Subverting BIND's SRTT Algorithm Derandomizing NS Selection

Roee Hay
IBM, Israel
roeeh@il.ibm.com

Jonathan Kalechstein
CS Department
Technion
kalechstain@gmail.com

Gabi Nakibly
National EW Research
& Simulation Center
gabin@rafael.co.il

Abstract. One of the defenses against DNS cache poisoning is randomization of the IP address of the queried name server. We present a newly found vulnerability in BIND, the most widely used DNS software on the Internet, which enables an attacker to easily and deterministically control the queried name server chosen by BIND's resolver. The vulnerability lies in BIND's SRTT (Smoothed Round Trip Time) algorithm. The attack reduces the time and effort required to successfully poison BIND's cache. The general lesson from this vulnerability is that a DNS resolver must never keep a global state shared between different domain names (in our case the SRTT values are kept as a global state).

I. Introduction

The Domain Name System (DNS) [Moc87a], [Moc87b] is a hierarchical distributed naming system which allows to resolve names to IP addresses. Generally, during a DNS resolution a resolver issues a query for a name which is responded to by a name server (NS). A resolver may be a client or a cache server handling queries on behalf of other clients. BIND is the most widely used DNS software on the Internet [Sas10] for name and cache servers.

The most common type of attack on DNS is cache poisoning. In this type of attack an attacker causes a victim resolver to cache a bogus DNS resource record (RR). This may enable further attacks, with an impact on both integrity and confidentiality. For example, integrity can be broken if the target cache is poisoned with a fake RR that resolves the name of a software update server to an IP address of an attacker controlled server. As another example, confidentiality can be compromised by poisoning a cache with a bogus RR that resolves the name of a web server to an attacker controlled IP address. When a user browses to that web server, he will connect the attacker's web server instead of the genuine one and may reveal private information.

A common method to achieve DNS cache poisoning is by generating a forged response to a DNS query sent by the victim resolver. To mitigate this attack method a resolver uses unpredictable values with each generated query. Since the corresponding values in the response must match the values sent in the query, it is difficult for a blind (off-path) attacker, who does not see the query, to forge a valid response. The most common values which the resolver randomizes are DNS transaction ID (TXID), UDP source port and the IP address of the queried name server [HvM09]. The IP address of the queried name server is chosen from a list of candidate name servers that are relevant for the domain name to be resolved.

In this work we present a newly discovered vulnerability in BIND which allows an attacker to determine (derandomize) the IP address of the name server a BIND resolver queries. The attack reduces the amount of information a blind attacker must guess to successfully poison BIND's cache. The vulnerability lies in BIND's algorithm that updates the SRTT (Smoothed Round Trip Time) of each name server. The vulnerability has been acknowledged by ISC (the maintainers of BIND).

II. The SRTT algorithm

BIND maintains a dynamic list of candidate name servers for resolving a particular query. From this set of candidate name servers one is chosen and queried. If the query fails for some reason, BIND chooses another NS from the remaining set and so on. BIND chooses the NS with the lowest SRTT (Smoothed Round Trip Time). The SRTT of a NS estimates the time the resolver may wait for a response from that NS. The SRTT value for each name server is stored in a global cache indexed by the name server's IP address and calculated as follows (the indicated source code locations are of `BIND 9.8.1-P1`):

1) **Init.**[1] When a candidate NS is first added to the global cache it receives a very low value: $SRTT_{init} = 1 + 31 \wedge R\ \mu s$ where R is a 32bit unsigned value returned by the `isc_random_get` routine. If we assert that the 5 least significant bits of R are distributed uniformly, then $SRTT_{init} \sim Uni(1, 32)\ \mu s$.
2) **Update.**[2] Whenever a response is received from a NS, the SRTT of that NS becomes a weighted sum of the old SRTT and new RTT value: $SRTT_{update} = 0.7 \cdot SRTT_{old} + 0.3 \cdot RTT_{new}$. The new RTT value is the time period from query origination to response receipt.
3) **Decay.**[3] In order to avoid starvation, for each query the resolver produces, the SRTT of the other NS candidates are multiplied with a decay factor of 0.98.
4) **Error.**[4] In case of an unanswered request, or a network error (e.g. an ICMP "Destination Port Unreachable" message is received), the candidate name server is punished by adding 200 ms to its SRTT value (with a maximum value of 1 s).

III. Related work

There are a couple of works that propose probabilistic attack methods to derandomize name server (NS) selection.

[Pet09] presents an attack which works by spoofing DNS responses from a victim NS while reducing the SRTT of that NS. By reducing its SRTT the BIND resolver is more likely to choose that NS for the next query. This attack targets the *Update* operation of the SRTT algorithm.

In [HS12], the attacker sends to the resolver a spoofed fragment of a DNS response originated by a NS, making the reassembled response packet corrupt. Hence, this response is discarded by the resolver. After several failed attempts, the IP address of that NS is marked by the resolver as non-responsive and is blocked for interval of time. The attack in launched on all but one NS from a candidate set of NSs. The remaining NS will be the one to be queried next by the resolver. To carry out the attack the DNS response from a victim NS must be large enough to be fragmented. In BIND, this attack targets the *Error* operation of the SRTT algorithm.

IV. The Attack

Our attack exploits the *Decay* operation and forces BIND to decrease the SRTT of any name server we choose to an arbitrary low value. This allows an attacker to have a BIND resolver query a target NS of his choosing. The attack we present does not require the attacker to spoof responses on behalf of other NSes as in the other attacks presented above. It is deterministic and does not require guesswork from the attacker. Moreover, the target name server is never contacted during the attack and therefore it is unaware of it.

A. First variant

Prerequisites. The attacker owns two arbitrary domain names correspondingly served by two NSes, A_1 and A_2, which are under his control. The NS A_2 must have lower latency to the target resolver, R, than the victim's NS, V. Specifically, if the attacker tries to lower the SRTT of name server V in the cache of R, it is required that A_2's SRTT value will be lower than V's SRTT value (denoted as $SRTT_{original}$). We note that these two name servers, A_1 and A_2, can be anywhere on the Internet, under this constraint.

Attack flow (see Figure 1).

1) The attacker inserts A_2 to R's SRTT cache. This can be done, for example, by querying R for the domain A_2 is authoritative of, say `a2.foo`.
2) R contacts A_2 which returns the answer to the attacker.
3) The attacker queries R for a domain name that A_1 is authoritative of, say `a1.foo`.
4) R contacts A_1 which replies with a delegation (namely, it refers R to other NSes) that includes:
    a) A fresh list of non-open name servers (i.e. external clients do not have access to their local cache), $C_1, \ldots, C_n$.
    b) A name server A_2.
    c) A name server V.
5) R queries in turn all non-open name server. All of them will refuse.

[1] `lib/dns/adb.c:1747 (new_adbentry)`
[2] `lib/dns/adb.c:3900 (dns_adb_adjustsrtt)`
[3] `lib/dns/adb.c:3898 (dns_adb_adjustsrtt)`
[4] `lib/dns/resolver.c:817 (fctx_cancelquery)`

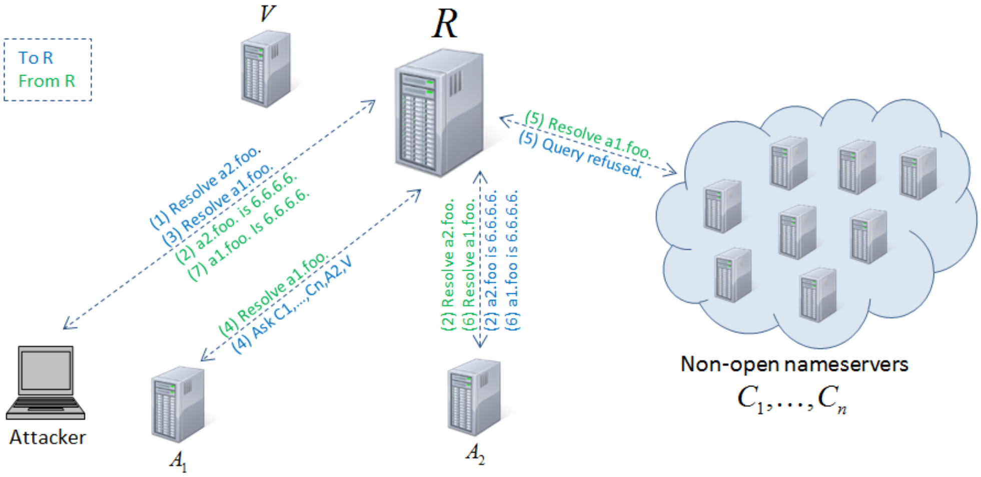

Figure 1.

6) R queries A_2 which returns a valid answer.
7) R returns a valid answer to the attacker.

When the resolver receives the delegation (4), it creates new SRTT entries for $C_1, \ldots, C_n$ with a low values ($SRTT_{init}$), thus they are queried before A_2 and V which already reside in the cache. After n queries to the non-open name servers, according to the SRTT algorithm, both A_2's SRTT and V's SRTT will be 0.98^n of their original value before the attack (n decay operations). Before the attack A_2's SRTT was lower than V's SRTT (as per the prerequisite above), A_2's SRTT will also be lower than V's SRTT after n decay operations, thus R choose to query A_2 before V. The iterative resolution ends by querying A_2. Thus V's SRTT results in a low bogus value ($0.98^{n+1} \cdot SRTT_{original}$).

B. *Second variant*

Prerequisites. The attacker owns an arbitrary domain name served by NS A_1, which is under his control. Here the attacker is not required to have a low-latency name server, however the amount by which the SRTT value is decreased is not deterministic. This may require the attacker to initiate several, m, attack attempts to lower the SRTT to the desired value. These attempts require m servers $A_{2,1}, \ldots, A_{2,m}$ controlled by the attacker (the attacker may choose to utilize the same server for each attempt but in such a case he has to wait for the SRTT entry of that server to expire from the cache before initiating the next attempt).

Attack flow (i^{th} attempt).

1) The attacker queries R for a domain name that A_1 is authoritative of, say `a1.foo`.
2) R contacts A_1 which replies with a delegation that includes:
    a) A fresh list of non-open name servers, $C_1, \ldots, C_n$.
    b) A name server $A_{2,i}$.
    c) A name server V.
3) R queries t non-open name server, where $0 \leq t \leq n$. All queried name server will refuse.
4) R queries $A_{2,i}$ which returns a valid answer.
5) R returns a valid answer to the attacker.

When the resolver receives the delegation (2), it creates new SRTT entries for $A_{2,i}$ and for every non-open name server, with $SRTT_{init} \sim Uni(1, 32)$ μs. Thus $A_{2,i}$'s SRTT must be lower than V's SRTT. Therefore, $A_{2,i}$ will be queried before V. However, the SRTT of $A_{2,i}$ may be lower than some non-open name servers. We denote the number of non-open names servers that has lower SRTT by $0 \leq t \leq n$. Therefore, according to the SRTT algorithm, the attack reduces V's SRTT by 0.98^{t+1}. We note that t is a uniform random variable with an average of $\frac{n}{2}$.

C. *Impact*

As noted above, this attack allows to derandomize the NS selection which can reduce the attack time of blind (off-path) DNS cache poisoning. In addition, the attack can help an attacker to act as a Man-in-the-Middle (MitM), in cases where he resides on the path between the resolver and only one of the candidate name servers. In such cases the attack can enable on-path poisoning. Moreover, the attack may assist in Denial-of-Service (DoS) of a specific target name server by directing more queries to it.

We note that since the update operation of the SRTT algorithm takes into account the previously defined SRTT, the recovery is not instant, thus subsequent requests to V will not cause it to immediately revert the SRTT to its original value.

V. Evaluation

The first variant of the attack was evaluated against `BIND 9.8.1-P1`. The vulnerability has been reported to and acknowledged by ISC. A description of how we evaluated the attack follows.

A. Lab setup

We tested the vulnerability on virtual machines with the following setup:

`192.168.19.201`	An NS serving some zone
`192.168.19.202`	V (serving the same zone)
`192.168.42.100`	R
`192.168.19.250`	A_1
`192.168.42.251`	A_2
`192.168.19.170-199`	$C_1, \ldots, C_{30}$
`192.168.19.50`	Mock root name server

Each machine ran Ubuntu 12.0.4.1 LTS. The non-open NSes were actually one virtual machine with multiple IP addresses. A latency of 80 ms and a jitter of 10 ms between the `192.168.19.0/24` (which included V, A_1 and C_i) and `192.168.42.0/24` networks (which included R and A_2) were emulated using WANEm[5]. Note that in this setup R's latency to A_2 is smaller than its latency to $C_1, \ldots, C_{30}$, as required.

B. Results

We dumped `BIND`'s SRTT cache using the command `'rndc dumpdb -cache'` which gave the following output:

- Before step (3) of the attack:

```
192.168.42.251 srtt 571    (A_2)
192.168.19.202 srtt 39363  (V)
192.168.19.201 srtt 38738
192.168.19.50  srtt 80957
```

- After the attack (the entries of the non-open resolvers are omitted)

```
192.168.42.251 srtt 747    (A_2)
192.168.19.202 srtt 21029  (V)
192.168.19.201 srtt 38738
192.168.19.50  srtt 80234
192.168.19.250 srtt 49935  (A_1)
```

The SRTT of V before the attack is 39363. The SRTT of V after the attack is 21029 which is approximately $0.98^{31} \cdot 39363$ as expected (it is not the exact value due to floating-point to integer casting).

VI. Conclusions

In this work we presented a newly discovered vulnerability in BIND's SRTT algorithm. A deterministic attack exploiting this vulnerability allows an attacker to determine the name server that will be chosen by a BIND resolver from a set of a candidate name servers. This allows the attacker to derandomize the IP address of the queried name server in order to reduce the time and effort required to successfully poison BIND's cache. The vulnerability stems from the fact that BIND stores the SRTT of all name servers in a global cache shared by all domain names. This allows an attacker to influence the name server selection for one domain name while issuing queries for a different domain name. A possible mitigation for the attack is to keep the SRTT entries separated by domain names.

References

[HS12] Amir Herzberg and Haya Shulman. Security of Patched DNS. In *ESORICS*, pages 271–288, 2012.

[HvM09] A. Hubert and R. van Mook. RFC 5452: Measures for Making DNS More Resilient against Forged Answers, 2009. http://www.ietf.org/rfc/rfc5452.txt.

[Moc87a] P. Mockapetris. RFC 1034: Domain Names - Concepts and Facilities, 1987. http://www.ietf.org/rfc/rfc1034.txt.

[Moc87b] P. Mockapetris. RFC 1035: Domain Names - Implementation and Specification, 1987. http://www.ietf.org/rfc/rfc1035.txt.

[Pet09] Emanuel Petr. An Analysis of the DNS cache poisoning attack, 2009. https://labs.nic.cz/files/labs/DNS-cache-poisoning-attack-analysis.pdf.

[Sas10] Geoffrey Sasson. DNS Survey: October 2010, 2010. http://dns.measurement-factory.com/surveys/201010/dns_survey_2010.pdf.

[5] http://wanem.sourceforge.net/