DNS POISONING VIA PORT EXHAUSTION

A security advisory

Roee Hay <`roeeh@il.ibm.com`> ⋄ Yair Amit <`yairam@gmail.com`>
IBM Rational Application Security Research Group

October 18, 2011

Contents

1 Introduction

1.1 DNS in a nutshell

DNS is a request/response protocol which resolves hostnames into IP addresses and vice versa. The DNS protocol is usually carried over UDP. Each DNS request is identified by a 16-bit attribute, called '*TXID*'. For a DNS response to be considered, a matching response must contain the same TXID value; if it does not it should be dropped by the resolver. The TTL attribute of the DNS response includes a "hint" at how long it should be kept in the resolver's cache, if there is one.

1.2 DNS poisoning

DNS poisoning occurs when a third-party, we'll call it Mallory, interferes with the request-response dialog between Alice (the resolver) and Bob (the DNS server). This can be achieved if Mallory knows or guesses correctly the following data:

1. The DNS request's TXID (16 bits)
2. The DNS request's UDP source port number (16 bits)
3. The IP address of Bob

In addition to that, Mallory must succeed in injecting the forged DNS response, before the legitimate one arrives from Bob. In an ideal environment the TXID and UDP source port number combine into a 32-bit random value, making a successful attack unlikely. However, by port exhausting [7] the attacked system, the nonce can be reduced to 16-bit only, which makes it more vulnerable to guessing.

2 Local poisoning in MS Windows

2.1 Environment

MS Windows includes a stub resolver service (named *"DNS Client"*) which maintains an internal cache of received responses. The DNS cache itself is shared between all users of the system. This means that if a normal (that is non-administrative) user successfully resolves a hostname, and later an administrator tries to resolve the same host, the DNS resolver will return the cached resolution.

2.2 Vulnerabilities

Two vulnerabilities exist in MS-Windows which combine to enable a normal (non-administrative) user to poison the DNS cache with a hostname of his/her choice:

1. A non-administrative user can listen on all available UDP ports of the system. As mentioned above, this will reduce the DNS requests' nonce to 16-bit only, making it feasible to conduct a DNS poisoning attack. Since MS Windows maintains a DNS cache, attacking a single host is not feasible (unless the TTL is very low), however this vulnerability can be used on its own to attack a range of domains, including non-existent domains.

2. A non-administrative user can *flush* the DNS cache, effectively neutralizing the TTL's impact on incoming responses. Although the `'ipconfig'` command (when run together with the `'/flushdns'` switch) does verify that the user has administrative rights (the verification is done by calling `advpapi32.dll!`*`CheckTokenMembership`* with `BUILTIN\Administrators`), this restriction can be bypassed either by calling `DnsApi.dll!`*`DnsFlushResolverCache`* or by directly issuing the appropriate RPC to the stub resolver service.

2.3 Impact

Since the DNS cache is shared between all users of the system, a successful DNS poisoning may be used by malicious users in the following ways:

2.3.1 Universal Cross-Site Scripting

Using the following methodology, the attacker can inject code that will run in the context of an innocent (arbitrary) domain:

1. The attacker poisons the DNS cache with the targeted domain, pointing to a server under his/her control. The bogus cache entry must remain alive while the next step takes place. This can be achieved by increasing the TTL value of the forged DNS response.
2. During the lifetime of the poisoned DNS entry, the user must be tricked into visiting the chosen (attacked) website. When he/she does so, he/she will actually see the attacker's web page. This page must then be cached by the victim's web browser (which can be achieved by adding a Cache-Control: max-age=<some big value> header to the HTTP response).
3. The attacker detects that the user has connected to his/her web server, and flushes the DNS cache of the attacked system, using the technique described above.
4. The user visits the same website, but this time it connects to the correct IP address. However, the browser will now display the attacker's cached page instead of the real one. Hence we now have the attacker's code running on the real domain (XSS).

2.3.2 Information disclosure of users' private data

The attacker may target the hostname of a website, so that when a user of the system browses to that website, he/she will see the attacker's controlled data, allowing the latter to steal cookies or conduct a phishing attack.

2.3.3 Privilege escalation

This can be achieved in several ways, including aiming at hostnames of insecure update servers, or favorite non-SSL download sites which might be used by system administrators.

2.3.4 NX domains poisoning

All payloads described in section 3.3 are also relevant here, however the attack method is different. The victim must be lured into visiting an attacker-controlled webpage, after the NX domain has been poisoned, in order to trigger the various payloads.

2.4 Proof-of-Concept

Our PoC [2] consists of two parts:

1. Poisoner: Server code which runs on an attacker-controlled machine. It must be capable of sending spoofed packets to the victim's machine. This server listens on a designated port and accepts commands.
2. Resolver: Code that runs on the attacked system as a limited (non-Admin) user, consumes all available UDP ports of the system, leaving only a single port open. Iteratively it does the following:
   (a) Flushes the DNS cache.
   (b) Commands Poisoner to start sending bogus DNS responses to the attacked machine.
   (c) Resolves the targeted domain by calling `ws2_32.dll!`*`getaddrinfo`*

3 Remote poisoning via Java Applets (CVE-2011-3552, CVE-2010-4448)

3.1 Environment

The following vulnerability is triggered when a user surfs the web and is lured into opening an attacker-controlled website. It is not browser-dependent.

3.2 Vulnerabilities

In Java, it is possible to create low-level system UDP sockets using the `java.net.Socket` API. These sockets can be bound on arbitrary ports. Furthermore, by using the Java Applet API, attackers can create Java code which is triggered upon visiting their web-page. This, together with the fact that Java fails to restrict the number of concurrent sockets, enables attackers to conduct a DNS poisoning attack on the visiting machine using the port exhaustion technique mentioned above. Unlike the Local Poisoning scenario described earlier, in this case the attacker cannot flush the victim's DNS cache, so he/she cannot poison a specific hostname. However, he/she can still target arbitrary domains, including non-existing ones.

3.3 Impact

3.3.1 Sub-domain poisoning

Attackers can spoof non-existing subdomains of a specific target in order to achieve the following:

1. *Cookie leakage*: If the target set a cookie with a wildcard domain which includes the poisoned domain's subdomain, the cookie would be added to all outgoing requests designated for the poisoned subdomain. This is true for all browsers. Moreover, in the case of Internet Explorer, the wildcard domain constraint is superfluous, so all cookies (except those with the `Secure` bit set) will be added to HTTP requests which target the subdomain. It should be noted that this is more powerful than XSS because `HTTPOnly` cookies can also be retrieved by the attacker. By leaking cookies the attacker can hijack the victim's web session, as the session identifier is frequently carried in a cookie.

2. *Cookie manipulation*: Attackers can modify cookies sent to the target host. Under Mozilla Firefox and Google Chrome, if a subdomain of the target attempts to set a cookie which has already been set by the target, the new cookie will be appended to the original one (that is, the HTTP requests will have two cookies with the same name). It's up to the server to decide which cookie is returned when calling a specific API. In the case of Internet Explorer, the original cookie is simply overwritten. Attackers may choose to target the session-id cookie so it points to an open session known to the attacker. If the target website is vulnerable to session fixation, then the session is not revoked after the user logs on, and the attacker can actually take over the session. If not, the manipulation can at least be used for phishing.

3. *Partial Cross-Site Scripting:* By default, Same-Origin Policy does not allow code coming from a subdomain to access the pages of its parent or sibling domains. However, if a page originating from a sibling or parent domain sets the document.domain property to a value which is an ancestor of the poisoned subdomain, the latter can do the same, which by specification [8] gives it read/write access to the former.

4. *Attack against Flash (CrossDomain.xml):* Flash movies adhere to Same-Origin Policy with an exceptional Cross-Domain bi-directional communication, which can be defined by creating a special file named *CrossDomain.xml*. This file defines which external domains are allowed to communicate, using Flash, with the domain making the definition. Frequently, when a domain contains multiple sub-domains which need access to the top domain, the latter includes a wildcard definition in the `CrossDomain.xml` file. Using our technique, we can attack any domain which makes such a definition. The idea is to poison an arbitrary sub-domain, and load a malicious Flash movie from it. This Flash file will have full access to the domain making the cross-domain definition. It should be noted that requests by the malicious Flash movie to the domain under attack will use existing credentials (i.e. stored cookies) if they exist. Therefore, a possible payload could be user impersonation, which can be done using ActionScript's `URLLoader`.

3.3.2 Intranet domain poisoning

Internet Explorer separates the world into the following security zones:

- Internet
- Local intranet
- Trusted sites
- Restricted sites

Each zone has different security constraints. URLs must be manually specified in order to be added to the lists of `Trusted` or `Restricted` sites. By default, Internet Explorer automatically detects intranet sites, and if the computer is a domain member, then all "dotless" URLs (such as "`localhost`", "`somehost`") are treated as intranet (otherwise the local intranet zone must be manually enabled). The Local Intranet zone is less secure that the Internet zone in a number of ways, including the following two:

1. Intranet sites can force Internet Explorer to authenticate in NTLM using the credentials of the logged on user (Single-Sign-On or SSO). During the NTLM authentication, the NT and LM hashes are sent to the authenticator, in addition to the username, domain name and machine name of the logged on user.

2. Intranet sites have read/write access to the Clipboard.

By attacking non-existing domains which are suffixed by the victim's machine DNS suffix, the attacker can provide pages in the Local Intranet zone as they can be accessed using the "dotless" notation. Therefore, the attacker has access to attributes (1) and (2). It should be denoted that under Windows, Google Chrome adheres to Internet Explorer's Intranet zone detection, and also provides SSO NTLM authentication for detected Intranet sites. Thus attackers can also target Google Chrome users in order to leak their NTLM credentials. Firefox supports NTLM however SSO sites have to be manually specified.

3.3.3 DNS rebinding

In general, DNS rebinding attacks occur if client-side code of some domain (which is resolved to some IP before the download process begins), abandons the original domain-to-IP resolution after the code has been downloaded, and permits the running code to communicate with the same domain mapped to a different IP. For example, a vulnerable web-browser would be susceptible to DNS rebinding if it referred to the DNS answers' TTL. Immune client technologies (such as web browsers or Java Applets) maintain a feature called *DNS pinning* which basically disregards the TTL value of responses received, and caches the original host-to-IP resolution until the session terminates. Another attack scenario which must be considered is where the DNS server returns an ambiguous resolution, for example in which code is downloaded from some IP address (`A.B.C.D`) which resolves to some hostname that resolves to multiple IP addresses, i.e. the connected IP (`A.B.C.D`) and some different IPs, the browser or browser plugin has to deny or take special care when the code attempts to connect to the additional resolutions. Java Applets technology implements this restriction by relying on reverse DNS lookups. As described by an earlier work [1], Java will grant permission for the downloaded code to intercommunicate with the other resolved IP(s) if and only if the reverse DNS look-up of `A.B.C.D` and the IP(s) in question resolve (i.e. reverse DNS) to the same host and the latter resolves to `A.B.C.D` and the IP(s) in question. This implementation is valid because same reverse (PTR) DNS records indicate that the same owner is in control of both `A.B.C.D` and the other IP(s).
An attacker can leverage the Java Applets vulnerability described above in order to rebind on the local loopback device. The attacker must be in control of both the DNS `A` record of his/her domain, and the DNS `PTR` record of his/her IP (or only over the `A` record of the domain pointed by an already existing `PTR` record of his/her IP). By controlling the `A` record of the attacker's domain, it can be arranged that the domain will resolve to multiple IPs: `A.B.C.D` (the attacker's web server's IP) and a large of number of subsets of `127.0.0.0/8`. In addition to this, `A.B.C.D` should resolve to the attacker's domain. Each time the attacker's code tries to connect to an IP which is a member of the chosen subset, Java would issue a reverse DNS request. By exploiting the aforementioned vulnerability, the attacker can win an arbitrary resolution during this iterative process in a feasible attack time in order to make an iterated `127.0.0.0/8` IP address resolve to the same host of which `A.B.C.D` resolves to and therefore bypass Java DNS rebinding protection. Now the attacker can connect to an arbitrary IP of the loopback device (`127.0.0.0/8`), giving it access to all services bound on `INADDR_ANY` (`0.0.0.0`) (this includes SMB, RDP, etc.). In effect, this enables the attacker to bypass intermediate or local firewalls that block incoming remote connections to the services on target. It should be denoted that on systems prior to MS09-013 & MS09-014 [9], this can become a vector for NTLM reflection [6].

3.4 Proof-of-Concept

Our PoC consists of two parts:

1. *Poisoner*: Server code which runs on an attacker's controlled machine. It must be capable of sending spoofed packets to the victim. This server listens on a designated TCP port accepting commands.

2. *Resolver*: Java Applet, JavaScript and HTML code which is hosted by the attacker and executed by the victim when he/she has been lured into visiting an attacker's controlled website. The following process takes place when the victim visits the malicious page:

   (a) Multiple instances of a UDP port consuming java applet are loaded, each of which is given a different range of ports. The ports are consumed by creating UDP sockets and binding as described above. Each applet is forced to be loaded on a different JVM, by adding the `separate_jvm` parameter to the `<APPLET>` HTML tag. We have found out empirically that the port-exhaustion procedure is faster when delegating it on several processes and it also would bypass any attempt to block the attack by limiting the number of concurrent sockets per process (instead of globally). In addition to that, in order to speed up the binding process, and reduce the amount of resources that the PoC consumes from the underlying OS (as described in other papers [7]), only a subset of `1024-65535` needs to be bound (`49152-65534` for Windows, and `32768-60999` for Linux), as other ports are anyway not used.

   (b) A second applet is loaded which, in general, iteratively commands *Poisoner* to start sending spoofed DNS responses (either PTR or A) using the designated TCP channel. A DNS request is then initiated, and if the poisoning is successful, the actual payload is executed. It should be denoted that since the attacking code knows which hostname is to be resolved, it has an advantage over the DNS server in order to win the race condition (therefore we instruct *Poisoner* to start sending responses *before* the actual request is made*).*
   To be more specific, different Java Applets are implemented for the various attacks, which are detailed below:

      i. Sub-domain poisoning [3]: Iteratively, it commands *Poisoner* to start sending spoofed DNS responses of arbitrary subdomains of a domain chosen by the attacker. After sending the command to the server, it resolves the target subdomain by calling JavaScript code using the interoperable `JSObject` API. On the JavaScript side, A DNS resolution is achieved by using Cross-Domain communication techniques such as `XMLHttpRequest` or `XDomainRequest`, giving the subdomain as a parameter. If the DNS poisoning is successful,
      `http://poisoned_domain/some_page.htm` is requested, and the response must to include an appropriate
      `Access-Control-Allow-Origin` header so the Cross-Domain request does not raise an exception. If the latter does not occur, it indicates that the resolution was successful (and unsuccessful otherwise). On older browsers, the resolution can be triggered by other techniques, such as changing the `src` attribute of an `IMG` tag, and by listening on its `onload` and `onerror` events.

      ii. Intranet domain poisoning [4]: This is the same as (i) but here arbitrary subdomains prefix the DNS suffix registered on the victim's Windows network settings. When the poisoning is successful,
      `http://poisoned_domain_with_no_suffix/some_page.htm` is requested. This elevates the browser (Internet Explorer, Chrome) to the Local Intranet Zone. In order to retrieve the NT/LM tokens the attacker's web server triggers the NTLM handshake process (3 phases) by returning a `401` error code escorted with a `WWW-Authenticate: NTLM` header.

iii. DNS rebinding [5]: At each iteration, it commands *Poisoner* to start sending spoofed DNS PTR responses which map an arbitrary local loop address to hostname which `A` record is controlled by the attacker. Then it attempts to connect to the arbitrary local loop address using the `java.net.Socket` API, on a protocol & port chosen by the attacker (such as TCP/445). This triggers a DNS PTR request of the arbitrary local loop address. If the resolution is successful, the attacker's IP resolves to the same random host, and the hostname which `A` record is controlled by the attacker has multiple resolutions which include the arbitrary local loop address, a connection is established (unless the port is closed).

4 Attack feasibility

4.1 MS Windows prior to Windows 7 / 2008 R2

Prior to Windows 7 / Windows 2008 R2, when the DNS resolver encountered a DNS reply with a mismatching `TXID`, it simply dropped it. This means that the success probability of a single DNS cache poisoning attempt, directly depends on the amount of packets the attacker manages to insert between the DNS query, and the genuine DNS response.

Therefore, we can claim that the success probability of a single poisoning attempt is given by the following formula:

$$p = \lfloor \frac{b * l}{s} \rfloor * \frac{1}{2^{16}}$$

where b stands for the victim's available bandwidth, l stands for the DNS responses' latency, and s reflects the forged responses' packet size. It should be denoted that p does not depend on k, the latency between the attacker and the victim, since the attacker may be aware of the time at which the victim produces the DNS request, and start poisoning k time units before the request is sent.

Considering the geometric properties of the attack, the average time for success is given by:

$$E(T) = \frac{t}{p} = \frac{t * 2^{16}}{\lfloor \frac{b*l}{s} \rfloor}$$

where t stands for the average time taken for a single attempt, and T is the attack time random variable. Interesting values (considering $s = 120bytes,\ t = 500ms$) are shown below:

Bandwidth (b)	Latency (l)	Attack time (E)
$1gb/s$	$1ms$	$31.477s$
$1gb/s$	$10ms$	$3.146s$
$1gb/s$	$50ms$	$0.629s$
$1gb/s$	$100ms$	$0.314s$
$100mb/s$	$1ms$	$5.251m$
$100mb/s$	$10ms$	$31.477s$
$100mb/s$	$50ms$	$6.291s$
$100mb/s$	$100ms$	$3.146s$

Bandwidth (b)	Latency (l)	Attack time (E)
$8mb/s$	$1ms$	$68.266m$
$8mb/s$	$10ms$	$6.580m$
$8mb/s$	$50ms$	$1.312m$
$8mb/s$	$100ms$	$39.337s$
$3mb/s$	$1ms$	$3.034h$
$3mb/s$	$10ms$	$17.617m$
$3mb/s$	$50ms$	$3.501m$
$3mb/s$	$100ms$	$1.75m$

4.2 MS Windows 7 / 2008 R2

Windows 7 / 2008 R2 has introduced a change to the resolution's process. If a DNS response contains a mismatching `TXID`, the relevant DNS query request (one with a source port that matches the DNS response's destination port) is dropped (to be more specific, it is re-requested and if this iterative process fails 4 times, the resolution returns an error). With respect to security, under Windows 7 / 2008 R2, we can claim that:

$$p = \frac{1}{2^{16}}$$

$$E(T) = t * 2^{16}$$

For the local poisoning attack, by delegating the resolution phase to multiple processes, we managed to decrease t down to $\sim 65ms$. This yields an expected time of around 71 minutes for success, which is still feasible for a local poisoning attack.

As for remote poisoning, further research may show that t can also be decreased to the same number (for example, by delegating the resolution phase to multiple JVMs). Despite that, 71 minutes is infeasible attack time for remote DNS poisoning.

4.3 Linux

Linux has two limits on the number of open file descriptors:

1. Per-process cap
2. System-wide cap

Since we can create multiple processes (both for the local attack and the remote one), the first cap does not protect against our port exhaustion attack. However, lowering the second cap makes the attack (both types) infeasible.
In addition to that, the range of source ports can be set (with root privileges), and has a default value of $32768 - 61000$ (28233 ports)

We have taken Ubuntu 11.04 as a sample distribution.
It ships with the following settings:

1. Per-process cap is 1024
2. System-wide cap is 199124
3. Default IP port range

By delegating the port exhaustion phase of the attack to 29 JVMs (i.e. ~1000 ports per JVM), we successfully remotely poisoned a Firefox client, running the latest Oracle JVM.

5 Fix recommendations and vendor response

The following describes a general solution for the aforementioned vulnerabilities which can be used when developing similar environments.

5.1 Local stub resolver DNS vulnerability

There are a few solutions to this issue, which can be used in conjunction for best protection.

1. Restrict the amount of ports a normal user may bind on. This solution can be weakened in case the malicious user has several accounts on the system.

2. Do not allow non-administrative (normal) users to flush the DNS cache. While this solution decreases the severity of the port exhaustion vulnerability, we have shown that attacking non-existent domains is still possible.

3. Separate the DNS cache as covered by the patent [12]. The idea is to break the shared (global) DNS cache of the stub resolver into (local) private ones. In other words, a different cache is created per user . Effectively this isolates each user within his or her own sandbox and makes it impossible for a malicious user to cache-poison another user by simply poisoning his or her own cache. A drawback of logical cache separation is a decrease in performance and an increase in storage requirements. However, as an optimization, the system may maintain another *global* cache which is shared between all users. Only trusted responses are stored in the global cache and they can be added either statically (i.e manually) or dynamically with the following heuristic: If the same cache entry exists under X local (private) caches, move it to the global cache.

It should be noted that although there is no official fix for this vulnerability, publication of this paper has been fully coordinated with Microsoft, and adheres to our responsible disclosure guidelines.

5.2 Remote poisoning via Java applets vulnerability

Client-side code (such as Java Applets) should not be allowed to listen on an unlimited number of ports. In the case of Java, a global restriction should be made to all Java Applets, and not to a single JVM, since a webpage can force the Java plugin to load a given applet in a new JVM using the `separate_jvm` parameter. Despite that, capping the number of ports to a low number is also a good solution (but not too low, so it doesn't affect functionality), since launching many JVMs is unfeasible.

Oracle first attempted to fix the vulnerability by capping the number of ports to 1024 per JVM (CVE-2010-4448, February 2011, Java 1.6u24 [10]). However, since we had managed to overcome this restriction, the cap was reduced to 50 ports per JVM (CVE-2011-3552, October 2011, Java 1.6u29 [11]).

6 Acknowledgments

- We would like to thank Oracle and Microsoft for their cooperation.
- We are grateful to the following people for their contribution to this paper:
  - Adi Sharabani
  - Jonathan Cohen

7 References

[1] David Byrne. Anti-DNS Pinning and Java Applets. http://seclists.org/fulldisclosure/2007/Jul/159.

[2] Roee Hay and Yair Amit. Demo of PoC: Local DNS poisoning via port exhaustion vulnerability in Windows. http://bit.ly/nxQqwn.

[3] Roee Hay and Yair Amit. Demo of PoC: Remote DNS poisoning via Java Applets: Cookie theft. http://bit.ly/ptNf73.

[4] Roee Hay and Yair Amit. Demo of PoC: Remote DNS poisoning via Java Applets: NTLM token/Clipboard theft. http://bit.ly/qvynLF.

[5] Roee Hay and Yair Amit. Demo of PoC: Remote DNS poisoning via Java Applets: Firewall bypass. http://bit.ly/qyOBKn.

[6] HD Moore. MS08-068: Metasploit and SMB Relay. http://blog.metasploit.com/2008/11/ms08-067-metasploit-and-smb-relay.html.

[7] Kris Kaspersky. DNS patches against Kaminsky's attack are still vulnerable. http://endeavorsecurity.blogspot.com/2008/08/dns-patches-against-kaminskys-attack.html.

[8] Michal Zalewski. Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Part2.

[9] MS09-013 and MS09-014: NTLM Credential Reflection Updates for HTTP clients. http://blogs.technet.com/b/srd/archive/2009/04/14/ntlm-credential-reflection-updates-for-http-clients.aspx.

[10] Oracle Java SE and Java for Business Critical Patch Update Advisory - February 2011. http://www.oracle.com/technetwork/topics/security/javacpufeb2011-304611.html.

[11] Oracle Java SE Critical Patch Update Advisory - October 2011. http://www.oracle.com/technetwork/topics/security/javacpuoct2011-443431.html.

[12] Roee Hay and Adi Sharabani. Protection Against Cache Poisoning, 2011.