ICANN Controlled Interruption IPv6 Research Study - Report 2

Author: Sinodun IT
Date: 7th Oct 2025
Version: 1.3

Version	Date	Comments
0.1	5 Aug 2025	Preliminary draft report
0.2	19 Aug 2025	Details of basic dual stack desktop and mobile testing
0.3	2 Sep 2025	Updated with details of dual stack coverage and error codes
0.4	9 Sep 2025	Include IPv6 only coverage and basic results
1.0	15 Sep 2025	Additional IPv6 only testing and OS testing
1.1	19 Sep 2025	Mobile IPv6 only testing
1.2	24 Sep 2025	Add links to other documents and context on CI
1.3	7th Oct 2025	Add paragraph on expanding IPv6 loopback prefix

Introduction

This study is to identify one or more IPv6 addresses that may be suitable for use during Controlled Interruption. This study will involve a review of the IANA registries for IPv6 to produce a list of candidate addresses, and technical tests of popular end user operating systems to determine whether or not DNS lookups that respond with those addresses result in external network traffic.

This document is part of a set of documents covering the results of the study:

- ICANN Controlled Interruption IPv6 Research Study - Report 1
- ICANN Controlled Interruption IPv6 Research Study - Report 2 (this document)
- ICANN Controlled Interruption IPv6 Research Study - Summary Report

Controlled Interruption

Controlled Interruption is a phase in the establishment of a new generic top-level domain that is designed to address the risk of Name Collision [ICANN-NC]

During Controlled Interruption, certain DNS resource records are published at and below the gTLD name that are designed to interrupt resolution processes in such a way as to minimize any harm that arises from such interruption.

For example, A, MX, TXT and SRV resource records are published at the apex of the DNS zone, and as wildcards, that contain information intended to inform network administrators that a name collision has occurred:

```
example. IN A 127.0.53.53
example. IN MX 10 your-dns-needs-immediate-attention.example.
example. IN SRV 10 10 0 your-dns-needs-immediate-attention.example.
example. IN TXT ("Your DNS configuration needs immediate attention
                 see https://namecollisions.icann.org/")

* IN A 127.0.53.53
* IN MX 10 your-dns-needs-immediate-attention.example.
```

```
* IN SRV 10 10 0 your-dns-needs-immediate-attention.example.
* IN TXT ("Your DNS configuration needs immediate attention see
                    https://namecollisions.icann.org/")
```

Controlled Interruption uses A resource records containing the IPv4 address 127.0.53.53. This address is within the 127.0.0.0/8 prefix, which is used for "loopback" interfaces (see RFC 1122, Section 3.2.1.2). The use of this prefix ensures that any application that uses the result of a DNS query that returns this IPv4 address will not initiate a network connection. 127.0.53.53 is used instead of 127.0.0.1 so that if a user or administrator performs a web search for this address, they are more likely to be shown links to the Name Collision resources on ICANN's web site.

When the concept of Controlled Interruption was originally developed in 2014, no suitable equivalent address could be found for IPv6-enabled hosts. While this was noted as a limitation, at the time the impact was limited since fewer than 5% of internet hosts supported IPv6. In the intervening decade, IPv6 adoption has accelerated, and now more than 40% of all internet hosts have IPv6 connectivity. As a result, the lack of an IPv6 solution for Controlled Interruption has a much greater impact. Even though most clients are dual-stack, some clients may not query for both IPv4 and IPv6 addresses.

ICANNs preferred approach would be to use an IPv6 loopback address analogous to the existing IPv4 `127.0.53.53` address, however, IPv6 has only a single loopback address allocated (::1 [RFC 4291]). Two proposals within the IPv6 Operations working group at the IETF to assign a larger loopback prefix have not gained consensus and as a result there is still no direct equivalent IPv6 address available today. Should such an address become available in future ICANN would intend to utilize it for Controlled Interruption. In lieu of that, and with a recognition of the current impact of not publishing a AAAA record for Controlled Interruption, an alternative option that can be deployed as an interim solution is highly desirable.

Scope of work

The study shall carry out a review of all the registrations in the Internet Protocol Version 6 Address Space [IANA-IPv6AS] and IANA IPv6 Special-Purpose Address Registry registries, and any other relevant resources, to develop a list of [IANA-IPv6SPAR] candidate address prefixes.

A prefix may be considered a candidate if the specifications indicate that packets with a destination address within that prefix will remain local to the host (i.e., packets will not be transmitted outbound from the host). If no such prefixes are found, second-best options should also be considered, such as prefixes within the link-local scope which are unlikely to exist on the local network.

The consultant will then perform benchtop tests of popular end-user operating systems (including but not limited to Windows 10 and 11, and all not end-of-life versions of macOS, iOS, Android, Debian, and Red Hat Enterprise Linux) and applications such as browsers (including

but not limited to Chrome, Firefox, Safari and Edge). Test systems must be freshly installed and up-to date, and configured with IPv6 connectivity only. Test systems should also be configurable as dual-stack in order to determine if system behaviour changes in case both IPv4 and IPv6 connectivity is available. The consultant will then test the candidate addresses using a range of browsers (including but not limited to Chrome, Firefox, Safari, and Edge, where available for the operating system), command-line tools (including but not limited to curl, ssh, netcat, openssl s_client), and popular applications (including those that act as clients even if they are categorized as servers, for example, HTTP proxy servers, and SMTP servers that relay email to other hosts) for those systems.

Deliverables

The study shall will produce a report in a mutually agreed format which includes:

1. A summary of the document review, which includes the key data points as to why specific IPv6 prefixes were selected/rejected as candidates;

2. A description of the methodology for performing benchtop tests, including equipment and software used and test procedures in sufficient detail to allow independent replication

3. Results of the tests indicating for each IPv6 address, operating system, and application, whether or not the tests resulted in network packets leaving the test system, and under what circumstances such network traffic occurred.

4. A clear recommendation to ICANN regarding what IPv6 address to choose for controlled interruption

Item 1 was covered in ICANN Controlled Interruption IPv6 Research Study - Report 1. This report constitutes items 2,3 and 4 of the above deliverables. A summary of the findings in a format suitable for public review are available in the draft Summary report.

Summary of Stage 1 testing

A range of types of candidate prefixes were identified for initial testing based on the criteria specified in Report 1.

Candidate class	Test Name	Address Block/Address	Notes
First class	Mapped	::ffff:0:0/96	IPv4-mapped IPv6 addresses [Section 2.5.5.2 of RFC4291]
	Dummy	100:0:0:1::/64	Dummy IPv6 Prefix [RFC9780]
	DocumentationA	2001:db8::/32	Documentation prefix [RFC3849]
	DocumentationB	3fff::/20	Documentation prefix [RFC9637]
	MultNodeInterface	ff01::1	Predefined multicast address - Node-Local Scope Multicast Address, All Nodes Address [RFC4291]

Candidate class	Test Name	Address Block/Address	Notes
	MultNodeLink	ff02::2	Predefined multicast address - Link-Local Scope Multicast Address, All Nodes Address [RFC4291]
	MultRouterInterface	ff01::2	Predefined multicast address - Node-Local Scope Multicast Address, All Routers Address [RFC4291]
Second class	Partially Unallocated	::/128	Partially Unallocate Address space [RFC4291]
	UniqueLocalUnicast	fc00::/10	Unique-Local Unicast [RFC4193] [RFC8190]
	LinkLocalUnicast	fe80::/10	Link-Local Unicast [RFC4291]

Specific test addresses within these prefixes were tested in basic black-box tests on a subset of OS versions (Windows 11, macOS Sequoia and Ubuntu server 24.04) for a subset of applications as described in the Stage 1 Test report. Most of the candidate prefixes resulted in network traffic being transmitted off the host machine and so were ruled out, see the Stage 1 Results.

The table below shows the 3 types of candidate prefix that passed the Stage 1 testing

- IPv4-mapped IPv6 addresses
- Pre-defined multicast addresses and
- Link local unicast addresses

and the 7 specific addresses tested. Note the IPv4 address 127.0.53.53 represented in (a partial) IPv6 presentation format is 7f00:3535.

Test Name	Test Address	Network traffic seen	Loopback traffic seen
Mapped1	::ffff:0:1	N	N
Mapped2	::ffff:7f00:3535	N	Y (IPv4 traffic to 127.0.53.53)
MultNodeInterface	ff01::1	N	N
MultNodeLink	ff01::2	N	N
MultRouterInterface	ff02::1	N	N
LinkLocalUnicast1	fe80::1	N	N
LinkLocalUnicast2	fe80::7f00:3535	N	N

Stage 2 testing overview

Methodology

Appendix A contains detailed methodologies for setting up test environments and executing tests. Some relevant details are listed here to provide context to the results.

Compared to the Stage 1 testing we refined our testing script and tested across a fuller matrix of OS versions and applications.

Status of testing

Overview

This stage of the study involved initial testing in a dual stack configuration across a suite of OS's, OS versions and applications. Further testing was performed in an IPv6 only configuration on a subset of the above testing matrix (based on observations from the initial testing).

The applications tested fell broadly into 3 categories. These are (specifics are dependant on the platform):

1. **CLI tools** - `ssh, curl, nc, openssl s_client`
2. **Browsers** - Chrome, Firefox, Microsoft Edge and Safari
3. **Proxies/Mail servers** - e.g. `nginx` or `exim`. These are 'servers' that can also behave as clients when connecting to other hosts.

The applications were invoked with a test name that resolved to the test address and then monitored. Alongside the test candidates we include several reference addresses in the testing:

1. The existing IPv4 Controlled Interruption address 127.0.53.53
2. The IPv6 loopback address ::1
3. The icann.org website which has both A and AAAA records
4. A name that resolves to NXDOMAIN for both A and AAAA record

These ensure the test environment is working and also provide a baseline of behaviour for such addresses.

The table below illustrates the testing coverage of this study.

- Blue is tested with candidate prefixes
- White is will not or cannot be tested

Environment	Dual stack	IPv6 only
Desktop - CLI tools + Browsers	✅	✅
Desktop - Proxies/Mail servers	✅	✅
Mobile - Browsers	✅	✅

Dual Stack testing

Desktop/server OSs

The following desktop/server OS's have been tested using VMs:

- Ubuntu 25-04-desktop
- Ubuntu 24.04-server
- RedHat 10.0
- MacOS Ventura (v13)
- MacOS Sonoma (v14)
- MacOS Sequoia (v15)
- Windows 10
- Windows 11

Dual stack desktop coverage

The following table shows the matrix of OS's and applications tested.

Application/Platform	U 24.04 server	U 25.05 desktop	RedHat 10	FreeBSD 14.3	macOS 13	macOS 14	macOS 15	Windows 10	Windows 11
ssh	✅	✅	✅	✅	✅	✅	✅	✅	✅
curl	✅	✅	✅	✅	✅	✅	✅	✅	✅
netcat	✅	✅	✅	✅	✅	✅	✅		
openssl s_client	✅	✅	✅	✅	✅	✅	✅		
Chrome		✅	✅		✅	✅	✅	✅	✅
Firefox		✅	✅		✅	✅	✅	✅	✅
Edge		✅	✅		✅	✅	✅	✅	✅
Safari					✅	✅	✅		
Sendmail	✅								
Postfix	✅								
Exim	✅								
nginx	✅						✅		
Apache	✅						✅		✅

Application/Platform	U 24.04 server	U 25.05 desktop	RedHat 10	FreeBSD 14.3	macOS 13	macOS 14	macOS 15	Windows 10	Windows 11
HAProxy	✅						✅		

Only applications supported directly on platforms were tested e.g. nothing was tested on Windows that required an additional environment (cygwin/mingw/etc.) to be configured. Only applications with stated support for the platform were tested, even if they could be run.

Mobile OS's

The following mobile platforms were tested:

- iOS 17 and 18 - Physical devices
- Android 14, 15 and 16 - Simulators
- Android 15 Pixel 9a - Physical device

Dual stack mobile coverage

The following table shows the matrix of mobile platforms and applications tested.

Application	iOS 17 - physical device	iOS 18 - physical device	Android 14	Android 15	Android 16	Android 15 - physical device
Chrome	✅	✅	✅	✅	✅	✅
Firefox	✅	✅	✅	✅	✅	✅
Safari	✅	✅				

IPv6 only testing

We note that there is currently no specific definition of a 'IPv6 only'(we note the `v6ops` Working Group at IETF has an action item to define IPV6 only) and that for our testing purposes we will simply use desktop hosts that, compared to the dual stack configuration:

- Have IPv4 disabled on the network interface
- Still have an IPv4 loopback interface and address.

Based on the results of the dual stack testing (see later) we ran the IPv6 testing on only the latest version of each desktop OS:

- Ubuntu 25-04-desktop/Ubuntu 24-04-server
- macOS Sequoia (v15)
- Windows 11

and on a subset of the original candidate prefix list:

- Mapped2: ::ffff:7f00:3535
- Multnodeinterface: ff01::1

- Linklocalunicast2: fe80::7f00:3535

IPv6 only desktop coverage

Application	U 24.04 server	U 25.05 desktop	FreeBSD 14.3	OpenBSD 7.7	macOS 15	Windows 11
ssh	✅		✅	✅	✅	✅
curl	✅		✅	✅	✅	✅
netcat	✅		✅	✅	✅	
openssl s_client	✅		✅	✅	✅	
Chrome		✅			✅	✅
Firefox		✅			✅	✅
Safari					✅	
Sendmail	✅					
Postfix	✅					
Exim	✅					
nginx	✅				✅	
Apache	✅				✅	✅
HAProxy	✅				✅	

IPv6 only mobile coverage

For this, we tested in an ‘IPv6-mostly’ wifi network i.e. the router is configured to recognise DHCP option 108 (RFC 8925 - IPv6-only preferred option) and so clients with a CLAT implementation (including Android and iOS) will not obtain an IPv4 address from the network.

Application	iOS 18 - physical device	Android 15 - physical device
Chrome	✅	✅
Firefox	✅	✅

Application	iOS 18 - physical device	Android 15 - physical device
Safari	✅	

Dual stack testing Results

General observation from dual stack testing

1. We see no difference in the behaviour between different versions any of the tested OS's

2. We see no difference between Ubuntu and RedHat.

3. We see no difference in the traffic patterns between FreeBSD and OpenBSD, but do see different error messages.

4. On a specific platform, we see no difference in the network traffic generated for each candidate prefix between different applications. However the network traffic patterns do vary between platforms. (The only exception is that Safari on macOS generates Neighbour solicitation traffic when other applications do not)

5. We see no difference between Chrome and Edge, which is not surprising since Edge is based on Chrome. For simplicity we just use Chrome from this point on in the document to refer to both browsers

6. We see no difference in behaviour between the 3 multicast addresses

7. We see no difference in behavior between the 2 link local unicast addresses.

8. We do see differences in the behaviour for different applications and OS's with respect to
    a. loopback traffic generated and
    b. error codes produced

Summary of results for network interface traffic only

The table below shows observed network traffic for tests on listed OS's. The behavior of the existing IPv4 Controlled interruption address is provided here as a reference point.

OS	IPv4 CI address (reference)		Mapped1	Mapped2	Mult* addresses	LinkLocalUnicast* addresses
Ubuntu/RedHat	N		Y	N	N	N
FreeBSD/OpenBSD	N			N	N	N
macOS	N		N	N	N	Y(1)
Windows 11	N		N	N	N	Y(1)
iOs	N		Y	N	N	Y(1)
Android	N		Y	N	N	N

(1) We see Neighbour solicitations for the candidate prefix for all apps on Windows and iOS and Safari on macOS.

Observations

1. `mapped1`: We see here that in some scenarios, the address `mapped1` resulted in traffic leaving the host with a destination address of the candidate prefix over the network connection, which rules this out as a candidate for further testing. It is not reported in later sections for simplicity.
2. `linklocalunicast`*: For both the `linklocalunicast` addresses tested we observed ICMPv6 Neighbour Solicitation traffic leaving the host for certain platforms/apps (presumably due to a default local scope for such addresses). These packets were missed in the Stage 1 testing and initial Stage 2 testing due to the packet capture processing filters being too limited. An example of such packet is:

```
fe80::5b9f:f644:a351:4f96 → ff02::1:ff00:3535 ICMPv6 86 Neighbor Solicitation for fe80::7f00:3535 from 52:54:00:cc:76:ee
```

The packet can be seen arriving at other hosts on the same network. While this traffic is not globally routable it is conceivable that a host on the local network could be configured with the candidate prefix as its link local address.

DISCUSSION POINT: We consider that the ICMPv6 traffic observed for the `linklocalunicast` addresses significantly reduces the attractiveness of them as candidate prefixes, even though it is highly unlikely that a host on the local network would actually be configured with the candidate prefix. We discuss this further in the conclusion.

Results for loopback interface traffic only

For completeness, we list here the behaviour with respect to traffic observed on the loopback interface.

1. Here we split the behaviour between the types of application as different behavior is seen for the various types of application.
2. As before behavior of the existing IPv4 Controlled interruption address is provided here as a reference point.

NOTE: With our current methodology we do not observe the loopback interface for the iOS mobile device

OS	App	IPv4 CI address (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
Ubuntu/RedHat	CLI commands	Y		Y	N	N
	Browsers	N		Y	N	N
	nginx	Y		Y	N	N
	exim	N		N	N	N
	exim-mod(1)	Y		Y	N	N
FreeBSD/OpenBSD	CLI commands	N		N	N	N
macOS	CLI commands	Y		Y(2)	N	N
	Browsers	Y(3)		Y(4)	N	N
	nginx	Y		Y	N	N
Windows 11	CLI commands	Y		Y	N	Y(5)
	Browsers	N		N	N	Y(5)
Android	Browsers	N		Y(6)	N	N

1. By default exim is configured to ignore multiple address ranges including some candidate prefixes, see App configuration. Here exim was tested with that configuration removed.
2. Only for `ssh`. Not observed for `curl`, `nc` or `openssl s_client`
3. Only for Safari.
4. Not seen for Firefox
5. ICMP packet observed with ‘Address unreachable’ seen for the address
6. Chrome but not Firefox

Observations

1. `mapped2`: This address behaves similarly to the `ipv4ci` address in that it produces IPv4 traffic on the loopback address in some scenarios.
2. `mult*`: None of the multicast addresses are observed to generate traffic on loopback
3. Interestingly we do observe ICMPv6 Destination Unreachable packets captured on the loopback interface on Windows that originate from the host network link local IPv6 address and have the Destination Address of the candidate prefix. This behaviour is possibly an artifact of the implementation of loopback interface on Windows.

```
fe80::5b9f:f644:a351:4f96 → fe80::5b9f:f644:a351:4f96 ICMPv6 124 Destination Unreachable (Address unreachable)
```

Error codes resulting from desktop name resolution

Since we have multiple candidates at this point in the testing, for completeness we list the error codes output by the tools tested when using the various candidates. More detailed tables and a sample of the detailed error code text is listed in Appendix B.

Since the results are a 3 dimensional matrix (OS, App, Address) we take the behaviour on macOS to be a baseline and present this first. The following tables only show content where the behaviour on that OS is different to that on macOS.

We group the error codes into a few general types:

- Application hangs, appear to try to connect or does try to connect with no error produced
- Connection failure/REFUSED/TIMEOUT
- Resolution failure
- Address not supported
- No route to host
- Unreachable
- Invalid argument/Address invalid

macOS baseline

App	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
ssh	Connecting…	Could not resolve		Connecting…	Address not supported	No route to host
curl	Connecting…	Could not resolve		Could not resolve	Can't connect	Can't connect
nc	Connecting…	Could not resolve		Could not resolve	Address not supported	No route to host
openssl	Connecting…	Can't connect		Can't connect	Can't connect	No route to host
Safari	Connecting…	Can't connect		Connecting…	Could not resolve	Can't connect
Firefox	Can't connect	Can't connect		Can't connect	Connecting…	Can't connect
Chrome	Could not resolve	Could not resolve		Connecting…	UNREACHABLE	UNREACHABLE
nginx	Connecting…	Could not resolve		Connecting…	Address not supported	No route to host
Apache	Could not resolve	Could not resolve		Could not resolve	Address not supported	No route to host
HAproxy	Can't connect	Could not resolve		Could not resolve	Can't connect	Can't connect

Ubuntu differences to macOS

App	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
ssh					UNREACHABLE	INVALID ARGUEMENT
curl	Can't connect			Can't connect		
nc	Can't connect			Can't connect	UNREACHABLE	INVALID ARGUEMENT
openssl	Can't connect	Could not resolve		Can't connect	Can't connect	Can't connect
Firefox					Can't connect	Connecting…
Chrome				Can't connect		INVALID ARGUEMENT
nginx	Can't connect			Can't connect	UNREACHABLE	INVALID ARGUEMENT
apache	Can't connect			Can't connect	UNREACHABLE	INVALID ARGUEMENT
haproxy						
exim	Could not resolve	No route to host		Could not resolve	UNREACHABLE	Could not resolve
exim-mod	Can't connect	No route to host		Can't connect	UNREACHABLE	INVALID ARGUEMENT
postfix	Loops back to self	Could not resolve		Loops back to self	UNREACHABLE	INVALID ARGUEMENT
sendmail	Can't connect	No route to host		Can't connect	UNREACHABLE	INVALID ARGUEMENT

FreeBSD differences to macOS

App	IPv4 CI address (reference)*	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
ssh	Can't assign requested address/ No route to host			INVALID ARGUEMENT		UNREACHABLE
curl	Can't assign requested address/ No route to host			INVALID ARGUEMENT	Address not supported	UNREACHABLE
nc	Can't assign requested address/ No route to host			INVALID ARGUEMENT		UNREACHABLE
openssl	Can't assign requested address/Address family not supported	Could not resolve		INVALID ARGUEMENT	Address not supported	UNREACHABLE

* We saw different error codes on the dual stack vs IPv6 only testing here. The dual stack result is given first.

OpenBSD differences to macOS

App	IPv4 CI address (reference)*	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
ssh	UNREACHABLE			Can't assign requested address	INVALID ARGUEMENT	UNREACHABLE
curl	UNREACHABLE			Can't assign requested address	INVALID ARGUEMENT	UNREACHABLE
nc	UNREACHABLE			Can't assign requested address	INVALID ARGUEMENT	UNREACHABLE
openssl	UNREACHABLE	Could not resolve		Can't assign requested address	INVALID ARGUEMENT	UNREACHABLE

Note that ssh, nc and openssl default to using IPv4 address if available. Also for security reasons, OpenBSD does not route IPv4 traffic to an AF_INET6 socket, and does not support IPv4 mapped addresses, see FreeBSD/OpenBSD

Windows differences to macOS

App	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
ssh					Unknown error	Connecting…
curl	Can't connect			Can't connect		Connecting…
Firefox					Can't connect	Connecting…
Chrome				Could not resolve	INVALID ARGUEMENT	Can't connect
Apache				Can't connect	INVALID ARGUEMENT	

Error codes resulting from mobile name resolution

We give full results here for each mobile OS.

iOS

App	IPv4 CI address (reference) (1)	Nonexistent (reference)		Mapped2 (2)	Mult* Addresses (1)	LinkLocalUnicast* addresses
Safari	Connecting…	Could not resolve		Connecting…	Connecting…	Can't connect
Firefox	Connecting…	Could not resolve		Connecting…	Connecting…	Can't connect
Chrome	Connecting…	Could not resolve		Connecting…	Can't connect	Can't connect

1. Produces 'Could not resolve' in IPv6 only testing' for all browsers

Android

App	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
Firefox	Could not resolve	Could not resolve		Could not resolve	Can't connect	Connecting…
Chrome	Could not resolve	Could not resolve		Can't connect	Can't connect (1)	INVALID ARGUEMENT

1. Produced 'Unreachable' in IPv6 only testing

Observations on error code patterns

1. `ipv4ci`: The existing IPv4 Ci address gives a mixture of connection attempts, connection failures and resolution failures, except on the *BSD systems where they give different errors.
2. `mapped2`: The `mapped2` address gives a similar but not identical pattern of errors.
3. `mult*` addresses: This set of addresses give a mixture of other error codes e.g Address not supported, Unreachable, Address invalid and Unknown error
4. `linklocalunicast`* addresses: These addresses give a mixture of other error codes e.g. No route to host, Invalid argument and timeout.

DNS lookups using Browser Developer tool

We note that Firefox and Chrome/Edge provide developer tools to see the results of DNS lookups, clear the DNS cache and also to do one-off DNS lookups through the browser. Here, we show the results of performing a one-off lookup using those tools. The tools are accessed by using the following URL in the browser

- Firefox: about:networking#dnslookuptool
- Chrome: chrome://net-internals/#dns
- Edge: edge://net-internals/#dns

This section is not directly applicable to Safari as it uses the System DNS resolver/cache and does not have a separate DNS tool for the browser, and so we present below the result of the system dns tool on macOS: `host <domain_name>`
(which provides an extended output of that shown by `dscacheutil -q host -a name <domain_name>`)

We see 3 categories of result:

- IP Address - Return of an IP address
- NXDOMAIN type error - NS_ERROR_UNKNOWN_HOST or ERR_NAME_NOT_RESOLVED
- Name Collision specific error - ERR_ICANN_NAME_COLLISION (see below)

macOS baseline

	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
Safari	IP Address	NXDOMAIN		IP Address	IP Address	IP Address
Firefox	NXDOMAIN	NXDOMAIN		NXDOMAIN	IP Address	IP Address
Chrome/Edge	Name Collision	NXDOMAIN		IP Address	IP Address	IP Address

Ubuntu to macOS differences

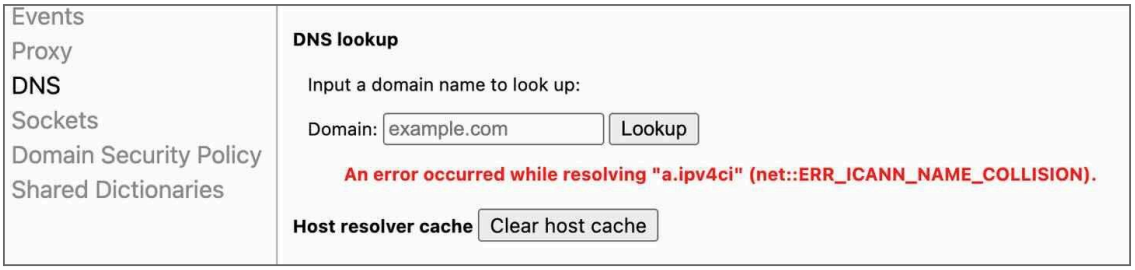
	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
Firefox				IP Address		
Chrome/Edge						

Windows to macOS differences

	IPv4 CI address (reference)	Nonexistent (reference)		Mapped2	Mult* addresses	LinkLocalUnicast* addresses
Firefox						
Chrome/Edge				Name Collision		

Observations

We note that Firefox fails to resolve the current IPv4 Name Collision A record and that Chrome currently has a specific Name Collision error code for it (which is interestingly triggered for `mapped2` on Windows):



Otherwise, the only differences seen are slight differences in behaviour for the `mapped2` address between browsers/platforms.

IPv6 only testing results

General observation from IPv6 testing desktop testing

1. In general we see no differences in the results compared to dual stack testing. The notable differences are described below, with error code differences also noted in the dual stack table results above.

2. We do see that on macOS, that when testing the current IPv4 Controlled Interruption address (127.0.53.53) we see only AAAA record DNS lookups in the packet traces which result in that address failing to resolve for most applications. `mapped2` does however resolve. (Apart from Chrome which may well be using its own DNS stack).

3. We also saw different error codes on FreeBSD when trying to connect to 127.0.53.53 on Dual stack compared to IPv6 only.

4. We also saw minor differences in error codes on iOS when running IPv6 only - both the current IPv4 Controlled Interruption address and Mapped2 failed to resolve when running in this mode (compared to attempting to connect in dual stack mode).

5. We saw on Android that Chrome reported the MultNodeInterface address as ‘unreachable’ for IPv6 only but reported ‘unable to connect’ for dual stack.

Conclusions

We started the Stage 2 testing with the following candidate prefixes:

Test Name	Test Address
Mapped1	::ffff:0:1
Mapped2	::ffff:7f00:3535
MultNodeInterface	ff01::1
MultNodeLink	ff01::2
MultRouterInterface	ff02::1
LinkLocalUnicast1	fe80::1
LinkLocalUnicast2	fe80::7f00:3535

1. Mapped1: We rule out Mapped1 address since it generates network traffic in some scenarios
2. Mapped 2 is still considered a viable candidate.
3. Of the 3 multicast addresses (which behave identically in the testing) we prefer MultNodeInterface as per its specification it is defined with the narrowest scope of those addresses
4. Of the 2 link local unicast addresses (which behave identically in the testing) we prefer LinkLocalUnicast2 due to its recognisability.

The table below gives a high level overview of the various characteristics of the 3 resulting candidate prefixes, considering both technical and usability/deployability aspects. The details are discussed below.

Test Name	Test Address	Globally routable traffic seen	Link local traffic seen	Loopback traffic seen	Recognisable?	Easy to troubleshoot?
Mapped2	::ffff:7f00:3535	N	N	Y	Yes	Generates only IPv4 traffic and errors similar to IPv4 CI address
MultNodeInterface	ff01::1	N	N	N	Not really	Generates no traffic at all and mostly reasonable error messages
LinkLocalUnicast2	fe80::7f00:3535	N	Y	N	Yes	Generates some LL traffic and some unusual error messages

1. **Mapped2**:
    a. Pros
        i. This address generates no network traffic at all but does often generate traffic on the IPv4 loopback interface to 127.0.53.53.
        ii. This address is attractive because of its similarity in behaviour to the existing IPv4 Controlled Interruption and it would be similarly recognisable if used in an Internet search following observation in DNS resolution or traffic logs.
    b. Cons
        i. Since this is not a true IPv6 address it might be viewed as a workaround solution not suitable for a future ‘IPv6 only’ world.
        ii. Its behaviour e.g. error messages might change if future systems implement different, specific behaviour for mapped addresses (as OpenBSD does today). Or on devices that lack an IPv4 loopback address. However it seems extremely unlikely this would result in network traffic leaving the host.
2. **MultNodeInterface**:
    a. Pros
        i. From a purely technical standpoint this is the most suitable of the candidates since it generates no traffic (network or loopback) and also generates reasonable error messages from the applications tested.

    b. Cons
        i. Its major drawback is that it is a generic looking address not well known by those who are already very knowledgeable about IPv6. An Internet search for the address might not immediately lead back to the ICANN Controlled Interruption website.
3. **LinkLocalUnicast2**:
    a. Pros
        i. This generates no globally routable traffic, no loopback traffic and reasonable error messages
        ii. Those with basic knowledge of IPv6 should recognise a link local address immediately and (as with Mapped2) if used in an Internet search following observation in DNS resolution or traffic logs it is likely to lead to the ICANN Controlled Interruption website.
    b. Cons
        i. On Windows and for some applications on macOS this generates ICMPv6 Neighbour Discovery Protocol packets on the local network. This is a strong argument for ruling this out as a candidate prefix.
        ii. After noticing this in the testing and then considering the link local unicast address space further we are aware that today, it is common practice to use only the lower /64 of the link local address. If the above traffic were deemed acceptable then to reduce the possibility of soliciting a response from a device on the local network a similar address from the high end of the link local address space could be chosen. For example, we conducted a limited set of testing on one other link local unicast address that we called LinkLocalUnicast2: febf:ffff:ffff:ffff:ffff:ffff:7f00:3535. Unsurprisingly this behaved identically to LinkLocalUnicast2.
        iii. The above lack of link local traffic on most systems is the result of there being no default scope defined for link local addresses today on those systems. That could change in future.

Recommendations and further considerations

With multiple candidates with various characteristics identified in this stage of testing the recommendations are are follows:

1. If the ‘recognisability’ of the address is deemed sufficiently important to be a major factor in a successful deployment then the Mapped2 address (::ffff:7f00:3535) would be the best candidate.

2. However, if ICANN’s priority is to select a candidate with the minimal traffic generation profile then the MultNodeInterface address (ff01::1) would be the best solution.

3. Since the findings of this study involve multiple candidates we suggest a review and consultation on the results by a wider audience before selection. A summary of the

findings in a format suitable for public review are available in the draft Summary report.

4. If further consultation and further testing is desirable, ICANN could consider creating test zones for each candidate where the content would be analogous to that tested here in terms of AAAA records and would allow any reviewer of the document to perform basic tests against the candidate prefix e.g.
    a. mapped.citesting.icann.org
    b. multicast.citeting.icann.org

5. If a candidate prefix finds consensus then ICANN should consider approaching the major browser implementers to implement an Controlled Interruption specific error code for that candidate prefix, as Chrome has done for 127.0.53.53.

Appendix A - Testing Methodology

Customised recursive resolver and authoritative server

On a dedicated VM we ran:

- `unbound` configured as a standard, validating recursive resolver, with rules to treat all of the tested TLDs as `stub-zones` to be resolved by our customised authoritative and additionally mark those domains as `domain-insecure`.
- `knot` authoritative configured with zones for the tested names with standard Controlled Interruption records for MX/SRV/TXT records and :
    a. No `A` record for the owner name (apart from `ipv4ci`, which has only an `A` record as per the current Controlled Interruption zone contents)
    b. A `AAAA` record for the owner name with the candidate prefix
    c. A `AAAA` wildcard record for the owner name with the candidate prefix

    For example:

```
linklocalunicast1.          300   IN    SOA   resolver.sinodun.com. jad.sinodun.com. 2022121730 30000 300 604800 300
```

```
linklocalunicast1. IN AAAA fe80::1
linklocalunicast1. IN MX 10 your-dns-needs-immediate-attention.linklocalunicast1.
linklocalunicast1. IN SRV 10 10 0 your-dns-needs-immediate-attention.linklocalunicast1.
linklocalunicast1. IN TXT ("Your DNS configuration needs immediate attention see https://namecollisions.icann.org/")

* IN AAAA fe80::1
* IN MX 10 your-dns-needs-immediate-attention.linklocalunicast1.
* IN SRV 10 10 0 your-dns-needs-immediate-attention.linklocalunicast1.
* IN TXT ("Your DNS configuration needs immediate attention see https://namecollisions.icann.org/")
```

Scripted testing

Control file

The full set of names in the control script used for testing was the following:

```
# The first 3 items are baseline tests to ensure
# the test framework is behaving correctly
icann.org               2001:500:88:200::7
a.ipv4ci                127.0.53.53
a.ipv6localhost         ::1
# This gives a baseline failure case for NXDOMAIN
a.nonexistant           NXDOMAIN
# Candidate prefixes
a.mapped1               ::ffff:0:1
a.mapped2               ::ffff:7f00:3535
a.multnodeinterface     ff01::1
a.multnodelink          ff01::2
a.multrouterlink        ff02::1
a.linklocalunicast1     fe80::1
a.linklocalunicast2     fe80::7f00:3535
```

The test names we assigned to the candidate prefixes are prepended by an additional `'a'` label in order to result in a response matched from the wildcard in the test zone. This both mimics the likely real world usage of a multi label domain name, and also avoids peculiarities associated with resolving a request for an A/AAAA record for a single label on Linux.

Test script

A bash script was written that performed the following steps

- Read the set of test names and associatedIPv6 addresses from a control file.
- For each test *name*:

    - Loop over a list of *commands* specified for the given platform. For each:
        - Flush the system DNS cache and any application DNS cache
        - Perform `dig/nslookup` lookups to ensure system resolution is returning the expected A/AAAA record for each test TLD name
        - Start traffic capture using `tshark` on both the loopback interface and the network interface
        - Launch the test *command* giving as an argument the test name from the control file and logging the output.
        - Wait (typically 5 seconds)
        - Shut down the packet capture and kill the application for the test *command*.
        - Parse the packet captures to:
            - Confirm DNS resolution occurred and the tested A/AAAA record was returned as expected
            - Identify any traffic where the destination IP address was the candidate prefix on either interface
            - (In later testing) Identify any ICMPv6 Neighbor Solicitation traffic which contains the candidate prefix as the destination address.
- Generate a report of observed traffic and logging output for each test *name/command* combination

The only dependencies of the test script beyond basic bash commands were `dig` or `nslookup` and `tshark`. On Windows the Windows Subsytem for Linux (wsl) was used as a testing environment using Powershell but all commands issued were native Windows commands.

A simplified test script for just the ssh and curl command is provided below for illustration (collapsed by default).

Testing of the mobile devices required some modifications to the above:

iOS Device testing

1. Connect device to a Mac via USB cable
2. Open the 'Devices and Simulators' window in XCode and find the device UUID
3. Create a virtual interface using the `rvictl` command
   `rvictl -s <DEVICE_UUID>`
4. Use this interface in the above test script to capture just the network traffic from the device
5. Clear the system DNS cache by turning Airport mode on and off
6. Set each browser as the system default in turn.
7. Manually click on a link to the webpage of the tested name at a prompt from the test script

Android simulator

1. Using Build #AI-251.26094.121.2512.13840223
2. Virtual phones used are a generic small phone running Android 15 and a Google Pixel 9a running Android 16
3. Clear the cache by turning Airport mode on and off
4. As with iOS, the test script is run and a link to the tested name is clicked manually
5. Packet capture is as described https://developer.android.com/studio/run/emulator-networking

Android Physical device

1. Root the device (details are specific to particular device/OS)
2. Download tcpdump from `https://www.androidtcpdump.com/android-tcpdump/downloads64bit`
3. Config the permissions
   `sudo chmod +x /opt/android-sdk/platform-tools/tcpdump`
   and push it into a read/write section of the phone file-system e.g.
   `adb push tcpdump /tmp`
   (You will need to repeat this if you reboot the phone)
4. Capture packets using a command like the following
   ```
   adb exec-out "su -c /tmp/tcpdump -i wlan0 -U -w - 2>/dev/null"
   | wireshark -k  -i - -w "$PWD"/${ETH_FILE} &
   ```
5. Use a command similar to the following to start Chrome/Firefox
   ```
   adb shell am start -n
   com.android.chrome/org.chromium.chrome.browser.ChromeTabbedActi
   vity https://icann.org
   ```
6. Use a command similar to this to kill Chrome/Firefox
   `adb shell su -c pkill`

Example test script and output

Example test script

```bash
#!/usr/bin/env bash
# set -x

# For macOS/Linux, must have sudo password pre-cached to run this script
# For Windows, run from Powershell
if [ "${USER}" == "root" ] ; then
    echo "Please do not run this script as root (sudo)"
    exit 1
fi

echo -n "Starting at "
```

```
date

if [[ "$(uname -r)" =~ "WSL2" ]] ; then
    PLATFORM="Windows"
else
    PLATFORM=$(uname -s)
fi

FLUSH_CACHE2=""
COMMANDS="ssh curl"

if [ "${PLATFORM}" == "Linux" ] ; then
    FLUSH_CACHE1="sudo resolvectl flush-caches"
    SUB_PLATFORM="ubuntu"
    SSH="ssh -T -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null"
    CURL="curl"
    TSHARK="/usr/bin/tshark"
    PIDWAIT="pidwait"
    ETH_IF="enp1s0"
    LBK_IF="lo"
elif [ "${PLATFORM}" == "Windows" ] ; then
    FLUSH_CACHE1="/mnt/c/Windows/System32/ipconfig.exe /flushdns"
    NSLOOKUP="/mnt/c/Windows/System32/nslookup.exe"
    SSH="/mnt/c/Windows/System32/OpenSSH/ssh.exe"
    CURL="/mnt/c/Windows/System32/curl.exe"
    TSHARK="/mnt/c/Program Files/Wireshark/tshark.exe"
    PIDWAIT="pidwait"
    LBK_IF=$(/mnt/c/Program\ Files/Wireshark/tshark.exe -D | grep "(Adapter for loopback traffic capture)" | awk -F . ' { print $1 } ')
    ETH_IF=$(/mnt/c/Program\ Files/Wireshark/tshark.exe -D | grep "(Ethernet)" | awk -F . ' { print $1 } ')
elif [ "${PLATFORM}" == "Darwin" ] ; then
    # Reliably flushing the cache on macOS appears to require doing these a 2 separate commands.
    FLUSH_CACHE1="sudo dscacheutil -flushcache"
    FLUSH_CACHE2="sudo killall -HUP mDNSResponder"
    SSH="ssh"
    CURL="curl"
    FIREFOX="open -g -a firefox"
    TSHARK="tshark"
    PIDWAIT="wait"
    LBK_IF="lo0"
    ETH_IF="en0"
else
    echo "Platform not supported"
    exit 1
fi

# Read target hostnames and addresses
while read P ; do
    A=( $P )
    if [[ "${A[0]}" == '#' ]]; then
        continue
    fi
    NAME=${A[0]}
```

```
    NAMES+=(${NAME})
    AAAA=${A[1]}
    AAAAS+=(${AAAA})
done<names.txt

DATE=$(date '+%Y-%m-%d_%H-%M-%S')
FOLDER=Results/"$PLATFORM"/"$DATE"
if [ "$3" == "clean" ] ; then
  rm -rf Results/*
fi

mkdir -p "$FOLDER"
cd "$FOLDER"
dig @8.8.8.8 icann.org A +time=1 +short
IPv4=$?
[[ $IPv4 == "0" ]] && IPv4_MODE="ON"   || IPv4_MODE="OFF"

echo "Running on ${PLATFORM} $(uname -r)"  ${SUB_PLATFORM}           "IPv4 is " $IPv4_MODE > packet_counts.txt
echo "Name                      Address                Command     ** RESULT **          ETH_DNS  (LBK_DNS)     ETHERNET TRAFFIC    LOOPBACK TRAFFIC" >> packet_counts.txt
echo "                                                             DNS  ETH/LBK  NS" >> packet_counts.txt

for i in "${!NAMES[@]}"; do
    NAME=${NAMES[i]}
    AAAA=$(echo ${AAAAS[i]} | tr -d '\r')
    echo >> packet_counts.txt
    echo
    echo "-------------------------"

    echo -n "Direct lookup result for $NAME: "
    TYPE="AAAA"
    WIN_TYPE="-type=aaaa"
    if [ "$NAME" == "a.ipv4ci" ]; then TYPE="A"  && WIN_TYPE=""; fi
    if [ "${PLATFORM}" == "Windows" ] ; then
        DIG_IP=$($NSLOOKUP $WIN_TYPE "$NAME" 2> /dev/null | awk '/^Address: /  { print $2}' | tail -n -1 | sed -e 's/\r//')
    else
        DIG_IP=$(dig $NAME +short $TYPE)
    fi
    echo -n $DIG_IP
    MAPPED_IP=$DIG_IP
    if [[ "$NAME" == "a.nonexistant" ]] && [[ -z "$DIG_IP" || "$DIG_IP" == "192.168.12.120" || "$DIG_IP" == "2a02:8010:6a36:102:20c:29ff:fef9:6158" ]] ; then
        echo -n "  NO ADDRESS RETURNED "
        DIG_IP=$NAME
    elif [[ "$NAME" == "a.mapped1" ]] && [[ "${PLATFORM}" == "Windows" ]] ; then
        MAPPED_IP="0.0.0.1"
    elif [[ "$DIG_IP" != "$AAAA" ]]; then
        # When dig (and tshark) display these addresses they are converted to a mapped representation that contains an IPv4 address. Except for mapped1 on Windows.
        if [[ "$NAME" == "a.mapped1" ]] && [[ "$DIG_IP" == "::ffff:0.0.0.1" ]] ; then
            echo -n "  (mapped result of  ::ffff:0:1) "
            MAPPED_IP="0.0.0.1"
        elif [[ "$NAME" == "a.mapped2" ]] && [[ "$DIG_IP" == "::ffff:127.0.53.53" ]] ; then
            echo -n "  (mapped result of  ::ffff:7f00:3535) "
            MAPPED_IP="127.0.53.53"
```

```
        else
            echo " ERROR: Incorrect DNS resolution - got |$DIG_IP| should be |$AAAA|"
            break
        fi
    fi
    echo

    for COMMAND in $COMMANDS ; do
        mkdir -p $NAME/$COMMAND
        # Flush the system resolver
        ${FLUSH_CACHE1}
        ${FLUSH_CACHE2}
        sleep 1

        # TSHARK can take a duration to run for and so you don't need to clean up or kill tshark
        TSHARK_DURATION=5
        ETH_FILE="$NAME/$COMMAND/$COMMAND-ethernet.pcap"
        LBK_FILE="$NAME/$COMMAND/$COMMAND-loopback.pcap"
        LOG_FILE="$NAME/$COMMAND/$COMMAND.log"
        date
        if [ "$SUB_PLATFORM" == "android" ]; then
            adb exec-out "su -c /tmp/tcpdump -i wlan0 -U -w - 2>/dev/null" | wireshark -k  -i - -w "$PWD"/${ETH_FILE} &
            adb exec-out "su -c /tmp/tcpdump -i lo    -U -w - 2>/dev/null" | wireshark -k  -i - -w "$PWD"/${LBK_FILE} &
        else
            "${TSHARK}" -n -i ${ETH_IF} -a duration:${TSHARK_DURATION} -Q -w ${ETH_FILE} -P > ${ETH_FILE}.txt  2>&1 &
            tshark1PID=$!
            "${TSHARK}" -n -i ${LBK_IF} -a duration:${TSHARK_DURATION} -Q -w ${LBK_FILE} -P > ${LBK_FILE}.txt  2>&1 &
            tshark2PID=$!
        fi
        sleep 2

        echo "Testing command: $COMMAND"
        case $COMMAND in
          ssh)        FULL_COMMAND="$SSH $NAME -v"
                      ;;
          curl)       FULL_COMMAND="${CURL} http://$NAME -v"
                      ;;

        *)            echo "Command $COMMAND not recognized"
                      break
                      ;;
        esac

        set -x
        $FULL_COMMAND > $LOG_FILE 2>&1 &
        COMMAND_PID=$!
        set +x

        # pidwait doesn't exist on Darwin
        if [ "${PLATFORM}" == "Darwin" ] ; then
          $PIDWAIT $tshark1PID
          $PIDWAIT $tshark2PID
```

```
        else
          $PIDWAIT tshark
        fi

        echo "Killing command: $COMMAND with pid $COMMAND_PID"
        # Clean up any processes still running
        if [ ${PLATFORM} == "Windows" ] ; then
            taskkill.exe /F /IM $COMMAND.exe
        else
            sudo kill -9 $COMMAND_PID
        fi
        sleep 1

        # Process the results
        # Double check that the app actually did a DNS lookup and got the test address
        ETH_DNS_TOTAL=$("${TSHARK}" -r $ETH_FILE -Y "dns" | wc -l)
        LBK_DNS_TOTAL=$("${TSHARK}" -r $LBK_FILE -Y "dns" | wc -l)
        ETH_DNS_SEEN=$("${TSHARK}" -r $ETH_FILE -Y "dns.resp.type==A or dns.resp.type==AAAA" | grep ${DIG_IP} | wc -l)
        LBK_DNS_SEEN=$("${TSHARK}" -r $LBK_FILE -Y "dns.resp.type==A or dns.resp.type==AAAA" | grep ${DIG_IP} | wc -l)

        # Check the traffic seen
        if [[ "$NAME" == "a.mapped1" || "$NAME" == "a.mapped2" ]] ; then
          IP_DEST="ip.dst==${MAPPED_IP} || ipv6.dst==::ffff:7f00:3535 || ipv6.dst==${DIG_IP}"
        elif [[ "$NAME" == "a.ipv4ci" ]]; then
          IP_DEST="ip.dst==${DIG_IP}"
        elif [[ "$NAME" == "a.nonexistant" ]]; then
          # create filter that gives no traffic since we have no address
          IP_DEST="not dns && dns"
        else
          IP_DEST="ipv6.dst==${DIG_IP}"
        fi
        ETH_TOTAL=$("${TSHARK}" -r $ETH_FILE -Y "not dns" | wc -l)
        LBK_TOTAL=$("${TSHARK}" -r $LBK_FILE -Y "not dns" | wc -l)
        ETH_SEEN=$("${TSHARK}" -r $ETH_FILE -Y "${IP_DEST}" | wc -l)
        LBK_SEEN=$("${TSHARK}" -r $LBK_FILE -Y "${IP_DEST}" | wc -l)

        ETH_NS_SEEN="-"
        if [[ "$NAME" == "a.linklocalunicast2" ]] ; then
          ETH_NS_TOTAL=$("${TSHARK}" -r $ETH_FILE -Y "icmpv6.type == 135 and ipv6.dst == ff02::1:ff00:3535" | wc -l)
          [[ $ETH_NS_TOTAL -eq 0 ]] && ETH_NS_SEEN="-"   || ETH_NS_SEEN="Y"
        fi

        [[ $ETH_DNS_SEEN -eq 0 ]] && DNS_OK="N"   || DNS_OK="Y"
        [[ $ETH_SEEN -eq 0 ]]     && ETH_TRAF="N" || ETH_TRAF="Y"
        [[ $LBK_SEEN -eq 0 ]]     && LBK_TRAF="N" || LBK_TRAF="Y"

        printf "%-24s  %-22s  %-8s    %s     %s/%s     %s    | %4s/%-4s(%4s/%-4s)     %6s/%-6s      %6s/%-6s" $NAME $DIG_IP $COMMAND  $DNS_OK  $ETH_TRAF $LBK_TRAF $ETH_NS_SEEN
$ETH_DNS_SEEN $ETH_DNS_TOTAL $LBK_DNS_SEEN $LBK_DNS_TOTAL $ETH_SEEN $ETH_TOTAL  $LBK_SEEN $LBK_TOTAL   >> packet_counts.txt
        echo >> packet_counts.txt

    done
    tail -n +1  $NAME/*/*.log > $NAME/"$NAME"_log_summary.log
```

```
done

echo
echo "******** RESULTS **********"
cat packet_counts.txt

# Generate file will all the logs in it
tail -n +1  */*.log > log_summary.log

echo -n "Finished at "
date
```

Example output

The text below shows an example summary file produced by the testing. A summary log file of the output of each command for each name was also produced, and full packet captures (both PCAP and txt format) were also generated.

```
Running on Windows 6.6.87.2-microsoft-standard-WSL2   IPv4 is  ON
Name                    Address                 Command     ** RESULT **          ETH_DNS  (LBK_DNS)     ETHERNET TRAFFIC    LOOPBACK TRAFFIC
                                                            DNS  ETH/LBK  NS

icann.org               2001:500:88:200::7      ssh         Y    Y/N     -    |    1/4   (   0/0   )         3/31              0/1
icann.org               2001:500:88:200::7      curl        Y    Y/N     -    |    1/4   (   0/0   )         5/34              0/10

a.ipv4ci                127.0.53.53             ssh         Y    N/Y     -    |    1/4   (   0/0   )         0/24              8/15
a.ipv4ci                127.0.53.53             curl        Y    N/Y     -    |    1/4   (   0/0   )         0/23              5/20

a.ipv6localhost         ::1                     ssh         Y    N/Y     -    |    1/4   (   0/0   )         0/25             15/15
a.ipv6localhost         ::1                     curl        Y    N/Y     -    |    1/12  (   0/0   )         0/25             10/20

a.nonexistant           a.nonexistant           ssh         N    N/N     -    |    0/4   (   0/0   )         0/16              0/0
a.nonexistant           a.nonexistant           curl        N    N/N     -    |    0/4   (   0/0   )         0/21              0/10

a.mapped2               ::ffff:127.0.53.53      ssh         Y    N/Y     -    |    1/12  (   0/0   )         0/30              8/15
a.mapped2               ::ffff:127.0.53.53      curl        Y    N/Y     -    |    1/4   (   0/0   )         0/19              5/20

a.multnodeinterface     ff01::1                 ssh         Y    N/N     -    |    1/12  (   0/0   )         0/22              0/0
a.multnodeinterface     ff01::1                 curl        Y    N/N     -    |    1/4   (   0/0   )         0/27              0/10

a.linklocalunicast2     fe80::7f00:3535         ssh         Y    N/Y     Y    |    1/4   (   0/0   )         0/25              1/1
a.linklocalunicast2     fe80::7f00:3535         curl        Y    N/Y     Y    |    1/4   (   0/0   )         0/24              2/12
```

Desktops VM configuration

All VMs are initially configured as dual stack, snapshotted and later converted to IPv6 only.

1. The VM's are hosted on an Intel(R) Xeon(R) w5-3425 @ 4.60 GHz server running Arch Linux x86_64 6.15.8-arch1-1 systemd 257.7-2-arch libvirt 11.5.0 QEMU 10.0.3.
2. Virtual machines are networked using a network bridge and obtain IPv4 addresses from DHCP and IPv6 using SLAAC.

In the Stage 1 testing we compared results from wired and wifi networks used on a physical macOS Sequoia macMini and also a Ubuntu 25.04 desktop VM - we found no difference in the results. The testing in Stage 2 was completed using VMs with wired network interfaces.

The sections below give brief details of any additional configuration beyond a basic install that was performed if required.

Ubuntu/RedHat

1. Installed Wireshark and enabled for non-superusers:
   ```
   $ sudo apt-get install tshark
   $ sudo dpkg-reconfigure wireshark-common
   $ sudo usermod -a -G wireshark $USER
   $ sudo reboot
   ```
2. (Desktop only) Disable `apparmour` for wireshark and then reboot
   `sudo ln -s /etc/apparmor.d/tshark /etc/apparmor.d/disable/`
3. All packages installed using `apt`
4. To disable IPv4:
   `sudo ip address del <IP_address>/22 dev enp1s0`
   To re-enable:
   `sudo netplan apply`

FreeBSD

1. We note that when installing FreeBSD 14.3 you can tell the installer to only configure IPv6 but you still get an IPv4 loopback interface.
2. During install deselect all optional components and do not select any hardening/security options.
3. Installation of basic packages as required: `sudo, git, bash, wireshark, curl, bind-tools-9.20.13`
4. To disable IPv4:
   `sudo ip address del <IP_address>/22 dev enp1s0`

OpenBSD

1. OpenBSD inet6 manual page:
   "For security reasons, OpenBSD does not route IPv4 traffic to an AF_INET6 socket, and does not support IPv4 mapped addresses, where IPv4 traffic is seen as if it comes from an IPv6 address like "::ffff:10.1.1.1". Where both IPv4 and IPv6 traffic need to be accepted, bind and listen on two sockets."

2. Installation of basic packages as required: `sudo, git, bash, wireshark, curl, bind-tools-9.20.13`
3. On OpenBSD we tested the application with and without '-6` flags for `ssh, nc` and `openssl s_client` as they appeared to default to IPv4 addresses to ensure our baseline test resolving icann.org over IPv6 worked. This is in contrast to the other platforms that prefer IPv6 if available.
4. To disable IPv4 edit /etc/hostname.<interface> and change inet autoconf to inet -autoconf then do
   `sh /etc/netstart <interface>`

macOS

1. Install Wireshark and all browsers by downloading the official macOS installer directly from the relevant website
2. Proxies installed using `homebrew`
3. Enable/disable IPv4 in `System settings->Network details->TCP/IP->Configure IPv4`

Windows 10/11

The following process was used to configure Windows 11, with Windows 10 specifics listed at the end.

1. Starts up saying there are no bootable disks.
2. Press any key to enter the boot manager.
3. Press down arrow twice to get to Boot Manager
4. Select UEFI QEMU DVD-ROM
5. Press any key to boot from CD or DVD
6. The system will boot
7. Select Language
8. Select keyboard
9. Select Install Windows 11 and click I agree everything will get deleted
10. Select I don't have a product key
11. Select Windows 11 Pro
12. Accept notices and licence terms
13. Just click next on Select location to install Windows 11
14. Click Install
15. Select Country (UK)
16. Select keyboard layout
17. Skip adding second keyboard layout
18. Give the device a name (IPv6-windows-11)
19. Select Set up for personal use
20. Sign in to unlock your Microsoft experience
21. Create a pin
22. Select No to allowing Microsoft using you location
23. Select No to find my device

24. Select Send Required diagnostic data
25. Select No to Improve inking and typing
26. Select No to Get tailored experiences
27. Select Set up as a new PC
28. Skip Let's customise your experience
29. Skip Use your phone from your PC
30. Select Only save files to this PC
31. Select Not now to importing from another browser
32. Decline trial of Microsoft 365
33. Decline Microsoft 365 Basic
34. Click Next on Did you know you can use Microsoft 365 for free
35. Skip Join PC Game Pass
36. Base install is done - now check for updates
37. Open ssh (to facilitate testing):
    a. Go to Settings > System > Optional features and click on View features.
    b. Locate "OpenSSH server" feature, select it, click Next, and then click Add.
    c. Set local network to private
38. Set local network to private
    a. Open Settings > Network and internet > Ethernet
    b. Select Private network
39. start openssh service
    a. Open services
    b. start OpenSSH SSH Server
    c. Set service to Startup type = Automatic
    d. In Windows, sshd reads configuration data from %programdata%\ssh\sshd_config but the defaults seem fine.
    e. Create "C:\ProgramData\ssh\administrators_authorized_keys" with your public key(s) in it.
40. We also enabled the Clipboard (to facilitate testing)
    a. Download https://www.spice-space.org/download/windows/spice-guest-tools/spice-guest-tools-latest.exe
    b. Install accepting all options
    c. Reboot
41. Make powershell the default shell for ssh (to facilitate testing)
    a. Open powershell in administrator mode and run
    ```
    New-ItemProperty -Path "HKLM:\SOFTWARE\OpenSSH" -Name DefaultShell -Value "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -PropertyType String -Force
    ```
42. Install wireshark https://www.wireshark.org/download.html
43. Install powershell 7 https://learn.microsoft.com/en-us/powershell/scripting/whats-new/migrating-from-windows-powershell-51-to-powershell-7?view=powershell-7.5

44. Install wsl https://learn.microsoft.com/en-us/windows/wsl/install

Windows 10 specifics

Follow the windows 11 instructions but

- Skip all questions asking you to setup a windows account. It is not necessary on Windows 10.
- Make sure to add a TPM v2 device to the VM and ensure it boots using UEFi
- To Install OpenSSH server use
    - Open settings → Apps →Optional Features → Add a feature → OpenSSH Server and click install and then restart

Application details

The following section contains a variety of details related to specific applications or specific platforms that we note here to aid anyone wanting to reproduce our results.

1. Command line tools
    a. We tested both `ssh` and `ssh -6` because we noticed different error responses between the two. In general `ssh` would return a specific error code but `ssh -6` would just report it was attempting to connect and not return. However this behaviour was consistent across the dual stack and IPv6 only testing.
    b. `openssl s_client` is actually LibreSSL on MacOS and so it requires that you specify a port to connect to.
2. Browsers:
    a. Each the browser was started multiple times after install to ensure all dialogs requiring user interaction were disabled and the simplest set up and homepage was configured.
    b. For all browsers except Safari clearing the DNS cache must be done manually via a button in the developer tools section.
    c. This page describes how to hide first run webpage in Edge https://learn.microsoft.com/en-us/deployedge/configure-microsoft-edge
    d. Firefox Safe Mode was disabled by setting the key `toolkit.startup.max_resumed_crashes = -1` in `about:config`
3. Proxies
    a. We note that with the testing for all our proxies we use a proxy listening only on port 8080. If the proxy were configured to listen on localhost on a standard port then those test cases for addresses on localhost would connect to that listening proxy rather than fail. And the error messages we report are those found in the log of the application, not what is seen when attempting to connect via e.g. a browser or curl.

b. In testing our proxies we did not run them as a service but rather from the command line with configuration options to log to stdout/stderr. This allowed us to capture specific errors for each test name.

c. `nginx` was given a custom nginx.conf file with a `proxy_connect_timeout 3` clause and a `proxy_pass` clause for each test name. We noted that by default it will try to resolve any host name specified in a `proxy_pass` directive on start up and so both `` `nonexistant` `` and `` `mapped2` `` resulted in fatal errors that stopped `nginx` starting. We added a `resolver` clause to the configuration and defined the `proxy_pass` target as a variable which causes `nginx` to instead resolve the test name at run time.
Interesting in this scenario `` `mapped2` `` resolved and caused a connection attempt.

```
location /a.mapped2 {
    set $target_host a.mapped2 ;
    proxy_pass http://$target_host:80;
}
```

d. Similarly HAProxy was also configured to delay resolution until runtime

```
resolvers mynameservers
  parse-resolv-conf

frontend http-in
            acl linklocalunicast2 path -i /a.linklocalunicast2
               use_backend proxy-linklocalunicast2 if linklocalunicast2
             ...
             backend proxy-linklocalunicast2
               mode http
               server server1 a.linklocalunicast2:80 resolvers mynameservers
             init-addr libc,none
```

e. Apache was configured to use the ProxyPass clause for each test name:

```
   ProxyPass        /icann.org  http://icann.org
   ProxyPassReverse /icann.org  http://icann.org
```

4. Mail servers
    a. As with the proxies the mail servers were tested from the command line (in verify mode)
    b. `exim` was configured as an internet mail server (not a local one). We note that the default install contains the following in the config files (`/etc/exim4/exim4.conf.template` and `/etc/exim4/conf.d/router/200_exim4-config_primary`) which causes exim to treat those hosts as ‘non-existant’ and does not attempt delivery:

```
ROUTER_DNSLOOKUP_IGNORE_TARGET_HOSTS = <; 0.0.0.0 ; 127.0.0.0/8 ; 192.168.0.0/16 ;
172.16.0.0/12 ; 10.0.0.0/8 ; 169.254.0.0/16 ; 255.255.255.255 ; ::/128 ; ::1/128 ;
fc00::/7 ; fe80::/10 ; 100::/64
```

c. We also tested `exim` with the address prefixes that coincided with any test candidate prefixes (update the config and then do `sudo update-exim4.conf`)
d. Postfix can be configured to log to stdout using: `postconf maillog_file=/dev/stdout`

Appendix B - Detailed Error codes

Desktop dual stack testing

A selection of short codes (which are further defined in a table below the results) is provided for illustrative purposes.

Square brackets indicate only see with -v or -debug flags

App	Short code	Error text
	Connecting…	App indicates attempting connection but hangs
ssh	Could not resolve	ssh: Could not resolve hostname *: nodename nor servname provided, or not known OR ssh: Could not resolve hostname a.mapped1: The requested name is valid, but no data of the requested type was found.
ssh	No route to host	ssh: connect to host * port 22: No route to host
ssh	Address not supported	ssh: connect to host * port 22: Address family not supported by protocol family
ssh	UNREACHABLE	ssh: connect to host a.mapped1 port 22: Network is unreachable
ssh	Unknown error	ssh: connect to host * port 22: Unknown error
ssh	INVALID ARGUEMENT	ssh: connect to host a.linklocalunicast1 port 22: Invalid argument
curl	Could not resolve	curl: (6) Could not resolve host: *
curl	Failed to connect	curl: (7) Failed to connect to * port 80 after * ms: Couldn't connect to server
nc	Could not resolve	nc: getaddrinfo: nodename nor servname provided, or not known
nc	Address not supported	[nc: connectx to * port 443 (tcp) failed: Address family not supported by protocol family]
nc	REFUSED	[nc: connect to * port 443 (tcp) failed: Connection refused]
nc	No route to host	[nc: connectx to * port 443 (tcp) failed: No route to host]
openssl	REFUSED	system library:BIO_connect:Connection refused:../crypto/bio/bio_sock2.c:114:calling connect() :error:10000067:BIO routines:BIO_connect:connect error:../crypto/bio/bio_sock2.c:116

openssl	Could not resolve	BIO routines:BIO_lookup_ex:system lib:../crypto/bio/bio_addr.c:738:Name or service not known connect:errno=2
Safari	Can’t find server	Safari can’t open the page * because Safari can’t find the server *
Safari	Domain error	Safari can’t open the page *. The error is “unknown error” (NSURLErrorDomain:-1)
Safari	Can’t connect	Safari can’t open the page * because Safari can’t connect to the server *
Firefox	Can’t connect	Hmm. We’re having trouble finding that site. We can’t connect to the server at *
Firefox	Unable to connect	Unable to connect Firefox can’t establish a connection to the server at *
Chrome/Edge	NXDOMAIN	This site can’t be reached/Hmm… can’t reach this page.. Check if there is a typo in *. DNS_PROBE_FINISHED_NXDOMAIN
Chrome/Edge	UNREACHABLE	This site can’t be reached/Hmm… can’t reach this page.. *. ERR_ADDRESS_UNREACHABLE
Chrome/Edge	REFUSED	This site can’t be reached/Hmm… can’t reach this page... * reduced to connect ERR_CONNECTION_REFUSED
Chrome/Edge	INVALID ARGUEMENT	This site can’t be reached/Hmm… can’t reach this page... The webpage at * might be temporarily down or it may have moved permanently to a new address. ERR_INVALID_ARGUEMENT
Chrome/Edge	TIMEOUT	This site can’t be reached/Hmm… can’t reach this page… * took too long to respond ERR_CONNECTION_TIMED_OUT
Chrome/Edge	ADDRESS INVALID	This site can’t be reached/Hmm… can’t reach this page... The webpage at * might be temporarily down or it may have moved permanently to a new address. ERR_ADDRESS_INVALID