OpenBSD DNS Cache Poisoning

and

Multiple O/S Predictable IP ID Vulnerability

Amit Klein

October-November 2007

Abstract

The paper describes a weakness in the pseudo random number generator (PRNG) in use by OpenBSD, Mac OS X, Mac OS X Server, Darwin, NetBSD, FreeBSD and DragonFlyBSD to produce random DNS transaction IDs (OpenBSD) and random IP fragmentation IDs (OpenBSD, Mac OS X, Mac OS X Server, Darwin, NetBSD, FreeBSD and DragonFlyBSD – the latter three only if the kernel flag net.inet.ip.random_id is 1). A technique is disclosed that allows an attacker to detect the algorithm used and predict its next values. This technique can be used to conduct DNS cache poisoning attack on OpenBSD DNS server (which is a modified BIND 9 server) in caching mode. A predictability algorithm is described that typically provides 8-10 possible guesses for the next transaction ID value, 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 the OpenBSD DNS server. The net effect is that pharming attacks are feasible against OpenBSD caching DNS servers, without the need to directly attack neither DNS servers nor clients (PCs). A similar technique is disclosed to detect the algorithm used for the IP fragmentation ID generation (thereby enabling fingerprinting, traffic analysis and host alias detection for OpenBSD, Mac OS X, Mac OS X Server and Darwin (and NetBSD, FreeBSD, DragonFlyBSD, if the kernel flag net.inet.ip.random_id is 1), as well as detecting "missing" IDs, which can be used in nmap's IdleScan method (as the "zombie" machine whose IP is used to scan the actual target host). IP fragmentation ID can also be used (in some cases) for TCP blind data injection.

Table of Contents

1 Introduction

Back in 1997, OpenBSD introduced randomization to some important protocol fields in the network stack. Such fields are the DNS transaction ID and the IP fragmentation ID. This was done in order to protect against attacks that exploit predictability in those fields, such as DNS cache poisoning and the IdleScan stealth scanning method.

Both DNS transaction ID and IP fragmentation ID are 16 bit quantities, and both should ideally not repeat themselves for as long as possible. The randomization algorithms for the DNS TRXID and the IPID fields were similar, yet with subtle differences. They have also gone through some modifications. Furthermore, other projects (Mac OS X, Mac OS X Server, Darwin, NetBSD, FreeBSD and DragonFlyBSD) started using these algorithms after a while. At present day, there are 3 flavors of the basic 16 bit algorithm in use, each such flavor is given a name in this paper, and is analyzed and shown to be predictable.

There are two aspects in which the algorithms differ: one is the amount of random external data used to randomize "hops" – this data varies from 3 bits (1-8 hops) to 0 bits (no random hops). The other aspect is whether a XOR operation is applied at some point (with *seed2*), or an ADD operation.

For the purpose of this paper, a designation system for the algorithm flavors is used, such that the operator (XOR/ADD) is designated by a capital letter (X and A, respectively), followed by the number of random bits comprising the hop randomization. The following table summarizes the flavors encountered "in the wild"; all three flavors are analyzed in the next sections.

	Algorithm X3	Algorithm X2	Algorithm A0
Caching DNS server (BIND 9) DNS transaction ID	**OpenBSD** 3.3-**4.2**		
libc resolver DNS transaction ID	OpenBSD 2.8-3.4		**OpenBSD** 3.5-**4.2**
IPv4 fragmentation ID		OpenBSD 2.6-3.4 **Mac OS X** 10.0-**10.5.1** **Mac OS X Server** 10.0-**10.5.1** **Darwin** 1.0-**9.1** **FreeBSD**[1] 4.4-**7.0**[2] **DragonFlyBSD**[1] 1.0-**1.10.1**	**OpenBSD** 3.5-**4.2** **NetBSD**[1] 1.6.2-**4.0**

[1] This O/S kernel has net.inet.ip.random_id flag which can be set to 1 to have the O/S randomize the IP fragmentation IDs. By default though, this flag is off, and

2 **Algorithm X3**: OpenBSD DNS server cache poisoning

2.1 The OpenBSD DNS server and DNS cache poisoning

OpenBSD's DNS server is based on ISC's BIND DNS server, with modifications to suit OpenBSD's needs (especially around security). Particularly, BIND's default transaction ID algorithm is replaced with an OpenBSD specific implementation. Of this implementation, the OpenBSD personnel stated ([4], with regards to [1]):

> "when BIND 9 was first imported into OpenBSD, we decided not to use the default ID generation algorithm (LFSR, Linear Feedback Shift Register) but to use a more proven algorithm (LCG, Linear Congruential [sic] Generator) instead. thanks to this wise decision, the BIND 9 shipped with OpenBSD does not have this weakness.
>
> the proactive security of OpenBSD strikes again"

As well as ([5], by the OpenBSD project coordinator):

> "We had gone through great efforts with the CORE guys (who did the math side of our non-repeating random number generator) to make sure that attacks of that kind [predicting DNS transaction ID] would not be feasable [sic]."

This section describes a DNS cache poisoning attack on OpenBSD DNS server. For a general historic overview of DNS cache poisoning, as well as relevant prior works and references, please refer to [1]. Hereby are listed only OpenBSD DNS server -specific issues:

1. UDP source ports - UDP source ports are predictable for OpenBSD DNS server implementation (the UDP port is unchanged, as briefly mentioned in [2] and was verified in the author's experiments). In fact, the source UDP port for outgoing queries from OpenBSD DNS Server is static.

IP IDs are sequential (which provides no security at all). Packet Filter IP randomization ("scrub out random-id") takes place regardless of this flag though.

[2] At the time the paper is released, FreeBSD 7.0 is in RC1. However, according to the FreeBSD team, the fix will be applied to released following 7.0, and not to FreeBSD 7.0 itself.

2. Historic notes and prior works pertaining specifically to OpenBSD's DNS Server:

- In April 1997, a paper was released [3], mentioning the problem of sequential transaction IDs in BIND 4.9.5-P1. The solution (a random number generator based on linear congruence generators) provided in the paper was developed by OpenBSD (as stated in the paper itself).
- Probably sometime in 1999, David Wagner reported some weakness of the 1997 algorithm in private to the OpenBSD project, upon which OpenBSD modified the algorithm (added "seed2" per David Wagner's suggestion [14]).
- As of 1997, the OpenBSD operating system incorporated their algorithm (instead of the native algorithm provided by ISC) in all OpenBSD releases ([5]). Particularly, OpenBSD 3.3 and above run BIND 9 with the 1999 algorithm.

The attacks described in this paper make use of the predictable nature of OpenBSD's DNS Server transaction IDs to poison its cache. It is assumed that the DNS server can be forced to perform DNS queries using a malicious web page. This is a real-life condition, but of course it limits the attacker's activity scope – 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.

2.2 The X3 algorithm

The code for the transaction ID algorithm is available to the public at [9]. Below is a simplified version C code:

```
n = 3 random bits drawn from an external source

for (i=0; i<=n; i++)
{
        x = (a*x + b) % M;
}
TRXID = (seed ^ pmod(g, seed2 ^ x, N)) | msb;
```

Where

M=31104 (=$2^7 \cdot 3^5$).

N=32749 (prime, so $\varphi(N)=(N-1)$ is an even number). $N-1=2^2 \cdot 3 \cdot 2729$.

a is a PRNG parameter (1…(M-1)), and by construction (a mod 48)=1.

b is a PRNG parameter (15 bits, but obviously since it is only used in modulo M arithmetic, one can think of it as in the range 0…(M-1)); also by construction (b mod 2)=1, and 3 doesn't divide b.

> *g* is a PRNG parameter – a generator for the invertible numbers in the mod (N-1) ring, thus 10912 different *g* values are possible.
>
> *seed* is a PRNG parameter (15 bits).
>
> *seed2* is a PRNG parameter (15 bits).
>
> *msb* is a PRNG parameter (two possible values: 0 or 0x8000).
>
> *x* is the PRNG state (0…(M-1)). Its initial state is a PRNG parameter.
>
> *pmod*(t,e,p) is the function ((t^e) mod p).

The PRNG is re-keyed every 180 seconds or 30,000 *x* steps (theoretically between 3,750 to 30,000 calls to TRXID generation, but obviously in most cases around 6,500-7,000 calls). *msb* is switched to its other value per re-keying.

The transaction ID is serialized into the DNS ID header field as big-endian (network order), i.e. bits 15…8 in the first byte and bits 7…0 in the second byte.

2.3 Observations on the X3 algorithm

This section describes important properties of the X3 algorithm. Additional properties (which are not used in the attack, but are nonetheless interesting) can be found in Appendix C.

2.3.1 *msb* removal

Note that *msb* doesn't change as long as the PRNG is not re-keyed. That is, in a sequence of TRXIDs generated by the same key, *msb* will be constant, and can be easily extracted (note that (`seed ^ pmod(g, seed2 ^ x, N)`) occupies the 15 lowest bits of TRXID and msb occupies the most significant, 16th bit of TRXID, so they're mutually exclusive). Henceforth we'll ignore *msb*, and for simplicity assume TRXID = `seed ^ pmod(g, seed2 ^ x, N)`.

2.3.2 Elimination techniques

Define $\{x_i\}$ a sequence of *x* values used for consecutive TRXID generation, i.e. each x_i is the value of *x* after the loop. Since the sequence is governed by modular arithmetic, and the modulo used (M) is a multiplicity of 2^4, we can look at a derived sequence, modulo 2^4=16. In this derived sequence, each consecutive value is obtained by applying the formula below *n* times, where *n* is between 1 and 8 (remember that *a* mod 16=1):

$$x \leftarrow x+b \pmod{16}$$

Or, in other words, the following equation holds:

$x_{i+1} = x_i+n_{i+1}\cdot b \pmod{16}$

So, if one can extract the derived sequence $\{x_i \bmod 16\}$, and one knows (or guesses) the 4 least significant bits of b, then it is possible to verify this guess, by calculating:

$n_{i+1}=b^{-1}\cdot(x_{i+1}- x_i) \pmod{16}$

And observing that each n_{i+1} must satisfy $1\leq n_{i+1}\leq 8$ (which yields 1 bit of elimination per a consecutive pair of x's). This technique also provides, as a by product, the exact number of "hops" between consecutive x's, namely n_{i+1}.

This technique can be easily extended up to mod 2^7, in which case it yields 4 bits of elimination per a consecutive pair of x's.

2.3.3 Indiscernible solutions

Define $\{y_i\}$ as a linear congruence sequence:

$y_{i+1}=a\cdot y_i+b \bmod K$

Define Z as the sequence $\{z_i\}$ derived from the "source" sequence $\{y_i\}$:

$Z(\{y_i\},a,b,S)=\{z_i\}= y_i \text{ XOR } S$

Three interesting observations are as following:

1. Assume a is odd and $K=2^m$, then the following holds:

   $Z(\{y_i\},a,b,S)= Z(\{y_i \text{ XOR } 2^{m-1}\},a,b,S \text{ XOR } 2^{m-1})$

   (the proof is trivial, since for every t, $(t \text{ XOR } 2^{m-1})=(t+2^{m-1}) \bmod 2^m$))

   That is, when observing the Z sequence (mod $K=2^m$), and if a is odd, it is impossible to distinguish between those two possible "solutions" to the source sequence.

2. Let $m>0$ be minimal such that $2^m-K\geq 0$. If it so happens that 2^m-K is "small" (compared to K), then "usually":

$Z(\{y_i\},a,b,S)= Z(\{((2^m-1)-y_i) \bmod K\},a,((2^m-1)\cdot(1-a)-b) \bmod K,(2^m-1)-S)$

Proof: it's easy to see that modulo K, the sequence $\{(2^m-1)-y_i\}$ is linear with parameters a and $((2^m-1)\cdot(1-a)-b)$. Now, if

$(2^m-1-y_i)<K$ (over the integers)

Then

$((2^m-1)-y_i) \bmod K = (2^m-1)-y_i$ (over the integers)

Finally, since

t XOR $(2^m-1)=(2^m-1)-t$ (for every t)

It follows that:

$((2^m-1)-y_i)$ XOR $((2^m-1)-S)$ = $((2^m-1)-y_i)$ XOR $((2^m-1)$ XOR $S)$ = $(((2^m-1)-y_i)$ XOR $(2^m-1))$ XOR S = $((2^m-1)-((2^m-1)-y_i))$ XOR S =

y_i XOR S

That is, if all y's involved obey $(2^m-1-y_i)<K$, then the two solutions are indistinguishable. If the y's are uniformly distributed (between 0 and $(K-1)$), then the probability of a single y value to obey the formula is

$1-((2^m-K)/K)$

And, if (2^m-K) is small, compared to K, then this probability can be pretty high. For example, if $K=M=31104$ then $m=15$ and the probability is 94.65%.

The probability of the complete sequence (or a sub-sequence) to be indistinguishable is therefore 0.9465^L, where L is the sequence (or the sub-sequence) length. If $L=15$, then this probability is 43.8%.

3. If follows from the previous observation that if $K=2^m$ then the solutions are always indiscernible:

$Z(\{y_i\},a,b,S)= Z(\{((2^m-1)-y_i) \bmod 2^m\},a,(a-b-1) \bmod 2^m,(2^m-1)-S)$

Observations #1 and #3 will be helpful for work reduction in the actual attack. Observation #2 is needed in order to understand why multiple solutions are sometimes possible.

Note that it is possible to combine two "transformations" to show that there are many indistinguishable solutions. For example, where $K=2^m$ (and when a is odd) it can be easily seen that there are 4 indistinguishable solutions (combining #1 and #3).

2.4 The basic attack

At large, the attack framework is identical to that of [1]. The attack target is an organization (e.g. a corporate, an ISP or an academic institute) with an internal OpenBSD 4.2 (results in this paper were obtained for OpenBSD 4.2 which uses a BIND 9 baseline; very likely all OpenBSD 3.3 and above are vulnerable since they use BIND 9; OpenBSD below 3.3 may also be affected) DNS caching server. This server does not answer DNS queries from the Internet, and no direct access to the internal network is available to 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. Let's skip the part where the name server obtains the authoritative name-server for attacker.com and focus on the query for www1.attacker.com. It is sent to the attacker's name server. This name sends back a CNAME record for the next host name (i.e. a CNAME that points at www2.attacker.com). The OpenBSD DNS server will then request www2.attacker.com with the next ID value. This process repeats itself (15 times). At this point, the attacker has the desired sequence and he/she can find the PRNG parameters.

The above technique is called CNAME chains[3]. 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

[3] CNAME chains are discouraged per the DNS RFC 1034 ([8]), 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 [7].

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.

Finding the PRNG parameters is the main challenge here. As explained above, it's trivial to extract *msb*, and indeed it can be assumed not to exist.

To clarify the attack, and to keep in order for the text to correspond to the code (in appendix A), the PRNG reconstruction is described as a series of phases.

Phase 1: The attacker guesses all possible 10,912 *g* values. For each *g*, the attacker creates a "log_g" table, mapping all possible values in 1…(*N*-1) to the power of *g* that creates them (note that the values 0 and *N*…2^{15}-1 are invalid, and also note that since *N*=32749 is slightly smaller than 2^{15}=32768, there are 20 values that will have 2 possible "sources"). Then, for each *g*, the attacker guesses all possible seed values (2^{15}). For each *g* and *seed*, the attacker also guesses all possible 8 values of (*b* mod 16), as well as (naively speaking) all possible 16 values of *seed2* mod 16. The attacker then eliminates consecutive pairs of values using the elimination technique described above. Note that each (*x* mod 16) is calculated as (log_g[*seed* XOR TRXID] XOR *seed2*) mod 16. Since each internal loop iteration eliminates half of the candidates, the expectancy of the loop length is 2. Hence, the total number of inner loop iterations is estimated at $10912\cdot2^{15}\cdot8\cdot16\cdot2$=91.5 billion. It is expected to produce around 2.8 million candidates for (*g*,*seed*,(*b* mod 16),(*seed2* mod 16)).

Phase 2: for each candidate found in phase 1, the attacker guesses the higher 3 bits in (*a* mod 2^7), (*b* mod 2^7) and (*seed2* mod 2^7). Elimination is again performed according to the above technique, at this time since the "hops" are known the elimination in each pair is around 3 bits. Since the elimination is now 3 bits, the expectancy of the loop length is 8/7. So the innermost loop body will be executed 2.8 million times $2^3\cdot2^3\cdot2^3$ times (8/7) = 1.6 billion times. The loop body consists of advancing the LCG by several hops. There are, therefore, 7.2 billion advances expected, though for optimization, the minimum *n* can be chosen for the first loop iteration, probably $n_{\min}$=1, getting LCG advance count of 1.6 billion. There will probably be one surviving candidate (up to indiscernible solutions).

Phase 3: for each candidate found in phase 2, the attacker guesses the remaining bits in *a*, *b* and *seed2* (243·243·256=15 million) and can fully verify the parameters.

At the end of phase 3, the attacker will be left with the correct PRNG parameters, and possibly (for a sequence of length 15, with probability around 45%) another set of parameters (indiscernible from the correct set, as explained above).

Now, given a TRXID from the same key, the attacker can quickly calculate a list of possible next values for this TRXID. It is done as following (assuming *msb* has been taken care of): with the correct *seed*, *g* and *seed2*, the attacker calculates *x*=log_g[TRXID XOR *seed*] XOR *seed2*. The attacker advances *x* 8 times (using *a* and *b*), and in each step, the attacker calculates a possible next TRXID by applying XOR with *seed2*, raising *g* to the power of the result (mod *N*) and XORing with *seed*. Each such value (up to msb adjustment) is a possible value of the next TRXID. There are 8 such values for the correct set of parameters. Additionally, if an additional set of parameters exist, then its 8 values should be added as well. However, it can easily be seen that in around 95% of the cases (per step), both sets produce the same value. Therefore, if there are two sets, the expected size of the possible next TRXID list is 8.4, and in around 98% of the cases, there will be no more than 2 additional values. Now, taking into account that there is 55% probability that only a single set will be found, the net result is that in 99% of the cases, there will be 8-10 possible next TRXID values.

Phase 1 and 2 can be easily optimized using observations #1 and #3. It is possible to reduce the guess space for *seed2* by 2 bits. Note that the most significant bit guessed in *seed2* is needless. It is enough to loop over *seed2* with least significant bit 0, since a solution with *seed2*'s most significant bit being 1 is mirrored into the same solution with most significant bit being 0, and vice versa. Likewise, the next to most significant bit in *seed2* can be chosen as 0 because each solution with that bit being 1 is mirrored in a solution with that bit being 0 (with a different *b* though) by applying observation #3 (followed by observation #1 to reset the most significant bit). This optimization reduces the amount of times the inner loop body needs to be executed in phase 1 to around 23 billion times.

The C program in appendix A implements this algorithm. As written, it is single threaded, and as such its runtime on a Dell PowerEdge SC1430 (with quad core Intel Xeon E5310 [15]) is around 360 seconds (6 minutes). However, this is a quad-core machine, so a multithreaded version (not provided in this document) was prepared for it, and the multithreaded version runtime is estimated at 90 seconds to yield the PRNG parameters (the runtime estimation of the multithreaded version on a quad-core are based on running two threads in a two core configuration which is otherwise idle, and dividing the runtime period thus obtained by two).

As can be understood, with such runtime, it is impossible for the attacker to force the DNS server to wait until the PRNG is reconstructed (due to DNS timeouts in the attacked DNS server). However, the attacker can easily force the DNS server to resolve a second domain name (once the PRNG parameters are known), in which case the attack can proceed immediately (once the PRNG parameters are known, producing the next TRXID values predicted for a given TRXID is done in less than a millisecond.

Once the current TRXID is known, the attacker's DNS server can respond in a CNAME record for www.example.com. Simultaneously, the attacker can start sending the forged 8-10 DNS responses, as rapidly as possible, cycling through them again and again. Even with a modest 256Kbit uplink and even 150 bytes per response it is possible to achieve a cycle in less than 40-50 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).

2.5 Attack variants

The CNAME chain is probably the most effective way to force the DNS server to rapidly iterate through IDs. However, it is not the only way to achieve this goal. It's possible (but less efficient) to force ID iteration via HTTP redirection (www1.attacker.com redirects to www2.attacker.com at the HTTP level, and so forth).

The CNAME chain can be also established via a "ping-pong" between two domains (this enables static configuration of the two name servers involved. So www1.attacker1.com will have a CNAME pointing at www2.attacker2.com, which will have a CNAME pointing at www3.attacker1.com, and so on. Just as with a single name-server chain, the OpenBSD DNS server will not follow more than 16 redirections. In this case, it's harder for security systems to detect that an anomalous DNS activity is taking place, what with the fact that some legitimate DNS servers do answer with a CNAME pointing at a name outside their authoritative domain (e.g. when content delivery network host is used to deliver the content).

If the gTLD server is too close to the OpenBSD DNS server to be poisoned (i.e. the round trip to it is very short), or if example.com's authoritative name server record is already cached by the OpenBSD DNS server, it may still be possible to poison the A record for say www.example.com. All the attacks above should work just the same.

The PRNG reconstruction algorithm as implemented in Appendix A is simplified, in the sense that the log_g table is 1:1, while in fact it should represent a one-to-many relation, since there are 20 values which have two log values (the rest 32,728 values have one source). This means that the algorithm as implemented in appendix A will have false negatives. For each TRXID provided to it, there's a probability of 20/32748 for its log_g to be incomplete. Hence, with a sequence of length 15, and an additional TRXID (16 TRXIDs altogether), the probability for a miss is 0.97%, or in other words, the probability of the algorithm as-is not to have a false negative is more than 99%. Still, it is possible to add the logic for the one-to-any relation into log_g – this shouldn't gravely affect the overall runtime of the algorithm.

The PRNG reconstruction algorithm can be further optimized. When hand-optimized, using the same machine with multithreading yields an estimated 20-25 seconds runtime (approximately 4× improvement). Quite likely, there's room for even more optimization, possibly moving to Assembler language.

In the experiments, a sub-$1000 platform (Dell PowerEdge SC1430) was used. An attacker may have access to a more powerful machine.

Distributing the work to many machines is another tactic that can vastly improve the attack runtime. For example, phase 1 can be paralleled easily up to 10912 cores. The algorithm's memory consumption is almost negligible (the g_log table in the appendix A's implementation consists of 32K four bytes entries, i.e. 128KB. It can probably be halved without noticeable impact on runtime) so it can fit in a multi-core platform.

A note about the sequence length: in the above discussion, a sequence length of 15 TRXIDs was assumed. An information theoretic argument can bound the sequence length from below as following: assume a sequence length L. Then the algorithm finds the effective key (66 bits) the initial value of x (14.9 bits), and (L-1) values of n (3 bits each). That is, the algorithm finds $77.9+3\cdot L$ bits. At the same time, the information provided to the algorithm is $15\cdot L$. From information theory, we have:

$$77.9+3 \cdot L \le 15 \cdot L$$

Therefore, $L \ge 7$.

In fact, the algorithm would work for any L, and will provide a possible set of candidates. Naturally with $L<7$, there will be many false positives. From runtime perspective, decreasing the sequence length will cause phase 2 to take more and more time, and around $L=11$ or $L=12$, phase 2 will become dominant. Decreasing the sequence length further will almost double the runtime per each decrement.

On the other hand, increasing the sequence length will not improve the algorithm runtime/success significantly. The one possible benefit of providing more data to the algorithm is the likelihood for eliminating the indiscernible solution.

3 **Algorithm A0**: OpenBSD (and NetBSD if the kernel flag net.inet.ip.random_id is 1) IP fragmentation ID prediction – fingerprinting and applicability for idle-scanning

Predictability of the IP fragmentation ID can be used for fingerprinting and stealthily probing hosts – this was first described in 1998 ([10]), and was later the subject of a detailed paper ([11]). The probing technique is now called idle-scan, and it is incorporated in the popular nmap scanner ([11]). In this attack, the machine whose IP ID is predictable plays the role of the "zombie", which isolates the real attacker from the scanned target (host). The target host will experience the port mapping as coming from the zombie's IP address.

Additional attacks that make use of predictable IP ID field (e.g. host alias detection and traffic analysis) are described at the bottom of [11].

A TCP blind data injection attack that exploits IP ID predictability (in some cases) is described in [27] and [28].

With respect to the idle-scan attack and IP ID randomization, OpenBSD states[4] ([26], pages 17-18):

[4] As a side note, it seems that [26] refers to the X2 algorithm that was used for IP ID generation in OpenBSD until December 2003. This can be understood from page 26, which states that ~12,000 IDs are expected in a single key - this corresponds to 2.5 iterations per ID, which in turn corresponds to a random hop

"Pseudo-random IPID since 1998"

"Attack (mostly) no longer possible"

This section described how the IP ID PRNG can be predicted, thus enabling idle-scanning, O/S fingerprinting, host alias detection, and in some cases, TCP blind data injection.

OpenBSD (and NetBSD, if the kernel flag for this feature is turned on) uses A0 to randomize the IP fragmentation ID field ([16], [17]). Compared to X3, A0 makes use of the ADD operator to introduce the *seed2* value into the randomization process. This has some interesting consequences. Since the addition is de-facto modulo (N-1), and (N-1) is divisible by 4, but not by higher powers of 2, the algorithm devised for X3 cannot work as-is (neither phase 1 as-is, nor phase 2). However, since there are no random hops, there is an interesting property that does emerge, and is very helpful.

Note that $g=2^j \bmod N$. Therefore:

$$\log_2(\mathrm{IPID}_i \text{ XOR } seed)=(j\cdot(x_i+seed2)) \bmod (N-1)$$

(where $\log_2$ is over the modulo N ring).

Now, taking mod 12 on both sides, and keeping in mind that gcd(M,N-1)=12 so 12 divides (N-1):

$$\log_2(\mathrm{IPID}_i \text{ XOR } seed) \bmod 12 = (j\cdot(x_i+seed2)) \bmod 12$$

Subtracting consecutive values yields $(j\cdot(x_{i+1}-x_i)) \bmod 12$

And since $(x_{i+1}-x_i)=((a-1)\cdot x_i+b) \bmod M$, and 12 divides M, and a-1=0 mod 12, the right side part becomes $(j\cdot b) \bmod 12$, which is constant. Note that by construction, j and b are even and are not divisible by 3, so $(j\cdot b)$ is even and not divisble by 3, i.e. modulo 12, it can only assume four values: 1, 5, 7 and 11.

The result is, that given the correct *seed*, the difference (mod 12) between consecutive $\log_2$(IPID XOR *seed*) values is constant (and one of four values). This can be used to easily reconstruct the correct seed value (given around 7 consecutive IPIDs). This is phase 1 of the algorithm in this case.

---

in the range 1-4 (2 bits random hops). The current algorithm (A0) does not incorporate such hops. Nevertheless, X2 is vulnerable just the same (see next section).

In fact, this result suffices for O/S fingerprinting, host alias detection[5], idle-scanning and probably firewall rule-set detection too. Using this method, and given seed, one can easily observe the "distance" (mod 12) of a given IPID from the previously observed IPID (see appendix B for implementation). For example, one can distinguish between a situation wherein two consecutive IPIDs are provided, and a situation wherein one IPID is skipped. Note that for the IdleScan method, it suffices to assume that M is even (i.e. no need for 12 to divide it), and use mod 2 arithmetic (this requires a longer sequence though – at least 16 consecutive IPIDs).

The false positive probability of this detection scheme is as following. Assume a random sequence of length k. There are (k-1) consecutive pairs in it, with the first pair used to calculate the constant difference. Therefore, there are (k-2) pairs that can be used for elimination, with each pair yielding $\log_2 12 \approx 3.6$ bits of elimination, and altogether $3.6 \cdot (k-2)$ elimination bits. There are additional $\log_2 3$ elimination bits due to the difference having 4 possible values (out of 12). Initially, there are 2^{15} candidates for *seed* value. A single seed candidate has probability of $1/3 \cdot 12^{-(k-2)}$ to "survive", or $1-1/3 \cdot 12^{-(k-2)}$ to be eliminated. The probability of all 2^{15} seed values to be eliminated is thus $(1-1/3 \cdot 12^{-(k-2)})^{2**15}$, so the false positive probability is $1-(1-1/3 \cdot 12^{-(k-2)})^{2**15})$. Using the approximation $(1-1/n)^n=e^{-1}$ (for large $n=3 \cdot 12^{(k-2)}$), this can be rearranged into $1-e^{-(2**15/3 \cdot 12**(k-2))}$. The following table summarizes the false positive probability for various sequence lengths:

Sequence length (k)	False positive probability
5	99%
6	41%
7	4.3%
8	0.36%
9	0.03%

Again, as can be seen, this method is good enough for O/S fingerprinting, host alias detection and idle scanning. However, other attacks involving IP ID (e.g. traffic analysis and TCP blind data injection) require a more complete prediction capability.

Phase 2 is as following: enumerate over all *seed2* values, enumerate over all g values (10912). Using $\log_g$(IPID XOR *seed*) to get (x+*seed2*) mod (N-1), and knowing *seed2*, it's possible to extract x mod (N-1), and since M<(N-1), this yields x mod M. Since M is divisible by 16, and using similar arguments as above, it follows that the difference between consecutive values of ((($\log_g$(IPID XOR *seed*)-*seed2*) mod (N-1)) mod 16 is exactly b mod 16. Observing that this difference is constant therefore yields further elimination. Note that there's no

[5] For host alias detection, one can start obtaining IP IDs from the first host, then move to the second host. If the combined series can still be processed successfully by the detection algorithm, then it's probably the same host.

need to calculate the full $\log_g$ table. Rather, one can calculate a global $\log_2$ table once, and (since $g=2^j \bmod (N\text{-}1)$) use the fact that $\log_g(x)=j^{-1}\cdot\log_2(x) \bmod (N\text{-}1)$.

At the end of phase 2, there are several candidates for ($seed$,g,$seed2$,$\{x_i\}$). Most likely, there will be single seed and g values, yet multiple seed2 values (and their corresponding $\{x_i\}$ sequence). This is due to the following observation: there are many almost-indiscernible sets of solutions. To begin with, (g,$seed2$,a,b) and ($g^{-1} \bmod N$,a,$M\text{-}b$,$(N\text{-}1)\text{-}seed2+(N\text{-}1)\text{-}M$) are indiscernible. Furthermore, (g,$(seed2\text{-}\delta) \bmod (N\text{-}1)$,$a$,$(b\text{-}(a\text{-}1)\cdot\delta) \bmod M$) is almost indiscernible from (g,$seed2$,a,b) for small values of $|\delta|$ (1 to few hundreds or thousands, and depending on the exact x values). This results in many alternative solutions. These properties can be potentially used to reduce the amount of work. Fortunately, those same sets rarely add false positive candidates (experimental results show a typical scenario of a single candidate).

Phase 3 involves simply solving two linear equations for a and b, e.g.

$$x_{i+1}=a\cdot x_i+b \bmod M$$
$$x_{i+2}=a\cdot x_{i+1}+b \bmod M$$

Which is easily solvable as long as $(x_{i+1}\text{-}x_i)$ is invertible modulo M. This is equivalent to requiring that $(x_{i+1}\text{-}x_i)$ is invertible modulo 2 and modulo 3. This is guaranteed (for the correct solution of the x series, a and b) because

$$x_{i+1}\text{-}x_i=(a\text{-}1)\cdot x_i+b \bmod 6$$

And since $(a\text{-}1) \bmod 6=0$ and $(b \bmod 6)$ is invertible (mod 6), it follows that $(x_{i+1}\text{-}x_i)$ is invertible modulo M.

The implementation in appendix B is naïve, and does not exploit those indiscernible sets. Still, it runs quite fast (less than 7 seconds on a single threaded implementation on a slow laptop). An optimized version which does incorporate those and other optimizations takes few dozen milliseconds (up to few hundred milliseconds when the sequence is short) to run on a slow laptop in order to extract the next IP ID. The algorithm in appendix B was tested with OpenBSD 4.2, and was found to work correctly.

A sequence length of 7 usually suffices to calculate the next IP ID (typically a single candidate).

Complete prediction can be used for traffic analysis as following (this mode is not supported in the code provided in appendix B): once the algorithm parameters are extracted, provide two samples with some time difference between them (but not too long as to avoid re-keying during this time period). The parameters can be used to extract the x of the first sample, and then it's easy to advance x and produce the next hundreds/thousands hops, and try to match them with the second sample. A match would yield the amount of hops (= used IP IDs) between the samples.

TCP blind data injection attack can be mounted as following: the attacker forces the target host to send a sequence of IP packets to the attacker, thus obtaining the necessary sequence of consecutive IP IDs. Depending on which algorithm is available to the attacker, the attacker can then either analyze it immediately (with the optimized algorithm) and predict the next IP ID, or analyze it in near real time (few seconds – with the un-optimized in Appendix B) and extract the PRNG parameters, then force the target host to send another IP packet (with a fresh IP ID) to the attacker, and use the extracted PRNG parameters to predict the next IP ID value.

NOTE: from source code analysis, it appears that OpenBSD uses this algorithm to randomize IP fragmentation ID not only in "regular" IP traffic (with TCP/UDP), but also in raw IP traffic, Ethernet-inside-IP encapsulation, IP-inside-IP encapsulation, the CARP protocol, IP multicast routing, pfsync interface protocol, packet filter (IP packet normalization), and network bridge (ICMP error packets). Likewise, NetBSD uses this algorithm to randomize IP fragmentation ID for CARP and packet filter (IP packet normalization), and (when the kernel flag net.inet.ip.random_id is 1) for "regular" IP traffic (with TCP/UDP), raw IP, IP multicast routing, IP-inside-IP encapsulation and IPsec encapsulation in IP.

NOTE: the OpenBSD resolver's[6] DNS transaction ID is supposed to be secure ([26] page 14) However, the implementation of the DNS transaction ID ([18]) can be predicted in the same manner, since it uses the same algorithm (A0, with the PRNG parameters being scoped to the running process instance, though). Fortunately, OpenBSD also uses randomized source (UDP) ports (see [13] and [26] page 28), and this randomization is based on the ARC4 algorithm. As such, spoofing responses for the resolver necessitates knowing the UDP source port, predicting the ARC4 algorithm's next output, or exploiting an implementation bug, none of which is currently known to apply.

NOTE: OpenBSD uses network order for the libc resolver DNS transaction ID, and all IP fragmentation ID fields except for packet filter (host order). NetBSD uses network order for all IP fragmentation ID fields except for packet filter, CARP and IP-inside-IP encapsulation.

4 **Algorithm X2**: Mac OS X, Mac OS X Server, Darwin (and FreeBSD and DragonFlyBSD - if the kernel flag net.inet.ip.random_id is 1) IP

[6] Page 14 of [26] probably refers to the resolver rather than to the DNS server, since it mentions random source port. Page 30 of [26] mentions ~12,000 IDs can be generated in a lifetime of a key, but this corresponds to random hops of 2 bits, and neither the resolver (A0 or X3) nor the DNS server (X3) implement random hops of 2 bits. This is probably a mistake in the text of [26].

fragmentation ID prediction – fingerprinting and applicability for idle-scanning

The X2 algorithm is in use for Mac OS X, Mac OS X Server and Darwin ([19]) for IP fragmentation ID. It is also used for IP fragmentation ID in FreeBSD and DragonFlyBSD ([20], [21]; only if the kernel flag net.inet.ip.random_id is set to 1; by default it is set to 0, in which case the IP fragmentation ID defaults to a sequential counter). In fact, one FreeBSD developer mentioned this option ([29]) as a way to combat the TCP blind data injection attack described in [27] and [28].

Naturally, since X2 is a weakened version of X3, the analysis of X3 can be applied as-is for X2. Of course, the basic algorithm can be optimized (probably by ×4) because there are less random bits per each iteration. Also, less sample data is needed: in experiments, 9-10 samples sufficed to run the algorithm with minimum false positives.

Usage for fingerprinting Mac OS X, Mac OS X Server and Darwin (and FreeBSD and DragonFlyBSD, if the kernel flag net.inet.ip.random_id is 1) is straight-forward: the algorithm for X3 needs to be run and if it returns positive results, then this is probably Mac OS X, Mac OS X or Darwin (or FreeBSD/DragonFlyBSD if the kernel flag net.inet.ip.random_id is 1). Of course, the algorithm can be improved (can be run faster since less entropy is introduced in each step). Some improvements are provided in appendix A. Note that for X2, typically only 4 candidates for the next value are provided (vs. 8 for X3). It may be possible to further distinguish between the operating systems based on byte order: in Mac OS X, Mac OS X Server and Darwin, the bytes are in host order. Ditto for FreeBSD's packet filter.

Usage for IdleScan is somewhat tricky. Since the algorithm predicts 1-4 next IP ID values, corresponding to 1-4 hops, and since there's no telling whether, say, 3 hops are the result of a single 3 hop iteration, or two iterations (1+2 hops, or 2+1 hops), or even three iterations (1+1+1 hops), in order to detect a "gap", one needs to repeat the spoofed SYN attempt at least 4 times, and then observe whether the next IP ID is one of the four predicted, or not. If it is, then the target port is closed. If it isn't, then the target port is open.

Usage for host alias detection is as explained for A0.

Usage for traffic analysis is similar to the technique described for A0. However, since each new IP ID consumes on the average 2.5 "hops", the number of hops calculated between the two consecutive samples should be divided by 2.5 to get an estimation of the traffic.

Usage for TCP blind spoofing is straight forward, though since 4 candidates for IP ID are provided, all 4 must be attempted.

NOTE: From source code analysis, it appears that FreeBSD uses this algorithm for packet filter (IP packet normalization), pfsync interface protocol and (if the kernel flag net.inet.ip.random_id is 1) for "regular" IP traffic (with TCP/UDP), raw IP, CARP, IP multicast routing, SCTP, IP-inside-IP encapsulation and IPv6 over GRE. Likewise, DragonFlyBSD uses this algorithm (if the kernel flag net.inet.ip.random_id is 1) for "regular" IP traffic (with TCP/UDP), raw IP, CARP, IP multicast routing and IP-inside-IP encapsulation. Mac OS X, Mac OS X Server and Darwin appear to use it (apart from "regular" IP) for DHCP IP fragmentation ID, raw IP, IP multicast routing and IPsec encapsulation in IP.

NOTE: FreeBSD uses host order for the IP fragmentation ID field, except for pfsync (which uses network order). DragonFlyBSD uses host order for this field except for SCTP. Mac OS X, Mac OS X Server and Darwin always use host order for this field.

NOTE: On top of the generic weakness in X2, there is an implementation bug in the Mac OS X, Mac OS X Server and Darwin implementation of the algorithm wherein *seed* is not initialized with random data (instead, it is initialized to 0) – it seems that there's a call to read_random() missing before *seed* is initialized. This makes *seed* initialize with the higher 16 bits of the "tmp" variable, which is always 0 at the beginning of the ip_initid() execution. With *seed*=0, it is much easier to attack Mac OS X, Mac OS X Server and Darwin (the single-threaded full prediction algorithm runs in 10-20 seconds, instead of several minutes, and further improvements reduced the run time to few dozen milliseconds, which makes TCP blind data injection easier, as explained above).

5 Other algorithm variants

Naturally, any variant of algorithm X3, with *seed2* removed, or with reduced hop entropy (e.g. 2 bit entropy: $1 \le n \le 4$, or 0 bit entropy: $n=1$) is vulnerable to the exact same attack, and in fact the run time can be reduced.

The generalized algorithm in [6] does not make use of *seed2*, neither of random hops. However, it does assume an unknown M. If M happens to be even (50% of the keys), then it's possible to use phase 1 (with modulo 2 arithmetic) from the IP ID algorithm to extract w (*seed*). Then it should be possible to guess the rest of M (14 bits at most), and g ($\varphi(p-1) \le (p-1)/2 < 2^{14}$ possible values), and solve the two linear equations for a and b (over the ring modulo M).

OpenBSD ([22]) and NetBSD ([23]) use a 20 bit and 32 bit versions of A2 for IPv6. FreeBSD ([24]) uses similar X2 variants for IPv6. NetBSD ([25]) also uses a 32 bit version of A2 for RPC XID (predictable XID is a known security issue, e.g. [31]). All those variants should be considered "suspicious", with the 20 bit versions almost certainly vulnerable (predictable) using present day hardware. Note that in general, an ADD-style algorithm with n-bit random hops is vulnerable to the techniques described above if $n < \log_2(\gcd(M,N-1))$. This is because the attack can look directly at $x \bmod \gcd(M,N-1)$ and eliminate due to the excess information ($n - \log_2(\gcd(M,N-1))$) per pair (up to guessing parts of a, b and perhaps j). With the XOR variant, the techniques can be used when (2^{width}-(N-1)) is "small", and n<gcd(M, 2^{width}), for similar reasons. In other words, the techniques should work in principle (given the respective M and N constants in the various flavors), for e.g. X6/16 (2^7 divides M, and 2^{15}-(N-1)=20), A2/20 (gcd(M,N-1)=36), X2/20 (2^7 divides M, and 2^{19}-(N-1)=20), A2/32 (gcd(M,N-1)=36) and X2/32 (2^7 divides M, and 2^{31}-(N-1)=20).

It should be noted that other BSD operating systems (and possibly non BSD operating systems as well) may have copied code from OpenBSD, and as such may have become vulnerable to this attack as well.

6 BSD O/S fingerprinting (based on IP ID algorithm)

As seen above, it's easy to obtain a sequence of consecutive IP fragmentation IDs and to run the various algorithms and observe which one succeeds to match the sequence. Here is a table that maps algorithms to operating systems (at present day):

Algorithm	Byte order	Operating Systems
A0	Big endian	• OpenBSD • NetBSD (with net.inet.ip.random_id=1)
A0	(platform determined)	• OpenBSD with pf ("scrub out random-id") • NetBSD with pf ("scrub out random-id")
X2	(platform determined)	• FreeBSD (with net.inet.ip.random_id=1) • DragonFlyBSD (with net.inet.ip.random_id=1)
X2 with seed=0	(platform determined)	• Mac OS X • Mac OS X Server • Darwin
++	Big endian	• NetBSD • FreeBSD • DragonFlyBSD

7 Summary of results

- OpenBSD 3.3-4.2
  - DNS server cache poisoning (predictable DNS transaction ID).

- OpenBSD 2.6-4.2
  - Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. (predictable IP fragmentation ID) in "regular" IP packets and raw IP packets.
  - Predictable IP fragmentation ID in Ethernet-inside-IP encapsulation, IP-inside-IP encapsulation, the CARP protocol, IP multicast routing, pfsync interface protocol, packet filter (IP packet normalization), and network bridge (ICMP error packets).
- OpenBSD 2.5-4.2
  - libc resolver predictable DNS transaction ID (the source UDP port is random though).
- Mac OS X 10.0-10.5.1, Mac OS X Server 10.0-10.5.1, Darwin 1.0-9.1
  - Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. (predictable IP fragmentation ID) in "regular" IP packets and raw IP packets.
  - Predictable IP fragmentation ID in DHCP, IP multicast routing and IPsec encapsulation in IP.
- NetBSD 1.6.2-4.0
  - Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. when the packet filter is used to normalize outbound IP packets (predictable IP fragmentation ID).
  - Predictable IP fragmentation ID in the CARP protocol.
- NetBSD 1.6.2-4.0 (if the kernel flag net.inet.ip.random_id is 1)
  - Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. (predictable IP fragmentation ID) in "regular" IP packets and raw IP packets.
  - Predictable IP fragmentation ID in IP multicast routing, IP-inside-IP encapsulation and IPsec encapsulation in IP.
- FreeBSD 4.4-7.0
  - Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. when the packet filter is used to normalize outbound IP packets (predictable IP fragmentation ID).
  - Predictable IP fragmentation ID in the pfsync interface protocol.
- FreeBSD 4.4-7.0 (if the kernel flag net.inet.ip.random_id is 1)
  - Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. (predictable IP fragmentation ID) in "regular" IP packets and raw IP packets.
  - Predictable IP fragmentation ID in the CARP protocol, IP multicast routing, SCTP, IP-inside-IP encapsulation and IPv6 over GRE.
- DragonFlyBSD 1.0-1.10.1 (if the kernel flag net.inet.ip.random_id is 1)

- Idle-scanning, O/S fingerprinting, host alias detection, traffic analysis, TCP blind data injection, etc. (predictable IP fragmentation ID) in "regular" IP packets and raw IP packets.
- Predictable IP fragmentation ID in the CARP protocol, IP multicast routing, and IP-inside-IP encapsulation.

8 Workarounds

It is possible to prevent a host from being used as a zombie for idle-scanning by filtering (dropping) incoming, "unexpected" SYN+ACK TCP packets to the host. This can be done using a stateful inspection firewall, as hinted in [11] and in other sources.

Usage of Path Maximum Transmission Unit (PMTU) Discovery (RFC 1191 – [30]) and the Don't-Fragment (DF) IP flag can reduce the risk of TCP blind data injection ([27]). A modern host is likely to have PMTU discovery turned on by default (it may be turned off though e.g. if the host cannot receive ICMP packets). This workaround however is incomplete: it cannot be used when incoming ICMP packets are blocked from reaching the host, and it does not guarantee that packets are not fragmented by a non-compliant device on the path.

For other issues, no workaround is hereby suggested.

9 Conclusions

9.1 OpenBSD's DNS server cache poisoning vulnerability

To quote from [1] with the necessary adaptations, it is saddening to realize that 10-15 years after the dangers of predictable DNS transaction ID were discovered, still one of the operating systems considered most secure (OpenBSD) contains a DNS cache server that does not incorporate strong transaction ID generation. It is particularly surprising that the transaction ID mechanism in use by OpenBSD for its DNS server is not based on industrial grade cryptographic algorithms.

The paper demonstrated that the "classic" DNS poisoning attack is still applicable to OpenBSD DNS server. The attack described is far more effective than any attack previously described for OpenBSD DNS server. 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 hundreds of queries, rendering the birthday attack almost ineffective when Split-Split DNS configuration is used.

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

9.2 Multiple BSD operating systems IP fragmentation ID vulnerability

Exploitations of the predictability of the IP fragmentation ID were made public almost a decade ago. Yet again, many operating systems (OpenBSD, Mac OS X, Mac OS X Server, Darwin) implemented a weak randomization algorithm for the IP fragmentation ID (with Mac OS X, Mac OS X Server and Darwin code containing an implementation bug that further weakens the algorithm!), while others (NetBSD, FreeBSD and DragonFlyBSD) do not randomize this field at all by default, and provide a kernel flag (net.inet.ip.random_id) that enables randomization through the weak algorithm.

The paper demonstrated that the O/S fingerprinting, host alias detection, traffic analysis, idle scanning and (in some cases) TCP blind data injection are still applicable to all the above mentioned operating systems.

Again, usage of industrial strength cryptographic algorithms is recommended for the IP fragmentation ID generation.

10 Disclosure timeline

October 2007-early November 2007 – Informal discussions with OpenBSD personnel regarding the strength of OpenBSD's DNS transaction ID PRNG.

November 21st-22nd 2007 – Formal report to OpenBSD, NetBSD, FreeBSD, DragonFlyBSD and Apple.

February 6th, 2007 – Paper announcement.

11 Vendor/product status

11.1 FreeBSD

FreeBSD committed a solution to the IPv4 ID to their source code tree on February 6th, 2008, simultaneously with the release of this paper ([33]).

11.2 Apple Mac OS X, Mac OS X Server and Darwin

On January 23rd, 2008 (two months after they were notified), Apple's Product Security Team provided the following statement:

```
Follow-up:  37108015

Hello Amit,

Given the high degree of affect that a change in this area
could have to system stability and compatibility, we cannot
provide you with a timeframe.  Feel free to publish your
advisory if you feel that you need to.  We are working to
address the issue, but at this time we simply cannot give you a
timeframe for a fix to be made available.

Best regards,

Apple Product Security Team
```

11.3 OpenBSD

On December 18th, 2007, OpenBSD's coordinator stated, in an email, that "[OpenBSD is] completely uninterested in the problem" and that "[the] problem […] is completely irrelevant in the real world". This is in direct contrast to statements and opinions made by the OpenBSD team recently, e.g. [4], [5] and [26].

11.4 NetBSD

NetBSD committed a solution to the IPv4 ID to their source code tree on February 6th, 2008 ([34]).

11.5 DragonFlyBSD

DragonFlyBSD committed a solution to the IPv4 ID to their source code tree on November 22nd, 2007 ([32]).

12 References

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

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

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

[2] "Source port allocation and named(8)" (OpenBSD-misc mailing list submission), Darren Spruell, August 3rd, 2007

http://marc.info/?l=openbsd-misc&m=118616429600518&w=2

[3] "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

[4] "Re: OpenBSD & BIND 9 cache poisoning" ("Undeadly" posting), Theo de Raadt (OpenBSD project coordinator), July 26th, 2007

http://undeadly.org/cgi?action=article&sid=20070725193920&pid=15

[5] "OpenBSD & BIND 9 cache poisoning" (OpenBSD-misc mailing list post), Jakob Schlyter, July 25th, 2007

http://marc.info/?l=openbsd-misc&m=118539211412877&w=2

[6] "Cryptography in OpenBSD: An Overview" (Usenix 1999 paper), Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, Niels Provos (The OpenBSD Project), June 1999

http://www.openbsd.org/papers/crypt-paper.ps (section 3.3)

[7] "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/

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

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

[9] "src/usr.sbin/bind/lib/isc/lcg.c - view - 1.2" (OpenBSD source tree)

http://www.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/bind/lib/isc/lcg.c?rev=1.2&content-type=text/x-cvsweb-markup

[10] "new tcp scan method" (BugTraq mailing list), "antirez" (Salvatore Sanfilippo – see http://invece.org/), December 18th, 1998

http://seclists.org/bugtraq/1998/Dec/0079.html

[11] "Idle Scanning and Related IPID Games" by "Fyodor" (Gordon Lyon – see http://insecure.org/fyodor/), September 15th, 2002

http://insecure.org/nmap/idlescan.html

[12] “src/sys/netinet/ip_id.c - view - 1.8” (OpenBSD source tree), December 10th, 2003

http://www.openbsd.org/cgi-bin/cvsweb/src/sys/netinet/ip_id.c?rev=1.8&content-type=text/x-cvsweb-markup

[13] “src/sys/netinet/in_pcb.c - view - 1.90” (OpenBSD source tree), function in_pcbbind()

http://www.openbsd.org/cgi-bin/cvsweb/src/sys/netinet/in_pcb.c?rev=1.90&content-type=text/x-cvsweb-markup

[14] “src/sys/netinet/ip_id.c - view - 1.2” (OpenBSD source tree), August 26th, 1999

http://www.openbsd.org/cgi-bin/cvsweb/src/sys/netinet/ip_id.c?rev=1.2&content-type=text/x-cvsweb-markup

[15] “Dell PowerEdge SC1430 Server Featured Systems” (Dell website)

http://www.dell.com/content/products/features.aspx/pedge_sc1430?c=us&cs=04&l=en&s=bsd

[16] “src/sys/netinet/ip_id.c - view - 1.14” (OpenBSD source tree)

http://www.openbsd.org/cgi-bin/cvsweb/src/sys/netinet/ip_id.c?rev=1.14&content-type=text/x-cvsweb-markup

[17] “src/sys/netinet/ip_id.c - view - 1.8.18.2” (NetBSD source tree)

http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/netinet/ip_id.c?rev=1.8.18.2&content-type=text/x-cvsweb-markup

[18] “src/lib/libc/net/res_random.c - view - 1.16” (OpenBSD source tree)

http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libc/net/res_random.c?rev=1.16&content-type=text/x-cvsweb-markup

[19] “sys/bsd/netinet/ip_id.c” (Darwin 9.0 source tree copy in watson.org website)

http://fxr.watson.org/fxr/source/bsd/netinet/ip_id.c?v=xnu-1228

[20] “File: [FreeBSD]/src/sys/netinet/ip_id.c Revision 1.9” (FreeBSD source tree)

http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet/ip_id.c?rev=1.9;content-type=text%2Fx-cvsweb-markup

[21] "File: [dragonfly]/src/sys/netinet/ip_id.c (download) Revision 1.6" (DragonFlyBSD source tree)

http://www.dragonflybsd.org/cvsweb/src/sys/netinet/ip_id.c?rev=1.6&content-type=text/x-cvsweb-markup

[22] "src/sys/netinet6/ip6_id.c - view - 1.4" (OpenBSD source tree)

http://www.openbsd.org/cgi-bin/cvsweb/src/sys/netinet6/ip6_id.c?rev=1.4&content-type=text/x-cvsweb-markup

[23] "src/sys/netinet6/ip6_id.c - view - 1.13.18.2" (NetBSD source tree)

http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/netinet6/ip6_id.c?rev=1.13.18.2&content-type=text/x-cvsweb-markup

[24] "File: [FreeBSD]/src/sys/netinet6/ip6_id.c Revision 1.8" (FreeBSD source tree)

http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet6/ip6_id.c?rev=1.8;content-type=text%2Fx-cvsweb-markup

[25] "File: [cvs.netbsd.org]/src/lib/libc/rpc/__rpc_getxid.c (download) Revision 1.3" (NetBSD source tree)

http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/rpc/__rpc_getxid.c?rev=1.3&content-type=text/x-cvsweb-markup

[26] "Time is not a secret: Network Randomness in OpenBSD" (Asia BSD Conference 2007), Ryan McBride

http://www.openbsd.org/papers/asiabsdcon07-network_randomness/

[27] "A new TCP/IP blind data injection technique?" (BugTraq mailing list post), Michal Zalewski, December 10th, 2003

http://www.securityfocus.com/archive/1/347130

[28] "Breaking the checksum (a new TCP/IP blind data injection technique)" (BugTraq mailing list post), Michal Zalewski, December 14th, 2003

http://www.securityfocus.com/archive/1/347556

[29] "Re: A new TCP/IP blind data injection technique?" (BugTraq mailing list post), Kris Kennaway, December 10th, 2003

http://www.securityfocus.com/archive/1/347216

[30] "Path MTU Discovery" (IETF RFC 1191), Jeffrey Mogul and Steve Deering, November 1990

http://tools.ietf.org/rfc/rfc1191.txt

[31] "Solaris rpcbind tricks" (BugTraq mailing list post), Ivan Acre, August 18th, 1999

http://seclists.org/bugtraq/1999/Aug/0211.html

[32] "File: [dragonfly]/src/sys/netinet/ip_id.c (download) Revision 1.7" (DragonFlyBSD source tree)

http://www.dragonflybsd.org/cvsweb/src/sys/netinet/ip_id.c?rev=1.7&content-type=text/x-cvsweb-markup

[33] "File: [FreeBSD]/src/sys/netinet/ip_id.c Revision 1.10" (FreeBSD source tree)

http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet/ip_id.c?rev=1.10;content-type=text%2Fx-cvsweb-markup

[34] "src/sys/netinet/ip_id.c - view – 1.12" (NetBSD source tree)

http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/netinet/ip_id.c?rev=1.12&content-type=text/x-cvsweb-markup

Appendix A – **Algorithm X3** (OpenBSD DNS server transaction ID, old OpenBSD libc DNS resolver transaction ID) **and X2** (Mac OS X, Mac OS X Server and Darwin IP fragmentation ID) prediction

This C/C++ program can be used for next value prediction: in this mode, the sequence is provided on file, and an additional value, "last" is provided in the command line. The program produces a list of candidates (8-16, typically 8) for the output of the X3 algorithm right after "last", and 4-8 candidates for X2. The choice between X2 and X3 is through a compilation flag (flag "N"). There's also compilation flag support for the Mac OS X, Mac OS X Server and Darwin seed initialization bug ("FORCE_SEED_0"). Finally, when host-order serialization takes place on a little-endian platform (e.g. Intel x86 architecture), the flag SWAP_BYTES must be defined. FreeBSD uses host order for the IP fragmentation ID field, except for pfsync (which uses network order). DragonFlyBSD uses host order for this field except for SCTP. Mac OS X, Mac OS X Server and Darwin always use host order for this field.

The algorithm needs at least 12-15 consecutive IDs (in the file) to be able to work properly (in X3 mode). Using the X2 mode with FORCE_SEED_0 yields a very quick run (less than 10 seconds on a fast machine).

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

/* choose exact flavor here */


/* OpenBSD BIND 9 DNS transaction ID prediction */

/*
#define N 3
*/



/* Mac OS X, Mac OS X Server, Darwin IP ID prediction */

/*
#define N 2
#define FORCE_SEED_0
*/
```

```
/* FreeBSD, DragonFlyBSD IP ID prediction (when kernel flag net.inet.ip.random_id is
1, or in packet filter IP normalization mode) */

/*
#define N 2
*/


/* In host-order fields (*ALL* IP ID fields except FreeBSD's pfsync and DragonFlyBSD's
SCTP) you need
to turn on SWAP_BYTES if the target hardware is little-endian (e.g. Intel x86
architecture) */

/*
#define SWAP_BYTES
*/


#ifndef N
#error "N (and possibly SWAP_BYTES, FORCE_SEED_0) must be specified"
#endif

/* maximum data */
#define MAX_DATA 30



/* code almost as-is from OpenBSD 4.2 */


#define RU_GEN 2              /* Starting generator */
#define RU_N   32749          /* RU_N-1 = 2*2*3*2729 */
#define RU_AGEN       7               /* determine ru_a as RU_AGEN^(2*rand) */
#define RU_M   31104           /* RU_M = 2^7*3^5 - don't change */

#define PFAC_N 3
static const int pfacts[PFAC_N] = {
       2,
       3,
       2729
};

/*
 * Do a fast modular exponation, returned value will be in the range
 * of 0 - (mod-1)
 */
int pmod(int gen, int exp, int mod)
{
       int s, t, u;

       s = 1;
       t = gen;
       u = exp;

       while (u) {
              if (u & 1)
                     s = (s*t) % mod;
              u >>= 1;
              t = (t*t) % mod;
       }
       return (s);
}

/* End of OpenBSD code */


typedef struct
{
       int cand[0x8000];   /* paranoid definition - in real life, 8-16 candidates are
expected */
       int size;
} candidate_list;
```

```
void calc_cand(int g,int log_g[0x8000],int seed,int seed2,int A,int B,int msb,int
last,candidate_list* list)
{
        int x,i,cand,j;

        printf("INFO: algorithm parameters extracted: g=0x%04x, seed=0x%04x,
seed2=0x%04x, a=0x%04x, b=0x%04x, msb=0x%04x\n",g,seed,seed2,A,B,msb);

        x=log_g[last^seed]^seed2;
        for (i=0;i<(1<<N);i++)
        {
                x=(A*x+B) % RU_M;
                cand=(pmod(g,x^seed2,RU_N)^seed)|msb;
                for (j=0;j<list->size;j++)
                {
                        if (list->cand[j]==cand)
                        {
                                break;
                        }
                }
                if (j==list->size)
                {
                        printf("INFO: Adding candidate: 0x%04x.\n",cand);
                        list->cand[list->size]=cand;
                        list->size++;
                }
        }
}

/* phase 3 */
void phase3(int data[],int len,int g,int log_g[0x8000],int seed,int seed2,int A,int
B,int hop[],int msb,int last,candidate_list* list)
{
        int a,b,seed2_,q;
        int x,y;
        int i;

        for (seed2_=seed2;seed2_<0x8000;seed2_+=0x40)
        {
                for (a=A;a<RU_M;a+=0x80)
                {
                        for (b=B;b<RU_M;b+=0x80)
                        {
                                for (q=1;q<len;q++)
                                {
                                        x=log_g[seed^data[q-1]]^seed2_;
                                        if (x>=RU_M)
                                        {
                                                break;
                                        }
                                        for (i=0;i<hop[q-1];i++)
                                        {
                                                x=(a*x+b) % RU_M;
                                        }
                                        y=log_g[seed^data[q]]^seed2_;
                                        if (y>=RU_M)
                                        {
                                                break;
                                        }
                                        if (x!=y)
                                        {
                                                break;
                                        }
                                }
                                if (q==len)
                                {
                                        calc_cand(g,log_g,seed,seed2_,a,b,msb,last,list);
                                }
                        }
                }
        }
}
```

```
/* phase 2 - go over the rest 3 bits of a, b and seed2. */
void phase2(int data[],int len,int g,int log_g[0x8000],int seed,int seed2,int B,int
msb,int last,candidate_list* list)
{
        int x_old,x_new,q,hop[MAX_DATA-1];
        int Binv[16];
        int seed2_,B_,A,x,skip;

        /* inverse values, mod 16 */
        Binv[1]=1; Binv[3]=11; Binv[5]=13; Binv[7]=7; Binv[9]=9; Binv[11]=3;
Binv[13]=5; Binv[15]=15;

        /* First, calculate the hops */
        x_old=log_g[seed^data[0]]^seed2;
        for (q=1;q<len;q++)
        {
                x_new=log_g[seed^data[q]]^seed2;
                hop[q-1]=(Binv[B]*((x_new-x_old) & 0xF)) & 0xF;
                x_old=x_new;
        }

        for (seed2_=seed2;seed2_<0x20;seed2_+=8)
        {
                for (B_=B;B_<0x80;B_+=0x10)
                {
                        for (A=1;A<0x80;A+=0x10)
                        {
                                for (q=1;q<len;q++)
                                {
                                        x=log_g[seed^data[q-1]]^seed2_;
                                        for (skip=0;skip<hop[q-1];skip++)
                                        {
                                                x=(A*x+B_) & 0x7F;
                                        }
                                        if (x!=((log_g[seed^data[q]]^seed2_) & 0x7F))
                                        {
                                                break;
                                        }
                                }
                                if (q==len)
                                {

        phase3(data,len,g,log_g,seed,seed2_,A,B_,hop,msb,last,list);

        phase3(data,len,g,log_g,seed,0x3F^seed2_,A,(0x3F*(1-A)-B_) &
0x7F,hop,msb,last,list);
                                }
                        }
                }
        }
}

/* Find seed, seed2's 3 lsb's, and b's 3 bits (b's lsb is always 1) */
void phase1(int data[],int len,int last,candidate_list* list)
{
        int i,j,g;
        int x,p,log_g[0x8000];
        int Binv[16];
        int seed, seed2;
        int B,Bi,q;
        int x_old,x_new,x_delta,n;
        int cand=0;
        int msb,m;

        /* inverse values, mod 16 */
        Binv[1]=1; Binv[3]=11; Binv[5]=13; Binv[7]=7; Binv[9]=9; Binv[11]=3;
Binv[13]=5; Binv[15]=15;

        /* truncate list */
        list->size=0;
        /* find and remove msb */
        msb=data[0] & 0x8000;
        for (m=0;m<len;m++)
        {
                if ((data[m] & 0x8000) != msb)
                {
                        printf ("ERROR: transition to new key detected at data[%d].
Can't process data.\n",m);
                        return;
                }
                data[m]&=0x7FFF;
        }
        if ((last & 0x8000)!=msb)
        {
                printf ("ERROR: transition to new key detected for last value. Can't
process data.\n",m);
                return;
        }
        last&=0x7FFF;

        /* main loop */
        for (j=0;j<RU_N;j++)
        {
                /* Check whether j is "good" */
                for (i=0;i<PFAC_N;i++)
                {
                        if (j % pfacts[i] == 0)
                        {
                                break;
                        }
                }
                if (i<PFAC_N)
                {
                        /* j is divisible by a factor of RU_N, so it's not good */
                        continue;
                }

                g=pmod(RU_GEN, j, RU_N);

                /* Create log_g table */
                x=1;
                for (p=0;p<0x8000;p++)
                {
                        log_g[p]=-1;
                }
                for (p=0;p<RU_N-1;p++)
                {
                        log_g[x]=p;
                        x=(x*g) % RU_N;
                }

                /* we're ready, let's go... */
#ifdef FORCE_SEED_0
                seed=0;
#else
                for (seed=0;seed<0x8000;seed++)
#endif
                {
                        for (seed2=0;seed2<4;seed2++)
                        {
                                for (B=1;B<16;B+=2)
                                {
                                        Bi=Binv[B];
                                        x_old=log_g[seed^data[0]]^seed2;
                                        for (q=1;q<len;q++)
                                        {
                                                x_new=log_g[seed^data[q]]^seed2;
                                                x_delta=(x_new-x_old) & 0xF;
                                                n=(Bi*x_delta) & 0xF;
                                                /* n is the number of hops */
                                                if ((n==0) || (n>(1<<N)))
                                                {
                                                        break;
                                                }
                                                x_old=x_new;
                                        }
                                        if (q==len)
                                        {

        phase2(data,len,g,log_g,seed,seed2,B,msb,last,list);
                                                phase2(data,len,g,log_g,seed,7^seed2,16-
B,msb,last,list);
                                        }
                                }
                        }
                }
        }
}

void swap_bytes(int *x)
{
        int tmp;

        tmp=((*x & 0xFF00)>>8)|((*x & 0xFF)<<8);
        *x=tmp;
}

int main(int argc, char* argv[])
{
        int data[MAX_DATA],num,last;
        FILE *fp;
        char line[100];
        candidate_list list;
        clock_t t1,t2;
        int i;

        if (argc<3)
        {
                printf("Usage: \n");
                printf("   %s file last   --- predict algorithm X3/X2 with data in file
(hex, one ID per line), what comes after last (hex)?\n",argv[0]);
                printf("\n(The X3 algorithm is in use for OpenBSD DNS transaction ID,
the X2 algorithm is in use for Mac OS X, Mac OS X Server, Darwin, FreeBSD and
DragonFlyBSD for IP fragmentation ID)\n");
                printf("\n\nCompiled as: X%d",N);
#ifdef SWAP_BYTES
                printf(" (rverese byte order)");
#endif
                printf("\n\n");
                return -1;
        }

        if (sscanf(argv[2],"%x",&last)!=1)
        {
                printf("PARAMETER ERROR: last is not a hex value.\n");
                return -1;
        }
#ifdef SWAP_BYTES
        swap_bytes(&last);
#endif
        if ((last<0) || (last>0xFFFF))
        {
                printf("PARAMETER ERROR: last is out of range.\n");
                return -1;
        }

        fp=fopen(argv[1],"r");
        if (fp==NULL)
        {
                printf("ERROR: Error opening file %s.\n",argv[1]);
                return -1;
        }

        for(num=0;(num<MAX_DATA) && (!feof(fp));num++)
        {
                if (fgets(line,sizeof(line),fp)==NULL)
                {
                        break;
                }
                if(sscanf(line,"%x",&(data[num]))!=1)
                {
                        printf("ERROR: At line %d, line is <%s>, could not read hex
data.\n",num+1,line);
                        return -1;
                }
                printf("INFO: At line %d, read 0x%04x.\n",num+1,data[num]);
#ifdef SWAP_BYTES
                swap_bytes(&data[num]);
#endif
                if ((data[num]<0) || (data[num]>0xFFFF))
                {
                        printf("ERROR: At line %d, data is 0x%x - it is out of
range.\n",num+1,data[num]);
                        return -1;
                }
        }
        if (!feof(fp))
        {
                printf("WARNING: only %d items read from file, additional data is
ignored (probably doesn't add much anyway).\n",num);
        }
        fclose(fp);

#if (N==3)
        if (num<15)
        {
                printf("WARNING: less than 15 items found in file. The algorithm may
suggest many false positives.\n");
        }
#else if (N==2)
        if (num<8)
        {
                printf("WARNING: less than 8 items found in file. The algorithm may
suggest many false positives.\n");
        }
#endif

        printf("\n");
        printf("INFO: Processing %d entries.\n\n",num);

        t1=clock();
        phase1(data,num,last,&list);
        t2=clock();

        printf("\nOUTPUT: %d candidate(s):  ",list.size);
        for (i=0;i<list.size;i++)
        {
#ifdef SWAP_BYTES
                swap_bytes(&list.cand[i]);
#endif
                printf("0x%04x ",list.cand[i]);
        }
        printf("\n\n");
        printf("INFO: Time elapsed: %f seconds.\n",((double)(t2-t1))/CLOCKS_PER_SEC);

        return 0;
}
```

Appendix B – **Algorithm A0** (OpenBSD and NetBSD IP fragmentation ID, OpenBSD DNS resolver transaction ID) Detection/hop calculation/prediction

This C/C++ program can be used in 3 modes:

- Algorithm detection: it detects whether the provided data is produced by A0 (calculating A0's seed as a by product). This mode takes few milliseconds to run. It can be used to fingerprint OpenBSD and NetBSD according to the IP ID sequence they generate.

- Hop calculation (mod 12): in this mode, a sequence is provided on file, as well as two data points (v1, v2) in the command line. The program calculates seed from the data in file, and then it outputs how many hops (mod 12) there were between the two values v1 and v2 (which can be part of the data on file, but not necessarily). This mode takes few milliseconds to run. It can be used for IdleScan (detecting whether the target machine has produced an additional IP ID) with OpenBSD and NetBSD.

- Next value prediction: in this mode, the sequence is provided on file, and an additional value, "last" is provided in the command line. The program produces a list of candidates (typically one) for the output of the A0 algorithm right after "last". This mode can be used for resolver spoofing (openBSD) and TCP blind data injection (OpenBSD, NetBSD).

The algorithm needs at least 7 consecutive IP IDs (in the file) for prediction, and 8 for detection/hop calculation to be able to work properly. Detection and hop calculation are very fast (milliseconds). Prediction can take few seconds on a slow machine when the sequence length is 9.

When host-order serialization takes place on a little-endian platform (e.g. Intel x86 architecture), the flag SWAP_BYTES must be defined. Host-order serialization takes place in OpenBSD for the IP fragmentation ID in the packet filter and in NetBSD's IP fragmentation ID field for packet filter, CARP and IP-inside-IP encapsulation.

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

/* In host-order fields (OpenBSD's packet filter, NetBSD's packet filter, CARP and IP-
inside-IP encapsulation) you need
to turn on SWAP_BYTES if the target hardware is little-endian (e.g. Intel x86
architecture) */

/*
#define SWAP_BYTES
*/
```

```
/* maximum data */
#define MAX_DATA 20



/* modes of operation */
#define DETECT 0
#define HOPS 1
#define PREDICT 2



/* code almost as-is from OpenBSD 4.2 */

#define RU_OUT  180           /* Time after wich will be reseeded */
#define RU_MAX 30000          /* Uniq cycle, avoid blackjack prediction */
#define RU_GEN 2              /* Starting generator */
#define RU_N   32749          /* RU_N-1 = 2*2*3*2729 */
#define RU_AGEN       7               /* determine ru_a as RU_AGEN^(2*rand) */
#define RU_M   31104          /* RU_M = 2^7*3^5 - don't change */

#define PFAC_N 3
const static int pfacts[PFAC_N] = {
       2,
       3,
       2729
};

/*
 * Do a fast modular exponation, returned value will be in the range
 * of 0 - (mod-1)
 */
int pmod(int gen, int exp, int mod)
{
       int s, t, u;

       s = 1;
       t = gen;
       u = exp;

       while (u) {
               if (u & 1)
                       s = (s*t) % mod;
               u >>= 1;
               t = (t*t) % mod;
       }
       return (s);
}

/* End of OpenBSD code */


typedef struct
{
       int cand[0x8000];   /* paranoid definition - in real life, ~1 candidate is
expected */
       int size;
} candidate_list;

/* prepare a candidate for the next value (after "last") */
void calc_cand(int g,int x_clean[],int seed,int seed2,int a,int b,int msb,int
last,candidate_list* list, int log_2[0x8000],int inv_j)
{
       int x,cand,j;

       x=((RU_N-1)+(inv_j*log_2[last^seed])-seed2)%(RU_N-1);
       if (x>=RU_M)
       {
               return;
       }
       x=(a*x+b) % RU_M;
```

```
        cand=(pmod(g,x+seed2,RU_N)^seed)|msb;
        for (j=0;j<list->size;j++)
        {
                if (list->cand[j]==cand)
                {
                        break;
                }
        }
        if (j==list->size)
        {
                printf("INFO: Adding candidate: 0x%04x.\n",cand);
                list->cand[list->size]=cand;
                list->size++;
        }
}

/* phase_c: solve the linear equations for a and b */
void phase_c(int data[],int len,int seed,int g,int x_plus_seed2[],int seed2,int B,int
msb,int last,candidate_list* list,int log_2[0x8000],int inv_j)
{
        int x;
        int x_clean[MAX_DATA];
        int i,q;
        int a,b,tmp,inv_tmp;
        int y;

        for (i=0;i<len;i++)
        {
                x_clean[i]=((RU_N-1)+x_plus_seed2[i]-seed2)%(RU_N-1);
                if (x_clean[i]>=RU_M)
                {
                        return;
                }
        }

        tmp=(RU_M+x_clean[1]-x_clean[0]) % RU_M;
        if (((tmp % 2)==0) || ((tmp % 3)==0))
        {
                /* this cannot happen in the corrects solution */
                return;
        }

        /* a's coefficient (tmp) is invertible mod RU_M so solve the two linear
equations for a and b */

        /* Lazy calculation of tmp^(-1) mod RU_M... t^phi(RU_M)=1 mod RU_M, so
t^phi((RU_M)-1)=t^(-1), and phi(RU_M)=RU_M/3  */
        inv_tmp=pmod(tmp,RU_M/3-1,RU_M);

        a=(inv_tmp*(RU_M+x_clean[2]-x_clean[1])) % RU_M;
        b=(RU_M*RU_M+x_clean[1]-a*x_clean[0]) % RU_M;

        for (q=1;q<len;q++)
        {
                x=(a*x_clean[q-1]+b) % RU_M;
                y=x_clean[q];
                if (x!=y)
                {
                        break;
                }
        }
        if (q==len)
        {
                calc_cand(g,x_clean,seed,seed2,a,b,msb,last,list,log_2,inv_j);
        }
}


/* phase_b: enumerate over g and seed2 - few g's will be found, each one with many
seed2's. b mod 16 is found too (but it's not used). */
void phase_b(int data[],int len,int seed,int msb,int last,candidate_list* list,int
log_2[0x8000])
{
        int i,j,g;
        int seed2;
        int v_old,v_new,q,b;
        int inv_j;
        int x_plus_seed2[MAX_DATA];

        for (j=0;j<RU_N-1;j++)
        {
                /* Check whether j is "good" */
                for (i=0;i<PFAC_N;i++)
                {
                        if (j % pfacts[i] == 0)
                        {
                                break;
                        }
                }
                if (i<PFAC_N)
                {
                        /* j is divisible by a factor of RU_N, so it's not good */
                        continue;
                }

                g=pmod(RU_GEN, j, RU_N);

                inv_j=pmod(j,10912-1,RU_N-1);  /* Lazy calculation of j^(-1)... */

                for (i=0;i<len;i++)
                {
                        x_plus_seed2[i]=(log_2[data[i]^seed]*inv_j) % (RU_N-1);
                }

                for(seed2=0;seed2<RU_N-1;seed2++)
                {
                        v_old=x_plus_seed2[0];
                        /* ugly but fast modulo since the value is < 2*(RU_N-1) */
                        if (v_old>=seed2)
                        {
                                v_old-=seed2;
                        }
                        else
                        {
                                v_old+=(RU_N-1)-seed2;
                        }

                        v_new=x_plus_seed2[1];
                        /* ugly but fast modulo since the value is < 2*(RU_N-1) */
                        if (v_new>=seed2)
                        {
                                v_new-=seed2;
                        }
                        else
                        {
                                v_new+=(RU_N-1)-seed2;
                        }

                        b=(v_new-v_old) & 0xF;

                        v_old=v_new;
                        for (q=2;q<len;q++)
                        {
                                v_new=x_plus_seed2[q];
                                /* ugly but fast modulo since the value is < 2*(RU_N-1)
*/
                                if (v_new>=seed2)
                                {
                                        v_new-=seed2;
                                }
                                else
                                {
                                        v_new+=(RU_N-1)-seed2;
                                }

                                if (((v_new-v_old) & 0xF) !=b)
```

```
                              {
                                      break;
                              }
                              v_old=v_new;
                      }
                      if (q==len)
                      {
        phase_c(data,len,seed,g,x_plus_seed2,seed2,b,msb,last,list,log_2,inv_j);
                      }
              }
      }
}

/* phase_a: enumerate over seed, find the correct one */
void phase_a(int mode,int data[],int len,int new_data[],candidate_list* list)
{
        int msb,m;
        int x,i,log2[0x8000];
        int seed,delta,q;
        int new_data_len;

        if (mode==PREDICT)
        {
                /* truncate list */
                list->size=0;
        }

        /* find and remove msb */
        msb=data[0] & 0x8000;
        for (m=0;m<len;m++)
        {
                if ((data[m] & 0x8000) != msb)
                {
                        printf ("ERROR: transition to new key detected at data[%d].
Can't process data.\n",m);
                        return;
                }
                data[m]&=0x7FFF;
        }

        switch(mode)
        {
        case DETECT: new_data_len=0; break;
        case HOPS: new_data_len=2; break;
        case PREDICT: new_data_len=1; break;
        default: printf("ERROR: unknown mode: %d\n",mode);
                return;
        }

        for (m=0;m<new_data_len;m++)
        {
                if ((new_data[m] & 0x8000)!=msb)
                {
                        printf ("ERROR: transition to new key detected at additional
data [%d]. Can't process data.\n",m);
                        return;
                }
                new_data[m]&=0x7FFF;
        }

        /* prepare log2 table */
        for (i=0;i<0x8000;i++)
        {
                log2[i]=-1;
        }
        x=1;
        for (i=0;i<(RU_N-1);i++)
        {
                log2[x]=i;
                x=(2*x) % RU_N;
        }
        for (seed=0;seed<0x8000;seed++)
        {
                /* The artificial addition of 12*0x8000 is just to make sure the whole
thing is positive before taking modulo */
                delta=(12*0x8000+log2[data[1]^seed]-log2[data[0]^seed]) % 12;
                /* Quick elimination */
                if (((delta & 1)==0) || (delta==3) || (delta==9))
                {
                        continue;
                }
                for (q=2;q<len;q++)
                {
                        if (((12*0x8000+log2[data[q]^seed]-log2[data[q-1]^seed]) %
12)!=delta)
                        {
                                break;
                        }
                }
                if (q==len)
                {
                        printf("\nOUTPUT: Algorithm A0 detected!
(seed=0x%04x).\n\n",seed);
                        if (mode==HOPS)
                        {
                                /* mod 12, delta==1/5/7/11 means delta==delta**(-1), so
to invert it, we multiply by it. */
                                printf("\nOUTPUT: There were %d hop(s) (mod 12) from
0x%04x to 0x%04x.\n\n",(12*0x8000+((log2[new_data[1]^seed]-
log2[new_data[0]^seed])*delta)) % 12,new_data[0],new_data[1]);
                        }
                        else if (mode==PREDICT)
                        {
                                phase_b(data,len,seed,msb,new_data[0],list,log2);
                        }
                }
        }
}

void swap_bytes(int *x)
{
        int tmp;

        tmp=((*x & 0xFF00)>>8)|((*x & 0xFF)<<8);
        *x=tmp;
}

int main(int argc, char* argv[])
{
        int data[MAX_DATA],new_data[2];
        int mode;
        char line[100];
        int num;
        FILE *fp;
        candidate_list list;
        int i;
        clock_t t1,t2;
        double tmp;

        if (argc<3)
        {
                printf("Usage: \n");
                printf("   %s -d file        --- detect algorithm A0 with data in file
(hex, one ID per line)\n",argv[0]);
                printf("   %s -h file v1 v2  --- calculate hops (mod 12) for algorithm
A0 with data in file (hex, one ID per line), how many hops between v1 and
v2?\n",argv[0]);
                printf("   %s -p file last   --- predict algorithm A0 with data in file
(hex, one ID per line), what comes after last (hex)?\n",argv[0]);
                printf("\n(The A0 algorithm is in use for OpenBSD,NetBSD IP ID, and for
OpenBSD libc resolver DNS ID)\n");
                return -1;
        }
        if (strcmp(argv[1],"-d")==0)
        {
                mode=DETECT;
        }
        else if (strcmp(argv[1],"-h")==0)
        {
                mode=HOPS;
        }
        else if (strcmp(argv[1],"-p")==0)
        {
                mode=PREDICT;
        }
        else
        {
                printf("PARAMETER ERROR: %s is not -d/-h/-p.\n",argv[1]);
                return -1;
        }

        if (mode==HOPS)
        {
                if (argc!=5)
                {
                        printf("PARAMETER ERROR: mode==HOPS, but no two values
provided.\n");
                        return -1;
                }

                if (sscanf(argv[3],"%x",&(new_data[0]))!=1)
                {
                        printf("PARAMETER ERROR: v0 is not a hex value.\n");
                        return -1;
                }
                if ((new_data[0]<0) || (new_data[0]>0xFFFF))
                {
                        printf("PARAMETER ERROR: v0 is out of range.\n");
                        return -1;
                }
#ifdef SWAP_BYTES
                swap_bytes(&new_data[0]);
#endif

                if (sscanf(argv[4],"%x",&(new_data[1]))!=1)
                {
                        printf("PARAMETER ERROR: v1 is not a hex value.\n");
                        return -1;
                }
                if ((new_data[1]<0) || (new_data[1]>0xFFFF))
                {
                        printf("PARAMETER ERROR: v1 is out of range.\n");
                        return -1;
                }
#ifdef SWAP_BYTES
                swap_bytes(&new_data[1]);
#endif
        }

        if (mode==PREDICT)
        {
                if (argc!=4)
                {
                        printf("PARAMETER ERROR: mode==PREDICT, but no last value
provided.\n");
                        return -1;
                }
                if (sscanf(argv[3],"%x",&(new_data[0]))!=1)
                {
                        printf("PARAMETER ERROR: last is not a hex value.\n");
                        return -1;
                }
                if ((new_data[0]<0) || (new_data[0]>0xFFFF))
                {
                        printf("PARAMETER ERROR: last is out of range.\n");
                        return -1;
                }
#ifdef SWAP_BYTES
                swap_bytes(&new_data[0]);
#endif
        }

        fp=fopen(argv[2],"r");
        if (fp==NULL)
        {
                printf("ERROR: Error opening file %s.\n",argv[2]);
                return -1;
        }

        for(num=0;(num<MAX_DATA) && (!feof(fp));num++)
        {
                if (fgets(line,sizeof(line),fp)==NULL)
                {
                        break;
                }
                if(sscanf(line,"%x",&(data[num]))!=1)
                {
                        printf("ERROR: At line %d, line is <%s>, could not read hex 
data.\n",num+1,line);
                        return -1;
                }
                if ((data[num]<0) || (data[num]>0xFFFF))
                {
                        printf("ERROR: At line %d, data is %x - it is out of 
range.\n",num+1,data[num]);
                        return -1;
                }
                printf("INFO: At line %d, read 0x%04x.\n",num+1,data[num]);
#ifdef SWAP_BYTES
                swap_bytes(&data[num]);
#endif
        }
        if (!feof(fp))
        {
                printf("WARNING: only %d items read from file, additional data is 
ignored (probably doesn't add much anyway).\n",num);
        }
        fclose(fp);
        if (num<7)
        {
                printf("WARNING: less than 7 items found in file. The algorithm may 
suggest false positives, and detection is less reliable.\n");
        }
        printf("\n");
        printf("INFO: Processing %d entries.\n\n",num);

        if (mode==DETECT)
        {
                tmp=((double)(1<<15))/(3.0*exp(log(12.0)*(num-2)));  /* 2 to the power 
of 15 divided by 3*(12 to the power of (num-2)) */
                printf("INFO: False positive probability (for a random sequence to 
yield positive result): %13.10f.\n",(1.0-exp(-1.0*tmp)));
        }
        t1=clock();
        phase_a(mode,data,num,new_data,&list);
        t2=clock();

        if (mode==PREDICT)
        {
                printf("\nOUTPUT: %d candidate(s):  ",list.size);
                for (i=0;i<list.size;i++)
                {
#ifdef SWAP_BYTES
                        swap_bytes(&list.cand[i]);
#endif
                        printf("0x%04x ",list.cand[i]);
                }
                printf("\n\n");
        }
        printf("INFO: Time elapsed: %f seconds.\n",((double)(t2-t1))/CLOCKS_PER_SEC);
}
```

Appendix C – Additional X3 algorithm properties

C.1 Not “non repeatable”

It is obvious that the 1997 design went to a great length in order to ensure that the PRNG output does not repeat itself over short spans of time/output stream. However, with *seed2* introduced in the 1999 version ([14]), this is no longer the case. The problem is that the value provided to the pmod() function can be thought of as de-facto being taken modulo (N-1). Since (N-1)<2^{15}, and since XORing with *seed2* is not commutative with modulo (N-1), there are in fact 29,440 *seed2* values (out of the possible 32,768 values) that cause 20 collisions between x=(k XOR *seed2*) and x'=((k–(N-1)) XOR *seed2*), where k=(N-1)…2^{15}; that is, (x XOR *seed2*) mod (N-1)= (x' XOR *seed2*) mod (N-1); note that $x<M$ and $x'<M$, so it should be possible to obtain such values from the LCG.

NOTE: this was probably noticed by Niels Provos around 2003, for the IP ID PRNG (see blow), as hinted in [12].

This means that during the lifetime of a key (6000-7000 outputs of the PRNG), and when 0x0680≤*seed2*<0x7980 (probability almost 90%), it is expected to find around one collision. This was verified by simulations.

Fortunately, those collisions do not appear in real-life BIND 9 outgoing requests, due to BIND 9’s post-PRNG collision removal mechanism. For IP ID uses (algorithm X2), however, no such safety net exists, and moreover, since with IP ID the same key can be used for longer sequences (due to its shorter hops), there’s more likelihood there for collisions.

C.2 Key size, effective key, key entropy, hop randomness

The key of the algorithm comprises of a,b,g,*seed*,*seed2*, and x_0. a can take 648 values (by construction), hence it’s 9.34 bits. b can take 10368 values (by construction), hence it’s 13.34 bits. g can take 10912 values (by construction), hence it’s 13.41 bits. *seed* and *seed2* are 15 bits each. Finally, x_0 can take 31104 values, so it’s 14.92 bits. Altogether, therefore, the key contains 81.01 bits.

However, it’s trivial to reconstruct the internal state (x) given a single output and a, b, g, *seed* and *seed2*. So a key enumeration attack for a, b, g, *seed* and *seed2* can succeed very well without a-priori knowledge of x_0 or x. In other words, the effective key size may be considered as 66.09 bits (without x_0).

Moreover, looking at how b and g are constructed, it is obvious that they are not uniformly distributed. b is firstly chosen as a random even number, but if it is divisible by 3, then 2 is added to it. Looking at (b mod 6), the initial value is 1, 3 or 5. But if it happens to be 3, then it is replaced by 5. That is, the probability of (b mod 6) to be 1 is 1/3, and the probability of the same quantity to be 5 is 2/3. An almost similar situation occurs with j (from which g is driven), so around half of the possible g values have probability twice as that of the other half. This reduces the entropy of b and g by around 0.08 bits each. The total effective key entropy is, therefore around 65.93 bits. This specific result is not used in the algorithm analysis.

Note that there is also a small non-uniformity factor introduced due to the uneven wrap-around when casting a random number evenly distributed among an integral power of 2 values, modulo a number which is not a power of 2. This deviation however is almost unnoticeable (reducing the entropy of each element by less than 0.02 bits).

Another issue is with the random hops. The way they are produced for X3 results in non-uniformity. Each time the local random source (32 bits) is “empty” (more on that below), it is filled with 32 fresh random bits. This means that the first 10 hops are random between 1-8, but the last hop is random between 1-4 (this is not entirely accurate, as explained right below).

Additionally, for X3 and X2, there’s another source of non-uniformity. The emptiness of the local random pool is determined by comparing it to 0. This means that values of all zeroes for the last 1-2 hops (in a 10-11 cycle) are less likely, as they will be skipped in preference of fresh random bits.

C.3 “Global” randomness/entropy

Ignoring the highest bit (which is trivially predictable), the algorithm usually produces 6000-7000 15 bit numbers in a single key lifetime (assuming that the key is exhausted before the time limit – 180 seconds). While indeed, in the X family of algorithms, a single collision (or few collisions, in case of X2) is expected, still, when considering a single PRNG output, the algorithm is likely to produce a unique value. This means that observing 6500 values from the same key reduces the space of the next value from 32,768 to 26,268, i.e. from 15 bits of entropy to 14.68 bits of entropy. In X2 it is worse, since there are (on average) 12,000 outputs in a single key lifetime; the last output will have a reduced space of 20,769 values - 14.34 entropy bits. And in A0 it’s worst – the last output will be after 29,999 iterations, i.e. will have its space reduced to 2679 values – merely 11.39 bits of entropy (compared to the expected 15 bits).

This is a direct consequence of the scheme used to ensure non-repetition. Any algorithm that applies a cryptographic permutation over 15 bits on an advancing counter would yield the same results. However, there are other ways to achieve (limited) non-repetition than this scheme, in which no entropy is lost beyond up to 1 bit (i.e. the entropy is guaranteed to be at least 15 bits).