 \\
\end{array}
\]

d[('a','tuple')]=0; hash(('a','tuple'))=2
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes $O(n^2)$ operations in the worst case.

$d[12345]=0xc, \ hash(12345)=1$
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes \(O(n^2)\) operations in the worst case.

d[‘astring’] = ‘foo’, hash(‘astring’) = 0
If the table is about as large as the number of elements to be stored (=n), insertion or lookup of n elements takes $O(n^2)$ operations in the worst case.

d[('a', 'tuple')] = 0; hash(('a', 'tuple')) = 2
Hash flooding:
Send to a server many inputs with a same hash (a *multicollision*) so as to enforce worst-case insert time
send **2MB of POST** data consisting of
200.000 colliding 10B strings

\[ \approx 40.000.000.000.000 \text{ string comparisons} \]
(at least **10s** on a 2GHz machine...)
Previous work


-> attack formalized and applied to Perl, Squid, etc.

Klink, Wälde. *Efficient Denial of Service Attacks on Web Application Platforms*. CCC 28c3

-> application to PHP, Java, Python, Ruby, etc.
Previous work

-> attack formalized and applied to Perl, Squid, etc.

Klink, Wälde. Efficient Denial of Service Attacks on Web Application Platforms. CCC 28c3
-> application to PHP, Java, Python, Ruby, etc.
n.runs AG
http://www.nruns.com/ security(at)nruns.com
n.runs-SA-2011.004 28-Dec-2011

  Oracle, http://www.oracle.com
  Microsoft, http://www.microsoft.com
  Python, http://www.python.org
  Ruby, http://www.ruby.org
  Google, http://www.google.com

Affected Products: PHP 4 and 5
Java
Apache Tomcat
Apache Geronimo
Jetty
Oracle Glassfish
ASP.NET
Python
Plone
CRuby 1.8, JRuby, Rubinius
v8

Vulnerability: Denial of Service through hash table multi-collisions

Tracking IDs: oCERT-2011-003
CERT VU#903934
Patches released consisting of a stronger hash with randomization (to make colliding values impossible to find)
MurmurHash2

“used in code by Google, Microsoft, Yahoo, and many others”

http://code.google.com/p/smhasher/wiki/MurmurHash

CRuby, JRuby
MurmurHash3

“successor to MurmurHash2”

Oracle’s Java SE, Rubinius
Hash-flooding DoS reloaded: attacks and defenses
1. Theory
for (i=0; i<nblocks; i++) {
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;
    k1 = ROTL32(k1, 15);
    k1 *= 0x1b873593;

    h1 ^= k1;
    h1 = ROTL32(h1, 13);
    h1 = h1 * 5 + 0xe6546b64;
}
Differential cryptanalysis strategy

1/ introduce a difference in the state $h_1$ via the input $k_1$

2/ cancel this difference with a second well chosen difference

```c
for (i=0; i<nblocks; i++) {
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;
    k1 = ROTL32(k1, 15);
    k1 *= 0x1b873593;
    h1 ^= k1;
    h1 = ROTL32(h1, 13);
    h1 = h1 * 5 + 0xe6546b64;
}
```
Differential cryptanalysis strategy

1/ introduce a difference in the state $h_1$ via the input $k_1$
2/ cancel this difference with a second well chosen difference

for (i=0;i<nblocks;i++) {  
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;  
    inject difference $D_1$
    k1 = ROTL32(k1,15);
    k1 *= 0x1b873593;  
    diff in $k_1$:0x00040000

    $h_1 ^= k1$;
    $h_1 = ROTL32 ( h_1 ,13)$;
    $h_1 = h_1 *5+0$ xe6546b64;}


for (i=0;i<nblocks;i++) {
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;
    k1 = ROTL32(k1,15);
    k1 *= 0x1b873593;
    h1 ^= k1;
    h1 = ROTL32(h1,13);
    h1 = h1 * 5 + 0xe6546b64;
}

Differential cryptanalysis strategy

1/ introduce a difference in the state h1 via the input k1
2/ cancel this difference with a second well chosen difference
Differential cryptanalysis strategy

1/ introduce a difference in the state \( h_1 \) via the input \( k_1 \)
2/ cancel this difference with a second well chosen difference

```c
for (i=0;i<nblocks;i++) {
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;  // inject difference \( D_1 \)
    k1 = ROTL32(k1, 15);
    k1 *= 0x1b873593;  // diff in \( k_1:0x00040000 \)
    h1 ^= k1;          // diff in \( h_1 \) \( 0x00040000 \)
    h1 = ROTL32(h1, 13);  // \( 0x80000000 \)
    h1 = h1 * 5 + 0xe6546b64;  // \( 0x80000000 \)
}```
Differential cryptanalysis strategy

1/ introduce a difference in the state $h_1$ via the input $k_1$
2/ cancel this difference with a second well chosen difference

for (i=0; i<nblocks; i++) {
    uint32_t $k_1$ = getblock(blocks, i);
    $k_1$ *= 0xcc9e2d51;  // inject difference D2
    $k_1$ = ROTL32($k_1$, 15);
    $k_1$ *= 0x1b873593;
    $h_1$ ^= $k_1$;
    $h_1$ = ROTL32($h_1$, 13);
    $h_1$ = $h_1$ *5+0 x6546b64;}

Differential cryptanalysis strategy

1/ introduce a difference in the state $h_1$ via the input $k_1$
2/ cancel this difference with a second well chosen difference

for (i=0; i<nblocks; i++) {
    uint32_t $k_1$ = getblock(blocks, i);
    $k_1$ *= 0xcc9e2d51;  // inject difference $D_2$
    $k_1$ = ROTL32($k_1$, 15);
    $k_1$ *= 0x1b873593;  // diff in $k_1$: 0x80000000
    $h_1$ ^= $k_1$;
    $h_1$ = ROTL32($h_1$, 13);
    $h_1$ = $h_1$ *5+0xe6546b64;
}
Differential cryptanalysis strategy

1/ introduce a difference in the state $h_1$ via the input $k_1$
2/ cancel this difference with a second well chosen difference

for (i=0;i<nblocks;i++) {
    i=1
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;  \text{inject difference D2}
    k1 = ROTL32(k1 ,15);
    k1 *= 0x1b873593;  \text{diff in $k_1$: 0x80000000}
    \text{diff in $h_1$: 0x80000000} \oplus \text{0x80000000} = 0
    h1 ^= k1;
    h1 = ROTL32 ( h1 ,13);  \text{COLLISION!}
    h1 = h1 *5+0 xe6546b64;}

2 colliding 8-byte inputs
Chain collisions $\Rightarrow$ multicollisions

$8n$ bytes $\Rightarrow 2^n$ colliding inputs
A multicollision works for any seed

=> “Universal” multicollisions

h1=seed;
for (i=0;i<nblocks;i++) {
    uint32_t k1 = getblock(blocks, i);
    k1 *= 0xcc9e2d51;
    k1 = ROTL32(k1,15);
    k1 *= 0x1b873593;
    // transform of k1 independent of the seed!
    h1 ^= k1;
    h1 = ROTL32 (h1,13);
    h1 = h1 *5+0 xe6546b64;}

Even simpler for **MurmurHash2**
Consequence:
Systems using MurmurHash2/3 remain vulnerable to hash-flooding
Other hash attacked
CityHash provides hash functions for strings. The latest stable version is cityhash-1.1.0.tar.gz. Differences between versions are explained in the NEWS file.

The functions mix the input bits thoroughly but are not suitable for cryptography. We provide reference implementations in C++, with a friendly MIT license. The code's portable; let us know if you encounter problems. To download the code use the .tar.gz file or use svn with these instructions.

The README contains a good explanation of the various CityHash functions. However, here is a short summary:

CityHash64() and similar return a 64-bit hash. Inside Google, where CityHash was developed starting in 2010, we use variants of CityHash64() mainly in hash tables such as hash_map<string, int>.

CityHash32() returns a 32-bit hash. It's mostly useful in 32-bit code (e.g., x86).

CityHash128() and similar return a 128-bit hash and are tuned for strings of at least a few hundred bytes. Depending on your compiler and hardware, it may be faster than CityHash64() on sufficiently long strings. It is known to be slower than necessary on shorter strings, but we expect that case to be relatively unimportant. Inside Google we use variants of CityHash128() mainly for code that wants to minimize collisions.

Even weaker than MurmurHash2...

Also vulnerable to hash flooding
CityHash64( BU9[85WWp/ HASH!, 16 ) = b82e7612e6933d2f
CityHash64( 8{YDLn;d.2 HASH!, 16 ) = b82e7612e6933d2f
CityHash64( d+nK&t?yr HASH!, 16 ) = b82e7612e6933d2f
CityHash64( {A.#v5i]V{ HASH!, 16 ) = b82e7612e6933d2f
CityHash64( FBC=/\hJeA!HASH!, 16 ) = b82e7612e6933d2f
CityHash64( $03$=K1.-H!HASH!, 16 ) = b82e7612e6933d2f
CityHash64( 3o'L'Piw\\!HASH!, 16 ) = b82e7612e6933d2f
CityHash64( duDu%qaUS@"HASH!, 16 ) = b82e7612e6933d2f
CityHash64( IZVo|0S=BX"HASH!, 16 ) = b82e7612e6933d2f
CityHash64( X2V|P=<u,=#HASH!, 16 ) = b82e7612e6933d2f
CityHash64( 9<%45yG]qG#HASH!, 16 ) = b82e7612e6933d2f
CityHash64( 6?4O:'<Vho#HASH!, 16 ) = b82e7612e6933d2f
CityHash64( 2u 2}7g^>3$HASH!, 16 ) = b82e7612e6933d2f
CityHash64( kqwnZH=cKG$HASH!, 16 ) = b82e7612e6933d2f
CityHash64( Nl+:rtvw}K$HASH!, 16 ) = b82e7612e6933d2f
CityHash64( s/pI!<5u*]$HASH!, 16 ) = b82e7612e6933d2f
CityHash64( f|P~n*<xPc$HASH!, 16 ) = b82e7612e6933d2f
CityHash64( Cj7TCG|G}}$HASH!, 16 ) = b82e7612e6933d2f
CityHash64( a4$>Jf3PF'%HASH!, 16 ) = b82e7612e6933d2f
2. Practice
Breaking **Murmur**:  

We’ve got the recipe –  

Now all we need is the (hash) **cake**
Where are hashes used?
Internally vs. Externally
Parser symbol tables
Method lookup tables
Attributes / Instance variables
   IP Addresses
Transaction IDs
   Database Indexing
   Session IDs
   HTTP Headers
   JSON Representation
URL-encoded POST form data
   Deduplication (HashSet)
   A* search algorithm
   Dictionaries
   ...
=> Where aren’t they used?
Can’t we use something different?
We could,
but amortized **constant time** is just too **sexy**
Possible real-life attacks
Attack *internal* use?
Elegant, but low impact
Need a high-profile target
Web Application
Example #1

Rails
First:
Attacking MurmurHash in Ruby
Straight-forward with a few quirks
Apply the recipe
Demo
Should work with Rails
out of the box, no?
Unfortunately, no
Demo
def POST
    ...
    @env["rack.request.form_hash"] = parse_query(form_vars)
    ...
end
def parse_query(qs)
    Utils.parse_nested_query(qs)
end
def parse_nested_query(qs, d = nil)

    params = KeySpaceConstrainedParams.new

    (qs || '').split(d ? /[#{d}] */n : DEFAULT_SEP).each do |p|

        k, v = p.split('=', 2).map { |s| unescape(s) }
        normalize_params(params, k, v)

    end

    return params.to_params_hash
end
def unescape(s, encoding = Encoding::UTF_8)
    URI.decode_www_form_component(s, encoding)
end
def self.decode_www_form_component(str, enc=Encoding::UTF_8)
    raise ArgumentError, "invalid %-encoding (#{str})"
    unless /\A[^%]*(?:%\h\h[^%]*)*\z/ =~ str

    str.gsub(/\+|%\h\h/, TBLDECWWWCOMP_).force_encoding(enc)

end
\A[^%]*(?:%\h\h[^%]*)\z/
Catches invalid % encodings
(e.g. %ZV, %%1 instead of %2F)
def parse_nested_query(qs, d = nil)

    params = KeySpaceConstrainedParams.new

    (qs || '').split(d ? /[#{d}] */n : DEFAULT_SEP).each do |p|

        k, v = p.split('=', 2).map { |s| unescape(s) }
        normalize_params(params, k, v)

    end

    return params.to_params_hash
end
def normalize_params(params, name, v = nil)

    name =~ %r(\A[^\[\]]*([^
]+)\]*

    k = $1 || ''

    ...
end
%r(`A\[\]\]*(`[^\[\]]+\]`\]*)

???
helps transform [[]] to []
idea:

pre-generate matching values
create **random values**

passing the regular expressions
that should do it, right?
Demo
CONFIDENCE: The feeling you experience before you fully understand the situation.
def parse_nested_query(qs, d = nil)

    params = KeySpaceConstrainedParams.new

    (qs || '').split(d ? /[\#{d}] */n : DEFAULT_SEP).each do |p|

        k, v = p.split('=', 2).map { |s| unescape(s) }
        normalize_params(params, k, v)

    end

    return params.to_params_hash
end
class KeySpaceConstrainedParams

def []=(key, value)

    @size += key.size if key && !@params.key?(key)

    raise RangeError, 'exceeded available parameter key space'
        if @size > @limit

    @params[key] = value

end

def
I am a clever little bastard
What now? Rails is safe?
Remember:

Hashes are used *everywhere*
So if `application/x-www-form-urlencoded` doesn't work, how about `application/json`?
Again, with the encoding...
Fast-forward...
Demo
Conclusion

Patchwork is not helping
too many places
code bloat
yet another loophole will be found
Fix it at the root
Example #2

Java
String(byte[] bytes)
public String(byte bytes[], int offset, int length, Charset charset) {
  ...

  char[] v = StringCoding.decode(charset, bytes, offset, length);
  ...

}
Tough nut to crack
What now? Java is safe?
String(char[] value)
public String(char value[]) {

    int size = value.length;
    this.offset = 0;
    this.count = size;
    this.value = Arrays.copyOf(value, size);
}

No decoding!
Substitute `byte[]` operations with equivalent operations on `char[]`
Demo
Disclosure

Oracle (Java): Sep 11
CRuby, JRuby, Rubinius: Aug 30
Hash-flooding DoS reloaded: attacks and defenses
SipHash: a fast short-input PRF

New crypto algorithm to fix hash-flooding:
• Rigorous security requirements and analysis
• Speed competitive with that of weak hashes

Peer-reviewed research paper (A., Bernstein). published at DIAC 2012, INDOCRYPT 2012
SipHash initialization

256-bit state $v_0 \ v_1 \ v_2 \ v_3$

128-bit key $k_0 \ k_1$

$v_0 = k_0 \oplus \text{736f6d6570736575}$
$v_1 = k_1 \oplus \text{646f72616e646f6d}$
$v_2 = k_0 \oplus \text{6c7967656e657261}$
$v_3 = k_1 \oplus \text{7465646279746573}$
SipHash initialization

256-bit state $v_0$ $v_1$ $v_2$ $v_3$
128-bit key $k_0$ $k_1$

$v_0 = k_0 \oplus \text{"somepseu"}$
$v_1 = k_1 \oplus \text{"dorandom"}$
$v_2 = k_0 \oplus \text{"lygenera"}$
$v_3 = k_1 \oplus \text{"tedbytes"}$
SipHash compression

Message parsed as 64-bit words $m_0, m_1, \ldots$

$v_3 \oplus= m_0$

$c$ iterations of SipRound

$v_0 \oplus= m_0$
SipHash compression

Message parsed as 64-bit words $m_0, m_1, \ldots$

$v_3 \oplus = m_1$

$c$ iterations of SipRound

$v_0 \oplus = m_1$
SipHash compression

Message parsed as 64-bit words $m_0, m_1, \ldots$

$$v_3 \oplus = m_2$$

$c$ iterations of SipRound

$$v_0 \oplus = m_2$$
Message parsed as 64-bit words \( m_0, m_1, \ldots \).
SipRound
SipHash finalization

\[ v2 \oplus = 255 \]

d iterations of SipRound

Return \[ v0 \oplus v1 \oplus v2 \oplus v3 \]
SipHash-2-4 hashing 15 bytes
Family SipHash-c-d

Fast proposal: SipHash-2-4

Conservative proposal: SipHash-4-8

Weaker versions for cryptanalysis: SipHash-1-0, SipHash-2-0, etc. SipHash-1-1, SipHash-2-1, etc. Etc.
Proof of simplicity

June 20: paper published online
June 28: 18 third-party implementations

C (Floodyberry, Boßlet, Neves); C# (Haynes)
Cryptol (Lazar); Erlang, Javascript, PHP (Denis)
Go (Chestnykh); Haskell (Hanquez)
Java, Ruby (Boßlet); Lisp (Brown); Perl6 (Julin)
Who is using SipHash?

OpenDNS

http://www.opendns.com/

Soon?

http://www.rust-lang.org/
Take home message

• DoS is doable with **only small data/bandwidth**

• Java- and Ruby-based web applications vulnerable to DoS (and maybe others...)

• SipHash offers both **security and performance**

*Contact us if you need to check your application*
Hash-flooding DoS reloaded: attacks and defenses

THANK YOU!