Ecdysis: Open-Source DNS64 and NAT64

Simon Perreault, Jean-Philippe Dionne, and Marc Blanchet
Viagénie, Québec City, Canada
{simon.perreault, jean-philippe.dionne, marc.blanchet}@viagenie.ca

February 8, 2010

Abstract

Ecdysis is an open-source projet that is developing open-source DNS64 and NAT64 implementations. These two protocols are used jointly for translation from IPv6 clients to IPv4 servers. The implementations are based on the Bind and Unbound DNS servers for DNS64, and on Linux's Netfilter and OpenBSD's pf for NAT64.

1 Introduction

IPv4 and IPv6 networks are "incompatible." The IETF recommendation has usually been to rely on dual-stack deployment: have both networks coexist until IPv6 takes over IPv4. However, IPv6 growth has been much slower than anticipated. Therefore, new IPv6-only deployments face an interesting challenge, that of communicating with the predominantly IPv4-only rest of the world. A similar problem is encountered when legacy IPv4-only devices need to reach the IPv6 Internet.

Translation between IPv4 and IPv6 is one framework engineered within the IETF as a solution to the problem of IPv6 transition. The general framework for IPv4/IPv6 translation is described in [4]. It also explains the background of the problem, and some expected uses. Another document describes the translation algorithm [6]. This stateless algorithm (one-to-one mapping between IPv4 and IPv6 addresses) has been deployed by CERNET and their experience is described in [9].

Given the increasing shortage of public IPv4 addresses, we will need to overload IPv4 addresses and share them among multiple IPv6 hosts. This is akin to the network address translation (NAT) function that is common in today's IPv4-only networks. NAT64 is the equivalent protocol for translating between IPv4 and IPv6 networks and is being developed by the IETF [7]. It makes use of a DNS application-layer gateway (ALG) protocol, DNS64 [8], that intercepts DNS AAAA queries sent by IPv6 hosts, performs an A queries on the IPv4 side, and synthesizes AAAA answers directing IPv6 hosts toward the NAT64. The DNS64 and NAT64 functions are completely separate, which is the key to understanding the superiority of NAT64/DNS64 over NAT-PT [2, 3].

The Ecdysis[1] project's goal is to develop open-source implementations of an IPv4/IPv6 gateway that run on open-source operating systems such as the various BSD flavours and Linux. The gateway is comprised of two distinct modules: the DNS64 and the NAT64. The DNS64 module was developed by modifying two open-source DNS servers: Bind and Unbound. The NAT64 module was developed by modifying pf (the firewall and NAT code in the OpenBSD kernel, which is used also in other BSD variants) and Netfilter (the firewall and NAT code in the Linux kernel). As part of the development process, stand-alone implementations of DNS64 and NAT64 were developed for experimentation purposes.

The deployment operation of an IPv4/IPv6 gateway need to be carefully planned and understood. Issues such as scalability, stability, and network management are exacerbated by the stateful nature of the protocols involved. Some of these issues are similar to those encountered when deploying IPv4 NAT devices, and are addressed in the same fashion, but other are different. In particular, the logical separation of the DNS64 and NAT64 functions will allow us to scale more efficiently. Also, NAT64

[1] http://ecdysis.viagenie.ca

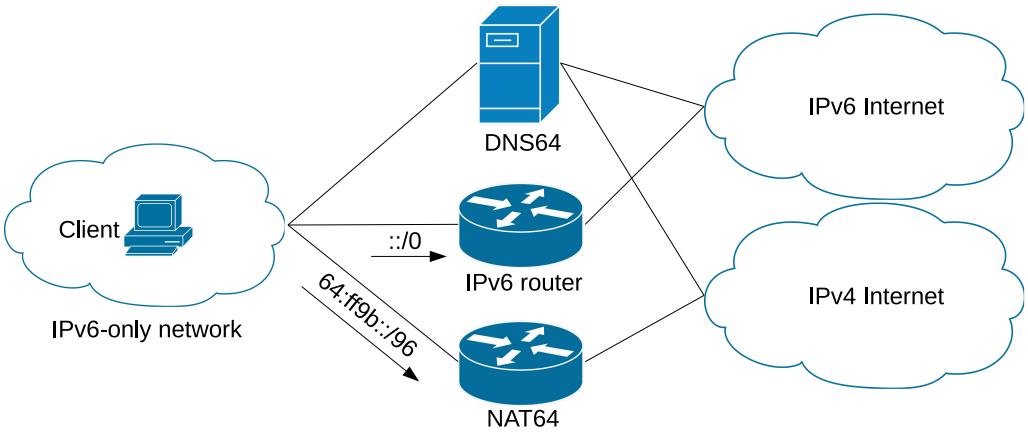

Figure 1: Expected use case. The default NAT64 prefix, 64:ff9b::/96, is routed to the NAT64 device. All other IPv6 traffic is routed to the default IPv6 gateway. The DNS64, which is provisioned to the clients as their default DNS resolver, has both IPv6 and IPv4 connectivity, which it uses for reaching authoritative DNS servers. Note that IPv6 router, DNS64, and NAT64 are logical functions and could conceivably be co-located on the same physical machine.

builds on a better understanding of NAT provided by years of experience. For example, it possesses characteristics allowing peer-to-peer communication in restricted but important cases. For example, it is expected that peer-to-peer SIP & RTP across the IPv4/IPv6 boundary will be attainable.

2 The Protocols

Here we give a brief overview of the DNS64 and NAT64 protocols. Please refer to Figure 1 on page 2, which illustrates an expected use case.

2.1 DNS64

In most cases, client-server connection establishment starts with a DNS query. This is the basic assumption upon which the IPv6 translation framework is built: the IPv6-only client will ask its DNS resolver for the AAAA record associated with the server's name.

This DNS resolver is augmented with DNS64 functionality. It first tries to resolve the AAAA query as usual. If the result of this is not an error but the answer section is empty, the resolver will initiate a new A query for the same host name. If this query succeeds and its answer section is not empty, then the A records are converted to AAAA records by prepending them with the NAT64 prefix. This new response is then sent to the client. Refer to Figure 2 on page 3 for an example.

It is important to note that DNS64 is partially incompatible with DNSSEC. If the client performs its own DNSSEC validation, and has no knowledge of the DNS64 function being performed by the server, then the validation will fail. However, if the client is somehow aware of the DNS64 function and knows how to convert synthetic AAAA records back to A records, then it may correctly validate them. Finally, the client may instead trust the server to perform DNSSEC validation in its stead. Since the server can do so before synthesizing AAAA records, this is compatible with DNSSEC.

2.2 NAT64

The specification of NAT64 is covered by two documents. The first part, stateless translation between IPv6 and IPv4 (in both directions), is described in [6]. It is an update to the Stateless IP/ICMP Translation Algorithm (SIIT) [1] which was also the basis of NAT-PT. It describes how to map IPv6 header fields to and from IPv4, as well as mapping ICMPv6 to and from ICMPv4.

Stateless translation can be used as-is when you can allocate one IPv4 address per host. For an ex-

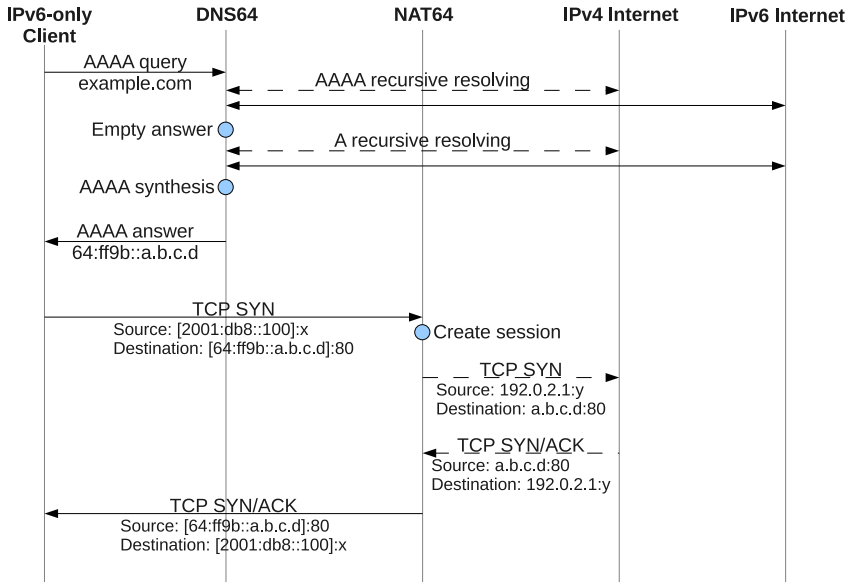

Figure 2: Example connection establishment flow. In this case, the IPv6-only client is attempting a TCP connection to example.com on port 80. There is no AAAA record associated with example.com, and so the DNS64 synthesizes one from the A record, which points to the IPv4 address *a.b.c.d*, by prefixing it with the NAT64 prefix, which in this case is the defautl 64:ff9b::/96. The client then sends a TCP SYN to 64:ff9b::a.b.c.d, port 80. The packet gets routed to the NAT64 device which creates a new session and allocates a binding. It changes the source address that of the binding created, and converts the IPv6 destination address to IPv4 by removing the prefix. When the SYN/ACK reply is received, the NAT64 does the reverse operation and forwards the IPv6 packet to the client. The rest of the connection operates in the same fashion.

ample, see [9]. Stateless translation is more robust than stateful translation because there is no state in the network. If a network element fails, it is easy to fail over to another without needing state synchronization. Furthermore, the absence of state makes it is easy to load-balance on multiple translators, employ asymmetric routing, etc.

However, in many cases it is impossible to allocate one IPv4 address per host. In these cases stateful translation [7] is needed to overload IPv4 addresses and have multiple IPv6-only hosts share them. The specification is limited to three transport protocols: UDP, TCP, and ICMP. Support for other protocols will be in separate documents. Other supporting functionality, such as an FTP ALG [10], is also in separate documents.

It is very important to note that connection initiation is only possible from the IPv6 side to the IPv4 side (except when statically configured otherwise). This simplifying assumption is the main difference with NAT-PT, where connection initiation was supported in both directions. Limiting the direction from the IPv6 side to IPv4 allows the complete separation between NAT64 and DNS64. Indeed, NAT64 could be used without DNS64 if the IPv6-only client is using another method to obtain IPv6 addresses of IPv4 servers (e.g. static configuration).

When an IPv6-only client initiates a connection, it sends a first packet (e.g. a TCP SYN) to an IPv6 address contained in a prefix routed to the NAT64 device. The default prefix is 64:ff9b::/96 (see [5]). The NAT64 device creates a new entry in its session table and allocates a binding, which indicates the source IPv4 address and port to use for the IPv4 side of the connection. The IPv4 destination address is extracted from the IPv6 destination address. Once the IPv4 source and destination addresses are known, the packet is translated following the stateless translation rules. It is

then routed and forwarded on IPv4. When packets belonging to the same connection arrive from the IPv4 or IPv6 side, the corresponding session entry is looked up in the session table. Its expiration is refreshed and its state is updated if necessary. The packet is then translated using the source and destination addresses and ports stored as part of the session entry.

See Figure 2 on page 3 for an example.

3 DNS64 Implementations

We implemented DNS64 functionality in the Bind[2] and Unbound[3] DNS servers.

3.1 Bind

The implementation in Bind modifies the core resolving state machine, namely the `query_find()` function. It also introduces a new configuration variable named `dns64-prefix`, which takes as argument the IPv6 prefix that will be prepended to IPv4 addresses contained in A records in order to synthesize AAAA records. This variable can be used either in the global options context or in a view context. When it is not present, the DNS64 functionality is disabled.

3.2 Unbound

Unbound's modular architecture let us implement DNS64 as a completely separate and self-contained dynamically loadable module. In Unbound, modules are arranged in a chain and requests traverse them as their resolving progresses. The usual modules are `iterator.so`, where the core DNS iterating algorithm takes place, and `validator.so`, which performs DNSSEC processing. Our module, `dns64.so`, is placed in front of these two. An incoming request is immediately passed to the next module in the chain. When it comes back, its status is checked. If DNS64 processing needs to take place, a new A request is generated and sent to the next module in the chain. When this request comes back, AAAA records are synthesized and the new response is returned to the previous module in the chain. Since we are usually the first module in the chain, this causes Unbound's core to send the response to the client.

Usage of the DNS64 functionality in Unbound depends on whether `dns64.so` is used or not. This is configured in `unbound.conf`. We also created a new configuration variable named `dns64-prefix`, which specifies the DNS64 prefix to be used.

4 NAT64 Implementations

We implemented NAT64 functionality as a standalone user-space program, as a Linux module, and as part of OpenBSD's pf [4].

4.1 User-Space

Our user-space implementation, `nat64d`, uses a `tun(4)` device to receive and send packets. This interface was chosen because it is fairly portable and is easy to use. When the program starts it creates the interface. The user then needs to manually configure IPv4 and IPv6 routing so that this interface receives packets intended to the NAT64 translator. This is usually done with `ifconfig(8)` and the exact syntax varies from one OS to another.

Our user-space implementation assumes it has complete ownership of the IPv4 address it is configured to use (address pools are for future development). It cannot share an IPv4 address assigned to another interface. Therefore, for most usage two IPv4 addresses will be needed: one for the physical interface and used to access the box, and another used by `nat64d`. This design choice was made for implementation simplicity.

The session table is implemented using red-black trees from `sys/tree.h`. Acceleration using a hash table is for future development.

When `nat64d` receives the SIGUSR1 signal, it prints its session table to stdout. The SIGHUP signal causes it to flush its session table.

4.2 Linux

Our Linux implementation takes the code from the user-space implementation and puts it in kernel-space, in a module called `nf_nat64.o`. Packet input using `tun(4)` is replaced by a Netfilter hook.

[2] https://www.isc.org/software/bind
[3] http://www.unbound.net/
[4] http://www.openbsd.org/faq/pf/

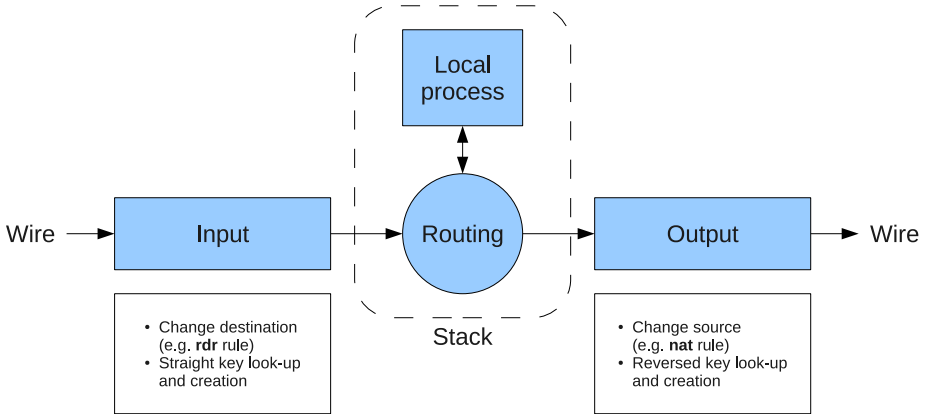

Figure 3: Pf processing.

Output is accomplished with a dummy `nat64` interface to which the translated packets are sent using `netif_rx()`.

4.3 OpenBSD's pf

Adding NAT64 functionality to pf meant fighting pre-existing assumptions in pf's code.

Pf processes forwarded packets in two steps: in the input direction and in the output direction (see Figure 3 on page 5). The existing code assumes that the packet's address family doesn't change as it traverses pf. For example, `ip6_input()` calls `pf_test6()` with the `dir` parameter set to `PF_IN` to perform input processing, then it passes the packet to `ip6_forward()`. There is no way for `pf_test6()` to signal to the calling function that the IPv6 packet has now become an IPv4 packet and that it should call `ip_forward()` instead. Furthermore, note that the destination address always needs to be changed before routing happens. If we try to change the source address in the input direction (instead of the output direction as usual), then returning packets in the same flow will match against the state key only in the output direction, which is too late to change the destination address.

The design that was adopted circumvents these issues by two means:

- We completely translate packets in the input direction, feed the translated packet back to the main input function (either `ipv4_input()` or `ip6_input()`), and stop processing of the original packet by replacing it by a null pointer. The necessary functionality to accomplish this was already present and this is rather clean.
- The state key that is created has a straight wire key but a reversed stack key. This makes the stack key match the wire key of returning packets so that we can do the reverse processing in the input direction (before routing) also. These special "half-reversed" state keys are recognized when the address family of the wire key doesn't equal that of the stack key. It was necessary to add such checks in various locations in pf's code.

The syntax of NAT64 rules in `pf.conf` is identical to that of regular NAT rule, except that the "nat64" action is used instead. For example:

```
nat64 from any to 64:ff9b::/96 -> (em0)
```

It is planned to also create an "rdr64" action for statically mapping IPv4 address-port combinations to IPv6.

4.4 Comparison

The user-space implementation tries to be as close to the IETF specification as possible. Since the Linux implementation reuses this code, it has the same behaviour. In contrast, the pf implementation is built on top of pf's already existing NAT code, which behaves differently. This has both advantages and disadvantages. For example, the IETF

specification mandates endpoint-independent mapping behaviour, which facilitates NAT traversal. On the other hand, it greatly reduces scalability. Since pf uses port- and address-dependent mapping, it can make more efficient use of IPv4 addresses at the cost of making NAT traversal harder.

Another difference is the level of integration. The Linux implementation is mostly separate from the existing Netfilter connection tracking code. The pf implementation is very tightly integrated, which makes it benefit automatically from compatibility with other parts of the system such as user-space tools (e.g. `pfctl(8)`, `systat(1)`) and state synchronization using pfsync.

5 Operational Issues

When deploying NAT in medium to large networks, there are usually two concerns that need to be addressed: fail-over and load balancing. These also apply to NAT64.

5.1 Fail-Over

The strategy is exactly the same as for usual NAT: replicate the state on a warm backup and automatically direct traffic to it when the main box fails. This is accomplished on OpenBSD using pfsync and CARP. These methods are also applicable with our NAT64 implementation.

5.2 Load Balancing

The separation of DNS64 and NAT64 functions enables a very powerful form of load-balancing.

Each NAT64 device, or pair of devices in a fail-over configuration, is assigned a different NAT64 prefix and pool of IPv4 addresses. The multiple prefixes are configured in the DNS64 which picks one of them when synthesizing AAAA records. Many strategies for choosing a prefix can be envisaged: round-robin, random, based on the source address of the client, using feedback from polling the load on the NAT64 devices, etc.

Our DNS64 patches currently do not support this feature. It is planned for future work.

5.3 Lessons Learned

While running NAT64 we have encountered surprising issues. For example, things that work behind an IPv4 NAT do not necessarily also work from behind a NAT64. This happens when a connection attempt is made directly to an IPv4 address, without using the DNS. For example, some web pages use IPv4 address literals in hyperlinks. To make these links work, one solution is to use an IPv6-enabled HTTP proxy. An clever hack using regular expressions is described in [11].

Another problem encountered was with authoritative DNS servers replying with an error to AAAA queries but normally to A queries. Originally, the DNS64 protocol did not attempt to synthesize records when any error was encountered. Since we reported this problem to the protocol authors, synthesis is attempted for all DNS errors except NXDOMAIN.

Finally, all protocols that transport IPv4 addresses in their payload cannot be assumed to work with NAT64. Besides the usual suspects (FTP, SIP, etc.), we encountered a few unexpected ones. For example, a very popular instant-chat protocol redirects the client to a server identified by an IPv4 address as soon as the client tries to register. This makes the program idle in the "Connecting..." phase.

There are a few minor issues with client operating systems when they are configured in IPv6-only mode. For example, we have seen Firefox think it is not connected and enter the "Offline" mode automatically. These are minor issues in that the functionality is still present but users may stumble on rough corners

Overall, NAT64 is very usable for the common "web and email" style. Some remaining issues, such as the instant-chat program example discussed above, can be mitigated fairly easily by developing a simple ALG until the application developers fix their bug. Others, such as SIP usage, require much more ingenuity.

6 Conclusion

The Ecdysis project has developed multiple open-source DNS64 and NAT64 implementations. They are available at http://ecdysis.viagenie.ca. Our ul-

timate goal is for the patches to evolve and eventually be accepted for inclusion by the Bind, Unbound, Linux, and OpenBSD projects.

With this code, you can now remove IPv4 from your network, go IPv6-only, and still be able to talk to the rest of the world. That's one less barrier in the way of IPv6 adoption.

Acknowledgements

This work was funded by the NLnet Foundation and Viagénie.

References

[1] Nordmark, E., *RFC2765: Stateless IP/ICMP Translation Algorithm (SIIT)*, February 2000.

[2] Tsirtsis, G. and P. Srisuresh, *RFC2766: Network Address Translation - Protocol Translation (NAT-PT)*, February 2000.

[3] Aoun, C. and E. Davies, *RFC4966: Reasons to Move the Network Address Translator – Protocol Translator (NAT-PT) to Historic Status*, July 2007.

[4] Baker, F., X. Li, C. Bao, and K. Yin, *draft-ietf-behave-v4v6-framework-05: Framework for IPv4/IPv6 Translation*, January 2010.

[5] Huitema, C., C. Bao, M. Bagnulo, M. Boucadair, X. Li, *draft-ietf-behave-address-format-04: IPv6 Addressing of IPv4/IPv6 Translators*, January 2010.

[6] Li, X., C. Bao, and F. Baker, *draft-ietf-behave-v6v4-xlate-06: IP/ICMP Translation Algorithm*, January 2010.

[7] Bagnulo, M., P. Matthews, and I. van Beijnum, *draft-ietf-behave-v6v4-xlate-stateful-08: NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers*, January 2010.

[8] Bagnulo, M., A. Sullivan, P. Matthews, and I. van Beijnum, *draft-ietf-behave-dns64-05: DNS64: DNS extensions for Network Address Translation from IPv6 Clients to IPv4 Servers*, December 2009.

[9] Li, X., C. Bao, and F. Baker, *draft-xli-behave-ivi-07: IP/ICMP Translation Algorithm*, January 2010.

[10] van Beijnum, I., *draft-ietf-behave-ftp64-00: IPv6-to-IPv4 translation FTP considerations*, December 2009.

[11] Wing, D., *draft-wing-behave-http-ip-address-literals-01: Coping with IP Address Literals in HTTP URIs with IPv6/IPv4 Translators*, October 2009.