iab. TECH LAB

ads.cert Open Source Software

Design Doc

Released for Public Comment September 2021

Presented by the IAB Tech Lab Cryptographic Security Foundations working group

Please email support@iabtechlab.com with feedback or questions. This document is available online at https://iabtechlab.com/standards/ads-cert/

Program Leaders:
Curtis Light, Staff Software Engineer - Google
Rob Hazan, Senior Director, Product - Index Exchange

Other Significant Contributions from:
Ben Antier, CEO - Publica
Nabhan El-Rahman, CTO - Publica
Joshua Gross, Senior Engineering Lead - Index Exchange
Bret Ikehara, Staff Software Engineer, Publica
Johnny Li, Software Engineer, Index Exchange
Amit Shetty, Programmatic Products & Partnerships - IAB Tech Lab
Sam Mansour, Principal Product Manager - Moat
Miguel Morales, CTO & Co-Founder - Lucidity Tech
Colm Geraghty, Principal Architect - Verizon Media Group
Mani Gandham, Engineering - Index Exchange
James Wilhite, Director of Product management, Publica

IAB Tech Lab Lead:
Amit Shetty
VP, Programmatic Products & Partnerships - IAB Tech Lab

About IAB Tech Lab

The IAB Technology Laboratory (Tech Lab) is a non-profit research and development consortium that produces and provides standards, software, and services to drive growth of an effective and sustainable global digital media ecosystem. Comprised of digital publishers and ad technology firms as well as marketers, agencies, and other companies with interests in the interactive marketing arena, IAB Tech Lab aims to enable brand and media growth via a transparent, safe, effective supply chain, simpler and more consistent measurement, and better advertising experiences for consumers, with a focus on mobile and TV/digital video channel enablement. The IAB Tech Lab portfolio includes the DigiTrust real-time standardized identity service designed to improve the digital experience for consumers, publishers, advertisers, and third-party platforms. Board members include AppNexus, ExtremeReach, Google, GroupM, Hearst Digital Media, Integral Ad Science, Index Exchange, LinkedIn, MediaMath, Microsoft, Moat, Pandora, PubMatic, Quantcast, Telaria, The Trade Desk, and Yahoo! Japan. Established in 2014, the IAB Tech Lab is headquartered in New York City with an office in San Francisco and representation in Seattle and London.

Learn more about IAB Tech Lab here: www.iabtechlab.com

TABLE OF CONTENTS

Documentation links

- ads.cert Primer
- ads.cert Open Source Software Design Doc (this doc)
- ads.cert Call Signs Protocol Specification
- ads.cert Authenticated Connections Protocol Specification
- ads.cert Open Source Software Implementer's Guide

Executive Summary

This document establishes the technical requirements and principles driving the design of the *ads.cert* open source software library, which implements the Authenticated Connections protocol and will be further leveraged to enable other ads.cert protocols in the future. It also provides design rationale behind underlying protocols.

Objective

The refreshed ads.cert protocols published by IAB Tech Lab let advertising industry participants secure programmatic ad buying and selling using industry-standard cryptographic security protocols.

Rather than asking every integrator to understand and write the complex code needed to generate and sign/verify messages adhering to these protocols, we're providing an industrial strength open source implementation that the community may use and improve over time which encapsulates these details. While we publish documentation for the underlying message formats and algorithms, we strongly encourage that implementers leverage and help us improve this common code base where we can minimize the effort needed to adopt security within programmatic ads.

Requirements

Functional

- Encapsulates and provides the ads.cert Authenticated Connections protocol services
- Obtains and parses counterparty public key configurations from DNS according to the ads.cert specifications
- Generates signature strings that vouch for HTTP request originating party and message integrity, included by the integrator within HTTP request headers
- Verifies signature strings presented by a counterparty
- Provides an operational control surface to configure/override system behavior
- Supports online and offline (logged) signature verification

Non-functional

- Deployment scalability: from 1's to 1,000's of ad servers signing and verifying messages within an enterprise
- Ease of use within enterprises of all sizes, with minimal deployment overhead where possible
- Compatible with all/most integrator software languages
- Continuity of operations: signing and verification process success should not be within the critical path of proper advertising delivery, so resilience must be designed at all stages, and signatures/verification should "fail fast" for individual ad requests
- Diagnostics: administrators need sufficient visibility into system operation to ensure proper functioning and diagnose issues.
- Minimal added latency to sign requests and verify signatures
- Hermetic builds and deployment: application images need to be built and deployable in a controlled fashion so that containerized execution environments can properly control and secure access to secure credentials needed by the application.
- Equitable access: enterprises of all sizes and resources to investigate towards security need to be able to benefit from this tooling with minimal investment
- Auditability: security-sensitive processes need to have auditable controls
- Security: a default deployment should benefit from robust security out-of-box, and the guidance provided by our documentation should give best practices that have been vetted by security experts
- Privacy: the protocols and systems should not create privacy risks to consumers, nor should they require disclosure of proprietary information to third parties

- Testability: a well-tested software product, providing appropriate hooks for letting integrators properly test their implementation using the libraries
- Usability: APIs should be easy to understand and difficult to use incorrectly
- Supportability: non-advertising technology businesses (e.g. publishers) may benefit from integrating this software, but they may need guidance in doing so. An advertising product's support team needs to be able to help guide these parties through the implementation process, so a simple and well-documented API should benefit this effort.

Out of scope

- The current implementation focuses on providing a minimum viable solution. Some architectural improvements and security hardening may wait until a future iteration of the software.
- We plan to provide conformance testing suites (if there is demand for this) to assist parties who want to build their own independent implementation to handle these protocols, although that will not be part of the initial implementation.

Related work

Original ads.cert efforts

We're retaining the "ads.cert" branding established from the prior IAB Tech Lab initiative to develop an "ads.cert Signed Bid Requests" protocol, but readers should view the current 2021 "ads.cert" as a complete departure from the design and direction pursued in the original 2018 strategy.

The main weaknesses in the first iteration of ads.cert were documented at that time (see "Limitations and Abuse Vectors" in the original doc) and thus limited the utility of the spec. To succeed, the specification required precise byte-for-byte reconstruction of certain critical fields present in the bid request risking brittleness and inflexibility to adapt to the changing landscape (e.g. privacy, identifiers) which requires intermediary canonicalization and redaction of information to conform to privacy (e.g. IP address truncation).

Most important: the original ads.cert signing scheme provided no protections to help assure that a bid actually got applied to the same genuine bid request origin. After implementing complex signing protocols, opportunities still exist for bids to get applied to unrelated traffic and ultimately circumvent the security mechanisms.

In the meantime, complementary protocol improvements have added substantial new transparency to the ecosystem. Features such as the OpenRTB Supply Chain Object (SCO) provide visibility into the participants facilitating the ad transaction. We have opportunities to

utilize this and other protocol improvements in upcoming standards work where we plan to provide round-trip, mutual authentication between all parties participating in an ad impression delivery.

Email security protocols

The original ads.cert effort was wisely inspired by the Domain Keys Identified Mail (DKIM) email security standard that has been in use for the past decade to reduce the volume of spam and falsified origin email. In this standard, an email system administrator creates a private key that's used to sign the headers and body of email messages sent by the organization's email servers. Receiving email servers can look up the sending server's corresponding public key in DNS and use it to verify the authenticity of the email origin and integrity (even if the message passed through multiple mail transfer agents).

Complementing this protocol is the "sender policy framework" (SPF) scheme: a record within DNS that indicates which IP addresses are permitted to forward email on behalf of a domain. Readers familiar with ads protocols can equate SPF to the seller authorizations provided in the ads.txt protocol, but authorizing IP addresses rather than seller IDs.

DKIM and SPF serve as a great foundation for the next generation of advertising security protocols, so the new ads.cert adopts from them heavily.

Prior designs

Many candidate designs and substantial trial-and-error went into the protocols we ultimately pursued.

Our original strategy for ads.cert Authenticated Connections planned to use TLS client certificates published in a way that servers receiving server-to-server HTTPS requests could authenticate the datacenter-originating client. The authenticating client would publish the certificate they use at a well-known URL on their website (similar to ads.txt) which would let the web server vouch for the authenticity of the client certificate. This automated discovery process would bring equitable mutual authentication solutions to all ad tech participants.

Google and others have used this approach to authenticate server-to-server integrations for many years, and this approach sounded great on paper. The main problem we encountered when trying to trial this solution with a few of the working group member companies was that TLS client certificates simply did not work well with cloud load balancing, CDN load balancing, and other such solutions. While those products could support TLS mutual authentication (mTLS) natively, they generally didn't provide a solution. Those that did required use of a common root certificate authority to be configured in the frontline TLS termination endpoints. This conflicted with the self-signed certificate strategy we were trying to implement.

Workarounds did exist where we could make this work for most implementers, but the solutions were quite undesirable. One option would be to configure load balancing solutions to perform lower “layer 4” (TCP transport level) load balancing instead of the higher “layer 7” (HTTP application layer) balancing. This would create many undesirable risks, though, including degradation of denial of service protections, connection hotspotting on individual servers, and other general inefficiencies. We worried that any IAB Tech Lab specification whose guidance included “make this intrusive change to your network topology” wouldn’t be well-received or easy to implement by the industry. This prompted exploring other options.

Glossary

Integrator-visible terms

- Signing party
- Verifying party
- ads.cert Call Sign domain
- Operational domain
- Signature message (consisting of a “message” and “signature”)
- ads.cert keys DNS record
- ads.cert delegation DNS record

Terms internal to open source implementation

- Integrator API stub
- Signatory service
- Counterparty manager
- DNS refresh loop

Infrastructure

Core Golang open source software application

As described later, our primary product will be software libraries/servers written in the Go software language. Go provides a controlled process for managing versioning with Go modules, capturing direct and transitive dependency version information in a manifest submitted alongside the application's source code.

Go applications can cross compile onto multiple OSes and CPU architectures (even compiling binaries intended for foreign operating systems/architectures).

We will target a minimum of Go version 1.XX (TODO: determine earliest version).

Containerization tools

Many enterprises deploy applications in containers such as Docker, and they may orchestrate these containers using infrastructure such as Kubernetes. Communication between these containers may be managed by a service mesh such as Envoy.

Other enterprises may not use any of those solutions, instead opting to run applications within a bespoke system administration setup.

The tooling we provide needs to support each of these environments. An integrator should be able to use our software in process within application code running on bare metal servers, and the source repository should include best practice implementations of containerization scripts such as a Dockerfile.

Storage platform

Initially we will focus on building an ephemeral deployment which doesn't persist DNS information. We will fast-follow with an upgrade to a basic file storage solution, primarily used for diagnostic purposes. Finally, to support distributing consistent DNS crawl information to large application clusters, we will support storing DNS responses in a centralized database (targeting future date). Further prototyping and experimentation will be needed to understand and design this latter persistent solution.

Secrets management platform

Various cloud and on-prem solutions exist for securely handling secrets used by applications. To promote security best practices, we will encourage--and make easy--the use of proper secrets management software. In an ideal configuration, an administrative action will generate

and store the private key used in our protocols so that no person (including sysadmin) gains access to this secret. For example, in a cloud environment, it should be possible to configure roles and permissions so that only applications running under a specified role have access to this secret key material. Any departures from these controls should lead to the cloud platform logging auditable events.

Success criteria

Industry adoption

We hope to achieve 100% industry adoption where platforms (e.g. SSAI providers) and S2S integrated publishers utilize this protocol for authentication. Similarly, we hope to achieve 100% adoption from verfiers receiving S2S HTTPS requests where they verify all (or a representative sample) of requests that have security significance.

To accelerate adoption, we plan to work with the Prebid.org organization and provide a Git pull request for a Prebid Server integration that uses our open source API. Any parties currently deploying Prebid Server should be able to quickly adopt this protocol with minimal configuration effort.

Other bespoke client integrations should be relatively quick to bring online due to the lightweight API provided by this OSS library.

Server integrations who receive and need to verify signatures benefit from the flexibility of being able to perform verifications online or offline. With minimal effort, a receiving party can begin logging the information required to verify all--or a sampling of--signatures they receive, performing the verification step in a batch logs processing step.

Reliability and performance

The ads.cert tooling should not introduce reliability or performance degradation within integrating applications. Most importantly, any failure or degraded performance within the ads.cert components must not impact the integrator's continuity of operations.

Security doesn't come completely free, so we must recognize that there will be some amount of latency and data transfer size increase, but we strive to keep it negligible.

DNS infrastructure issues are the largest risk outside of our control. We have designed the protocols to have some resilience to extended DNS outages, especially when deployed with application-level DNS caching. Public key versioning native to the protocols lets parties continue to utilize prior key versions in the event that key rotation isn't successful for a subset of counterparties.

Invalid/unauthorized traffic reduction

Implementing these protocols should result in a reduction of IVT. Fewer resources should be needed to monitor traffic for signs of illegitimate origins, especially in a forensic analyst capacity. These protocols remove the "lowest hanging fruit" opportunities to introduce IVT, instead pushing these activities to other surface areas.

Once fully adopted, ad platforms and advertisers should be comfortably positioned to stop buying ads from data center traffic sources that do not participate in these protocols

It's still the buyer's responsibility to perform their due diligence on the counterparties they buy from, as these protocols only help identify the organization originating the traffic rather than vetting the underlying businesses legitimacy. It only takes five minutes to register an ads.cert Call Sign domain and add the required DNS record to start sending ad requests that use this signature scheme, so buyers should by no means trust traffic simply because it contains validating signatures. Buyers must always build up a trust relationship with the domain signing the ad requests.

False-positive rejection/filtration performance

Once the ads.cert Authenticated Connections protocol enjoys wide adoption, we expect that participants may want to begin making ad buying decisions based on the signal this provides. Participants can deploy automation to measure the signed request signature verification success rate and use this information to inform their risk management systems. It's important to make sure deploying these protocols does not risk incorrectly filtering ad buying opportunities due to a misinterpretation of client signatures: either due to signatures being incorrectly applied by the client, or due to the server not accurately evaluating the signatures provided. A successful system should result in little to no false-positive rejection/filtration of otherwise legitimate traffic. The protocols and implementation should provide fast feedback to genuine participants that some problem is present needing remediation to prevent this overfiltration risk.

Actionable security risk reports

The information security community contains many thousands of security researchers with a keen eye for security vulnerabilities and a strong motivation to discover and disclose these weaknesses. As a computer security protocol, ads.cert must stand up to rigorous scrutiny from security researchers worldwide and be capable of defending against various attack vectors. No system is completely secure, and it takes the combined effort of many creative individuals to root out vulnerabilities in security protocols and mechanisms.

We encourage security researchers to analyze, poke holes, and vet the strength of the ads.cert protocols and corresponding open source software implementation. We ask that researchers follow responsible disclosure procedures, allowing the maintainers to patch vulnerabilities and distribute updates.

Security reports are an interesting success measure for opposing reasons. On one hand, not receiving vulnerability disclosures could mean that the protocols and software stand up to attempted abuses. On the other hand, actually receiving vulnerability reports and then incorporating those findings into the protocols/software demonstrates that we've garnered security researcher interest.

Protocol resource overhead

Factors such as bandwidth, CPU usage, and logging requirements could add costs for implementers that might discourage adoption. Our goal is to not have resource utilization be a material factor in deployment of these solutions.

Ease of implementation

While the core protocol logic is straightforward, it takes a non-trivial amount of time and investment to implement this solution from scratch across every implementer. We have attempted to encapsulate as much detail as possible behind a well-defined and intuitive API so that implementers do not need to spend extensive time to build and debug their solution. Additionally, we hope that this design document will provide useful material for implementers crafting their own design documents when planning their deployment into their own application.

Administration overhead

We want to ensure that it's easy to administer this software, especially given the need to deploy it within organizations of all sizes. For example, a publisher using S2S and deploying ads.cert will not have significant time to invest in tending this software. An advertising platform that discovers counterparty relationships with thousands of domains needs to have an automated solution that requires little-to-no intervention in dealing with exceptions. A successful solution minimizes human intervention where possible, and it provides actionable feedback to quickly address issues in the event they do surface.

Detailed design

The system consists of:

- An API exposed to the integrating application to sign or verify HTTP requests,
- Logic for generating signatures,
- A component for maintaining a list of domains that are signature/verification counterparties,
- A component for maintaining refreshed counterparty DNS records containing public keys,
- An index of the shared secrets calculated from those public keys

These components integrate using different network topologies depending on the size of deployment and data consistency requirements.

Component architecture

The system needs public keys obtained from DNS to function. A signing/verification component uses this data to operate, signing/verifying events provided by the integrating application code.

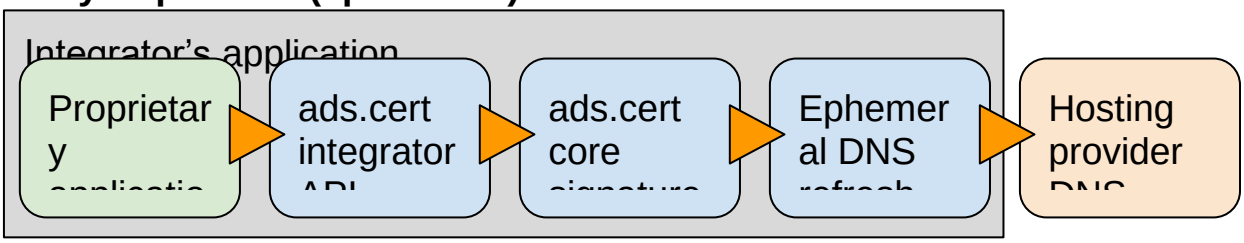
There are three main integration scenarios. All use an identical API from the integrating application code's perspective, but the underlying architecture differs per deployment.

At a high level, this list summarizes the three options for deploying ads.cert within an ad delivery environment, listed from lowest to highest degree of complexity.

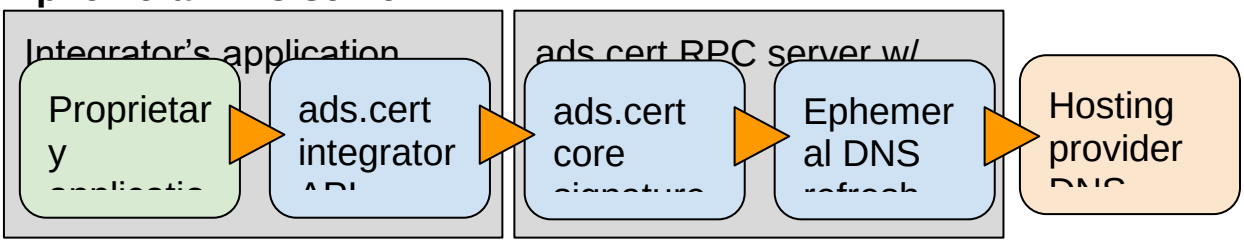
- **Fully in-process (ephemeral):** integrator's application maintains an in-memory cache of counterparty shared secrets derived from DNS.
- **Separated, ephemeral RPC server:** the core ads.cert signing and DNS refresh logic remain separated from the integrator's application, useful for two key reasons:
    - Mismatch between implementation language (Golang) of the core ads.cert signature and DNS logic versus the integrator's application, and
    - Improves security by limiting access to the private key/shared secret material.
- **Post MVP: Regionally/globally centralized, consistent RPC server with persistence:** the most complex but most consistent solution where the system maintains a central, persistent repository of last-known-good DNS lookup information. These processes aggregate counterparty domain lookup requests from many (potentially thousands) of ad server processes, periodically lookup and store results from DNS in a centralized fashion, and push updates to the serving environment. DNS crawl info gets reused across job restarts, potentially creating a more-consistent serving behavior across the fleet of serving jobs.

The separation of these components across applications in each scenario:

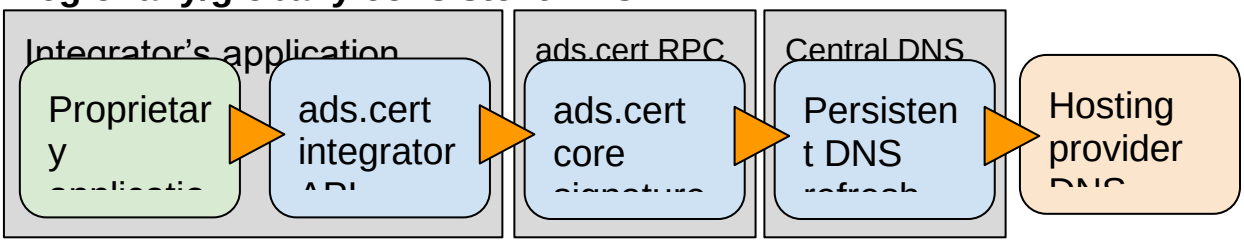
Fully in-process (ephemeral)



Ephemeral RPC server



Regionally/globally consistent RPC

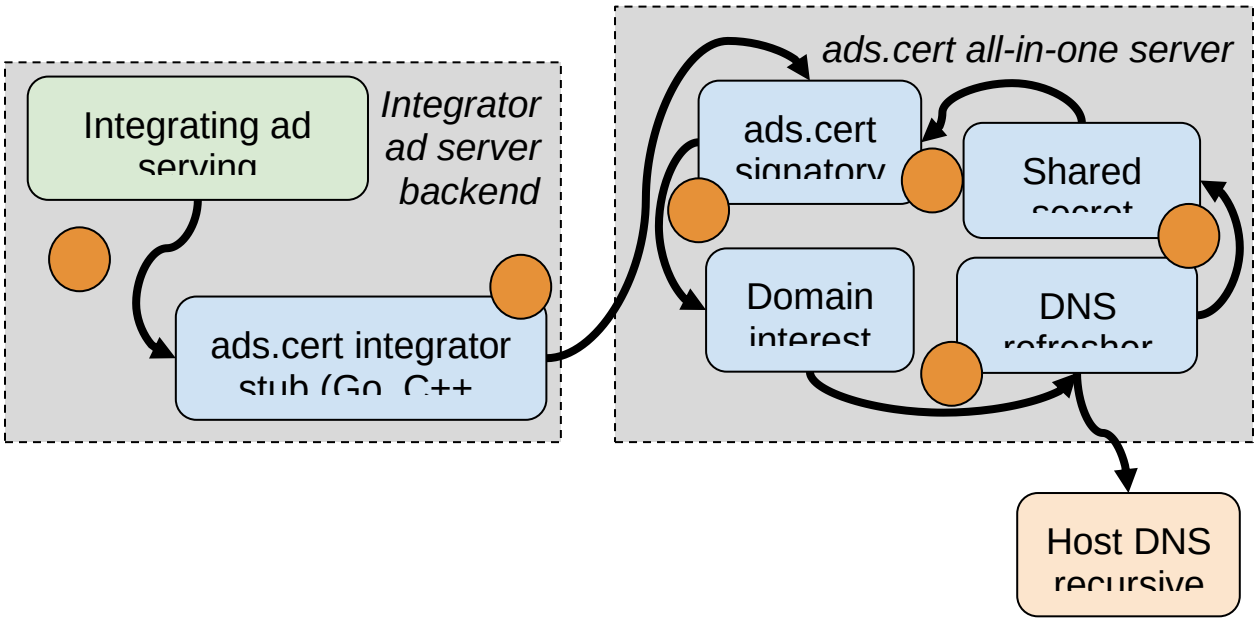


In particular, the RPC server based deployment can operate over a network so that one instance can service many serving processes, or it can operate as a sidecar deployment.

Fully in-process and all-in-one ads.cert server implementation process flow

The ephemeral topologies in option 1 and 2 rely on ongoing ad traffic to prompt each standalone server to re-learn the latest counterparties from DNS based on the requests it receives and processes. Because these operate in an ephemeral mode and cache counterparty shared secret info in memory, all DNS record lookup details get lost between server restarts. Sharing this information across instances and restarts might be possible using a gossip protocol, persistent cache such as Redis, or file-based solution, but we would want to collect details about how the naive implementation behaves before trying to prematurely optimize this.

The process flow works as follows:

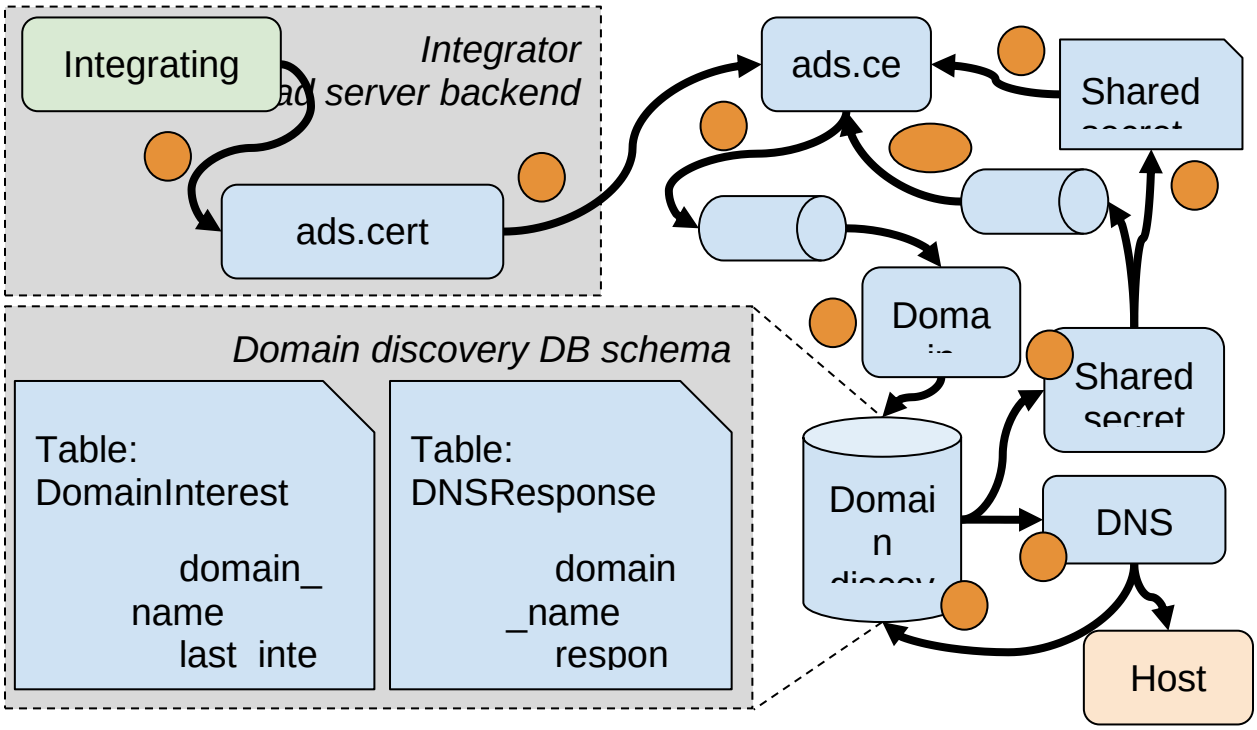


1. Integrating application needs to sign or verify a signature, so it invokes the ads.cert integrator stub API provided for the integrator's respective language
2. Stub creates an RPC request containing a SHA256 hash each of the destination URL and request body and sends over RPC to signatory server (or stays in-process for that variant)
3. Signatory server looks up whether there is any pre-cached shared secret for the counterparty domain
    a. If found, this component embosses (or verifies) the request and returns the result
    b. If NOT found, the service returns a result with the appropriate status and enqueues a domain discovery request
4. Domain interest coordination logic initiates DNS lookup, periodically refreshed by DNS refresher logic
5. Shared secret indexer logic calculates shared secrets for keys obtained from DNS
6. Updated shared secrets get applied to the in-memory shared secret cache used for subsequent requests

Persistent, centralized DNS refresh management option

Post MVP: This added complexity need not be included in the initial MVP implementation, but we should design the code to support adding these enhancements.

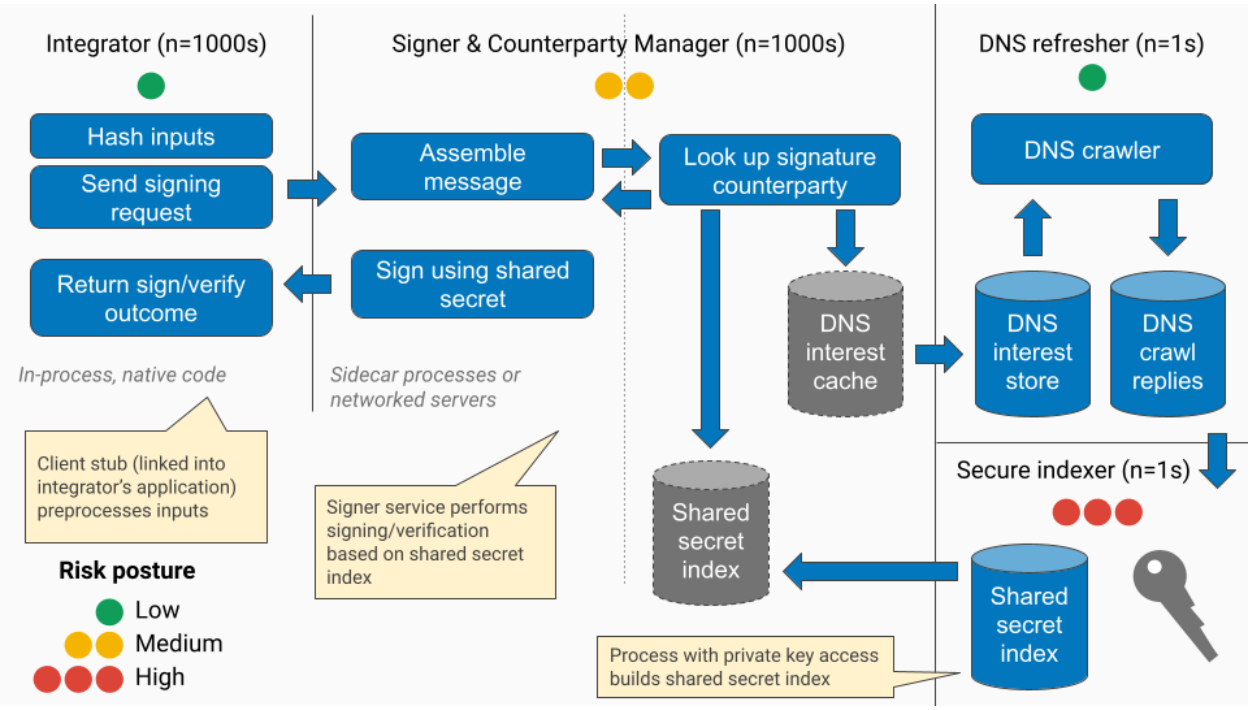
Larger enterprises and integrators requiring a higher degree of reliability may choose to deploy a more-complex architecture that permits consistent, resilient behavior within a global deployment or cloud availability zone. This approach isolates the serving components from infrastructure outages such as local or remote DNS resolver issues by retaining the last-known-good DNS record info within a persistent datastore and pushing updates out to the serving environment in an atomic or eventually consistent manner.



1. The ads.cert signatory RPC server reads a file containing last-known-good shared secret calculations for each domain
2. Integrating application needs sign or verify a signature, so it invokes the ads.cert integrator stub API provided for the integrator's respective language
3. Stub creates an RPC request containing a SHA256 hash each of the destination URL and request body and sends over RPC to signatory server
4. Signatory server looks up whether there is any pre-cached shared secret for the counterparty domain
    a. If found, this component embosses (or verifies) the request and returns the result in the RPC response
    b. If NOT found, the service returns an RPC response with the appropriate status and then forwards the domain discovery request via PubSub queue (noting this in local memory)
5. Domain interest coordinator reads from queue and writes domain interest to DB
6. Event monitoring and periodic polling trigger DNS refresh process which requests latest DNS records from DNS recursive resolver
7. DNS fetch results written to discovery DB
8. Shared secret distributor identifies last-known-good DNS response for each domain, calculates shared secrets using private key, and maintains in-memory index to identify changes
9. Shared secret distributor writes updates over PubSub and periodically updates shared secret snapshot files
10. ads.cert signatory server consumes PubSub updates and indexes in memory

The exact arrangement of this architecture may require prototyping and experimentation.

This diagram may serve as a better representation of the process flow, illustrating a division of the system into four major components with differing risk postures:



The “DNS interest cache” depicted here may, in practice, be better implemented as a logs processing pipeline depending on the integrator’s preferences. If integrators prefer this option, they should log the domain(s) specified in the API response.

Public key distribution

The ads.cert protocols rely on DNS for distributing public key details and pointers from a domain receiving ad queries to the company’s ads.cert Call Sign domain representing their business identity.

DNS record format

An ads.cert key record looks like the following:

```
$ host -t TXT _delivery._adscert.ssai-serving.tk
descriptive text "v=adcrtd k=x25519 h=sha256 p=w8f3160kEklY-nKuxogvn5PsZQLfkWWE0gUq_4JfFm8"
$ host -t TXT _delivery._adscert.exchange-holding-company.ga
descriptive text "v=adcrtd k=x25519 h=sha256 p=bBvfZUTPDGIFiOq-WivBoOEYWM5mA1kaEfpDaoYtfHg"
```

It consists of an “`_delivery._adscert`” subdomain of the ads.cert Call Sign domain: the company’s identity within the ads.cert scheme. The example above shows the records for two

fictitious counterparty companies. Each record contains a boilerplate v/k/h parameter indicating the version, key algorithm, and hash algorithm, followed by one to four public keys listed in order of preference (permitting key rotation), fitting within the 255 byte value size limit for a DNS TXT record.

In addition, ads.cert permits delegation of signing authority using an "ad policy" record (modeled after the sender policy framework, SFP, used for email) that looks like the following:

```
$ host -t TXT _adscert.ad-exchange.tk
descriptive text "v=adpf a=exchange-holding-company.ga"
```

If "ssai-serving.tk" needs to invoke the URL https://ads.ad-exchange.tk/impression?auction=6d8a826b02a2715e44 to record an ad impression, then they would first look up the DNS record for `_adscert.ad-exchange.tk` to find that exchange-holding-company.ga is the signing authority (the ads.cert Call Sign domain) for this entity.

Generating a key

With curve25519, any 32 byte random value can be used as a private key. To be secure, it's only necessary that these values be obtained from a cryptographically secure random number generator and the value remains protected from improper disclosure. As an initial iteration during this project's low-security MVP stance, we'll provide a simple command line tool for generating a random 32 byte key, calculating the corresponding public key using the X25519 library, and emitting these values in base64 encoding to the console.

```
$ go run cmd/keygen/main.go
Randomly generated key pair
Public key:  pi1lCNA0MThJ4tWnCp0BgUKDdyoT1aSXcHzwZYteGTc
Private key: ZzpzEGOe2xCTI6U7zf3mFvdExnHhKsJ2nh5vE20e99o
```

Post MVP: As our process becomes more secure, we will discourage this key management method, instead transitioning to a solution where cloud hosting IAM and production deployment processes limit access to private key material so that it's only accessible to production jobs running within limited roles. In this arrangement, we can utilize utilities that run as these privileged roles that can create, store, and access secrets within the cloud hosting environment's secrets manager solution. Ideally, no human user will have access to the private key material using this model: the interface to this component would only provide the public key that the administrator should publish in DNS.

Securely storing keys

Post MVP: The ads.cert tooling uses a "configuration as code" approach to key management, where the keyring used by the application is meant to be checked into one's source control system in the same way that other system configuration files are managed in a "configuration as code" DevOps workflow. The application uses external key management system (KMS)

infrastructure to securely decrypt the private keys on application startup. For example, the applications needing access to keys can use the KMS to decrypt the private key material by specifying the KMS URI for the “key encryption key” (KEK) (e.g. `gcp-kms://projects/example1/locations/global/keyRings/adscert-kek/cryptoKeys/kek`) and then using the cloud access management controls to limit which roles can request decryption operations that use this key.

Keys are managed by a tool and stored in a JSON configuration file that looks like the following mockup (dates shown for illustrative purposes and do not imply expected key rotation schedules):


```
{
    "domain": "exchange-holding-company.ga",
    "keyset": [
        {
            "key_id": "mPwKMd",
            "public_key": "mPwKMdHt-UZvzYdL5oAYzwxhS5SZHpcj2jhFy_c9zQk",
            "encrypted_private_key": "fpe4_04gwqohjjjOpidicRG.....iIv8M",
            "status": "KEY_STATUS_ARCHIVED",
            "timestamp_created": "2021-03-04",
            "timestamp_activated": "2021-03-07",
            "timestamp_primaried": "2021-03-10",
            "timestamp_secondaried": "2021-03-28",
            "timestamp_archived": "2021-05-04"
        },
        {
            "key_id": "mBVKMG",
            "public_key": "mBVKMGsixPFQRmLu7cjqYqvTmATPfpU_rymUYNTAOlU",
            "encrypted_private_key": "NPF_gZ4ettyH1AcN0TA0yRC.....F5NSE",
            "status": "KEY_STATUS_ACTIVE_SECONDARY",
            "timestamp_created": "2021-03-18",
            "timestamp_activated": "2021-03-27",
            "timestamp_primaried": "2021-03-29",
            "timestamp_secondaried": "2021-04-12"
        },
        {
            "key_id": "z44FDk",
            "public_key": "z44FDkoTru2wab9-WQu0oL3A4IdC-zlQkmd3CA9OQRo",
            "encrypted_private_key": "I6ZNrf7xdvVcPq7Xfb6nCkY.....kSktY",
            "status": "KEY_STATUS_ACTIVE_SECONDARY",
            "timestamp_created": "2021-04-07",
            "timestamp_activated": "2021-04-09",
            "timestamp_primaried": "2021-04-12",
            "timestamp_secondaried": "2021-05-08"
        },
        {
            "key_id": "It3XaO",
```

```
            "public_key": "It3XaOlX99qwHqUGQ_cHuoArmmXUU8oichIBHSJj7BQ",
            "encrypted_private_key": "KKC6SQhCBeI-tmEB5toG1Wc.....Zj_qU",
            "status": "KEY_STATUS_ACTIVE_PRIMARY",
            "timestamp_created": "2021-05-01",
            "timestamp_activated": "2021-05-08",
            "timestamp_primaried": "2021-05-10"
        },
        {
            "key_id": "w2FrwA",
            "public_key": "w2FrwATf0QDwOKk5eUyeCn1YiF4F2dmggvZAjVK0xzQ",
            "encrypted_private_key": "L_OoU7nTp-nAHKvcHf3b3_g.....BHYOc",
            "status": "KEY_STATUS_PUBLISHED",
            "timestamp_created": "2021-05-07",
            "timestamp_activated": "2021-05-11"
        },
        {
            "key_id": "S5nPtY",
            "public_key": "S5nPtYZdI_4inpaBAA7YHDASWFntCq0AfPhYGSDb6Tk",
            "encrypted_private_key": "L745k5BkclNaUuUPx5hSL34.....ly6zQ",
            "status": "KEY_STATUS_UNPUBLISHED",
            "timestamp_created": "2021-05-15"
        }
    ]
}
```


Newer keys are added at the bottom. Older keys get removed from the top.

The ads.cert tooling can read this configuration file and deterministically transform it to corresponding DNS records. While initially the tooling will simply emit the record that a system administrator should copy and paste into their DNS config, later revisions could automate the process of updating DNS based on the state of this file. Thus, the tooling would generate a record with the following 211 byte value (newest keys appearing first):

```
v=adcrtd k=x25519 h=sha256 p=w2FrwATf0QDwOKk5eUyeCn1YiF4F2dmggvZAjVK0xzQ
p=It3XaOlX99qwHqUGQ_cHuoArmmXUU8oichIBHSJj7BQ p=z44FDkoTru2wab9-WQu0oL3A4IdC-
zlQkmd3CA9OQRo p=mBVKMGsixPFQRmLu7cjqYqvTmATPfpU_rymUYNTAOlU
```

Because of application release processes, DNS caching, application caching, and other delays, key rotation must be performed with caution. A party publishing a new key should refrain from using that new key until it is safely picked up by counterparties. Because of this measured rollout requirement, the ads.cert keyring tracks each key through its lifecycle, and the revision history for its changes can be tracked through the implementer's source control system.

The lifecycle of a key is as follows (with corresponding key status):

- Private key newly minted (`KEY_STATUS_NEW`) - Private key written to party's keystore but not yet installed within serving systems
- Private key installed (remains `KEY_STATUS_NEW`) - All serving systems globally have access to the new private key for signing and verification operations
- Public key published (`KEY_STATUS_PUBLISHED`) - New corresponding public key written into the party's DNS record, advising counterparties of its availability for their use
- Public key distributed (`KEY_STATUS_ACTIVE_PRIMARY`) - After ample time has passed for allowing counterparties to pick up and index the new public key, the party may begin ramping up use of the new key for signing operations
- Public key dormant (`KEY_STATUS_ACTIVE_SECONDARY`) - The public key remains published in DNS but appears further down the list of published values
- Public key archived (`KEY_STATUS_ARCHIVED`) - The public key is removed from DNS, although the corresponding private key remains installed within serving systems for continued use in stray signature verification still using the old key
- Private key uninstalled - The private key is fully removed from serving processes so that it can no longer used to verify signatures online, and the key is removed from the keyring configuration file
- Private key destroyed (out of system scope) - Key encryption key destroyed within the implementer's key management system, removing party's ability to recover and retroactively evaluate signatures using that key

Consult with your company's privacy and legal counsel to establish an appropriate key retention policy.

The command line tooling lets an administrator perform key management operations. To protect keys from unauthorized access, the tooling does not give administrators direct access to keys. A key generator tool, deployed into a container environment operating as a privileged role, will generate a new random key, encrypt that key using its privileged access to the KMS, and then return the encrypted private key with corresponding public key so that the pair can be imported into the keyring. The following is a mockup for how this tool should work:

```
$ KMS=gcp-kms://projects/example1/locations/global/keyRings/adscert-kek/cryptoKeys/kek
$ ~/adscert/bin/adscert-keygen --kms_uri=$KMS
Generated key with the ID: S5nPtY

Use the adscert-keyring-tool command to add this secured key to the desired
keyring.

c62MGxeGDIkuCNhbCAQjDOxEoKAjkh8M9t2hsi9cOEkG2CDC-
t5OggQIgFIj9YqKs_pHsnh4hoy0hMMX7I3OkgnD1eHQMyBD556yn5VtQgA9uv63ZxZTlN0k9qNdiyFxEs9zCic
EFCZ9k5WDdplfHkAy4eoxNaDqcV9-zNuhssFw
```

The base64 encoded JSON value[1] contains a plaintext representation of the public key and the corresponding encrypted private key, intended for consumption by the subsequent tooling.

With this value protecting the private key from human prying, the keyring management tool permits importing it into a local keyring copy.

This command creates a new adscert-key-config.json file if one does not already exist at the specified location:

```
$ ~/adscert/bin/adscert-keyring-tool --action=create-keyring \
> --config=adscert-key-config.json --domain=exchange-holding-company.ga
Keyring created for domain exchange-holding-company.ga
Use the import-key action to import the first wrapped key.
```

This command reads the adscert-key-config.json file, modifies the data to include the newly imported key payload, and overwrites the file.

```
$ ~/adscert/bin/adscert-keyring-tool --action=import-key \
> --config=adscert-key-config.json --key_wrapper="c62MGxeGD...zNuhssFw"
Imported key ID S5nPtY into keyring.
```

This command summarizes the keys contained in your keyring, validates the configuration, and outputs the required DNS record that will reflect this configuration in production. It also checks to see if the value found in DNS matches the value required for this configuration. If it does not match, the tool provides additional information about the changes to make.

```
$ ~/adscert/bin/adscert-keyring-tool --action=list-keys \
> --config=adscert-key-config.json
Keys for domain: exchange-holding-company.ga

These are the keys found in your keyring config:

Key ID  ?  Status     Last action
mPwKMd     ARCHIVED   2021-05-04
mBVKMG  *  SECONDARY  2021-04-12
z44FDk  *  SECONDARY  2021-05-08
It3XaO  *  PRIMARY    2021-05-10
w2FrwA  *  PUBLISHED  2021-05-11
S5nPtY     PENDING    2021-05-15

An asterisk (*) indicates which public keys will be added to the generated DNS
record for this configuration, shown below.

----------------------
DNS diagnostics report
----------------------
```

[1] The current mockup is not actually base64 encoded JSON and is instead just random gibberish, so don't try to decode it. :-)

```
DNS update status:

>>> NO MATCH - AWAITING DNS UPDATE <<<

A DNS record was found for the required entry, but the value does not reflect
your current key configuration.

--------------------
Required DNS record
--------------------

Based on this current key configuration, your DNS should contain the following.


DNS TXT record key:

_delivery._adscert_.exchange-holding-company.ga.


DNS TXT record value:

v=adcrtd k=x25519 h=sha256 p=w2FrwATf0QDwOKk5eUyeCn1YiF4F2dmggvZAjVK0xzQ
p=It3XaOlX99qwHqUGQ_cHuoArmmXUU8oichIBHSJj7BQ p=z44FDkoTru2wab9-WQu0oL3A4IdC-
zlQkmd3CA9OQRo p=mBVKMGsixPFQRmLu7cjqYqvTmATPfpU_rymUYNTAOlU


Please update your DNS to contain the specified key and value shown above,
ensuring that the value appears as one continuous line of text.

After applying this change, you may re-run this tool using the same command to
verify that your changes were applied successfully.  Allow sufficient time for
the current DNS record version to expire from caching.
```

Re-running the command to verify DNS updates isn't required, but it may be useful for diagnosing issues.

Other key administration operations:

Set a key to PUBLISHED status:

```
$ ~/adscert/bin/adscert-keyring-tool --action=set-key-published \
> --config=adscert-key-config.json --key_id=S5nPtY
Key ID S5nPtY now has PUBLISHED status.
```

Set a key to PRIMARY status:

```
$ ~/adscert/bin/adscert-keyring-tool --action=set-key-primary \
> --config=adscert-key-config.json --key_id=w2FrwA
Key ID It3XaO transitioned to SECONDARY status.
Key ID w2FrwA now has PRIMARY status.
```

Set a key to ARCHIVED status:

```
$ ~/adscert/bin/adscert-keyring-tool --action=set-key-archived \
> --config=adscert-key-config.json --key_id=mBVKMG
Key ID mBVKMG now has ARCHIVED status.
```

Remove a key from the keyring:

```
$ ~/adscert/bin/adscert-keyring-tool --action=remove-key \
> --config=adscert-key-config.json --key_id=mPwKMd
Key ID mPwKMd has been removed from the keyring.
```

Integrator API

We will provide a lightweight GRPC API client in each common language used within ad serving environments (e.g. Go, C++, Java). The API design will minimize complexity for implementers. Initially, the API will expose two methods:

- Sign Authenticated Connection
- Verify Authenticated Connection

Future iterations of the ads.cert protocols may extend the list of APIs exposed, so we must preserve the ability to add new methods and parameters without causing compile errors or introducing regressions in existing implementations.

APIs will accept/return a request and response data structure rather than raw parameters to permit adding defaulted parameters if needed in the future. This permits upgrades without breaking builds.

For example, the following API accepts two values and returns one, but we can change it to accept/return more values without disrupting existing implementations.

```
type AuthenticatedConnectionsSignatory interface {
      SignAuthenticatedConnection(
            params AuthenticatedConnectionSignatureRequest)
                  (AuthenticatedConnectionSignatureResponse, error)
      ...
}

type AuthenticatedConnectionSignatureRequest struct {
      DestinationURL string
      RequestBody    []byte
}
```

```
type AuthenticatedConnectionSignatureResponse struct {
	SignatureMessages []string
}
```

API initialization and configuration

Integrators must make a call at application startup which initializes the integrator API based on the desired configuration. This can be:

- Local configuration, where the API must know the configured ads.cert Call Sign to declare upon signing/look for upon verification along with the required private key ring
- Remote configuration, where the API needs a connection string to dial the appropriate backend (via TCP, Unix sockets, etc.) which will be separately initialized using the settings above

Some organizations prefer using command line flags to pass in runtime parameters, while others prefer configuration files in YAML, etc. Rather than dictate a specific configuration technique, the ads.cert implementation will provide initialization parameters through a generic data structure, and different pluggable strategies may be used to build it via flags, files, databases, etc.

```
import (
	"flag"
	"github.com/InteractiveAdvertisingBureau/adscert/api/golang/adscert"
)

func main() {
	flag.Parse()
	...
	config := adscert.ConfigureIntegratorFromFlags()
	signer := adscert.NewAuthenticatedConnectionsSigner(config)
	...
}
```

This technique also lets integrators configure the signer in a predictable way for use within testing environments, as the keys, wall clock, and pseudorandom number generator can be seeded with predictable values while still allowing for broader end-to-end code execution.

API utilization

Refer to the Implementer's Guide and examples for details about the intended API usage patterns.

Signing requires providing the destination URL and request body, generating signatures:

```
signature, _ := signatory.SignAuthenticatedConnection(
```

```
      adscert.AuthenticatedConnectionSignatureRequest{
            DestinationURL: destinationURL,
            RequestBody:    []byte{}})

req.Header["X-Ads-Cert-Auth"] = signature.SignatureMessages
```

The verifier receiving the request reconstructs the URL that was invoked by the signer. It passes that URL, the request body, and the signature header to the ads.cert verification API to check for a valid signature. The verifier may then act on that verification outcome accordingly, although the safest initial policy may be to just log the outcome.

```
signatureHeaders := req.Header["X-Ads-Cert-Auth"]
reconstructedURL := …
body, _ := ioutil.ReadAll(req.Body)
verification, _ := signer.VerifyAuthenticatedConnection(
      adscert.AuthenticatedConnectionSignatureParams{
            DestinationURL: reconstructedURL,
            RequestBody:    body,
            SignatureMessageToVerify: signatureHeaders})
```

This concludes the API surface area that integrators will utilize. Subsequent sections summarize implementation details about the underlying open source software.

Calling backend signatory RPC server

Regardless of the operating mode, the integrator shim will access an RPC service interface to request signing and verification operations. This uniform pattern will permit plugging in different implementations into the client shim to support the local-vs-remote topology.

To perform a signing operation, the client shim generates a random nonce, timestamp, and hash of the URL and body (described in depth later).

Signatory RPC service IDL

The signatory service exposes two RPC methods which return operation status and any applicable signatures/verification outcomes.

```
service AdsCertSignatory {
    rpc EmbossSigningPackage(AuthenticatedConnectionSigningRequest)
        returns (AuthenticatedConnectionSigningResponse) {}

    rpc VerifySigningPackage(AuthenticatedConnectionVerifyRequest)
        returns (AuthenticatedConnectionVerifyResponse) {}
}

message AuthenticatedConnectionSigningRequest {
```

```
    SigningContextInfo signing_context_info = 1;
    string timestamp = 2;
    string nonce = 3;
}

message AuthenticatedConnectionSigningResponse {
    repeated SignatureInfo signature_info = 1;

    // TODO: include structured outcome and monitoring feedback.
}

message AuthenticatedConnectionVerifyRequest {
    SigningContextInfo signing_context_info = 1;
    repeated string signature_message = 2;
}

message AuthenticatedConnectionVerifyResponse {
    repeated VerifyInfo verify_info = 1;

    // TODO: include structured outcome and monitoring feedback.
}

message SigningContextInfo {
    string invocation_hostname = 1;
    bytes url_hash = 2;
    bytes body_hash = 3;
}

message SignatureInfo {
    SigningStatus signing_status = 1;
    string signature_message = 2;
}

message VerifyInfo {
    VerifyStatus verify_status = 1;
    VerificationOutcome url_verification_outcome = 2;
    VerificationOutcome body_verification_outcome = 3;
}

enum SigningStatus {
    SIGNING_STATUS_UNDEFINED = 0;
    SIGNING_STATUS_OK = 1;
    SIGNING_STATUS_SIGNATORY_DEACTIVATED = 2;
    SIGNING_STATUS_SIGNATORY_INTERNAL_ERROR = 3;
```

```
    SIGNING_STATUS_MISSING_REQUIRED_PARAMETER = 4;

    SIGNING_STATUS_COUNTERPARTY_NOT_YET_FETCHED = 10;
    SIGNING_STATUS_COUNTERPARTY_NOT_PARTICIPATING = 11;
    SIGNING_STATUS_COUNTERPARTY_DEACTIVATED = 12;
    SIGNING_STATUS_COUNTERPARTY_INVALID_DOMAIN = 13;

    SIGNING_STATUS_INVOCATION_HOST_NOT_YET_FETCHED = 20;
    SIGNING_STATUS_INVOCATION_HOST_NOT_PARTICIPATING = 21;
    SIGNING_STATUS_INVOCATION_HOST_DEACTIVATED = 22;
    SIGNING_STATUS_INVOCATION_HOST_INVALID_DOMAIN = 23;
}

enum VerifyStatus {
    VERIFY_STATUS_UNDEFINED = 0;
    VERIFY_STATUS_OK = 1;
    VERIFY_STATUS_SIGNATORY_DEACTIVATED = 2;
    VERIFY_STATUS_SIGNATORY_INTERNAL_ERROR = 3;
    VERIFY_STATUS_MISSING_REQUIRED_PARAMETER = 4;

    VERIFY_STATUS_COUNTERPARTY_NOT_YET_FETCHED = 10;
    VERIFY_STATUS_COUNTERPARTY_NOT_PARTICIPATING = 11;
    VERIFY_STATUS_COUNTERPARTY_DEACTIVATED = 12;
    VERIFY_STATUS_COUNTERPARTY_INVALID_DOMAIN = 13;
    VERIFY_STATUS_COUNTERPARTY_INCORRECT = 14;
}

enum VerificationOutcome {
    VERIFICATION_OUTCOME_UNDEFINED = 0;
    VERIFICATION_OUTCOME_VERIFIED = 1;
    VERIFICATION_OUTCOME_UNCHECKED = 2;
    VERIFICATION_OUTCOME_SIGNATURE_MISMATCH = 3;
    VERIFICATION_OUTCOME_SIGNATURE_MISSING = 4;
}
```

DNS interest loop

The system must retrieve records from DNS on discovering new interest in a particular domain and on a periodic basis. While DNS records themselves can have a relatively short TTL specified by the authoritative name server, we can set the expectation for the ads.cert standards that implementers should tolerate DNS records being cached for much longer (hours, days) and updates not necessarily available immediately within counterparty ad serving fleets.

Domain discovery goes through the following lifecycle:

1. Signatory logic tries to look up a counterparty domain from the internal counterparty thread-safe map.
    a. If found, the code uses the available info for performing a signing/verification operation.
    b. If not found, the code inserts a stub counterparty entry into the map using a thread-safe operation. It signals the availability of a new domain, consumed by a separate goroutine
2. A DNS refresh loop (awakened by the above signal and by scheduled ticker) begins an update sweep through the list of entries in the counterparty map.
    a. It compares the last DNS fetch time and status for the entry, skipping domains that have been recently fetched or do not need error retries yet.
    b. It performs the DNS fetch when required
    c. It updates the struct with the new counterparty details, applying this into the main counterparty map using an atomic copy-and-swap operation
3. The new DNS information becomes available for future requests using that domain as a counterparty.

In the initial MVP, this DNS interest loop will operate independently on every ephemeral ads.cert deployment instance (in-process or RPC server). Each process will learn from its individual domains exposure which domains need to be looked up and monitored.

To avert a “thundering stampede” of replicas all trying to perform (potentially identical) DNS lookups at the same time (e.g. in a mass server restart scenario), the configuration parameters will provide an option for jittering the timing per instance when DNS lookups occur. This may improve utilization of DNS caching.

Post MVP: To improve consistency across an ad serving fleet, future iterations should centralize DNS crawl so that a consistent picture of DNS information can be pushed out to servers uniformly.

Persistence

To support analysis and diagnostics, the DNS interest loop library will initially support basic file-based persistence and reconstitution so that administrators can snapshot and understand the system’s behavior. This could also be used as a lightweight way to bootstrap a system on startup so that it doesn’t have to rediscover domains and records on its own.

One area to investigate is whether or not leveraging Prebid Cache as a local temporary persistence solution might make sense, as that software currently enjoys broad deployment in environments already using Prebid Server. We would need to make sure that this use of Prebid Cache infrastructure does not add risk or degrade performance for its main workload.

Deriving shared secrets

The ads.cert protocols will use the RFC 7748 X25519 Diffie-Hellman key exchange algorithm to derive a shared secret between the local private key and the counterparty public key.

It's a recommended best practice to process shared secrets through a key derivation function (KDF) rather than use the raw output of the Diffie-Hellman key exchange. We will use the RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) to generate scoped shared secrets for specific operations. This way, we can have one shared secret dedicated to signatures we generate and another for signatures designating us as a recipient. The application-specific "info" value supplied to the HKDF operation will be a SHA256 hash of the string with the format:

```
adscert connection from=<<origin>> to=<<destination>>
```

The "from" and "to" fields indicate the signing domain and recipient domain, respectively. For example, if a party "me.com" signs a message to be sent to "them.com", they would use:

```
adscert connection from=me.com to=them.com
```

Likewise, a counterparty sending a signature to "me.com" for verification would use:

```
adscert connection from=them.com to=me.com
```

In the future, we can extend this scheme to support other ads.cert subprotocols with a different prefix (e.g. "adscert delivery" for the Authenticated Delivery protocol).

This code sample (https://play.golang.org/p/9PTeRl0Za75) provides an end-to-end example of key generation, key exchange, and KDF application.

Signed message construct

A message consists of the following information:

- Domain of the party originating the request
- Originating key alias (first six characters of the base64 encoded public key used to sign)
- Domain component of the URL being invoked on the remote web server
- Domain of the counterparty to this request
- Counterparty key alias
- Second-resolution timestamp
- Random nonce
- Status code (useful for encoding errors)

These values are encoded into a querystring format of key/value pairs.

Additionally, the signed payload implicitly contains a SHA256 hash each of the post body (if applicable) and the URL being invoked.

A cryptographically secure signature is then computed over these details. Two HMACs are constructed over the data, converted to base64, and truncated to arrive at a respective signature value.

A complete signature message looks like the following:

```
from=request-origin.com&from_key=ABCDEF&invoking=destination-service.com&non
ce=aV0V80ofk_Nv&status=OK&timestamp=210429T142944&to=destination-
business.com&to_key=UVWXYZ; sigb=s3rT3pkYou0I&sigu=esa6jS65xSPj
```

(Exact layout/field naming not yet finalized.)

The first signature (sigb) consists of an HMAC over the message concatenated with the body SHA256 hash:

$$bodySignatureBytes = HMAC_{SHA256}(sharedSecret, message \,||\, bodyHash)$$

Note: creating an HMAC which directly included the body content would have been preferable.

$$bodySignatureBytes = HMAC_{SHA256}(sharedSecret, message \,||\, \boldsymbol{body})$$

This avoids hashing a hash (generally not desirable).

The alternative scheme, however, avoids needing to send the entire request payload and URL content to the remote signing server over RPC. Only the 32 byte hash values need to be sent using this scheme. It also helps obfuscate the content of the body and URL being signed from the remote signing server, but it does not fully protect against the server obtaining knowledge about the messages that were signed (e.g. if the signatures are over non-unique messages). This should be an acceptable compromise. The hash itself isn't encoded in the message provided in the HTTP request header, so it is not possible to learn what URL/body was signed from the signed message alone.

The URL signature (sigu) also concatenates the URL SHA256 hash into the message.

$$urlSignatureBytes = HMAC_{SHA256}(sharedSecret, message \,||\, bodyHash \,||\, urlHash)$$

Appending to the prior HMAC construct lets the algorithm reuse the prior hash calculation:

```
h := hmac.New(sha256.New, counterparty.SharedSecret().Secret()[:])

h.Write([]byte(message))
h.Write(bodyHash)
bodyHMAC := h.Sum(nil)

h.Write(urlHash)
urlHMAC := h.Sum(nil)
```

This reuse reduces CPU resource requirements and latency.

Message variations for abnormal operations

Various abnormal operating modes may require reporting a different signature than a standard message. For example, the client may have signing deactivated. Persistent DNS errors may lead to the client using an older counterparty public key. The counterparty DNS record could be malformed. For foreseeable scenarios, the protocol includes a set of status codes reportable with the signature message.

```
enum AuthenticatedConnectionProtocolStatus {
    AUTH_CONNECTION_PROTOCOL_STATUS_UNDEFINED = 0;
    AUTH_CONNECTION_PROTOCOL_STATUS_OK = 1;
    AUTH_CONNECTION_PROTOCOL_STATUS_DEACTIVATED = 2;
    AUTH_CONNECTION_PROTOCOL_STATUS_UNAVAILABLE = 3;
    AUTH_CONNECTION_PROTOCOL_STATUS_TESTING = 4;
    AUTH_CONNECTION_PROTOCOL_STATUS_KEY_FETCH_PENDING = 5;
    AUTH_CONNECTION_PROTOCOL_STATUS_REVIEW_PENDING = 6;
    AUTH_CONNECTION_PROTOCOL_STATUS_DNS_RETURNED_RCODE = 7;
    AUTH_CONNECTION_PROTOCOL_STATUS_ADPF_PARSE_ERROR = 8;
    AUTH_CONNECTION_PROTOCOL_STATUS_ADCRTD_PARSE_ERROR = 9;
    AUTH_CONNECTION_PROTOCOL_STATUS_ADVISORY_ONLY = 10;
    AUTH_CONNECTION_PROTOCOL_STATUS_SUPPRESSED = 11;
    AUTH_CONNECTION_PROTOCOL_STATUS_DELAYED = 12;
}
```

In particular, there will be scenarios where the client has not yet fetched the counterparty's public key from DNS, so no signature can be generated. So that the client uniformly identifies itself to servers, the ads.cert signatory will always generate a message to transmit, but this message may not always be signed if no key material is present.

For example, when invoking a server not yet seen by the client (or across restarts of an ephemeral client instance), the client may report a signature message with a numeric status such as the following:

```
from=request-origin.com&invoking=destination-service.com&status=5
```

This declares (in a non-secure fashion) that ssai-serving.tk has initiated the key fetch process against ad-exchange.tk, but the process has not completed in time to handle this request.

Similarly, there may be scenarios where the client isn't able to get current DNS record details for the counterparty, so the client is currently relying on potentially stale DNS replies for the counterparty key info. The message can be signed as usual, but the client uses this status field to inform the server that there's a reason the key may be out of date. The field can also be used to give the server hints that there may be a persistent issue with the server's own DNS deployment.

The system will provide options for controlling what types of status values get reported to servers to reduce unwanted oversharing of information, if warranted.

Server deployment

Runtime configuration

This system requires the following configuration parameters to be specified by the implementer:

- ads.cert Call Sign domain name: what domain name does the integrator use to represent its organization to other ad tech participants?
- Keyring containing private key material: which private keys does the system intend to actively use for signing requests and verifying signatures?

Additionally, there are administrative settings that will allow for overriding behavior for exception cases:

- Domain blocklist/allowlist: which counterparty domains does the system explicitly ignore for signing-related operations? During experimental ramp-up, which domains does the system permit for selective signing/verification operations?
- DNS record augmentation: manual overrides for public key info obtained from a domain to augment/replace values obtained through normal DNS lookups, used for overriding incorrect behavior
- Counterparty key selection override: manual override to permit use of another (potentially stale) key for signing/verification with a specific counterparty

Identity and Access Management (IAM) integration

To improve security, certain secure operations should only be available to applications running as a specified role and fully inaccessible to individual administrators. Features such as cloud secret management can be isolated in this way. Configuration settings will need to be provided

to inform the application where to look up these secure configuration data. For example, this may be in the form of a cloud key management system resource path.

Application container

The ads.cert signatory RPC server should be wrapped with appropriate containerization options for deployment within a containerized environment (e.g. Kubernetes, Docker). The ephemeral deployments permit individual servers to be brought online and turned down as needed in response to demand change, although these will lack counterparty key information until they see an ad request using that counterparty for the first time after a restart. A persistent last-known-good DNS dataset can be read into the server upon startup, allowing the containerized environment to bring up additional instances without behavior degradation.

Batch operation

To simplify implementation, some deployments may choose to perform signature verification as a centralized batch processing operation over logged signature messages as opposed to attempting online signature verification. Participants may find that this technique provides better reliability and consistency, as DNS resolution gaps can be filled in advance of performing the logs processing step. This method requires logging three pieces of data:

- The received signature message
- Hash of body
- Hash of URL

This provides sufficient information to reconstruct and verify the signatures. Implementers may choose to log all signatures, a uniform sampling, or samplings weighted to problematic segments.

TODO: Evaluate if logging hashes will necessitate salting them with the request nonce and timestamp.

Project information

Code location

All code will be hosted in a repository at https://github.com/IABTechLab/adscert including the server implementation and open source APIs for each target language.

Repository layout

Given the multiple language nature of this project, this is lightly influenced by the unofficial suggestions in https://github.com/golang-standards/project-layout (which is subject to much debate, but various recommendations seem suitable).

- github.com/IABTechLab/adscert
  - api
    - cpp
    - golang
    - java
    - …
  - internal (golang)
    - discovery
    - signatory
    - formats
  - proto
    - discovery
    - signatory
  - cmd (golang)
    - keygen
    - signatory-server
  - docs
  - examples
    - cpp
    - golang
    - java
    - …
  - conformance
    - authconnections

Notable features of the layout:

- Integrators utilize a thin API implementation for their respective application language.
- All implementation within the /internal directory will be restricted from import into packages outside of this project.
- The “discovery” package focuses on fetching DNS records and handling the replies. It needs to support options for storing DNS crawls to a storage system (initially just files).
- The “signatory” package focuses on performing efficient signatures based on an in-memory shared secrets cache.
- The “conformance” package provides a series of format and DNS tests that help ensure consistent handling of the protocol for any parties choosing to build a custom implementation of these libraries (albeit discouraged).

Code review and change management procedures

Upon launch readiness preparations, we will institute mandatory code reviews for all changes. Before that, code reviews will be encouraged but not strictly required.

Open source software dependencies

Because of the sensitive security of any servers or libraries built from this code, we will strive to minimize the number and origins of direct and transitive dependencies imported by the project. Whenever possible, this project will rely upon the core set of libraries published by the Go Maintainers team due to the controlled change management process.

Vulnerability disclosure process

We will establish a process for security researchers to disclose security concerns about the protocols and open source implementation through a formal IAB Tech Lab process. The materials will ask that researchers provide a standard 90 day remediation period before publishing their research.

Measuring improvements

While our libraries may provide basic support for surfacing some of this information to implementers in a structured fashion, we'll be relying on implementers to perform their own analysis based on their proprietary logging and analytics solutions. We encourage implementers to voluntarily report improvements attained to the IAB Tech Lab ads.cert working group so that our team may iterate on the designs and understand industry impact.

Measurable objectives for implementers:

- Reduction in opaque traffic (bid requests, impression pings, etc.) received from cloud and data center IPs
- Ratio of traffic containing fully verifiable signatures, measured by relevant dimensions (request count, advertising spend, etc.)
- Ratio of traffic where signatures exhibit no technical issues (e.g. URL canonicalization issues), thereby minimizing data analyst/ad ops workloads
- Efficiencies from buyers being able to leverage signed server-to-server creative fetches and impression pings as an additional inventory quality signal, including:
    - Advertisers/Agencies
    - Programmatic buyers/DSPs
    - Publishers
    - Sell-side platforms
    - Third-party verification platforms

- Ad spend improvements shared by the aforementioned parties
- Operational improvements that save the aforementioned parties' resources
- Policy changes implemented by demand partners to require server-to-server traffic being authenticated using this or equivalent protocol
- Platforms providing publisher solutions (e.g. server-side ad insertion products) facilitate more ad insertions from legitimate origins, therefore receive more slots filled/auction pressure
- Reduction in publicized incidents of schemes generating invalid traffic at scale

Caveats

All implementers must continue to maintain a defensive posture and watch for invalid activity. As we harden security, abusive behavior will attempt to find weaknesses in defenses and exploit them. These protocols cannot and do not intend to replace vigilant monitoring of advertising activities. We encourage knowledge sharing about novel abuse schemes--especially those designed to circumvent security controls we add--with the IAB Tech Lab working groups who maintain these protocols.

Monitoring

Infrastructure

We will instrument this software with the OpenTelemetry framework which enables collection of telemetry (time-series metrics, logs, and traces) using standardized APIs and integrates into various monitoring platforms.

TODO: validate that OpenTelemetry is ready for adoption with Go, or if the Prometheus APIs are preferred.

Note: monitoring systems expose metrics which can either be expressed as counters or gauges. (Gauges can decrease in value, while counters only increment.) Monitoring systems permit slicing of metrics by attributes. Because the monitoring infrastructure builds an in-memory map containing each attribute tuple seen for the metric, there are limits on attribute cardinality, and some monitoring solutions may implement a hard cap (e.g. 5,000 tuples) that the instrumented application will collect for a metric before it begins discarding values. The outlined metrics appearing below may have the option to activate higher cardinality options, but these will need to be managed with caution, and we will want to have collection of the attribute be configurable (maybe targeting specific domains).

Signatory API operational and performance metrics

- Signing operations
  - sliced by:
    - Signing outcome status
    - Invocation PS+1 domain *(optional, with caution)*
    - Counterparty ads.cert Call Sign domain *(optional, with caution)*
  - measuring:
    - Invocations (counter)
    - Latency *(histogram counter, no slicing)*
- Verification operations
  - sliced by:
    - Verification outcome status
    - Counterparty ads.cert Call Sign domain *(optional, with caution)*
  - measuring:
    - Invocations (counter)
    - Latency *(histogram counter, no slicing)*

DNS refresh loop metrics

- Domain interest list
  - measuring:
    - Number of items (gauge)
- DNS fetch attempts
  - measuring:
    - Number of attempts (counter)
    - Latency *(histogram counter, no slicing)*

Counterparty participation and performance analytics

- Census and query weighted participation by counterparty domain and invocation domain
- Persistent errors/abandonment from participating domains
- DNS record format errors
- DNS latency
- DNSSEC participation
- Change of keying events

Crawl quality

One of the most important aspects of a robust DNS crawl solution is a proper active crawl quality monitoring solution. From one direction, we are collecting an ongoing stream of “demand” to crawl DNS for a particular domain. This “demand” can be satisfied by receiving crawl responses from DNS providing the DNS records located (or lack thereof).

Similar to other protocols such as ads.txt, the crawling system needs to account for transient failures that can occur within some local or remote part of this process. There can be a transient outage somewhere: in the local DNS recursive resolver, in the remote authoritative server, in the network between these. The system needs to be resilient to ignore these transient errors, surfacing them to administrators only if the issue remains persistent. To help administrators prioritize investigations, this crawl quality monitoring should include an indication whether the domain has previously demonstrated participation within the ads.cert scheme. The solution should also include a way to let the integrator mix in impact metrics such as traffic volumes related to each domain. In particular, mixing in signing or verification outcomes into this domain-level data will help measure the current impact to the system. These data will help inform actionable alerting strategies.

Latency

Signing/verification latency objectives

The core effort to lookup counterparty information in a thread-safe fashion, calculate message hashes, and generate the HMAC signature for the message needs to remain as fast as reasonably possible, and we currently target a 10 microsecond latency for these operations. One virtual CPU should be able to sign and verify approximately 100K messages per second.

DNS fetch latency

For initial DNS fetches of a newly discovered domain, fetch latency can be as fast as a normal DNS lookup, with authoritative nameserver responses available within <200ms (faster if results cached).

While we don't deliberately slow down this process, we don't want to build an overreliance on speedy lookups to the point that counterparties reject what is otherwise good ad inventory due to lack of signatures on legitimate requests.

We will generally assume that smaller scale deployments will lean towards the simpler, ephemeral DNS configuration where each serving process maintains its own DNS fetch state. These implementations should benefit from fast (<200ms) DNS fetching that will incorporate those public keys into subsequent ad request signatures/verifications.

Larger deployments (particularly those using centralized DNS fetching solutions) may exhibit substantially longer lookup times for new domain discoveries, as it would be up to the central system--rather than distributed workers--to perform the fetch. On the other hand, these centralized deployments would be persisting the DNS fetch results they receive, so this initial

fetch latency penalty would only be encountered on a domain's first discovery. Subsequent restarts of worker systems would take advantage of the persistent DNS fetch info.

Verifiers need to be tolerant of the fact that these two deployment models will exist within the ecosystem. Some percentage of authentication messages may not contain signatures due to this DNS fetching requirement. We will collect data from implementers to understand the extent that this is a problem and adapt the infrastructure/guidance accordingly.

Application startup-to-healthy latency

We will target a <1s server startup latency. We will implement testing within our continuous integration environment to monitor for performance regressions here and either address those problems or update the SLO if fixing isn't practical.

Scalability

Microbenchmarks

TODO

Load expectations and testing

TODO

Responsiveness to traffic shifts

TODO

Internationalization

User messaging

The core functions of the system do not generate user-facing interfaces, so internationalization concerns for messaging generally aren't applicable.

We don't currently have plans to localize messaging for administrator tooling related to these components, although this localization work could be performed if needed and assists the community.

Domains

Due to the considerable security risks involved with humans trying to interpret extended character set domains, our protocols require use of counterparty ads.cert Call Sign domains within the ASCII character set. Any localized domain participating in this scheme must be expressed in Punycode format as a canonical representation. This applies to DNS record content (already limited to ASCII) and signature messages. Limiting domains to ASCII characters helps keep them accessible to the broadest audience within the advertising ecosystem while protecting from social engineering that's possible through supporting extended characters.

Documentation

As is standard practice with IAB Tech Lab documentation to publish in English, we do not anticipate the need to localize the documentation for this project, but translations could be created if there is demand for this service.

Logging plan

DNS crawl logging

The DNS crawl component will provide the option to generate structured logs based on the crawl results, with fields including:

- DNS query subdomain name
- DNS response
- Latency
- Errors

Auditable security events

Beyond the responsibility of the OSS package but still important, the documentation should suggest implementing a sensible auditable security event policy around sensitive actions, such as:

- KMS administration operations
- KMS encryption operations
- IAM role grant changes related to the KMS service or roles running privileged jobs
- Launch of jobs running under privileged roles

Failure modes

Serving-related failures

Integrator stub failure to connect to RPC signer

We must assume complete or degraded ability to connect to the RPC-based signer process at application startup or during operation.

- Integrator RPC client stub must fail fast with tight latency profile and export monitorable metrics regarding these failures
- Stub must log the failure issue with enough detail and appropriate frequency for the administrator to diagnose the issue
- Where possible, the categorized failure mode should be surfaced as a dimension in the exported client monitoring metric so that fleet-wide consoles can aggregate the reason
- The RPC client should employ appropriate exponential backoff and persistent reconnect attempts
- Because signing and verification are a stateless, deterministic operation, there is the option to send concurrent RPCs to multiple servers if doing so improves reliability and reduces tail latency

RPC signer overload/latency increase

VMs and containers operate on shared resources that are subject to resource contention. This may introduce latency.

Memory exhaustion of indexing quota

To reduce risk of runaway memory utilization by the components that index the shared secrets derived from DNS responses, the OSS index needs to include configurable parameters that limit the quantity of entries that can be inserted.

The OSS needs to export the configured and current quota utilization for each instance as monitoring metrics.

Failure to index updates

Most relevant to centralized DNS crawling: the age of indexed data being used within signing/verification processes must be monitored to ensure that these jobs obtain up-to-date information. We have not yet designed an open source centralized crawling mechanism, so the details here are currently vague. This solution will likely track and export metrics at the production and consumption points regarding the timestamp of indexing data streams.

HTTP request related failures

Web servers typically allocate a fixed buffer to receive HTTP request headers. This can be 8KB-16KB by default on some web servers, but some defaults are as small as 4KB. Additionally, increasing the buffer size could push the web server into using an alternative code path (for example, NGINX sets a distinction for “large” header buffer thresholds, defaulting at 1KB).

When activating the feature for a new domain, clients should monitor the responses returned by the counterparty web server, rolling back the change for that domain if the update results in a reduction of successful requests processed.

Based on feedback and demand for such features, we may be able to include options in the integrator API to facilitate this rollout experiment process and automated restriction per domain.

DNS related failures

This protocol relies heavily on DNS functioning properly (although the same can be said for normal service operations). This summarizes where failures might occur.

Failures can be caused by:

- Authoritative DNS server outage or misconfiguration
- Recursive resolver unavailability
- DNSSEC misconfiguration by one’s own DNS zone or the parent TLD

The systems will be more exposed to this risk when operating in an ephemeral DNS crawl mode. Later, as we provide options to use persistent DNS crawl data, transient DNS outages will have reduced impact.

KMS related failures

We rely on connectivity to the key management system to decrypt private key material. This action occurs on system startup, and it could be triggered again during operation if a dynamic configuration change were pushed. Failure to complete this action will result in the instance not being able to generate signatures.

There are two configuration options to consider for responding to this outage scenario automatically:

- Instance generates unsigned messages indicating "SIGNING_STATUS_SIGNATORY_INTERNAL_ERROR" so that the process continues to process the workload
- Instance reports an "unhealthy" state to the RPC load balancing infrastructure so that the system attempts to shift load to healthy instances that have functioning keys cached.

The KMS could be unavailable for various reasons:

- Failure to connect to KMS upon app startup (KMS service outage; KMS network connectivity issue)
- KMS data loss (infrastructure failure; accidental key deletion through KMS admin console)

As it is by design, humans should not have access to the encrypted private key material nor the keys used to encrypt it. With that said, there may be desire to add redundancy by maintaining a backup encryption of the private key material using a separate key stored in an alternative location (separate cloud provider; on-prem HSM solution; key printed on paper, held in tamper evident sealed envelope and stored in a locked safe/cabinet). Currently we do not have a mechanism for this, but one could be built with a sufficient business case.

An option in the case of a prolonged or permanent outage could be to simply rotate in a new key using a less secure configuration (e.g. private key passed in as command line flag). For signers, the risk in simply rotating in a key is that counterparties may temporarily be unable to verify requests until DNS updates propagate.

The best policy could be to keep one key in DNS "in reserve" and available for use in a "break-glass" scenario. There are four key slots in our DNS protocol, so this is feasible in a key rotation strategy. In the event of an outage, an out-of-band process could be used to obtain and deploy this key within serving, bypassing KMS.

Redundancy and reliability

TODO

Dependency considerations

DNS recursive resolver unavailability

TODO

Risk considerations and mitigations

Impact to DNS recursive resolvers

Because this solution relies heavily on DNS, and because an ad tech operation's processes rely on DNS to function properly for normal ad delivery operations, we must evaluate and ensure that this system's design does not pose a risk to the infrastructure or allocated quotas associated with cloud, on-prem, or external DNS recursive resolver resources. In general, this should not be a problem, as DNS is built to scale and cache at many layers throughout the network. Any counterparties that an advertising client already calls out to for normal HTTP request handling will require DNS lookups to establish those connections.

For proper due diligence, we should assemble details on recursive resolver quotas.

Data integrity

Preventing loss of private key material

The main vector for irrecoverably losing one's private key material is through accidental or deliberate KMS key destruction, performed through the KMS administration interface. Limiting this access may mitigate this risk.

Preventing loss of DNS crawl responses

TODO: revise when we implement persistent DNS crawl

Data retention

DNS crawl responses

DNS responses are based on public data. Retention policies are up to the implementer.

SLA requirements

DNS unavailability tolerances

TODO

Security considerations

Cryptanalysis

The algorithms we use should be quite robust against cryptographic analysis attacks against the public keys and the signature scheme.

The 256-bit keys used by the X25519 elliptic curve algorithm produce a 256-bit shared secret output and provide 128-bit theoretical security.

Application of truncated HMAC SHA-256 provides strong protection of key material, as the truncation action alone destroys significant structure from any oracle that would confirm correct selection of the key.

Signature collision risk

Use of a 12 character signature provides a 9 byte (72-bit) range of values, a technique performed in accordance with NIST recommendations regarding hashing algorithm truncation [1]. For purposes of protecting low-value ad impressions and identifying fraudulent traffic sources, this should be sufficient protection, as the impact of this signature guess would be minimal even if we used a much shorter value. However, if the signature protects something of particularly high value (e.g. used as an access control mechanism for an administrative function or larger value funds transfer), then a shorter signature length might not be sufficient for protecting that use case if the attacker can generate substantial volumes. A 72-bit signature provides a reasonable tradeoff between avoiding values that are short enough to guess with enough tries versus using unnecessary bandwidth to transfer them.

Implementers should limit using these signatures to protect lower value activities such as access control on individual bid requests and identifying potential invalid traffic from purchased ad impressions. Transactions of that nature are what the protocol was designed to protect, where we trade using a more compact signature against having a relatively remote risk of guessing one value correctly. Implementers **SHOULD NOT** use this 72-bit signature scheme for gating access to high value operations such as account access, funds transfer, or facilitating substantial value bid requests: the protocol wasn't designed for that purpose (even though the risk should be relatively low).

Due to the volume of signatures transferred over the course of a day under normal operations, an attacker with full access to all data could try to mine the data for collisions over signatures if there were an algorithm to mine it, and those numbers. Even though there is a high probability that there will be some signature value collision over the set of all signatures received by a service (and thus shouldn't be used as a unique identifier), the HMAC process absorbs the message in computing the value. There would be little to no opportunity for someone to leverage the birthday paradox [2] because there isn't a method for the attacker to map an input to a collision and the birthday paradox focuses on the probability of there being any collision among a set of values. Thus, even with universal knowledge of all signatures ever generated using a given key, we do not risk key wear out since the attacker cannot predict the HMAC output.

The ads.cert Authenticated Connections specification requires that verifiers accept signatures up to the maximum length generated by a 256-bit HMAC (43 character base64-encoded values). Per the specification, verifiers MUST NOT accept signatures shorter than 12 characters.

While the protocol allows signature length extension to gain additional security, this protocol isn't necessarily designed for the purpose of authenticating high-value actions. If a use case surfaces for performing such high-value authentication, then we should evaluate alternative methods, or we should at least consider allocating a separate high-value key namespace so that those actions can remain in a closely monitored and separated security realm. High value authentication operations should use full-length 43-byte signatures.

In-memory secrets protection

This security scheme will only be as secure as the protection of the signer/verifier private key and the artifacts derived from it. To keep a tight latency profile, we want to calculate the shared secret value ephemerally and retain it for the lifetime of the process. Risk exists of secret compromise either by obtaining the private key or derived secrets from memory. There are various risk vectors that could allow this compromise:

- Triggering the application to dump memory contents, including
    - OS level debugging action
    - Snapshotting the container/VM memory state
    - Forcing application to page memory to disk in a location that can be compromised (potentially by exploiting a memory-consuming system weakness)
    - Exploiting a remote vulnerability (e.g. a Heartbleed-style bug) in the application, infrastructure, container, or OS

Prior to finalizing the specification, we need to evaluate whether the static/static shared secret derivation is sufficient for shared secret calculations, or if we should introduce a further

partitioning by using a key derivation function (KDF) [3] applied to the shared secret. Use of a KDF is generally recommended with X25519 [4] when used as an encryption key, although this should be less of a concern when using it as a key for HMAC.

With that said, we may be able to take advantage of a KDF to create some degree of shared secret rotation that's based on the signature timestamp. For example, the key used in the HMAC could be a function:

```
mesage_timestamp = "210531T123456"
salt = substring(message_timestamp, 6) // "210531"
hmac_key = HKDF(salt, X25519(my_private, their_public))
```

In this model, the effective key for the HMAC operation could rotate daily (or hourly, etc) allowing for architectural refinements that can further protect the private key and limit the time period for which a compromised shared secret could be abused. For instance, it could be the responsibility of a separate, higher-security process to hold the only access to the organization's private key, and that system could be used to push time-constrained shared secrets to the backend processes that need them for signing. As long as that high-value process isn't compromised, the compromise of any other component would have a much more limited impact as long as the compromise isn't persistent. A change of this nature would make the code and architecture more complex, though.

Availability risks

An attacker could take actions that try to make the signing infrastructure unavailable for creating/verifying signatures. There are a few potential attack vectors:

- A naive verifier implementation will attempt to look up DNS records for any and all counterparties specified in signature messages. For example, an attacker could present signatures claiming to be from millions of gibberish, generated domain names. This could result in the system exhausting various resources, including:
    - In-memory caching of domain interest lists and calculated shared secrets
    - Hosted or external DNS recursive resolver quota used to resolve the hostnames (potentially creating collateral impact to other applications)
- DoS attempts against a signing/verification counterparty's authoritative DNS servers could result in DNS being unavailable to serve public key details.
    - This may be more of a concern for lightly provisioned DNS servers or zone delegation to an underpowered instance.
- DoS attempts against signing/verification infrastructure by flooding these components with signature requests
    - Individual CPU cores should be able to perform signature generation/verification scalably, but a DoS attack could overwhelm RPC quota/induce load shedding for these components if too many signature-related operations get attempted.

To mitigate DNS DoS risks, verifiers should accumulate and vet the client counterparties they wish to verify based on other complementary signals about the inbound signed traffic. Verifiers need not attempt to immediately verify every new domain they encounter. The ads.cert OSS will provide verifiers with a means of defining this domain allowlist through integrator automated processes and/or manual configuration.

Signers could also be susceptible to DNS DoS, although the surface area should be much smaller. The main vector for compelling a signer to attempt signing against a new counterparty would be through passing URLs via VAST responses (VAST wrapper, creative fetch, impression tracking, etc.) which the signer's HTTP client will try to invoke and thus add domains to the DNS discovery process.

Key compromise risks

The ads.cert OSS encourages maintaining private keys as encrypted values within files submitted to the integrator's source code repository. There are two main risks associated with this approach to mitigate:

- Compromise of the encrypted keys due to weak security
- Attacking the configuration file in its source repository, adding an unauthorized key to the keyring

These risks should be sufficiently mitigated as long as the keys are properly encrypted using modern authenticated encryption, ideally with keys managed by a KMS.

All key encryption flavors (KMS, local) provided by the ads.cert OSS will use authenticated encryption with associated data (AEAD) to simultaneously assure confidentiality and authenticity of the keys maintained in the ads.cert keyring file. Implementation details are up to the specific KMS platform, but generally these will use 256-bit Advanced Encryption Standard (AES-256) keys in Galois Counter Mode (GCM), padded with KMS-internal metadata, as the method for implementing AEAD. This prevents an attacker from crafting arbitrary encrypted key material and having the system attempt to decrypt it for use: the KMS will reject attempts to decrypt a key that fails this AEAD check. This limits attacks on the configuration file source code.

Implementers have the choice of relying on dedicated hardware security module (HSM) backed KMS solutions to protect this key encryption key. These commercially available solutions may already be in use within on-prem/hosted environments, and various cloud providers offer HSM-backed KMS in a pay-per-use model.

Implementers should deploy sensitive applications within confidential computing environments (those which encrypt containers in memory at runtime, such as cloud offerings using AMD EPYC-CPU or Intel SGX), where available.

Network ingress/egress should be appropriately firewalled.

RPC server signer deployment risks

The RPC server needs to provide adequate protection against unauthorized invocation. While less severe than outright key compromise, being able to generate arbitrary signatures for unauthorized traffic could be an insider risk. The RPC mechanism will natively support its own authentication method that the deployment needs to use to provide access control.

The integration must provide adequate protection against man-in-the-middle attacks between the system requesting verifications and the verifier RPC backend. This MitM could occur within a network component or within the integrating application (e.g. replacing the RPC client software with one that marks malicious traffic as verified). Mutual authentication will help address this concern from the network risk perspective.

DNS integrity risks

A party's public key must survive intact between the authorized system administrator's original intent and the counterparty using that information for authentication. Places where this could be violated:

- Compromise of the authoritative DNS server's configuration (e.g. introduction of attacker-controlled records)
- DNS spoofing/cache poisoning attacks

Change control procedures and system security for an organization's DNS configuration affect much more than what's in scope for the ads.cert protocols, so we must delegate those concerns to the organization as part of their broader security posture.

As previously discussed, implementers have the option to use DNSSEC as a mechanism to further protect their DNS-distributed public keys from unauthorized modification.

Message attack risks

An attacker could attempt to compromise the system using various malformed messages required by the underlying protocols. We have three message formats specified:

- The signature message
- The key distribution DNS TXT record
- The signing authority delegation DNS TXT record

Parameter Pollution

HTTP Parameter Pollution [5] is a method that can be used to confuse parsing of parameterized key/value pair messages. By including duplicate keys within the message, systems that rely on multiple message parsing implementations--particularly when multiple software libraries, languages, or software packages attempt to parse the same message and could arrive at different interpretations.

Except where noted, the ads.cert protocol parsers expect a fixed set of parameters in the protocol messages. The OSS includes unit tests covering these validation checks.

To permit protocol enhancements over time, parsers will not reject parameters unknown to their programming.

Server-side request forgery (SSRF) risks

Web server applications that make their own outbound HTTP requests can be vulnerable to server-side request forgery (SSRF): a scenario where an attacker provides the application with a URL to invoke that goes against the application's intended use. The ads.cert protocols don't address SSRF risks on their own, but they do have hardening that helps assure the verifier that URLs aren't being invoked under certain exploit circumstances. By requiring that the signed message include a signature over the entire URL string that the client is trying to invoke, the verifier receiving the request can be better assured that the signature has been solicited under a constrained set of expectations rather than under duress.

Open source supply chain attack risks

Being an open source Go project, we will be relying on other Go modules as dependencies within the build process. These dependencies could introduce security vulnerabilities by accident or deliberately.

Surface areas include:

- Core key management components (signer/verifier; DNS fetch process)
- Language-specific integrator API
    - Primary Golang implementation
    - Adapters for other languages

Potential risks include:

- Exfiltration of key material
    - Via network protocols such as HTTP/HTTPS; DNS; raw TCP/UDP, etc.
    - Via encoding into signature messages

- Infiltration of bogus public key material to trick verification (include a public key controlled by attacker as trusted)
- False reporting of signatures from a specific source as being “valid” (introduced either in the core signing components or the language-specific integrator API)
- Other nefarious actions unrelated to ads.cert

To minimize this risk, we will tightly constrain the direct and indirect dependencies that the open source software introduces.

Risks could exist for introducing compromises via independent open source dependencies not introduced by our OSS supply chain. For example, a language that allows hot-swapping implementations of a library at runtime could replace the integrator API with a wrapper that always approves certain bogus signatures.

Insider risks

An insider with privileged access to systems could use this access maliciously. Risks include:

- Exfiltration of private key material, letting the attacker generate valid signatures for improper traffic
- Modification to system configuration, including key configurations, DNS records, counterparty authentication administrative override policies
- Modification of IAM policies to permit unauthorized KMS access for decrypt or encrypt operations
- Introduction of code that modifies ads.cert component behaviors

Much of these concerns are outside the scope of our open source software, but we can advise on practices to mitigate these risks, including:

- Role-based access controls to sensitive operations
    - Configuring to prohibit administrator direct access to KMS encrypt/decrypt functions
- Limiting deployment to builds created from a protected workflow, including:
    - Built from code submitted to organization’s official source code version control system
    - Code review process
    - Passing test automation and configuration constraint verifications

Cryptographic agility

As time progresses, security researchers tend to find weaknesses in existing cryptographic algorithms and develop new ones which resolve those weaknesses. While the algorithms we've chosen are widely utilized industry standard solutions, we must anticipate the need to upgrade algorithms in the future.

We do not need a specific strategy for this at the moment beyond outlining a few reasonable approaches. A few options:

- Try to pack additional algorithms into the same `_delivery._adscert.<<call_sign_domain>>` DNS record, knowing that we will have to modify the implementation to concatenate multiple 255 byte values.
- Simply use a separate DNS name when we need to update to a different crypto scheme, e.g. `_deliveryv2._adscert.<<call_sign_domain>>` or `_deliveryx448._adscert.<<call_sign_domain>>`

Spam and abuse considerations

HTTP request header spam

As mentioned in availability risks, the signature HTTP request headers create a considerable abuse surface area if not carefully handled by verifiers. A naïve implementation could lead to the system attempting to excessively look up DNS records.

Invalid signature spam

A malicious party could attempt to undermine the credibility of an honest party by flooding the verifier with:

- Signatures which do not validate correctly
- Messages missing a corresponding signature, such as those falsely reporting an error status within the message

One mitigation for this issue could be to evaluate IP address reputation. Presumably we would expect a server to receive a consistent stream of events from an individual client IP address that either all/mostly contain valid signatures, or none contain valid signatures at all. Establishing IP address reputation info may help a verifier wave through requests that present abnormal status codes but are otherwise interleaved with valid requests. Implementing this falls outside the responsibility of the ads.cert OSS package, but we can advise implementers about this risk and potential strategies to handle it.

Privacy considerations

Avoiding public key signing over consumer activity

Using symmetric keys for signing means that signers aren't generating a public key signature cryptographic record of consumer activity.

Public key cryptographic signature artifacts are an issue with the DKIM protocols used to sign email. In its current form, DKIM encourages attackers to compromise mailboxes so that they can obtain long-life evidence that a particular email provider originated a specific email from the sender claimed in the email message. It need not even be the sender's mailbox that gets compromised: any party receiving the email can attempt to leverage the DKIM signature as evidence of authenticity. This limitation in the DKIM standard is not one that we want to emulate.

Because the ads.cert protocols use symmetric key signatures, we gain two protections against this type of attack.

- Only parties to the signature are able to make any use of the signature; otherwise, the signature cannot be distinguished from a randomly-generated number
- Either the sender or receiver could have generated the signature associated with a message, creating plausible deniability about the message authenticity if attempting to use it within a third party context

Signatures are further tunneled within HTTPS connections using TLS, so they are not available to a man-in-the-middle observation of the transmitted signature without some form of TLS compromise.

Preventing using ads.cert for B2B non-repudiation

Using symmetric keys for signing protects participants from signatures being used as non-repudiation artifacts. The ads.cert protocols aren't designed to be used--nor are they intended--as proof of events that can be subsequently shared with another party as evidence of some business-to-business transaction or activity. We explicitly designed this protocol as an authentication and message integrity mechanism so that one party can attest its identity to another. Both parties possess the shared, symmetric secret, meaning that both parties would be in a position to "forge" a signature over an arbitrary message. This trapdoor undermines being able to present a signature to a third party as proof that a specific party originated the activity, as either party could have crafted the message.

Logging of URL and body hashes

Offline verification of signatures will require logging of the full signature message and the body+URL hashes. These hashes will provide some degree of information about the body/URL invoked. For a body or URL values containing few or predictable values, these hashes will provide information that can reconstruct the input behind the event so long as the input can be guessed and compared against the logged hash. A privacy attack against this scheme could thus attempt to build a rainbow table over possible inputs to the SHA-256 hashing algorithm. Mitigations that could attempt to make the rainbow table approach marginally more difficult could require updating the protocol to salt the hashes with some variable input (e.g. the timestamp and nonce present in the message). Adding this enhancement only makes this analysis technique more computationally expensive than computing a single rainbow table that could interpret a wide range of predictable inputs.

The best protection against this attack is the addition of entropy into the URL or body. This entropy could occur from natural, existing sources, such as impression tracking URLs containing auction identifiers or encrypted strings. To reduce risk of preimage attacks against the hashes (e.g. through the use of a length extension attack), implementers should avoid reliance on techniques that could allow an attacker to include arbitrary data with the goal of obtaining a hash collision. (Obtaining a hash collision, while something that has been accomplished with weaker algorithms such as SHA-1 or MD5, is an expensive effort and likely not a practical risk for implementations of our protocol. No SHA-256 hash collisions are known to have been generated.)

Product inclusion and equality

Technology accessibility to businesses

It's important that we make this technology accessible to businesses of all sizes. Just as individuals operating their own website can enjoy protection of their visitor traffic using HTTPS over TLS and free certificates issued by authorities such as Let's Encrypt, that same individual should have the ability to utilize ads.cert protocols for signing or verification while growing a new publishing or ad technology business. We've attempted to make the adoption of ads.cert protocols require minimal time, effort, and hosting resources. Materials such as the open source library and these documents help all participants achieve a consistent deployment using open security techniques available for inspection and feedback from the community.

Administrative controls

TODO

Documentation

User's guide

TODO: Publish using readthedocs.org or gitbook.com as an automated formatting solution?

Testing plan

Load testing environment

TODO

OSS CI/CD environment

We will use the IAB Tech Lab's subscription to the Travis CI hosted CI/CD platform as a means for presubmit testing of pull requests.

Test bid request traffic environment

TODO

Hosted compatibility testing solution

TODO

Work estimates

TODO

Launch plans

Publication of open source software suite

IAB Tech Lab would launch this product by publishing these documents, the open source software suite, and an appropriate user's guide.

Any implementers incorporating this product will have the option to start generating unsigned HTTP request headers informing the servers they invoke of their claimed identity and

participation in this standard. This will make those counterparties aware that they can start receiving requests with signatures once they start publishing their public key in DNS.

Integration into Prebid Server

To facilitate rapid adoption, we propose that the OSS implementation be integrated into Prebid Server so that any server-to-server requests being sent from that platform can be decorated with ads.cert signatures. A draft of the changes required to add this integration can be found here, although a small amount of additional work would be required to add conditional logic for activating this functionality via configuration files.

Software release process

To facilitate deployment, we will configure the CI/CD environment to generate container snapshots on an ongoing basis, labeled as a “nightly” or “development” build label. Upon review and vetting with the team, we will designate an appropriate snapshot from this process as a versioned release.

(TODO: figure out how to label these releases with versions so that parties building from source can upgrade to those numbered releases.)

Rollback/degradation/safe mode strategy

TODO

Alternatives considered

Key distribution protocols

Generally speaking, we wanted to avoid distributing public keys from HTTP endpoint well-known URLs due to the complexity that crawling these URLs introduces in practice. We have extensive (negative) experience with crawling ads.txt, app-ads.txt, and sellers.json IAB Tech Lab standards formats and find that there are various reoccurring issues that we want to avoid entirely by leveraging DNS. For example, it’s very common that a corporation’s root domain configuration (e.g. example-company.com) will resolve to IP addresses operated by their website content delivery network. The files that they publish often have to be hosted on a redirect location and can result in multiple crawler redirects. We encounter misconfigurations, geographic restrictions, HTTP downgrades, robots.txt crawl restrictions, and a whole list of other issues that we won’t encounter by avoiding HTTP-based services entirely.

DNS is the original distributed, cached, scalable key/value store, and its organization lends to delegating all of the `_adscert` subdomain activities to a separate DNS zone if desired. The key algorithms we use are compact and let us distribute four public keys within one 255-byte DNS TXT record. X25519 is a highly-specified standard that leaves no room for introducing additional parameters such as the base point: all are defined by the RFC.

JSON Web Keys (JWK)

JWK was designed to be a general envelope format for storing and sharing key material, with attributes annotated using JSON markup. The protocol can contain public keys, private keys, and symmetric keys. The protocol can specify well-known algorithms outlined in the RFC 7518 JSON Web Algorithms (JWA) [6] spec. In some circumstances, having this degree of flexibility gives options for changing cryptographic algorithms over time: a good goal for an Internet standard. This modular approach allows adding new algorithms and deprecating old ones.

Flexible standards like this can, however, introduce security risks. Typically one manages the complexity of working with such a protocol by introducing libraries that can parse the message into the key material needed for the application. These libraries become a surface area, either by misconfiguration or by malicious message attack.

JWK doesn't lend well to distribution using DNS (something we want to reduce key crawling moving parts).

X.509 certificates

We initially explored and prototyped using X.509 certificates as a means for distributing public keys via well-known URIs on websites. The rationale here is that X.509 is a widely adopted standard that predates the Internet and has support within a wide range of tools.

X.509 suffers from many of the same problems as JWK. In addition to having a large number of parameters for defining the key, there are a litany of additional metadata fields and extensions that are immensely confusing to understand and use correctly. In addition, certificates are encoded in a binary format (ASN.1) that requires special tools to interpret, and none of the default options for interpreting this are great (e.g. OpenSSL command line tools aren't very informative, as they present an elided view). X.509 relies upon either using a library to parse the encoded data into fluent constructs, or it requires parsing the format using low-level ASN.1 parsing libraries. Both are quite painful to work with. These are details that the OSS implementation could hide from the integrator, but it just doesn't seem worth the hassle to use this protocol when we are no longer pursuing a mutual TLS protocol for client authentication which would have required X.509 certificates.

PKCS/PKIX standardized formats

Falling within the same category of X.509 complaints, there's an option to use the core ASN.1 encoding formats for the public key alone. Again, this is painful to parse, and it increases the encoded message size.

Signature protocols

JSON Web Tokens (JWT)

JWT is a widely used standard for authentication mechanisms across a variety of applications, and we initially investigated using this protocol. Various off-the-shelf open source software packages can be used to generate, sign, and verify authentication tokens using this protocol: a valuable proposition for a set of protocols that would need to be ported to different environments.

JWT suffers from many of the same problems as JWK, though. Because it intends to be a generalized solution for many cryptographic primitives, it creates a large surface area for attacking weaknesses in handling the protocol. The most widely known problems with JWT stem from the fact that "none" is an option that can be specified for the signature algorithm! (This has caused so many security compromises in OAuth and other integrations that a security researcher by the handle "zofrex" created https://www.howmanydayssinceajwtalgnonevuln.com/ to poke fun at the fact that JWT "alg=none" security vulnerabilities are so common.)

In a blog post [7], "zofrex" provides a succinct description of the protocol:

> You don't need to know everything about JWTs to follow along here, but you do need to know this much: A JWT consists of three parts: a header, a payload, and an optional signature. Here's an example:
>
> ```
> eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6IjEifQ.eyJuYW1lIjoiem9mcmV4IiwibWVzc2FnZSI6ImhlbGxvLCB3b3JsZCEifQ.40IhxsQj3rVqfQSpdzZTroSs1onXrTJUsy6yC1SadK0
> ```
>
> Each part is Base64 encoded, which isn't important here. Once decoded, it looks like this:
>
> ```
> {"alg":"HS256","typ":"JWT","kid":"1"}.{"name":"zofrex","message":"hello, world!"}.<inscrutable cryptographic soup>
> ```
>
> The header tells whoever is verifying the JWT which cryptographic algorithm was used to sign it (alg), and optionally an identifier for the key it was signed with (kid).

For application developers and administrators, JWT presents a significant usability issue. JWTs must be processed by a decoder to obtain any information about the encoded authentication message (human base64 fluency notwithstanding). This creates an additional impediment to quickly isolate the relevant signatures and diagnose problems without processing the events using specialized tooling.

Simply avoiding the JWT protocol is preferred since we can craft a solution that optimizes for brevity (important when being sent on bid requests) and contains the attributes relevant to our problem space. With a customized protocol, we have more flexibility to specify constraints appropriate for a protocol that focuses on combating invalid traffic.

TLS Mutual Authentication

While this would have been an ideal solution in many ways, as discussed earlier in prior designs, the main problem with trying to use mTLS is the fact that we would very much be pushing the boundaries of what is a normal, typical use of the protocol. Doing so introduces risks, summarized here:

- Significant risk in creating consumer-visible authentication prompt if the hostname for the mTLS endpoint were ever exposed to end-user devices (incidents that have happened before in ad tech)
- Risks introduced by requiring network topology changes to work around limitations in HTTPS load balancing
    - Requires introduction of dedicated mTLS termination solution past load balancing
    - Requires deploying reverse proxy instances (e.g. NGINX) which will increase security surface area
    - Potentially bypasses existing denial of service protection mechanisms
- TLS client certificates get passed in plain text for protocols prior to TLS 1.3

If we were able to successfully standardize around mTLS (without creating a supportability nightmare), the protocol would have given us a well-utilized foundation built off of the widely deployed TLS standards. We would gain many of the benefits achieved in the current ads.cert protocol, such as avoiding creating public key signatures over communications.

One downside of the mTLS security model is the fact that clients have a passive role in negotiating authentication to web servers. Forms of SSRF attacks exist where a client could be “tricked” into signing a URL invocation that’s unrelated to the ad impression being delivered. Standard HTTP client libraries simply present the client certificate to any subsequent URL redirect from an initial invocation. By instead putting the signing operations more directly in control of the application logic triggering URL invocations, choosing which events to sign becomes a much more deliberate action.

TLS client authentication may be vulnerable to notary based privacy attacks (although this is a speculative concern, as no evidence was found thus far regarding attempts to notarize client communications: only server). The now obsolete TLS 1.0 and 1.1 protocol versions were susceptible to having a "witness" system escrow a portion of the key exchange material as a means of "proving" that a website served some content. The now-defunct TLSnotary [8] project is one example of this method. Newer projects such as TLS-N [9] and PADVA [10] claim to support TLS 1.3, with both including a public blockchain verification component.

Ring signatures

Ring signatures [11] let an individual generate a signature associated with a group of public keys where the signature can be verified as originating from one of the keys within the group, but it's not possible to identify which private key was used to generate the signature.

It would be possible to maintain our repudiation requirement by implementing signatures using this scheme, but we would not be able to retain our signature confidentiality requirements. Anyone with knowledge of the participants' public keys would be able to verify that at least one of the parties signed the message, but we don't want this feature given our requirements. Anyone who doesn't possess the shared secret value cannot verify HMAC-based signatures, and this behavior is explicitly what we want. A truncated HMAC signature requires substantially fewer bytes to transfer and less computational resources to calculate.

References

[1] Quynh Dang. 2012. NIST Special Publication 800--107 Revision 1: Recommendation for Applications Using Approved Hash Algorithms. https://csrc.nist.gov/publications/detail/sp/800-107/rev-1/final

[2] Birthday Paradox, via https://en.wikipedia.org/wiki/Birthday_problem

[3] Key derivation function, via https://en.wikipedia.org/wiki/Key_derivation_function

[4] RFC 8418: Use of the Elliptic Curve Diffie-Hellman Key Agreement Algorithm with X25519 and X448 in the Cryptographic Message Syntax (CMS). https://datatracker.ietf.org/doc/html/rfc8418#section-2.2

[5] HTTP Parameter Pollution, via https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/04-Testing_for_HTTP_Parameter_Pollution

[6] RFC 7518: JSON Web Algorithms (JWA), via https://datatracker.ietf.org/doc/html/rfc7518

[7] zofrex. 2020-10-20. How I Found An alg=none JWT Vulnerability in the NHS Contact Tracing App. https://www.zofrex.com/blog/2020/10/20/alg-none-jwt-nhs-contact-tracing-app/

[8] “TLSnotary - a mechanism for independently audited https sessions” via https://tlsnotary.org/TLSNotary.pdf

[9] H. Ritzdorf, “TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing for Disintermediation”, 2018 Network and Distributed System Security Symposium (NDSS), 2018. via https://spiral.imperial.ac.uk/bitstream/10044/1/85569/2/ndss2018ritzdorf.pdf

[10] P. Szalachowski, "PADVA: A Blockchain-Based TLS Notary Service," 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS), 2019, pp. 836-843, doi: 10.1109/ICPADS47876.2019.00124. via https://ieeexplore.ieee.org/document/8975811

[11] Ring signature, via https://en.wikipedia.org/wiki/Ring_signature