Port Number Exhaustion of a 464XLAT Implementation in a Virtual Environment

Norbert Nagy, Gábor Lencse
Department of Networked Systems and Services
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics
Műegyetem rkp. 3, Budapest, H-1111, Hungary
Email: norbert.nagy@edu.bme.hu, lencse@hit.bme.hu

***Abstract*—There are several IPv6 translation technologies, and we chose one of them, 464XLAT. In this paper, we focused on how we can perform a successful DoS (Denial of Service) attack and exhaust the available port number range on one of its components, the provider-side translator (PLAT). To achieve this, we built a testbed using virtual machines. We implemented the client-side translator (CLAT) with Tayga software and used a combination of Tayga and the Netfilter framework for the provider-side translator (PLAT). We measured the limitations of our testbed and carried out successful tests using a DNS query generator program, dns64perf++, thus exhausting the port number range on the PLAT.**

***Keywords*—464XLAT; Connection tracking table; DoS attack; Netfilter, Stateful NAT64, Tayga.**

I. INTRODUCTION

Because of the depletion of the IPv4 address pool, the use of IPv6 addressing is inevitable. This translation cannot go overnight, and because of this reason, several IPv6 transition solutions have been created, one of them being 464XLAT [1]. 464XLAT is usually used in wireless (cellular) environments, but in our research work, we tested it in a wired testbed. Nowadays, more and more service providers have an IPv6 network, but they also have users who need IPv4 service, as there are some IPv4-only applications like Skype or Spotify [2]. 464XLAT solves the problem when IPv4 users want to connect to other IPv4 servers on the internet over an IPv6 network. It achieves this with double translation. It uses SIIT translation on the customer side device called CLAT, and stateful NAT64 on the provider side device called PLAT. For the stateful behavior, PLAT uses NAPT (Network Address and Port Translation), which means for the connection tracking it stores the source and destination IP addresses, port numbers, and the protocol (TCP or UDP) that was used [1].

On the one hand, IPv6 transition technologies make the cooperation between the two incompatible versions of the Internet Protocol (IPv4 and IPv6) possible. However, on the other hand, their application involves various security vulnerabilities [3]. The vulnerabilities of the different IPv6 technologies are actively researched [4]. The *Denial of Service* (DoS) attack is the one that is "absorbing resources needed to provide service" [5]. As for the different opportunities of a DoS attack against 464XLAT, there are multiple ways to prevent its PLAT component from performing translations. One can simply exceed the CPU capability of the device, thus it can't translate or translate the packets properly. This has been done in [6]. Another way to prevent the proper behavior is to fill up the connection tracking table of the PLAT device until it is full. There is a third option, which takes advantage of the fact that PLAT works in a stateful way, thus all customers share the resources that are used in the translation, such as port numbers and public IP addresses. In this paper, we focused on how one or multiple hosts can exhaust the PLAT port number range so the PLAT can't operate normally.

There are multiple open-source 464XLAT implementations on the internet. But we used Tayga [7] a stateless NAT64 software for the CLAT. For the PLAT, we also used Tayga to do a stateless NAT64 translation and Netfilter to perform the stateful NAT44 translation, thus together they implemented a stateful NAT64. We chose this implementation to accurately demonstrate how PLAT operates.

The size of the available port number range is: 65536-1024=64512, where 65536 is the number of all available port numbers because port numbers are represented on 2 bytes, which is 16 bits, thus the number of representable port numbers is 2^{16}=65536. 1024 is the number of the well-known ports, which are not used as source port numbers. To exhaust the available port number range, we needed to generate a significant volume of network traffic. For this reason, we used the **`dns64perf++`** program [8] (documented in [9]). It can generate DNS AAAA query requests with a variable query name field and sends them to a specified destination. Meanwhile, this program measures the response time of the query. We have chosen this program because we were able to specify the number of requests per second.

The remainder of this paper is organized as follows. In section II, we describe the testbed used to exhaust the port number range of the PLAT. In section III, we determine the limitations of our testbed and detail the constraints that we had to deal with. In section IV, we detail the method of how 464XLAT's vulnerability can be exploited and we show one example. Finally, in section V, we summarize and conclude our paper.

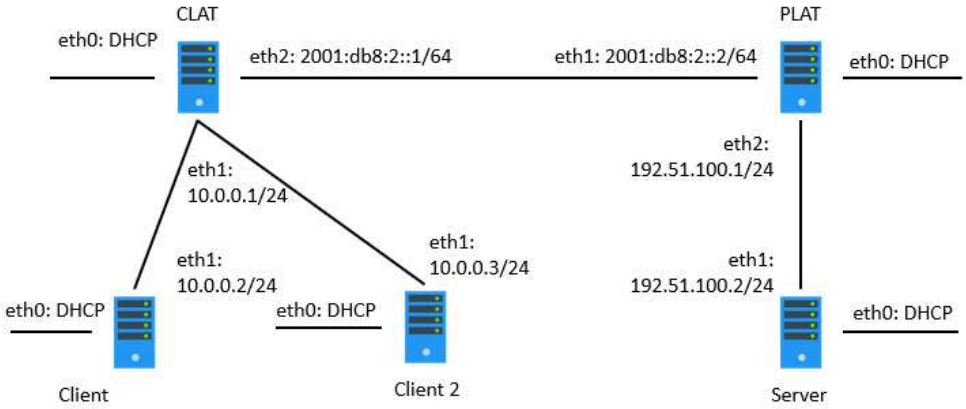


Fig. 1. Topology of the test network.

TABLE I. LINUX AND VMWARE NETWORK SETTINGS FOR VIRTUAL MACHINES.

Virtual machine name	Client	Client 2	CLAT	PLAT	Server
eth0 Linux conf	DHCP	DHCP	DHCP	DHCP	DHCP
eth0 VMware conf	NAT	NAT	NAT	NAT	NAT
eth1 VMware conf	VMnet11	VMnet11	VMnet11	VMnet12	VMnet13
eth2 VMware conf	N/A	N/A	VMnet12	VMnet13	N/A

II. 464XLAT TESTBED

A. *The architecture*

Virtual machines can be quickly created, and they are easy to set up, for these reasons, we used VMware virtual machines for the testbed. These virtual machines were run by VMware Workstation Player. As shown in Table I, 5 VMs were created and assigned to various VMnets. Two of these represented IPv4 hosts, we called them Clients. The other two of them were the CLAT and the PLAT. They were connected by an IPv6 network. CLAT was connected to the clients. The last virtual machine was connected to the PLAT's IPv4 interface and behaved as an IPv4 server on the Internet.

B. *Testbed implementation*

The clients, server, and the CLAT had 256 MB memory and 1 CPU core, while the PLAT had 3 CPU cores and 512 MB memory. More resources were needed on the PLAT since it did the stateful NAT64 translation (actually a stateless NAT64 plus a stateful NAT44). For the virtual machine operating systems, we installed Debian 8 and configured them according to Table I.

C. *End device configurations.*

On the server, we only configured the network interface according to Fig. 1 and set the necessary route. We were not interested in the DNS query answers, thus, we didn't set up a DNS server on this machine. On the clients' machines, we also configured the IP addresses and the default route. We also downloaded the **`dns64perf++`** program from GitHub. Due to the fact that our program uses a high number of different source port numbers (and thus also sockets) for packet generation, we had to increase the limit on the number of open files in Linux. To achieve this, the following line was inserted into the **`/etc/security/limits.conf`** file:

```
root       hard   nofile        100000
```

D. *CLAT configuration*

For the stateless NAT64 function, we used Tayga, and the following lines were modified in the configuration files.

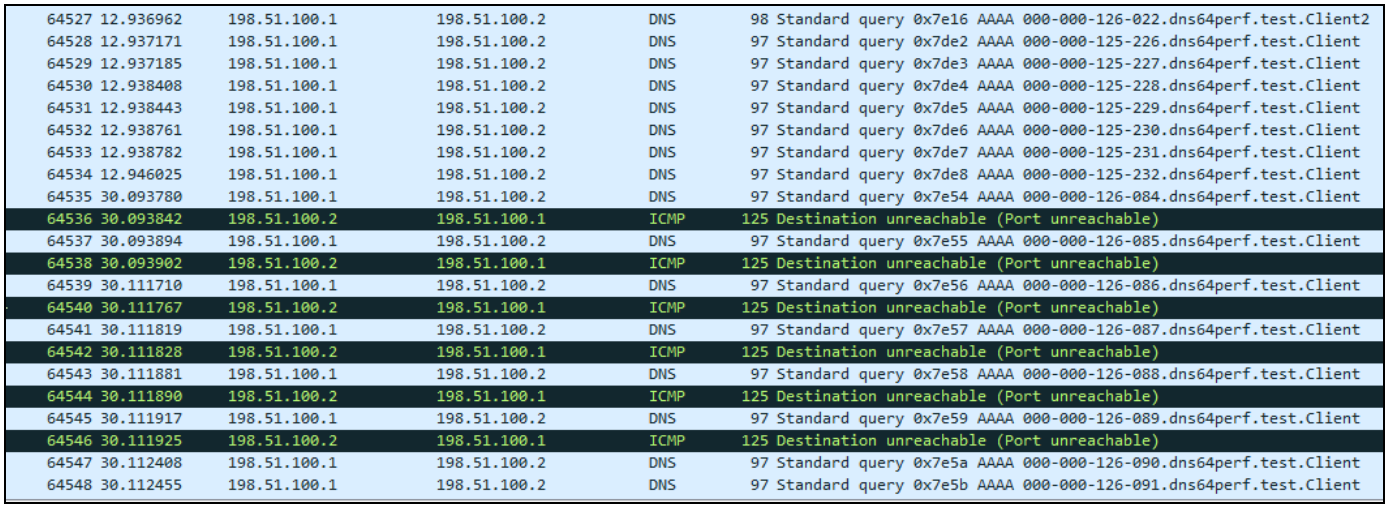
In **`/etc/default/tayga`**, we switched off the activation of the stateful NAT44 function and started Tayga.

```
RUN="yes"
CONFIGURE_NAT44="no"
```

In **`/etc/tayga.conf`** file, we made the following modifications.

```
ipv4-addr 192.0.2.9
ipv6-addr 2001:db8:2::9
prefix 2001:db8:a::/96
#dynamic-pool 192.168.255.0/24
#data-dir /var/spool/tayga
map 10.0.0.2 2001:db8:c::10.0.0.2
map 10.0.0.2 2001:db8:c::10.0.0.3
```

Finally, we created a bash shell script to set up the IP forwarding and necessary routes.



64527	12.936962	198.51.100.1	198.51.100.2	DNS	98 Standard query 0x7e16 AAAA 000-000-126-022.dns64perf.test.Client2
64528	12.937171	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de2 AAAA 000-000-125-226.dns64perf.test.Client
64529	12.937185	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de3 AAAA 000-000-125-227.dns64perf.test.Client
64530	12.938408	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de4 AAAA 000-000-125-228.dns64perf.test.Client
64531	12.938443	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de5 AAAA 000-000-125-229.dns64perf.test.Client
64532	12.938761	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de6 AAAA 000-000-125-230.dns64perf.test.Client
64533	12.938782	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de7 AAAA 000-000-125-231.dns64perf.test.Client
64534	12.946025	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7de8 AAAA 000-000-125-232.dns64perf.test.Client
64535	30.093780	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e54 AAAA 000-000-126-084.dns64perf.test.Client
64536	30.093842	198.51.100.2	198.51.100.1	ICMP	125 Destination unreachable (Port unreachable)
64537	30.093894	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e55 AAAA 000-000-126-085.dns64perf.test.Client
64538	30.093902	198.51.100.2	198.51.100.1	ICMP	125 Destination unreachable (Port unreachable)
64539	30.111710	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e56 AAAA 000-000-126-086.dns64perf.test.Client
64540	30.111767	198.51.100.2	198.51.100.1	ICMP	125 Destination unreachable (Port unreachable)
64541	30.111819	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e57 AAAA 000-000-126-087.dns64perf.test.Client
64542	30.111828	198.51.100.2	198.51.100.1	ICMP	125 Destination unreachable (Port unreachable)
64543	30.111881	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e58 AAAA 000-000-126-088.dns64perf.test.Client
64544	30.111890	198.51.100.2	198.51.100.1	ICMP	125 Destination unreachable (Port unreachable)
64545	30.111917	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e59 AAAA 000-000-126-089.dns64perf.test.Client
64546	30.111925	198.51.100.2	198.51.100.1	ICMP	125 Destination unreachable (Port unreachable)
64547	30.112408	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e5a AAAA 000-000-126-090.dns64perf.test.Client
64548	30.112455	198.51.100.1	198.51.100.2	DNS	97 Standard query 0x7e5b AAAA 000-000-126-091.dns64perf.test.Client

Fig. 2. Taffic observed at the **eth2** interface of PLAT.

```
#!/bin/bash
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
ip route add 198.51.100.0/24 dev nat64
ip route add 2001:db8:c::/96 dev nat64
ip route del 2001:db8:a::/96 dev nat64
#needed to be deleted as tayga set it automatically
ip route add 2001:db8:a::/96 via 2001:db8:2::2
```

E. PLAT configuration

The stateful NAT64 translation is handled by the PLAT. We implemented this with two programs. First, we used Tayga to do the translation from IPv6 to IPv4. But Tayga works in a stateless way, so we used Netfilter [10] to implement the stateful function.

Tayga's configuration is very similar to the previous case. The relevant part of the **/etc/tayga.conf** file is as follows:

```
ipv4-addr 192.0.2.9
ipv6-addr 2001:db8:2::9
prefix 2001:db8:a::/96
#dynamic-pool 192.168.255.0/24
#data-dir /var/spool/tayga
map 10.0.0.2 2001:db8:c::10.0.0.2
map 10.0.0.2 2001:db8:c::10.0.0.3
```

The **/etc/default/tayga** has the same content.

```
RUN="yes"
CONFIGURE_NAT44="no"
```

In order to make the stateful part up and running, we added the following **iptables** rule.

```
iptables -t nat -A POSROUTING -o eth2 -j MASQUERADE
```

This rule will change the source IP address so the outgoing packet will have the PLAT's IP address instead of the clients' IP addresses.

We also set up the forwarding and routing with the following script.

```
#!/bin/bash
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
ip route add 10.0.0.0/24 dev nat64
ip route add 2001:db8:c::/96 via 2001:db8:2::1
```

III. CONSTRAINS AND LIMITATIONS

A. Testbed limitations

Our measurements were performed on a single physical computer. The resources of the computer were split between the virtual machines. Thus, the virtual environment that we used had its limitations. We carried out some measurements in order to define the rate at which we can send the DNS queries so that they are transmitted without loss. 60,000 packets were sent from one of the clients with a 1ms delay between each packet, and we measured the number of packets that arrived at the server with TShark. After that, we repeated the test with a lower delay between the consecutive packets, down to 0.1ms. We observed that the minimum delay that we were able to set up for all the packets to arrive at the server was 0.2ms. This is equivalent to 5000 query/s, so our testbed was able to send packets with a maximum 5000query/s rate.

B. Constraints on the PLAT

The stateful function was implemented with Netfilter. It stores the connections in a connection tracking table. It can be viewed in **/proc/net/ip_conntrack** [11].

Because the DNS queries use UDP, it was important to know the UDP session timeout, which means, that after 30 seconds, the ports can be re-used again. It can be viewed in **/proc/sys/net/ipv4/netfilter/ip_conntrack_udp_timeout** [12], the timeout value was 30s, and we didn't change that.

There is a hash table that stores a list of NAT table entries, and the size of the table can be viewed or modified in **/sys/module/nf_conntrack/parameter/hashsize**.

The size of the connection tracking table was modified in **`/proc/sys/net/ipv4/netfilter/ip_conntrack_max`**. It is important to know that the size of the NAT table should be 8 times larger than the hash table size. The hash table size is recommended to be a power of 2, thus we set this at 16384. The connection tracking table's size was set to 131072 [13]. This size is a high enough number, so the table will not fill up before all the port numbers are in use.

IV. 464XLAT VULNERABILITY

A. *Vulnerability*

We are aware that there are multiple vulnerabilities in 464XLAT, but in this paper, our goal is to send the packets through the PLAT so that it will use all its available port numbers and reach a point where it can't work properly anymore because it doesn't have any available source port number to translate an incoming packet. It is possible because all devices behind the PLAT share a pool of connection descriptors that are used during the NAT translation.

B. *Sample attack*

To exploit the above-stated vulnerability, we used two clients and sent 60000 queries at a 2500 query/s rate from each machine. Thus, the traffic remained within our limitation of 5000 query/s at the PLAT. This resulted in 120,000 connections at the PLAT. The connection table size was 131072. This size prevents the possibility that the NAT table will be full. During the attack, we did a packet capture with TShark on the PLAT's outgoing interface. We examined the packet capture and found that 64534 packets left the PLAT in 12.9s. In these packets there were 24 ARP or ICMP, thus 64512 DNS query packets were translated. This is the exact number of available source port numbers. After this, no packets were captured until the 30.09s. Let us recall that the UDP session timeout was the 30s. After 30.09s, only a few hundred packets left the PLAT.

C. *Explanation*

What happened was, that when 64512 packets arrived, they got translated and all available source port numbers were used. These connections got into the connection tracking table and stayed there until the UDP timeout expired. Meanwhile, the packets kept coming. We believe a couple of hundred packets were stored in some buffer and the rest packets were dropped. Further work is needed to make sure what happened with the incoming packets and why it happened. But we do know that after the UDP timeout expired, the buffered packets were translated and sent out on the PLAT's IPv4 interface.

V. CONCLUSION

We successfully exhausted the available source port number range on the PLAT using DNS query messages in our testbed. Thus, we showed a possible DoS attack against the 464XLAT. It is worth mentioning that with the proper mitigations (multiple public IP addresses on the IPv4 interface of the PLAT or limiting connections based on source IP addresses), 464XLAT can be made more resilient to Denial of Service attacks, but we proved that it is possible to use all available port numbers to prevent the provider side device from working properly.

ACKNOWLEDGMENT

The authors thank Ameen Al-Azzawi for reviewing and commenting the manuscript of this paper.

REFERENCES

[1] M. Mawatri, M.Kawashima, C.Byrne: 464XLAT: Combination of Stateful and Stateless Translation, IETF RFC 6877, April 2013, DOI: 10.17487/RFC6877

[2] G. Lencse and Y. Kadobayashi, "Comprehensive survey of IPv6 transition technologies: A subjective classification for security analysis", *IEICE Transactions on Communications*, vol. E102-B, no.10, pp. 2021-2035. DOI: 10.1587/transcom.2018EBR0002

[3] G. Lencse and Y. Kadobayashi, "Methodology for the identification of potential security issues of different IPv6 transition technologies: Threat analysis of DNS64 and stateful NAT64", *Computers & Security* (Elsevier), vol. 77, no. 1, pp. 397-411, August 1, 2018, DOI: 10.1016/j.cose.2018.04.012

[4] A. Al-Azzawi, "Towards the security analysis of the five most prominent IPv4aaS technologies", *Acta Techica Jaurinensis*, vol. 13, no. 2, pp. 85–98, Mar. 2020. DOI: 10.14513/actatechjaur.v13.n2.530

[5] A. Shostack, *Threat Modeling: Designing for Security*, Wiley & Sons, Indianapolis, Indiana, USA, 2014.

[6] A. Al-Azzawi and G. Lencse, "Identification of the possible security issues of the 464XLAT IPv6 transition technology", *Infocommunications Journal*, vol. 13, no. 4, pp. 10-18, December 2021, DOI: 10.36244/ICJ.2021.4.2

[7] N. Lutchansky , "TAYGA: Simple, no-fuss NAT64 for Linux", http://www.litech.org/tayga/

[8] D. Bakai: "DNS64perf++: A C++14 DNS64 tester program", free software under GPLv2 license, https://github.com/bakaid/dns64perfpp

[9] G. Lencse, D. Bakai, "Design and implementation of a test program for benchmarking DNS64 servers", *IEICE Transactions on Communications*, vol. E100-B, no. 6. pp. 948-954, June 2017. DOI: 10.1587/transcom.2016EBN0007

[10] R. Rosen, "Netfilter". In: Linux Kernel Networking. Apress, Berkeley, CA. 2014. DOI: 10.1007/978-1-4302-6197-1_9

[11] O. Andreasson, "IP tables tutorial", chapter 4, The conntrack entries http://www.faqs.org/docs/iptables/theconntrackentries.html

[12] M. Sauter, "UDP NAT timeouts and how to chane them on Linux", https://blog.wirelessmoves.com/2015/06/udp-nat-timeouts-and-how-to-change-them.html

[13] H. Eychenne, "Netfilter conntrack performance tweaking v0.8", https://wiki.khnet.info/index.php/Conntrack_tuning

.