

# Hash Collision Complexity Attacks

Perl vs. Other Languages

- David Larochelle  
10 January, 2012  
Boston Perl Mongers

# About Myself

- David Larochelle
- Current Perl Developer
  - [Mediacloud.org](http://mediacloud.org)
- Also worked in numerous other languages

# About Myself 2

- Former security weenie
  - Splint
  - OunceLabs
- On a cleanse diet and craving junk food

# Credits

- Crosby & Wallah Denial of Service via Algorithmic Complexity Attacks USENIX'03  
<http://www.cs.rice.edu/~scrosby/hash/CrosbyWallah>
- n.runs-SA-2011.004 28-Dec-2011  
[http://www.nruns.com/\\_downloads/advisory281220](http://www.nruns.com/_downloads/advisory281220)

# Hashes

- Usually implemented as an array
  - Insert position is  $\text{Hash}(\text{key}) \% \text{num\_buckets}$
  - Store a linked list if there are collisions

# Why we love hashes

- Convenient / Programmer Friendly
- Expected insert time is  $O(1)$

# Hashes: The Dark Side

- Worse case insertion time is  $O(n)$
- Worse case  $O(n^2)$  to insert  $n$  elements
- However, for random data this is exponentially unlikely:
  - $(1/b)^n$
- But what if the data isn't random? Can we cause the worst case behavior?

# Finding Collisions

- Duplicate values of  $\text{Hash}(\text{key}) \% \text{num\_buckets}$
- If number of buckets is small and known
  - Brute force approach can pre-compute resulting bucket
  - i.e. hash random data until we've found enough collisions
  - Could send attack data for multiple bucket sizes
  - Internal resizing of the hash might also trigger the worst case
  - Still hard to do in practice – bucket size not usually knowable

# Attack the Hash function

- If we can generate collisions for  $\text{hash}(\text{key})$  then we don't care about the number of buckets
  - e.g. find  $k_1, k_2, k_3, \text{etc}$  such that
    - $\text{hash}(k_1) = \text{hash}(k_2) = \text{hash}(k_3)$

# Attacks

- Hash functions designed for speed rather than security
- Techniques:
  - Equivalent substrings
  - Meet-in-the-middle attack
  - Reduce difficult from  $1/2^{32}$  to  $1/2^{16}$

# Sending an attack payload

- Most web frameworks store request parameters in a hash
- Requests stored in the hash before being processed
- Post requests can be very large – thousands of parameters
- XSS another site could also send the request

# Mitigations

- Web request timeouts
- Maximum request sizes
- However, the attacker can just send more requests

# How Effective are These Attacks?

Bandwidth to DoS 1 CPU Core

- Python (Plone 32 bit) 20 kbit/s
- Ruby – (CRuby 1.8 or JRuby) - 850 bit/s
- Java (Tomcat) – 6 kbit/s
- ASP.NET – 30 kbit/s
- V8 – depends on framework
- PHP - 70-100 kbit/s



# Cores DoSable with Gbit Connection

- Python (Plone 32 bit) – 50,000
- Ruby – (CRuby 1.8 or JRuby) - 1,000,000
- Java (Tomcat) – 100,000
- ASP.NET – 30,000
- V8 – depends on framework
- PHP – 10,000

Source:

[http://www.nruns.com/\\_downloads/advisory281220](http://www.nruns.com/_downloads/advisory281220)

# What about Perl?

- Fixed in Perl 5.8.1
- Random perturbation
- As of 5.8.2 Perl detects pathological data switches on randomness for individual hashes
- PERL\_HASH\_SEED variable
  - Disable random hashes
  - Consistent ordering of hashes between runs
  - Allows the DoS to work on Perl

# Caveats

- 64 bit Python – Uses 64 bit hash keys and may not be vulnerable in practice
- CRuby 1.9 also uses random hashes

# Why were Other Languages Vulnerable?

- Basic issue known since 2003
- Ruby had fixed in 1.9 but didn't back port until exploit

# Conclusion

- Thoughts?
- Questions?