n.runs AG
http://www.nruns.com/ security(at)nruns.com
n.runs-SA-2011.004 28-Dec-2011

Vendors: PHP, http://www.php.net
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

Vendor communication:
2011/11/01 Coordinated notification to PHP, Oracle, Python, Ruby, Google
via oCERT
2011/11/29 Coordinated notification to Microsoft via CERT

Various communication with the vendors for clarifications, distribution of PoC code, discussion of fixes, etc.

Overview:

Hash tables are a commonly used data structure in most programming languages. Web application servers or platforms commonly parse attacker-controlled POST form data into hash tables automatically, so that they can be accessed by application developers.

If the language does not provide a randomized hash function or the application server does not recognize attacks using multi-collisions, an attacker can degenerate the hash table by sending lots of colliding keys. The algorithmic complexity of inserting n elements into the table then goes to O(n**2), making it possible to exhaust hours of CPU time using a single HTTP request.

This issue has been known since at least 2003 and has influenced Perl and CRuby 1.9 to change their hash functions to include randomization.

We show that PHP 5, Java, ASP.NET as well as v8 are fully vulnerable to this issue and PHP 4, Python and Ruby are partially vulnerable, depending on version or whether the server running the code is a 32 bit or 64 bit machine.

Description:

= Theory =

Most hash functions used in hash table implementations can be broken faster than by using brute-force techniques (which is feasible for hash functions with 32 bit output, but very expensive for 64 bit functions) by using one of two “tricks”: equivalent substrings or a meet-in-the-middle attack.

== Equivalent substrings ==

Some hash functions have the property that if two strings collide, e.g. hash('string1') = hash('string2'), then hashes having this substring at the same position collide as well, e.g. hash('prefixstring1postfix') = hash('prefixstring2postfix'). If for example 'Ez' and 'FY' collide under a hash function with this property, then 'EzEz', 'EzFY', 'FYEz', 'FYFY' collide as well. An observing reader may notice that this is very similar to binary counting from zero to four. Using this knowledge, an attacker can construct arbitrary numbers of collisions (2^n for 2*n-sized strings in this example).

== Meet-in-the-middle attack ==

If equivalent substrings are not present in a given hash function, then brute-force seems to be the only solution. The obvious way to best use brute-force would be to choose a target value and hash random
(fixed-size) strings and store those which hash to the target value. For a non-biased hash function with 32 bit output length, the probability of hitting a target in this way is 1/(2^32).

A meet-in-the-middle attack now tries to hit more than one target at a time. If the hash function can be inverted and the internal state of the hash function has the same size as the output, one can split the string into two parts, a prefix (of size n) and a postfix (of size m). One can now iterate over all possible m-sized postfix strings and calculate the intermediate value under which the hash function maps to a certain target. If one stores these strings and corresponding intermediate value in a lookup table, one can now generate random n-sized prefix strings and see if they map to one of the intermediate values in the lookup table. If this is the case, the complete string will map to the target value.

Splitting in the middle reduces the complexity of this attack by the square root, which gives us the probability of 1/(2^16) for a collision, thus enabling an attacker to generate multi-collisions much faster.

The hash functions we looked at which were vulnerable to an equivalent substring attack were all vulnerable to a meet-in-the-middle attack as well. In this case, the meet-in-the-middle attack provides more collisions for strings of a fixed size than the equivalent substring attack.

= The real world =

The different language use different hash functions which suffer from different problems. They also differ in how they use hash tables in storing POST form data.

== PHP 5 ==

PHP 5 uses the DJBX33A (Dan Bernstein's times 33, addition) hash function and parses POST form data into the $_POST hash table. Because of the structure of the hash function, it is vulnerable to an equivalent substring attack.

The maximal POST request size is typically limited to 8 MB, which when filled with a set of multi-collisions would consume about four hours of CPU time on an i7 core. Luckily, this time can not be exhausted because it is limited by the max_input_time (default configuration: -1, unlimited), Ubuntu and several BSDs: 60 seconds) configuration parameter. If the max_input_time parameter is set to -1 (theoretically:
unlimited), it is bound by the max_execution_time configuration parameter (default value: 30).

On an i7 core, the 60 seconds take a string of multi-collisions of about 500k. 30 seconds of CPU time can be generated using a string of about 300k. This means that an attacker needs about 70-100kbit/s to keep one
i7 core constantly busy. An attacker with a Gigabit connection can keep about 10.000 i7 cores busy.

== ASP.NET ==

ASP.NET uses the Request.Form object to provide POST data to a web application developer. This object is of class NameValueCollection. This uses a different hash function than the standard .NET one, namely CaseInsensitiveHashProvider.getHashCode(). This is the DJBX33X (Dan Bernstein's times 33, XOR) hash function on the uppercase version of the key, which is breakable using a meet-in-the-middle attack.

CPU time is limited by the IIS webserver to a value of typically 90 seconds. This allows an attacker with about 30kbit/s to keep one Core2 core constantly busy. An attacker with a Gigabit connection can keep about 30.000 Core2 cores busy.

== Java ==

Java offers the HashMap and Hashtable classes, which use the
String.hashCode() hash function. It is very similar to DJBX33A (instead of 33, it uses the multiplication constant 31 and instead of the start value 5381 it uses 0). Thus it is also vulnerable to an equivalent substring attack. When hashing a string, Java also caches the hash value in the hash attribute, but only if the result is different from zero.
Thus, the target value zero is particularly interesting for an attacker as it prevents caching and forces re-hashing.

Different web application parse the POST data differently, but the ones tested (Tomcat, Geronima, Jetty, Glassfish) all put the POST form data into either a Hashtable or HashMap object. The maximal POST sizes also differ from server to server, with 2 MB being the most common.

A Tomcat 6.0.32 server parses a 2 MB string of colliding keys in about
44 minutes of i7 CPU time, so an attacker with about 6 kbit/s can keep one i7 core constantly busy. If the attacker has a Gigabit connection, he can keep about 100.000 i7 cores busy.

== Python ==

Python uses a hash function which is very similar to DJBX33X, which can be broken using a meet-in-the-middle attack. It operates on register size and is thus different for 64 and 32 bit machines. While generating multi-collisions efficiently is also possible for the 64 bit version of the function, the resulting colliding strings are too large to be relevant for anything more than an academic attack.

Plone as the most prominent Python web framework accepts 1 MB of POST data, which it parses in about 7 minutes of CPU time in the worst case.
This gives an attacker with about 20 kbit/s the possibility to keep one Core Duo core constantly busy. If the attacker is in the position to have a Gigabit line available, he can keep about 50.000 Core Duo cores busy.

== Ruby ==

The Ruby language consists of several implementations which do not share the same hash functions. It also differs in versions (1.8, 1.9), which − depending on the implementation − also do not necessarily share the same hash function.

The hash function of CRuby 1.9 has been using randomization since 2008 (a result of the algorithmic complexity attacks disclosed in 2003). The CRuby 1.8 function is very similar to DJBX33A, but the large multiplication constant of 65599 prevents an effective equivalent substring attack. The hash function can be easily broken using a meet- in-the-middle attack, though. JRuby uses the CRuby 1.8 hash function for both 1.8 and 1.9. Rubinius uses a different hash function but also does not randomize it.

A typical POST size limit in Ruby frameworks is 2 MB, which takes about
6 hours of i7 CPU time to parse. Thus, an attacker with a single 850 bits/s line can keep one i7 core busy. The other way around, an attacker with a Gigabit connection can keep about 1.000.000 (one million!) i7 cores busy.

== v8 ==

Google's Javascript implementation v8 uses a hash function which looks different from the ones seen before, but can be broken using a meet-in- the-middle attack, too.

Node.js uses v8 to run Javascript-based web applications. The querystring module parses POST data into a hash table structure.

As node.js does not limit the POST size by default (we assume this would typically be the job of a framework), no effectiveness/efficiency measurements were performed.

Impact:

Any website running one of the above technologies which provides the option to perform a POST request is vulnerable to very effective DoS attacks.

As the attack is just a POST request, it could also be triggered from within a (third-party) website. This means that a cross-site-scripting vulnerability on a popular website could lead to a very effective DDoS attack (not necessarily against the same website).

Fixes:

The Ruby Security Team was very helpful in addressing this issue and both CRuby and JRuby provide updates for this issue with a randomized hash function (CRuby 1.8.7-p357, JRuby 1.6.5.1, CVE-2011-4815).

Oracle has decided there is nothing that needs to be fixed within Java itself, but will release an updated version of Glassfish in a future CPU (Oracle Security ticket S0104869).

Tomcat has released updates (7.0.23, 6.0.35) for this issue which limit the number of request parameters using a configuration parameter. The default value of 10.000 should provide sufficient protection.

Workarounds:

For languages were no fixes have been issued (yet?), there are a number of workarounds.

= Limiting CPU time =

The easiest way to reduce the impact of such an attack is to reduce the CPU time that a request is allowed to take. For PHP, this can be configured using the max_input_time parameter. On IIS (for ASP.NET), this can be configured using the “shutdown time limit for processes”
parameter.

= Limiting maximal POST size =

If you can live with the fact that users can not put megabytes of data into your forms, limiting the form size to a small value (in the 10s of kilobytes rather than the usual megabytes) can drastically reduce the impact of the attack as well.

= Limiting maximal number of parameters =

The updated Tomcat versions offer an option to reduce the amount of parameters accepted independent from the maximal POST size. Configuring this is also possible using the Suhosin version of PHP using the suhosin.{post|request}.max_vars parameters.

Credits:
Alexander Klink, n.runs AG
Julian Wälde, Technische Universität Darmstadt

The original theory behind this attack vector is described in the 2003 Usenix Security paper "Denial of Service via Algorithmic Complexity Attacks" by Scott A. Crosby and Dan S. Wallach, Rice University

---

References:
This advisory and upcoming advisories:
http://www.nruns.com/security_advisory.php

About n.runs:
n.runs AG is a vendor-independent consulting company specialising in the areas of: IT Infrastructure, IT Security and IT Business Consulting.