The Internet Protocol Journal

June 2017 Volume 20, Number 2

A Quarterly Technical Publication for Internet and Intranet Professionals

In This Issue

You can download IPJ back issues and find subscription information at:
www.protocoljournal.org

ISSN 1944-1134

From The Editor

Every day we seem to read another news story about some form of *cyber attack*, be that Denial of Service incidents, ransom ware, malware, website intrusions, compromised databases, so-called *phishing*, leaked e-mails, election hacking, and much more. The underlying opportunities for such attacks are varied, ranging from human factors like easy-to-guess passwords to poorly designed and insecure technologies, as we have discussed many times in this journal. As you might expect, making the Internet more secure and robust involves numerous efforts at every layer of the protocol stack.

Encryption is a time-tested method for securing end-to-end communication as well as for storing information in a manner that prevents unauthorized access. Encryption is also used in the generation of trusted *certificates* for secure web communication. In our first article, Daniel McCarney presents an overview of the *Automatic Certificate Management Environment* (ACME).

The *Domain Name System* (DNS) is one of the core components of the Internet. We have covered many aspects of the DNS over the years, but not looked closely at the *root server system* until now. Geoff Huston describes the history and evolution of the DNS and its root servers.

As announced in the previous edition of IPJ, the *Latin America and Caribbean Network Information Centre* (LACNIC) has agreed to translate selective articles from IPJ and provide summaries in Spanish. This service is now available at: **http://lacnic.net/ipjournal**

If you have a print subscription to this journal, you will find an expiration date printed on the back cover. For the last couple of years, we have "auto-renewed" your subscription, but starting with this issue, we ask you to log in to our subscription system and perform this simple task yourself. You should receive an e-mail with instructions for how to access this system. When logged in, you can update your mail and e-mail address and change your delivery options. For any questions, e-mail us at **ipj@protocoljournal.org**

—Ole J. Jacobsen, Editor and Publisher
ole@protocoljournal.org

A Tour of the Automatic Certificate Management Environment (ACME)

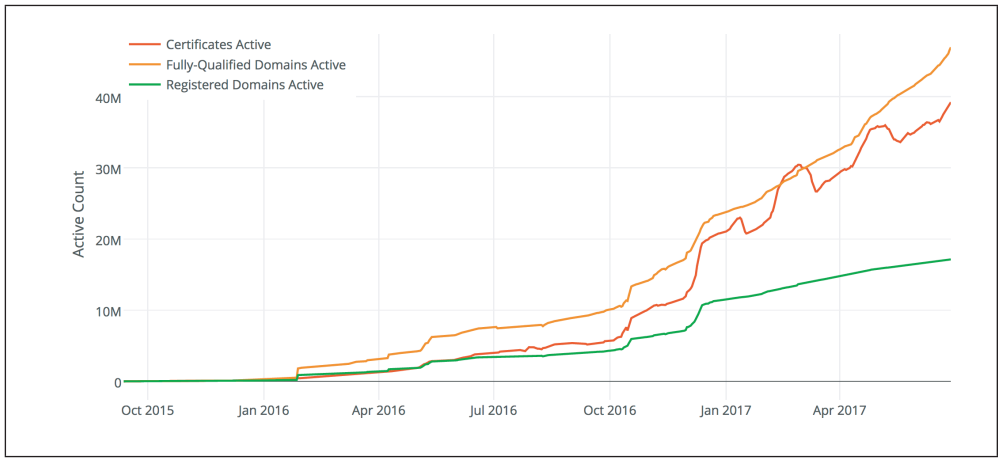
by Daniel McCarney, Internet Security Research Group

The introduction of *Let's Encrypt* has helped bolster *HyperText Transport Protocol Secure* (HTTPS) adoption by providing an easy-to-use and fully automated process for getting a trusted certificate for a domain name, all free of cost. Let's Encrypt is a service provided by the *Internet Security Research Group* (ISRG), a nonprofit organization with a mission to reduce financial, technological, and education barriers to secure communication over the Internet. To date, Let's Encrypt has issued certificates for more than 30 million websites.

Prior to Let's Encrypt, acquiring a certificate for a website was a difficult and error-prone process requiring frequent Google searches for obscure command-line incantations. Worse yet, this process typically had to be repeated manually with large periods of time elapsed between attempts—a recipe for disaster. One solution to both the usability and reliability issues created by placing this manual burden on a human is to augment the process with automation. A computer program will dutifully repeat complicated series of instructions at fixed intervals without missing a beat.

Frequently the methods of domain validation used by a certificate authority were difficult to automate at all (for example, e-mail-based validation) or required locking yourself in to a specific provider's *Application Program Interface* (API). Unlocking interoperable automation is critical to alleviating the burden on system administrators, promoting greater ease of use and helping create a fully encrypted web. Acknowledging this need for automation has been a core focus of Let's Encrypt from the very earliest days. The progress that Let's Encrypt has made in this area helps validate the premise that automation helps scale usage and has been reflected in the overall increase in HTTPS adoption observed by Mozilla Firefox telemetry, see Figure 1[0].

Figure 1: Let's Encrypt Statistics



While much has been said about Let's Encrypt and the choice to provide the service free of cost, less attention has been directed at the work underpinning the automation aspect. Underneath the service provided by Let's Encrypt is hidden a larger effort to bring a secure, automated *Internet Engineering Task Force* (IETF) standardized protocol for certificate issuance. This protocol, called the *Automatic Certificate Management Environment* (ACME) protocol, is a multi-organizational standards effort currently in IETF Working Group *Last Call*. This article briefly describes the core aspects of ACME and some of the ecosystem that surrounds it.

ACME

In the context of IETF *Request for Comments* (RFC) documents, ACME is unfinished. To date the ACME Working Group has published six drafts as the standard has received external feedback, and lessons from Let's Encrypt have been folded back into the overall design. It is currently in a phase of development where no new major changes are expected and the primary work remaining is editorial and interoperability-related.

This article focuses on ACME as defined in `draft-07` [1], the most current draft at the time of writing. As outlined in the next section, this focus means that our discussion of ACME will not perfectly match the Let's Encrypt service, which most closely resembles Version 3 (`draft-03`) of the draft specification for the moment while updates to the newer drafts are implemented.

The ACME Ecosystem

While ACME is a new protocol that is not yet standardized as an RFC, it already exists within an ecosystem of client and server implementations.

On the server side, Let's Encrypt has contributed an open source ACME implementation called *Boulder*[2]. It's written in *Go* and as the production backend for the service provided by Let's Encrypt it is the most "battle-tested" ACME server codebase to date. At peak so far Let's Encrypt and the Boulder codebase have issued more than 1.2 million certificates in a single day. Internally Boulder is divided into subcomponents, each responsible for a portion of the responsibilities of a *Certificate Authority* (CA). Notable components include the CA, the *Validation Authority,* the *Registration Authority,* and the *Web Front End*. Components "talk" to each other using a universal *Remote Procedure Call* (RPC) framework called *gRPC*.

Boulder has evolved alongside the draft ACME protocol and so necessarily has some divergences where specification moved faster than implementation or vice versa. The Boulder developers document these divergences in the Boulder repository[3] and adjust the codebase as the bounds of backwards compatibility with existing ACME clients allow. Since many ACME clients specifically target compatibility with Let's Encrypt and the Boulder implementation of ACME, most existing ACME clients will share these protocol divergences.

Our work to implement `draft-07`[1] fully is not yet complete at the time of writing, but is expected to be available by the end of 2017 as a separate API endpoint to ease migration from older clients using the existing (largely `draft-03`-based) API.

One of the benefits of an open standard is the ease with which clients can be written to interact with Let's Encrypt and issue certificates. Better yet, if other CAs adopt ACME, these clients will be able to interact with those CAs with minimal modification. A plethora of clients have been written since the launch of Let's Encrypt targeting a variety of platforms, programming languages, and use cases. Whether you're looking to issue a single certificate from an embedded device or thousands of certificates on-demand for a large platform integration, an ACME client is available for your needs.

Most well-known of these clients is *Certbot*[4]. Formerly known as the *letsencrypt* client, Certbot development has since been taken over by the *Electronic Frontier Foundation* (EFF) under its new name. As the first ACME client built for use with Let's Encrypt and the Boulder ACME server, it is still thought of as a reference client for Let's Encrypt. Certbot is an end-to-end solution capable of performing as much of the complicated administrative work as possible that is required to request, issue, and install a certificate for an HTTPS webserver.

Domain Validation

Let's Encrypt and the ACME protocol are both focused on *Domain Validated* (DV) certificates. *Organization Validated* (OV) and *Extended Validation* (EV) certificates are outside of the current scope of both the protocol and this article. OV and EV certificates require verification procedures that would be difficult if not impossible to automate programmatically. Unlike OV and EV certificates, a DV certificate does not attest identity but that the possessor of the private key corresponding to the public key in the certificate has demonstrated control of the domain names the certificate includes. The attestation from the CA that validated the domain control means you can be sure that your authenticated and encrypted *Transport Layer Security* (TLS) connection to the remote server is to a party in control of the domain name and not a *Man in the Middle* (MITM). It is important to know that this attestation does not vet anything about the trustworthiness of the domain owner.

Historically the technical method of domain validation that a CA employed was largely left to its own discretion and fairly ad-hoc. One method was to e-mail a token or activation link to an e-mail address believed to be authoritative for a given domain to prove ownership for an issuance request. The question of which addresses should be authoritative is the crux of this validation method, and mistakes in this decision process have led to certificates being issued to unauthorized parties in the past[5].

Other popular methods involved generating a token to be placed in a well-known location in the HTTP webroot of the domain, or in a *Domain Name System* (DNS) record for the domain. Recent changes in the baseline requirements[6] that CAs must meet have standardized the acceptable methods for domain validation and added some guardrails against mis-issuance.

Much of ACME directly addresses the domain validation process. This fact might be surprising if you were expecting to find a great deal of complicated cryptography related to the issuance of certificates themselves. Automating the act of issuing a certificate for a set of names is not the true challenge to scaling the web *Public Key Infrastructure* (PKI). The task of turning certificate signing requests into certificates with a software pipeline is well-understood. The larger challenge that must be addressed is how to scale the determination of whether the party requesting the certificate is authorized to act on behalf of all of the names the certificate includes. This contribution is one of the crucial ones of the ACME protocol: the introduction of clearly specified and peer-reviewed domain validation methods.

ACME Requests

All ACME requests are made over HTTPS. Protocol messages are primarily POSTed to HTTPS endpoints as *JavaScript Object Notation* (JSON) data. The JSON request data is authenticated and provided integrity through the application of *JSON Web Signatures* (JWS), as described in RFC 7515[10]. GET requests do not have a JWS body and are not authenticated by the ACME account key; therefore only public resources are available via GET.

To provide anti-replay protection, all ACME server responses provide a *nonce* header. The value of this header must be provided in the next request to the server. A dedicated *new-nonce* endpoint also exists to request a fresh nonce without performing a throw-away request only to look at the nonce reply header.

Since JWS will not cover the *Uniform Resource Identifier* (URI) of an HTTPS request, the URI is also contained in all request bodies and must be verified by the server to ensure that an entity terminating the ACME HTTPS request (for example, a *Content Distribution Network* (CDN) or *Load Balancer*) did not modify the request URI from the one intended to be used by the client contained in the authenticated request body. The ACME draft threat model section covers these considerations with more detail.

Components of ACME

ACME was designed to be a *Representational State Transfer* (REST) ful protocol, so one way to approach understanding it is by examining the resources the protocol specifies. At its core ACME is made up of *Accounts, Orders, Authorizations,* and *Challenges*.

At a high level, issuing a certificate is a matter of creating an ACME account, submitting an order for a certificate containing a set of DNS identifiers, satisfying authorizations for each of the identifiers by solving challenges, and finally, polling the ACME server until a signed certificate satisfying the order is produced.

A special directory resource serves as the entry point for the account creation and certificate issuance flows of the ACME protocol. ACME clients identify servers by their directory URI and make an initial request to this resource in order to learn the URIs used for other resources and to get a first nonce value. The directory also contains metadata related to the ACME server (for example, terms of service requirements, *Certification Authority Authorization* (CAA) identifiers, etc.).

An example of Let's Encrypt's current `/directory` endpoint, as generated by Boulder, follows:

```
curl https://acme-v01.api.letsencrypt.org/directory
{
   "key-change": "https://acme-v01.api.letsencrypt.org/acme/key-change",
   "new-authz": "https://acme-v01.api.letsencrypt.org/acme/new-authz",
   "new-cert": "https://acme-v01.api.letsencrypt.org/acme/new-cert",
   "new-reg": "https://acme-v01.api.letsencrypt.org/acme/new-reg",
   "revoke-cert": "https://acme-v01.api.letsencrypt.org/acme/revoke-cert"
}
```

Accounts

The account resource is a container for information about a user and that user's account with the ACME server. Most importantly, an account contains a public key encoded as a *JSON Web Key* (JWK)[11]. This public key is associated with the account at the time of account creation and is used to authenticate future requests. Like the other resources we'll see, an account is identified by its URI per usual REST practice. Account resources also contain additional metadata such as an e-mail address to contact for the account and whether a required Terms of Service Agreement has been acknowledged. In the earlier stages of the ACME draft accounts were called *registrations*, and you may still see references using this term in older material.

Accounts are created by POSTing an account resource to the new-account resource of the ACME server. Future updates (for example, to update contact information) are handled in a similar fashion. Notably you cannot view your current account information by sending a GET request to the account URI; instead you must use a POST request with an empty body. The rationale for this decision is rooted in the security model of the protocol. Only POST requests carry the required JWS to authenticate the request as coming from the account owner. If you use a JWS signed empty body in a POST request to retrieve account information, only the authorized account can view contact information.

An example POST body for a new account follows:

```
{
  "terms-of-service-agreed": true,
  "key”: "...",
  "contact": [
    "mailto:example@example.com",
  ]
}
```

Orders

Orders encapsulate the request of an account for a certificate to be issued by the ACME server. The most important field of an order object is the *Certificate Signing Request* (CSR). You might be familiar with non-ACME CAs; the ACME CSR is a standard RFC 2986[12] CSR, meaning existing tools (for example, *openssl*) can generate CSRs for use with ACME. For use within an Order the CSR is base64url-encoded, a practice used elsewhere in the protocol when binary data needs to be represented in a request.

The other important field of an Order object is the Authorizations field, containing an array of Authorization URIs. The ACME server is responsible for populating this field in the Order object returned to the client when a new order is created. Completing an order to obtain a certificate requires first completing each of the authorizations the order links to.

An example of a request body to create a new order for two DNS identifiers would resemble the following:

```
{
   "csr":”MIICmTCCAYECAQAw....cUc5i8XK-OBEMe",
}
```

Resulting in:

```
{
   "status": "pending",
   "expires”: "2017-03-14T12:41:37-04:00",
   "csr":"MIICmTCCAYECAQAw....cUc5i8XK-OBEMe",
   "authorizations": [
      "/authZ/k4jO5648Y-qqrQ_F-bD6JLgtrfV4TJb6vef9GrlybvQ",
      "/authZ/c71yuTUHsuwIVeCk9B4DrsFA1MlCZMLtt4FDZ71KI20"
   ]
}
```

Presently, as described in the Boulder divergences document, Let's Encrypt does not implement the order resource. Instead clients must explicitly create authorization objects for each of the domains they wish to issue for themselves using the *new-authz* endpoint, as opposed to creating an order and receiving the URI of authorizations required from the server.

Authorizations

Authorizations are the core of the domain validation process in ACME. For an account to receive a certificate valid for an identifier, the CA needs to verify control of that identifier. If control is established, then the account is said to be authorized to request a certificate valid for the domain. In ACME an authorization starts its life in a *pending* status, indicating that the account has not yet completed the authorization process. In order to progress from the pending state to a *valid* state, the account holder must complete a set of required challenges. Authorizations also contain an expiry date, and both pending and valid authorizations fall out of usefulness after their expiry date. In the case of pending authorizations, this requirement keeps the challenges fresh. In the case of valid authorizations, it means that control must be reestablished through a fresh authorization and new challenges if the expiry has passed.

An example authorization follows:

```
{
   "status": "pending",
   "identifier": {
      "type": "dns",
      "value": "www.example.com"
   },
   "challenges": [
      {
        "type": "dns-01",
        "token": "T50nPYe3YNdKeqlqae1egDVftLpqG5D8klP_K7inCHY",
        "status": "pending",
        "error": {}
      }
   ],
   "expires": "2017-03-14T12:41:37-04:00"
}
```

Challenges

One or more challenges are embedded directly into authorizations and are identified by a type and a URI. Solving a challenge of an authorization will demonstrate the ACME account key holder's control over the identifier the authorization refers to, allowing issuance for that identifier. Each challenge type has its own method for demonstrating control, but all share the use of a random *token* and a key authorization.

The token is a random value used to identify the challenge. The token is always expressed in the *base64url* alphabet used throughout ACME, and to facilitate the usage in various challenge types it must not contain any padding characters.

The challenge key authorization is used to concretely link a specified ACME account key with the challenge for the purpose of validating an identifier. It is created by concatenating the random token present in the challenge and the Base64 URL encoding of the JWK thumbprint of the ACME account. The key authorization is provided in the subsequent JWS signed request from the ACME client to update a challenge, asking the server to attempt to verify control by performing the challenge verification process as required by the challenge type.

An example challenge follows:

```
{
   "type": "dns-01",
   "token": "T50nPYe3YNdKeqlqae1egDVftLpqG5D8klP_K7inCHY",
   "status": "pending",
   "error": {}
}
```

Getting a Certificate

After challenges have been completed successfully for each of the authorizations embedded in an order resource, the order is considered valid and the certificate can be issued. The ACME server proactively monitors order resources, and when an order is ready to be issued, it is responsible for issuing a certificate matching the domains from the CSR/authorizations. The order resource is then updated with a URI at which the client can download the issued certificate. After a client completes all of the authorizations the order requires, a polling state can be entered so the certificate URI can be added to the order to allow fetching the produced certificate.

Presently, as described in the Boulder divergences document, Let's Encrypt does not implement the order resource, so the issuance process is slightly different. Instead, clients must explicitly create authorization objects for each of the domains they wish to issue for themselves using the *new-authz* endpoint. After the authorizations are validated by completing challenges, the client can submit a CSR to the *new-cert* endpoint and will receive a certificate as a response provided the server is able to validate that the correct unexpired authorizations are in place.

Challenge Types

The ACME standard defines four distinct challenge types, each identified by the draft that it was introduced in: HTTP-01, DNS-01, TLS-SNI-01, and TLS-SNI-02. An additional *Out-of-Band* (OOB) challenge exists for integration with existing CAs. Let's Encrypt and Boulder presently do not implement TLS-SNI-02 or the OOB challenges.

HTTP-01 Challenges

The HTTP-01 challenge allows authenticating a domain by making an externally visible change to the domain website. The primary idea is that the ACME client must sign the requested key authorization and place the result in a pre-specified location in the domain webroot. The name of the file is the token value from the challenge, and the contents of the file will be the same computed key authorization that is included in the JWS signed POST body asking the server to validate the challenge.

For ACME, the pre-specified location for the challenge file is in `/.well-known/acme-challenge/`, a prefix registered with the *Internet Assigned Numbers Authority* (IANA) for the purpose of ACME domain validation. When the challenge is POSTed by the ACME client with the correct key authorization, the ACME server will make a GET request to this location on the domain referenced in the challenge authorization. Using the HTTP response, the server can validate the contents of the HTTP challenge file. If the correct key authorization was present at the correct location and signed by the correct ACME account key, then the challenge is completed and the account is considered to possess a valid authorization for this domain until the point at which it expires. Both the challenge and authorization objects are updated server side with a valid status and an expiry date.

The HTTP-01 challenge is a great fit if you are already running a world-accessible webserver on port 80 of your domains. Since the challenge requests are standard HTTP requests and will always be directed to a well-known path prefix, it is possible to implement more complex validation systems with ease. For instance, you could use the URL rewriting capabilities of a webserver to divert HTTP-01 challenge requests to a centralized server responsible for Let's Encrypt challenge validation. Many ACME clients (Certbot included) can start up a standalone HTTP server explicitly for the purpose of solving HTTP-01 challenges; this feature may be beneficial for issuing certificates for non-HTTPS services like the *Extensible Messaging and Presence Protocol* (XMPP) without needing to configure a heavyweight HTTP server.

DNS-01 Challenges

The DNS-01 challenge is conceptually similar to the HTTP-01 challenge but instead of provisioning a file at a well-known location the challenge responder provisions a TXT record at a well-known label.

For ACME, the required record is a TXT record for the label **`_acme-challenge.`** concatenated onto the domain being authorized. Rather than placing the entire key authorization as the value of this TXT record, the DNS-01 challenge asks that a SHA256 digest of the computed key authorization be used as the TXT record value.

When the ACME client POSTs the challenge with the JWS signed key authorization, the ACME server will verify the details of the key authorization and token match, and proceed to validate the TXT record by issuing a DNS query against one of the authoritative DNS servers for the domain being authorized. If the contents of that TXT record match the expectation of the server of the SHA256 of the challenge key authorization, then the account is considered to possess a valid authorization for this domain.

The DNS-01 challenge is often used in situations where ports 80 and 443 are not globally accessible (for example, because of corporate firewall policies), ruling out the use of HTTP-01 and TLS-SNI-02 challenges. Since the DNS-01 challenge requires only that a TXT record be updated, there's no requirement for a direct connection to the domain name that a certificate is to be issued for. Instead the challenge is validated through a query to the authoritative nameservers. The DNS-01 challenge is also well-suited to centralized management of certificate issuance. Many DNS providers support programmatic updates through an API, or with more traditional dynamic DNS updates through *nsupdate*-like tools. Using an ACME client that exposes hooks for adding and removing the required TXT records makes it easy to centrally issue certificates by automatically adjusting DNS as required by the challenges. Certbot presently supports DNS-01 in a manual-only mode, but some other ACME clients have fully automatic support with a variety of DNS providers.

TLS-SNI-02 Challenges

The TLS-SNI-02 challenge is perhaps the most unfamiliar of the ACME challenge types. For this challenge type the requester must configure a TLS server accessible at the domain to be authorized such that it will use a special self-signed certificate when processing TLS requests with a specific *Server Name Indication* (SNI)[14] value.

The ACME client creates the self-signed certificate when it wishes to use a TLS-SNI-02 challenge to authorize a domain. The contents of the certificate are unimportant except for one crucial detail: the certificate must have two special *Subject Alternate Name* (SAN) values.

The first SAN value is a domain of the form **`x.y.token.acme.invalid`**, where **`x`** and **`y`** are computed as the SHA256 digest of the challenge token value, split into two labels. The second SAN value is a domain **`x.y.ka.acme.invalid`**, where **`x`** and **`y`** are computed as the SHA256 digest of the key authorization, split into two labels.

The TLS server used for responding to the TLS-SNI-02 challenge should be configured such that it returns the crafted challenge certificate whenever a TLS request arrives with the SNI value of the first SAN (for example, `x.y.token.acme.invalid`).

When the ACME client POSTs the challenge to begin the validation process, the ACME server will compute both SAN entries the same way the client did, and will send a TLS request to the domain using a SNI value of the first computed SAN. The ACME server can then validate that the challenge server presents a self-signed certificate with the two required SAN values verifying the challenge token and the key authorization.

The TLS-SNI-02 challenge is a good fit for environments where a webserver is already configured for HTTPS and you do not want to accept HTTP requests for HTTP-01 challenges or place files in the webroot of the domains. Similar to HTTP-01, Certbot and some other ACME clients can run a standalone TLS server for the purpose of solving TLS-SNI-02 challenges in place without requiring a heavier-weight server. The TLS-SNI-02 challenge uniquely employs the mechanics of certificates and TLS in order to provide authorization for the issuance of certificates; this symmetry of process and result is unique and satisfying from the perspective of an interested engineer.

The astute reader will note that unlike HTTP-01 and DNS-01, the TLS-SNI-02 challenge is on its second revision. Let's Encrypt and the Boulder codebase still use the original TLS-SNI-01 challenge from earlier drafts, but it suffers from one design flaw whereby all of the information required to complete the challenge was present in the request. This situation allows for a broken design where a TLS-SNI-01 challenge response server could be built that automatically replies to a challenge request without *a priori* knowledge of the challenges. To combat this design and its unintended security implications, the TLS-SNI-02 challenge requires that the key authorization, which isn't part of the challenge request, be returned as part of the challenge response.

What's Next?

We've covered the core of the ACME protocol, but the existing drafts have a great deal more information. Readers are encouraged to investigate the key rollover and revocation features of ACME from the standard since they were not covered in this article. Similarly the draft content offers more in-depth coverage of security considerations that may be of interest to readers with the hacker mindset.

The ACME standardization process is still underway. The IETF Working Group has most recently published `draft-07` and is undergoing a last-call process for interoperability testing. Sometime after this point we can expect the ACME draft will proceed to full RFC standard status.

Plenty of work remains to upgrade existing ACME servers and clients to support the latest iterations of the draft since most of the ecosystem is presently implementing ACME closer to the `draft-03` standard. Let's Encrypt intends to support the newer draft and final RFC version as independent directory endpoints alongside the current legacy `draft-03` era endpoint. This support will allow clients to gradually adopt support for the newest protocol features while continuing to renew legacy certificates produced with the `draft-03` endpoint.

The ACME protocol itself has left room for future improvements. Work is underway to develop a companion document[7] describing additions to the CAA standard, RFC 6844[13], that would allow domain owners to specify policy related to acceptable ACME account keys or challenge types. This work could allow for, as an example, adoption of a policy whereby only DNS-01 challenges could be used to issue certificates for a given domain name using ACME.

Standardization on challenges for non-DNS identifierss—such as IP addresses—is also an avenue for future ACME work. ACME was designed to handle additional identifier types and new challenges, and it will be interesting to see how the protocol evolves to handle use cases beyond domain validation of DNS identifiers.

Development of an open standard helps move the Web towards a world where HTTPS encryption is the norm. Certificates from Let's Encrypt are one avenue available to system administrators looking to increase the security of their websites. Adoption of ACME by other CAs and tools ensures that the decision to use HTTPS doesn't induce vendor lock-in and allows users the chance to change providers without abandoning automation. The future of ACME is still being written, and it's not too late to participate in the IETF Working Group[8]. Readers are encouraged to subscribe to the mailing list[9] and provide feedback as they envision integrating ACME into their own software and environments.

References

[0] Let's Encrypt Statistics: `https://letsencrypt.org/stats`

[1] Jacob Hoffman-Andrews, James Kasten, and Richard Barnes, "Automatic Certificate Management Environment (ACME)," Internet Draft, work in progress, `draft-ietf-acme-acme-07`, June 2017.

[2] Github Repository for Boulder: `https://github.com/letsencrypt/boulder`

[3] "Boulder divergences from ACME," `https://github.com/letsencrypt/boulder/blob/e81f7477a3169f77fd7247a6cdb8822fb29433aa/docs/acme-divergences.md`

[4] "Automatically enable HTTPS on your website with EFF's Certbot, deploying Let's Encrypt certificates," **https://certbot.eff.org/**

[5] Wayne Thayer, "Information about SSL Bug," Godaddy Blog, **https://www.godaddy.com/garage/godaddy/information-about-ssl-bug/**

[6] CA Browser Forum, "Ballot 169, Revised Validation Requirements," **https://cabforum.org/2016/08/05/ballot-169-revised-validation-requirements/**

[7] Hugo Landau, "CAA Record Extensions for Account URI and ACME Method Binding," Internet Draft, work in progress, **draft-ietf-acme-caa-01**, February 2017

[8] ACME Working Group Charter, **https://datatracker.ietf.org/wg/acme/charter/**

[9] ACME Mailing List Archive, **https://mailarchive.ietf.org/arch/browse/acme/**

[10] Nat Sakimura, Michael Jones, and John Bradley, "JSON Web Signature (JWS)," RFC 7515, May 2015.

[11] Michael Jones, "JSON Web Key (JWK)," RFC 7517, May 2015.

[12] Burt Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7," RFC 2986, November 2000.

[13] Rob Stradling and Phillip Hallam-Baker, "DNS Certification Authority Authorization (CAA) Resource Record," RFC 6844, January 2013.

[14] Donald Eastlake 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066, January 2011.

[15] Josh Aas, "Wildcard Certificates Coming January 2018," **https://letsencrypt.org/2017/07/06/wildcard-certificates-coming-jan-2018.html**

DANIEL MCCARNEY is a developer for the *Internet Security Research Group* (ISRG), where he works full-time on *Boulder*, the server-side software powering the Let's Encrypt certificate authority. Prior to the ISRG Daniel was a security architect for a large content delivery network and focused on TLS and application security. He has a Masters in Computer Science from Carleton University, where his research touched both Android system security and password managers. Daniel resides in Montréal, Canada, where he enjoys long snowy walks with his dog Bart. Daniel can be reached at: **cpu@letsencrypt.org**

The Root of the Domain Name System

by Geoff Huston, APNIC

Few parts of the *Domain Name System* (DNS) are filled with such levels of mythology as its *root server system*. In this article I will explain what it is all about and ask the question whether the system we have is still adequate, or if it's time to think about some further changes.

The namespace of the DNS is a hierarchically structured label space. Each label can have an arbitrary number of immediately descendant labels and only one immediate parent label. Domain names are expressed as an ordered sequence of labels in left-to-right order starting at the terminal label and then enumerating each successive parent label until the root label is reached. In domain name expresions, the ASCII period character denotes a label delimiter. *Fully Qualified Domain Names* (FQDNs) are names that express a label sequence from the terminal label through to the apex (or *root*) label. In FQDNs this root is expressed as a trailing period character at the end of the label sequence. But there is a little more than that, and that's where the hierarchal structure comes in. The sequence of labels, as read from right to left, describes a series of name delegations in the DNS. If we take an example DNS name, such as **`www.example.com`**, then **`com`** is the label of a delegated zone in the root. Here we will call a *zone* the collection of all defined labels at a particular delegation point in the name hierarchy. The label "**`example`**" is the label of a delegated zone in the **`com.`** zone. And **`www`** is a terminal label in the **`www.example.com.`** zone.

But that is not all there is to the DNS. There are many more subtitles and possibilities for variation, but as we want to look specifically at the root zone, we're going to conveniently ignore all these other matters here. If you are interested, RFC 1034[1] from November 1987 is still a good description of the way the DNS was intended to operate, and the recently published RFC 7719[2] provides a good compendium of DNS jargon.

The most common operation performed on DNS names is to *resolve* the name; resolving is an operation to translate a DNS name to a different form that is related to the name. This most common form of name resolution is to translate a name to an associated *IP address,* although many other forms of resolution are also possible. The resolution function is performed by agents termed *resolvers,* and they function by passing queries to, and receiving results from, so-called *name servers*. In its simplest form, a name server can answer queries about a particular zone. The name itself defines a search algorithm that mirrors the same right-to-left delegation hierarchy.

Continuing with our simple example, to resolve the name **www.example.com.**, we may not know the IP addresses of the authoritative name servers for **example.com.**, or even **com.** for that matter. To resolve this name, a resolver would start by asking one of the *root zone name servers* to tell it the resolution outcome of the name **www.example.com.** The root name server will be unable to answer this query, but it will refer the resolver to the **com.** zone, and the root server will list the servers for this delegated zone, as this delegation information is part of the DNS root zone file for all delegated zones. The resolver will repeat this query to one of the servers for the **com.** zone, and the response is likely to be the list of servers for **example.com.** Assuming **www** is a terminal label in the **example.com.** zone, the third query, this time to a server for the **example.com.** zone, will provide the response we are seeking.

In theory, as per our example, every resolution function starts with a query to one of the servers for the root zone. But how does a resolver know where to start? What are the IP addresses of the servers for the root zone?

Common DNS resolver packages include a local configuration fragment that provides the DNS names and IP addresses of the authoritative name servers for the root zone. Another way is to pull down the current root hints file from **https://www.internic.net/domain/named.root**.

But it may have been some time between the generation of this list and the reality of the IP addresses of the authoritative root servers today, so the first actions of a resolver on startup will be to query one of these addresses for the name servers of the root zone, and use these name servers instead. This query is the so-called *priming query*[3].

This priming implies that the set of root server functions includes supporting the initial bootstrap of recursive DNS resolvers into the DNS framework by responding to priming queries of the resolver, as well as anchoring the process of top-name name resolution by responding to specific name queries with the name server details of the next-level delegated zone. This role is critical in so far as if none of the root servers can respond to resolver queries, then at some point thereafter, as local caches of the resolvers expire, resolvers will be unable to respond to any DNS queries for public names. So, these root servers are important in that you may not know that they exist, or where they may be located in the net, but their absence, if that ever could occur, would definitely be noticed by all of us!

Moderating all considerations of the DNS is the issue of local caching of responses. For example, once a local resolver has queried a root server for the name **www.example.com.**, it will have received a response listing the delegated name servers for the **com.** zone.

If this resolver were to subsequently attempt to resolve a different name in the `com.` zone, then for as long as the `com.` name servers are still held in the resolver cache, the resolver will use the cached information and not query any root server. Given that the number of delegated zones in the root zone is relatively small (1,528 zones as of the start of 2017), then a busy recursive resolver is likely to assemble in its local cache the name servers of many of the top-level domain names. Then one would expect that it would have no further need to query the root name servers, except as required occasionally to refresh its local cache, assuming that it is answering queries about DNS names that exist in the DNS.

In that respect, the root servers would not appear to be that critically important in terms of the resolution of names, and certainly not so for large recursive name servers that have a large client population and therefore have a well-populated local cache. But this conclusion would not be a good one. If cached information of a recursive resolver for a zone has expired, it will need to refresh the cache with a query to a root server. At some point, all of the locally cached information will time out of the cache, and then the resolver will no longer be able to respond to any DNS query. To keep the DNS operating, recursive resolvers need to be able to query the root zone, so there is a requirement that collectively the root servers always need to be available.

In this respect, the root servers "anchor" the entire DNS system. They do not participate in every name resolution query, but without their feed of root zone information into the caches of recursive resolvers the DNS would stop. So these servers are important to the Internet, and it might be reasonable to expect a role of such importance to be performed by hundreds or thousands of such servers. But there are just 13 such root server systems.

Why 13?

The primary reason to have more than a single root server, and use multiple root servers, was diversity and availability. The root servers are intentionally located in different parts of the network, within different service provider networks. The intended objective is that in the case where a DNS resolver is incapable of contacting a root name server, then unless the resolver was itself completely isolated from the Internet, then the desired number of root servers was such that the likelihood that it could not reach any of the root name servers was considered to be acceptably low. By this reasoning, two is probably not enough, and three could well be insufficient as well, but perhaps hundreds of thousands of such root servers may well be a case of overkill!

This line of thought assumes that each named root server has a unique name, a unique IP address, and a single location. But perhaps we are assuming too much. There is a technique that places identically named and addressed servers at various locations across the Internet, called *anycast*[5,6].

Using anycast, a user attempting to send an IP packet to an anycast service would be directed to the "closest" instance of the family of servers that share a common anycast IP address. Why not just use anycast for a collection of root servers and put as many root servers as we want behind a single IP address?

For a considerable time, anycast was viewed with some caution and trepidation, particularly in the days before *Domain Name System Security Extensions* (DNSSEC) of a signed root zone. What would stop a hostile actor from setting up a fake root server and publishing incorrect DNS information if the IP addresses the root servers used could be announced mutiple times from any arbitrary location? There was also some doubt that the *Transmission Control Protocol* (TCP) would be adequately robust in such anycast scenarios. The original conservative line of thinking was that we needed multiple unitary DNS root zone servers, each with its own unique IP address announced from known points in the network fabric.

But needing "multiple" DNS root zone servers and coming up with the number 13 appears to be somewhat curious. It seems such an odd limitation in the number of root servers given that a common general rule in computer software design is Willem van der Poel's *Zero, One, or Infinity Rule,* which states a principle in computer science that either an action or resource should not be permitted (zero), should happen uniquely (one), or should have no arbitrary limit at all (infinity). For root servers, it appears that we would like more than one root server. But why set the limit to 13?

The reason may not be immediately obvious these days, but when the DNS system was designed, the size limit of DNS responses using the *User Datagram Protocol* (UDP) was set to 512 bytes (Section 2.3.4 of RFC 1035). It seems a ludicrously small limit these days, but you have to also account for the fact that the requirement for IPv4 hosts was (and still is) that it accepts IPv4 packets up to 576 bytes long[4]. Working backwards, that would imply that if you account for a 20-octet IPv4 packet header and an 8-byte UDP header, then the UDP payload could be up to 548 octets long, but no longer if you wanted some degree of assurance that the remote host would accept the packet. If you also allow for up to 40 bytes of IP options, then in order to ensure UDP packet acceptance under all circumstances the maximal UDP payload size should be 508 octets. The DNS use of a maximum payload of 512 bytes is not completely inconsistent with this assumption, but it is off by 4 bytes in this corner case!

This 512-byte size limit of DNS packets still holds, in that a query without any additional signal—that is, in today's terms, a query that contains no DNS extension mechanisms that signal a capability to use a larger UDP response size—is supposed to be answered by a response with a DNS payload no greater than 512 octets long. If the actual response would be greater than 512 octets, then the DNS server is supposed to truncate the response to fit within 512 octets, and mark this partial response as *truncated*.

If a client receives a truncated response, then the client may repeat the query to the server, but use TCP instead of UDP, so that it could be assured of receiving the larger response.

The desire in the design of the DNS priming query and response was to provide the longest possible list of root name servers and addresses in the priming response, but at the same time ensure that the response was capable of being passed in the DNS using UDP, and not rely on the use of any form of optional DNS extension mechanism. The largest possible set of names that could be packed in a 512-octet DNS response in this manner was 13 such names and their IPv4 addresses—so there are at most 13 distinct root name servers in order to comply with this limit.

These days every root name server has an IPv6 address as well as an IPv4 address, so the DNS priming response that lists all these root servers and their IPv4 and IPv6 addresses is now 811 octets. If the resolver also requests that the response should include the DNSSEC signatures, then the size of the response would expand to 1,097 bytes. But if you pass a simple priming query to a root server without a UDP buffer size extension in the query, then you will still receive no more than 512 octets in response. The size-limited response will still list the names of all 13 root name servers, but will not list all of their IPv4 and IPv6 addresses in the additional section of the response.

The partial set of these additional records of root server names and their IPv4 and IPv6 addresses is passed back without any particular indication of what is missing. The decision as to which records to include and which to omit to meet the size restriction also varies between root name servers. Some root name servers provide the IPv6 addresses of root servers A through J in a 508-byte response, while others give all 13 IPv4 addresses and add the IPv6 addresses of A and B in a 492-byte response. The remainder provide the IPv4 and IPv6 addresses for A through F and the IPv4 address of G in a 508-byte response. I suppose that the message here is that recursive resolvers should support the *Extension Mechanisms for DNS* (EDNS(0)) as specified in RFC 6891[14], and offer a UDP buffer size that is no less than 1,097 bytes if they want a complete DNSSEC-signed response to a root zone priming query.

However, even then the story is incomplete. These additional records are not DNSSEC-signed in the priming response, so if a resolver wants to assure itself that the IP addresses that are provided in this response are the actual IP addresses of the root servers, it needs to separately query these names and request DNSSEC credentials in the response. However, as of the time of writing of this article the zone **`root-servers.net`** is not DNSSEC-signed, so right at the heart of the DNS there is still a leap of faith that all resolvers need to make in order to link into the DNS through the priming process.

We are also entirely comfortable with anycast these days, and the root server system has enthusiastically adopted anycast, where most of the root servers are replicated in many locations. The overall result is that hundreds of locations host at least one instance of one of the root server anycast constellations, and often more. Part of the reason that our comfort level with anycast has increased is the use of a DNSSEC-signed zone, and recursive resolvers should be protecting their clients by validating the response they receive to ensure that they are using the genuine root zone data, to the extent that this data has been signed in the first place.

Should we do more?

It would certainly make some sense to sign the `root-servers.net` zone to further protect recursive resolvers from being led astray.

But what about the specification of 13 unique root server names and their associated anycast constellations? If we had more root servers, would it make everything else better? Should we contemplate further expanding these anycast constellations into thousands or even tens of thousands of root servers? Should we open up the root server letter set to more letters? Is there a limit to "more" or "many"? Where might that limit be, and why?

These days the response that recursive resolvers receive in 512 bytes or less is a partial view of the root name server system. From that perspective, 13 is not a practical protocol-derived ceiling on the number of distinct root server letters. Whether the partial response in 512 bytes reflects 6, 10, or 13 root name servers out of a pool of 13 or 14 or any larger number is largely no longer relevant. The topic has moved beyond a conversation about any numeric ceiling on the letter count into a consideration of whether more root server letters would offer any incremental benefit to the Internet, as distinct from the current practice of enlarging the root server anycast constellations. Indeed, rather than more root name servers, whether by adding more letters or enlarging anycast constellations, should we consider alternative approaches to the DNS that can scale and improve resilience under attack through answering root queries but not directly involving these root name servers at all? In other words, can we look at DNS structures that use the root servers as a distribution mechanism for the root zone data and use the existing recursive resolver infrastructure to directly answer all queries that relate to data in the root zone?

The reason to contemplate this question is that it is not clear that more root server letters or more root server anycast instances, or even both measures, make everything else better. Reducing the latency in querying a root name server has only a minimal impact for end users.

The design objective of the DNS system is to push the data as close to the user as possible in the first place, so that every effort is made to provide an answer from a local resolver cache.

It is only when there is a cache miss that the resolver query will head back into the authoritative DNS server infrastructure, a situation that would normally affect only a very small proportion of queries over time. The DNS derives its performance and efficiency through resolver caches, so the overall intention is to limit the extent to which resolvers query these root name servers to the minimal level possible.

Secondly, a local root name server may not necessarily provide any additional name resolution resilience in the case of local network isolation. Secondary root name servers also have an expiry time on the data they serve, and in the case of extended isolation the server will also time out a case to be able to respond. This timeout is as true for the root zone as it is for any other zone.

In many ways, the net effect of a local root name server on local users' Internet experience is minimal, and could well pass completely unnoticed in many cases.

In terms of the primary objectives of the root name server system, diversity and availability, there is little to be gained by adding additional root name letters. A significant expansion of the number of uniquely named root servers would ultimately make a complete priming response exceed 512 bytes, meaning either forcing all priming queries into TCP by signalling that the UDP response was truncated, or dropping some named root servers from a non-EDNS(0) priming query response.

But rather than resisting the hard limits imposed by protocol specifications in some early RFCs, perhaps we are asking the wrong question. Rather than trying to figure out how to field even more instances of root servers and keep them all current, there is perhaps a different question: Why do we need these special dedicated root zone servers at all?

If the only distinguishing feature of these root servers is the proposition that any response with a source address of any of these 26 distinguished IP addresses is by simple unfounded assertion the absolute truth, then it is laughably implausible. Anyone who has experienced DNS interceptors would have to agree that DNS lies are commonplace, and nation states as well as service providers across the entire Internet practice lying.

Enter DNSSEC

The DNSSEC-signing of the root zone of the DNS introduced further possibilities to the root zone service to resolvers. If a resolver has a validated local copy of the current *Key Signing Key* (KSK), then it can independently validate any response provided to it from any signed zone that has a chain of signing back to this KSK, including of course any signed response about the root zone itself.

A validating resolver no longer needs to obsess that it is querying a genuine root name server, and no longer needs to place a certain level of blind faith in the belief that its DNS queries are not being intercepted and that faked responses are not being substituted for the actual response. With DNSSEC it simply does not matter in the slightest how you get the response. What matters is that you can validate responses with your local copy of the root zone key. If you can perform this validation successfully, then the answer is much more likely to be genuine!

The ubiquitous use of DNSSEC casts the root server system in an entirely different light, and the relationship between recursive resolvers and the root servers can change significantly.

A relevant observation here is that some 75% of responses from the root zone are "no such domain" NXDOMAIN responses (for example,[7]). Recursive resolvers could absorb much of the root server query load and answer these queries directly with NXDOMAIN responses if they used this form of response synthesis. The way resolvers could answer the queries is to use so called "aggressive NSEC caching[11]." This approach uses the *Next Secure* (NSEC) records provided in the responses relating to the nonexistence of a name in the root zone to allow recursive resolvers to synthesise an authoritative NXDOMAIN response for queries relating to any name in the range specified in the NSEC data. Rather than caching a root zone NXDOMAIN answer for each individual nonexistent domain name, caching the NSEC response allows the recursive resolver to cache a common signed response for the entire span of query names as described in each NSEC response. With a cache of 1,528 defined top-level domains and another 1,528 NSEC records, a recursive resolver would be able to provide authoritative responses for any query that would otherwise be passed through to a root server.

Another approach is to use *local secondaries* for the root zone. This approach is not an architectural change to the DNS, or at least not intentionally so. For recursive resolvers that implement this approach, this change is a form of change in query behaviour in so far as a recursive resolver configured in this manner will no longer query the root servers for queries it would normally direct to an instance of the root. Instead, it directs these queries to a local instance of a slave server that is listening on the loopback address of the recursive resolver. This slave server is serving a locally held instance of the root zone, and the recursive resolver would perform DNSSEC validation of responses from this local slave to ensure the integrity of responses received in this manner. In effect, this technique loads a recursive resolver with the entire root zone into what is functionally similar to a local secondary root zone server cache. For users of this recursive resolver there is no apparent change to the DNS or to their local configurations. Obviously, there is no change to the root zone either.

This proposal provides integrity in the local root server through the mechanism of having the recursive resolver perform DNSSEC validation against the responses received from the local root slave. If the recursive resolver is configured as a DNSSEC-validating resolver, then this mechanism is configurable on current implementations of DNS recursive resolvers.

The advantage here is that the decision to set up a local slave root server or to use aggressive NSEC caching is a decision that is entirely local to the recursive resolver, and the impacts of this decision affect only the clients of this recursive resolver. No coordination with the root server operators is required, nor is any explicit notification. The outcomes are only indirectly visible to the clients of this recursive resolver, and no other.

Where does this leave the root server system?

In the light of increasing use of DNSSEC, the root server system is declining in relevance as a unique source of authoritative responses for the root zone, and we can forecast a time when their role in resolving queries would be largely anachronistic. A validated response can be considered a genuine response regarding the contents of the root zone, regardless of how the recursive resolver learned this response. It is no longer necessary to have a dedicated set of name servers running on a known set of IP addresses as the only means to protect the integrity of the root zone.

It is also true that the root servers are no longer being used as cache refresh for recursive resolvers for delegated domains. Today we see much of the time, effort and energy, and cost of root server operation being spent to ensure that NXDOMAIN answers are provided promptly and reliably. This use of time really does not make any sense these days. The use of local secondary root servers and the use of NSEC caching can remove all of these specific queries relating to undefined names to the root servers, and what would be left is the cache priming queries. If all recursive resolvers were able to use either of these measures, then the residual true role of the root server system would not be to respond to individual queries, but simply to distribute current root zone data into the resolver infrastructure.

If the functional intention of the root server system is to distribute signed root zone data to recursive resolvers, then perhaps we could look more widely for potential approaches. Regularising the times that changes are made to the root zone would help reduce opportunistic polling of the root servers to detect when a change might have occurred. Or using an approach based on *Incremental Zone Transfer* (IXFR) that would allow recursive resolvers to request incremental changes to the root zone based on differences between zone *Start of Authority* (SOA) numbers may be more efficient.

Maybe we can look further afield for additional ways to distribute the root zone contents. Social networks appear to be remarkably adept in their ability to distribute updates, and a thought is that the small set of incremental changes to the signed root zone would be highly amenable to similar techniques or even using the same social networks. One can readily imagine a feed of incremental root zone updates on media such as Twitter, for example!

I also can't help but wonder about the wisdom of the root zone servers being promiscuous with respect to whom they answer. Root zone query data points to some 75% of queries seen at the root zone servers generating NXDOMAIN responses, meaning that three-quarters of the responses from root servers are nonsensical questions in the context of the root zone. It's not clear to what extent the other 25% of queries reflect actual user activity. In an APNIC measurement exercise using synthetic domain names that included a time component, it was evident that more than 30% of the queries seen at the authoritative servers of the measurement reflected "old" queries, generated by query log replay or other DNS forms of stalking activities.

One way to respond to this situation is to farm out the query volume currently seen at the root servers into the existing recursive resolver infrastructure, so that all root zone responses are generated by these recursive resolvers, rather than passing queries onward to the root servers. If the root servers exclusively served some form of incremental zone transfer and did not answer any other query type directly, then we would see a shift in query traffic away from the root servers as a crucial DNS query attractor, leaving only a lower profile role as a server to recursive resolvers.

There is much to learn about the DNS, and there is still much we can do in trying to optimise the DNS infrastructure to continue to be robust, scalable, and accurate—all essential attributes to underpin the continued growth pressures of the Internet.

References and Further Reading

[1] P.V. Mockapetris, "Domain names - concepts and facilities," RFC 1034, November 1987.

[2] Kazunori Fujiwara, Paul Hoffman, and Andrew Sullivan, "DNS Terminology," RFC 7719, December 2015.

[3] Peter Koch, Matt Larson, and Paul Hoffman, "Initializing a DNS Resolver with Priming Queries," RFC 8109, March 2017.

[4] J. Postel, "Internet Protocol," RFC 791, September 1981.

[5] Kurt Erik Lindqvist and Joe Abley, "Operation of Anycast Services," RFC 4786, December 2006.

[6] David Oran, Dave Thaler, Eric Osterweil, and Danny McPherson, "Architectural Considerations of IP Anycast," RFC 7094, January 2014.

[7] **http://stats.dns.icann.org/rssac/2017/01/rcode-volume/l-root-20170130-rcode-volume.yaml**

[8] "RSSAC023: History of the Root Server System—A Report from the ICANN Root Server System Advisory Committee (RSSAC)," November 4, 2016.
https://www.icann.org/en/system/files/files/rssac-023-04nov16-en.pdf

[9] Marc Blanchet and Lars-Johann Liman, "DNS Root Name Service Protocol and Deployment Requirements," RFC 7720, December 2015.

[10] Paul Hoffman and Warren Kumari, "Decreasing Access Time to Root Servers by Running One on Loopback," RFC 7706, November 2015.

[11] Akira Kato, Warren Kumari, and Kazunori Fujiwara, "Aggressive use of DNSSEC-validated Cache," May 2017, Internet Draft, work in progress, **draft-ietf-dnsop-nsec-aggressiveuse-10**

[12] Geoff Huston, "Workshop on DNS Future Root Service," December 2014.
http://www.potaroo.net/ispcol/2014-12/futureroots.html

[13] DNS RFCs: **https://www.isc.org/community/rfcs/dns/**

[14] Paul Vixie, Joao Damas, and Michael Graff, "Extension Mechanisms for DNS (EDNS(0))," RFC 6891, April 2013.

GEOFF HUSTON, B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for building the Internet within the Australian academic and research sector in the early 1990s. He is author of numerous Internet-related books, and was a member of the Internet Architecture Board from 1999 until 2005. He served on the Board of Trustees of the Internet Society from 1992 until 2001. At various times Geoff has worked as an Internet researcher, an ISP systems architect, and a network operator. E-mail: **gih@apnic.net**

Fragments

ISOC Issues Paper on Content Blocking

The *Internet Society* (ISOC) recently voiced its commitment to keeping the Internet on for everyone, in response to the increasing number of government orders to temporarily shut down or restrict access to Internet services. Speaking out at *RightsCon 2017,* the world's leading conference on Internet and human rights that took place in late March in Brussels, the organization underscored that any deliberate attempt to interrupt Internet communications or control the flow of information over the Internet puts society at risk.

Internet shutdowns, including those that impact social media sites or entire networks, occur when governments intentionally disrupt the Internet or mobile apps, often used in the context of elections, demonstrations or other tense social contexts. According to *Access Now,* there were 56 Internet shutdowns recorded worldwide in 2016, an upward trend from previous years.

A paper entitled "Internet Society Perspectives on Internet Content Blocking,"[1] explores the most common Internet restriction techniques and highlights the shortcomings and collateral damage from the use of such measures. "From censorship to SMEs going out of business, the human, economic and technical costs of Internet shutdowns are just too high," explains Nicolas Seidler, Senior Policy advisor at the Internet Society.

The paper describes and evaluates the most common content blocking techniques used by governments to restrict access to information (or related services) that is either illegal in a particular jurisdiction, is considered a threat to public order, or is objectionable for a particular audience.

According to Freedom House's *Freedom on the Net report 2016,* governments in 24 of the 65 countries assessed impeded access to social media and communication tools, up from 15 the previous year.

"Before they take action, we are calling policymakers to think twice: Internet shutdowns and content filtering are not the answer," said Constance Bommelaer, Senior Director for Global Internet Policy at the Internet Society. "We are at a crossroads, and the actions we take today will determine whether the Internet will continue to be a driver of empowerment, or whether it will threaten personal freedoms and rights online," added Bommelaer.

The Content Blocking paper can be downloaded in various formas and languages from ISOC's website[1]. Quoting from the Foreword: "The use of Internet blocking by governments to prevent access to illegal content is a worldwide and growing trend. There are many reasons why policy makers choose to block access to some content, such as online gambling, intellectual property, child protection, and national security.

However, apart from issues relating to child pornography, there is little international consensus on what constitutes appropriate content from a public policy perspective.

The goal of this paper is to provide a technical assessment of different methods of blocking Internet content, including how well each method works and what are the pitfalls and problems associated with each. We make no attempt to assess the legality or policy motivations of blocking Internet content.

Our conclusion, based on technical analyses, is that using Internet blocking to address illegal content or activities is generally inefficient, often ineffective and generally causes unintended damages to Internet users.

From a technical point of view, we recommend that policy makers think twice when considering the use of Internet blocking tools to solve public policy issues. If they do and choose to pursue alternative approaches, this will be an important win for a global, open, interoperable and trusted Internet."

[1] **https://www.internetsociety.org/doc/internet-content-blocking**

IAB Issues RFC on Protocol Adoption and Transition

The *Internet Architecture Board* (IAB) has recently published a *Request for Comments* (RFC) on Protocol Adoption and Transition[1]. The abstract states: "Over the many years since the introduction of the Internet Protocol, we have seen a number of transitions throughout the protocol stack, such as deploying a new protocol, or updating or replacing an existing protocol. Many protocols and technologies were not designed to enable smooth transition to alternatives or to easily deploy extensions; thus, some transitions, such as the introduction of IPv6, have been difficult. This document attempts to summarize some basic principles to enable future transitions, and it also summarizes what makes for a good transition plan."

[1] Thaler, D., Ed., "Planning for Protocol Adoption and Subsequent Transitions," RFC 8170, May 2017.

Thank You!

Publication of IPJ is made possible by organizations and individuals around the world dedicated to the design, growth, evolution, and operation of the global Internet and private networks built on the Internet Protocol. The following individuals have provided support to IPJ. You can join them by visiting **http://tinyurl.com/IPJ-donate**

Fabrizio Accatino
Scott Aitken
Antonio Cuñat Alario
Matteo D'Ambrosio
Jens Andersson
Danish Ansari
David Atkins
Jaime Badua
John Bigrow
Axel Boeger
Kevin Breit
Ilia Bromberg
Christophe Brun
Gareth Bryan
Stefan Buckmann
Scott Burleigh
Jon Harald Bøvre
Olivier Cahagne
Roberto Canonico
Lj Cemeras
Dave Chapman
Stefanos Charchalakis
Greg Chisholm
Narelle Clark
Steve Corbató
Brian Courtney
Dave Crocker
Kevin Croes
John Curran
Morgan Davis
Freek Dijkstra
Geert Van Dijk
Ernesto Doelling
Karlheinz Dölger
Andrew Dul
Holger Durer
Peter Robert Egli
George Ehlers
Peter Eisses
Torbjörn Eklöv
ERNW GmbH
ESdatCo
Steve Esquivel
Mikhail Evstiounin
Paul Ferguson
Christopher Forsyth
Craig Fox
Tomislav Futivic
Edward Gallagher
Andrew Gallo
Chris Gamboni
Xosé Bravo Garcia
Kevin Gee
Serge Van Ginderachter
Greg Goddard
Octavio Alfageme Gorostiaga
Barry Greene
Martijn Groenleer
Geert Jan de Groot
Gulf Coast Shots
Sheryll de Guzman
Martin Hannigan
John Hardin
Edward Hauser
Headcrafts SRLS
Robert Hinden
Edward Hotard
Bill Huber
Hagen Hultzsch
Karsten Iwen
David Jaffe
Dennis Jennings
Edward Jennings
Jim Johnston
Jonatan Jonasson
Daniel Jones
Gary Jones
Amar Joshi
Merike Kaeo
David Kekar
Shan Ali Khan
Nabeel Khatri
Henry Kluge
Carsten Koempe
Alexader Kogan
Mathias Körber
John Kristoff
Terje Krogdahl
Bobby Krupczak
Warren Kumari
Darrell Lack
Yan Landriault
Markus Langenmair
Fred Langham
Richard Lamb
Tracy LaQuey Parker
Robert Lewis
Sergio Loreti
Guillermo a Loyola
Hannes Lubich
Dan Lynch
Miroslav Madic´
Alexis Madriz
Carl Malamud
Michael Malik
Yogesh Mangar
Bill Manning
Harold March

David Martin
Timothy Martin
Gabriel Marroquin
Carles Mateu
Juan Jose Marin Martinez
Brian McCullough
Joe McEachern
Carsten Melberg
Kevin Menezes
Bart Jan Menkveld
William Mills
Thomas Mino
Mohammad Moghaddas
Charles Monson
Andrea Montefusco
Fernando Montenegro
Soenke Mumm
Tariq Mustafa
Stuart Nadin
Mazdak Rajabi Nasab
Krishna Natarajan
Darryl Newman
Ovidiu Obersterescu
Mike O'Connor
Carlos Astor Araujo Palmeira
Alexis Panagopoulos
Manuel Uruena Pascual
Ricardo Patara
Dipesh Patel
Alex Parkinson
Craig Partridge
Dan Paynter
Leif-Eric Pedersen
Juan Pena
Chris Perkins
Rob Pirnie
Blahoslav Popela
Tim Pozar
David Raistrick
Priyan R Rajeevan
Paul Rathbone
Bill Reid
Rodrigo Ribeiro
Justin Richards
Mark Risinger
Ron Rockrohr
Carlos Rodrigues
Lex Van Roon
William Ross
Boudhayan Roychowdhury
Carlos Rubio
RustedMusic
Babak Saberi
George Sadowsky
Scott Sandefur
Sachin Sapkal
Arturas Satkovskis
Phil Scarr
Jeroen Van Ingen Schenau
Carsten Scherb
Roger Schwartz
SeenThere
Scott Seifel
Yury Shefer
Yaron Sheffer
Tj Shumway
Jeffrey Sicuranza
Thorsten Sideboard
Henry Sinnreich
Geoff Sisson
Helge Skrivervik
Darren Sleeth
Mark Smith
Job Snijders
Ignacio Soto Campos
Peter Spekreijse
Thayumanavan Sridhar
Matthew Stenberg
Adrian Stevens
Clinton Stevens
Viktor Sudakov
Edward-W. Suor
Vincent Surillo
Roman Tarasov
David Theese
Sandro Tumini
Phil Tweedie
Steve Ulrich
Unitek Engineering AG
John Urbanek
Martin Urwaleck
Betsy Vanderpool
Surendran Vangadasalam
Alejandro Vennera
Luca Ventura
Tom Vest
Dario Vitali
Randy Watts
Andrew Webster
Tim Weil
Jd Wegner
Rick Wesson
Peter Whimp
Jurrien Wijlhuizen
Pindar Wong
Bernd Zeimetz

Call for Papers

The *Internet Protocol Journal* (IPJ) is a quarterly technical publication containing tutorial articles ("What is...?") as well as implementation/operation articles ("How to..."). The journal provides articles about all aspects of Internet technology. IPJ is not intended to promote any specific products or services, but rather is intended to serve as an informational and educational resource for engineering professionals involved in the design, development, and operation of public and private internets and intranets. In addition to feature-length articles, IPJ contains technical updates, book reviews, announcements, opinion columns, and letters to the Editor. Topics include but are not limited to:

- Access and infrastructure technologies such as: Wi-Fi, Gigabit Ethernet, SONET, xDSL, cable, fiber optics, satellite, and mobile wireless.
- Transport and interconnection functions such as: switching, routing, tunneling, protocol transition, multicast, and performance.
- Network management, administration, and security issues, including: authentication, privacy, encryption, monitoring, firewalls, troubleshooting, and mapping.
- Value-added systems and services such as: Virtual Private Networks, resource location, caching, client/server systems, distributed systems, cloud computing, and quality of service.
- Application and end-user issues such as: E-mail, Web authoring, server technologies and systems, electronic commerce, and application management.
- Legal, policy, regulatory and governance topics such as: copyright, content control, content liability, settlement charges, resource allocation, and trademark disputes in the context of internetworking.

IPJ will pay a stipend of US$1000 for published, feature-length articles. For further information regarding article submissions, please contact Ole J. Jacobsen, Editor and Publisher. Ole can be reached at **ole@protocoljournal.org** or **olejacobsen@me.com**

Supporters and Sponsors

Supporters

Internet Society

cisco

Diamond Sponsors

Google

ICANN

Ruby Sponsor

RIPE NCC
RIPE NETWORK COORDINATION CENTRE

Sapphire Sponsors

Your logo here!

Emerald Sponsors

Afilias

APNIC

APRICOT
Asia Pacific Regional Internet Conference on Operational Technologies

cisco

CNNIC
中国互联网络信息中心
CHINA INTERNET NETWORK INFORMATION CENTER

COMCAST

DE-CIX

EQUINIX

JUNIPER NETWORKS

lacnic

LinkedIn

NSRC
Network Startup Resource Center

NTT Communications

TEAM CYMRU

VERISIGN

WIDE PROJECT

Corporate Subscriptions

AFRINIC
The Internet Numbers Registry for Africa

amsix
amsterdam internet exchange

DNS-OARC

InterWorking Labs®

ISC
Internet Systems Consortium

Limelight NETWORKS

netnod

qa cafe

SIDN

For more information about sponsorship, please contact `sponsor@protocoljournal.org`

The Internet Protocol Journal
NMS
535 Brennan Street
San Jose, CA 95131

ADDRESS SERVICE REQUESTED

The Internet Protocol Journal is published quarterly and supported by the Internet Society and other organizations and individuals around the world dedicated to the design, growth, evolution, and operation of the global Internet and private networks built on the Internet Protocol.

Email: ipj@protocoljournal.org
Web: www.protocoljournal.org

Printed in the USA on recycled paper.