Secure Production Identity Framework For Everyone

Joe Beda

Written: March 2, 2016
Last Edit: August 1, 2016

This document lives at spiffe.io. As the project matures, more will be hosted at that site beyond this design document. Feel free to follow @SPIFFEio on twitter or join the (nascent) mailing list spiffe-dev.

This is a work in progress draft. I'd love discussion and comments. If you want to comment please reach out to joe@spiffe.io and I'll give you permission.

For a softer introduction that sets some context, there is a slide deck with speaker notes from my presentation at GlueCon 2016.

Introduction

SPIFFE proposes a new set of protocols and conventions to securely communicate and identify service to service communication in modern production environments.

Modern production environments are quickly moving to a world that is much more dynamic than existing IT systems. Specifically, workloads are being dynamically placed and scaled (often via container orchestration). At the same time, parts of applications are being broken out as micro-services and being reused widely across larger organizations. Security systems and processes need to adapt to this more fluid environment.

One common approach to securing communication between services is being described as "micro-segmentation". Micro-segmentation consists of programming the network to only allow traffic between specific endpoints as a matter of policy. Eventually it is expected that as part of describing and running a service, the user will describe which other endpoints that service expects to both receive and send network traffic to. This approach is valuable but also very limited. The lack of context at the packet and firewall level not only makes it a blunt instrument but is also limited to just the security domain. If one relies on just micro-segmentation, it implies that "reachability is authorization".

Right now, many folks see micro-services as a way to break large monolithic applications down into more manageable parts. But as the architectures mature, we will see micro-services being used widely from multiple other services. This can create a network of connected services that spans an organization. As this happens, the protections of micro-segmentation decline. A first class identity system becomes necessary.

Another way to look at this: just as DevOps has redefined the lines between developers and operations roles, so too must we redefine the roles between application developers and security operations. While DevOps has many many definitions, it is clear that developers are now expected to understand and play a role in how applications are deployed and managed in production. Similarly, operations teams must be more aware of the applications that are being managed. As we move to a more evolved security stance, **we must offer better tools to application developers so they can play an active role in building securable applications**.[1]

Here are the qualities that we will look for in a such a system:

- **Easy to use.** Cryptography can be very difficult to use and even more difficult to use correctly. Above all, we must fix that.
- **Wide Integration.** The more frameworks and tools that build in support for SPIFFE the more useful it will be. Built in support in RPC/micro-service frameworks, storage systems and orchestrators will multiply its usefulness. There are parallels to the OpenTracing project.
- **Secure.** Modern proven cryptographic systems should be leveraged to verify identity of both clients and servers.
- **Identity and privacy.** Users should be able to use the system for both identity and privacy of communication. There are different costs for each of these and we should be open to exploring trading off costs.
- **Reliable.** The single points of failures in the system should be minimized and the system should degrade gracefully when any SPOF is down. All "steady state" operations shouldn't have requirements off of a specific node. This precludes any centralized signing/verification service.
- **Flexible Bootstrap.** Identity is about trust and that trust needs to be bootstrapped. The system should have multiple mechanisms including being able to hook into hardware security mechanisms (such as TPMs) as that makes sense..
- **Scoped trust roots.** While PKI may used as part of the solution, there should be no hardcoded, global trust roots as we see in the web browser world. It is up to users to explicitly decide which roots they will trust for which identities. Typically, these roots will be set per organization.
- **Federatable.** It should be possible to use these identity mechanisms across organizations. However, federation trust will have to be explicitly configured. The nitty gritty of how to establish this trust may be out of scope for the first cut of these specifications.
- **Legacy Friendly.** Legacy is perhaps not the right word here. Regardless, users should be able to apply SPIFFE without rewriting applications or moving them to a new management system. It should be possible to adapt this system as separable infrastructure to secure existing systems with little or no code change.

[1] It would be great to come up with a name for evolving the standard relationship between security teams and the rest of the IT org. It parallels the evolution embodied with "DevOps". Perhaps "SecDevOps"?

- **Container Friendly.** The system should work well in dynamically orchestrated containerized environments.
- **Minimal Knowledge.** A compromised machine should only expose any secrets for workloads that happen to be running on that machine.
- **Optional/Future** qualities that should be taken into consideration but may perhaps have to be secondary to other considerations:
    - **Delegatable.** Things are never as simple as A talking to B. There are often other processes and proxies in between. We should allow for and explore ways to enable identity to be layered in interesting ways.
    - **Scopeable.** There is lots to like about a capabilities based authorizations system. But as SPIFFE is just about identity and authentication. We should explore ways to have intermediaries scope down what identification tokens can be used for. This goes hand in hand with delegation.

This document lays out a sketch of how we can start to address this problem. There are plenty of open questions and decisions to be made. Hopefully this sketch will light the path to a more rigorous set of conventions/standards and reference implementations.

There are certain things that are explicitly out of scope for the initial version of SPIFFE. This includes:

- **Authorization.** SPIFFE is about identity. It is up to individual programs to decide how to grant access to resources based on that identity.
- **Directory Services.** Things like grouping and metadata about identities are out of scope. We don't want to reinvent AD or LDAP. However, users can pick a scheme for their SPIFFE IDs (defined below) so that they can use SPIFFE identity as a key to looking up information in a directory.

SPIFFE in Practice

SPIFFE provides all of the plumbing so that a workload can easily obtain 2 things:

- A set of certificates and private keys used to prove identities that the workload has access to. A specific identity is marked as default to ease configuration.
- A set of root certificates along with data for which identities those certificates apply to.

When a workload is launched, it refers to a SPIFFE Node CA to obtain this information. The address of the Node CA is specified in an environment variable or command line flag. If neither of these are present the workload must assume it is not operating in a SPIFFE enabled environment.

From the end user point of view, there are two common ways that this certificate can be used:

- Without any application changes whatsoever, use a proxy (forward and reverse) that will apply and verify identity. As we see proxies take on more and more interesting behavior (linkerd, Weave Flux, CoreOS jwtproxy) this behavior can be bundled in.
- Use a micro-services or RPC framework that automatically recognizes when it is run in a SPIFFE environment and uses the certificates.

The certificates should be flexible enough to be used in multiple different ways. Note also that just because two workloads are SPIFFE enabled doesn't mean that they can communicate effectively. They must agree on how to use those certificates.

- Secure the link via TLS. It is possible for clients to use these certificates and keys at the socket layer. This becomes even more compelling with HTTP/2 as server TLS is pretty much required.
- Payload signing. It is easy to use the cert and key to sign a payload directly. Something like JSON Web Signatures might work well here. However, doing public key cryptography for each message may be overly expensive.
- Establish a shared secret followed by signed payloads. This is more efficient but assumes peer to peer negotiation of a shared secret by the client and server (using Diffie Hellman key exchange?). There are sure to be subtle cryptography considerations here and is out of scope of this plan. This also starts to duplicate much of TLS at the payload level. NaCL/box looks interesting here but requires further study.

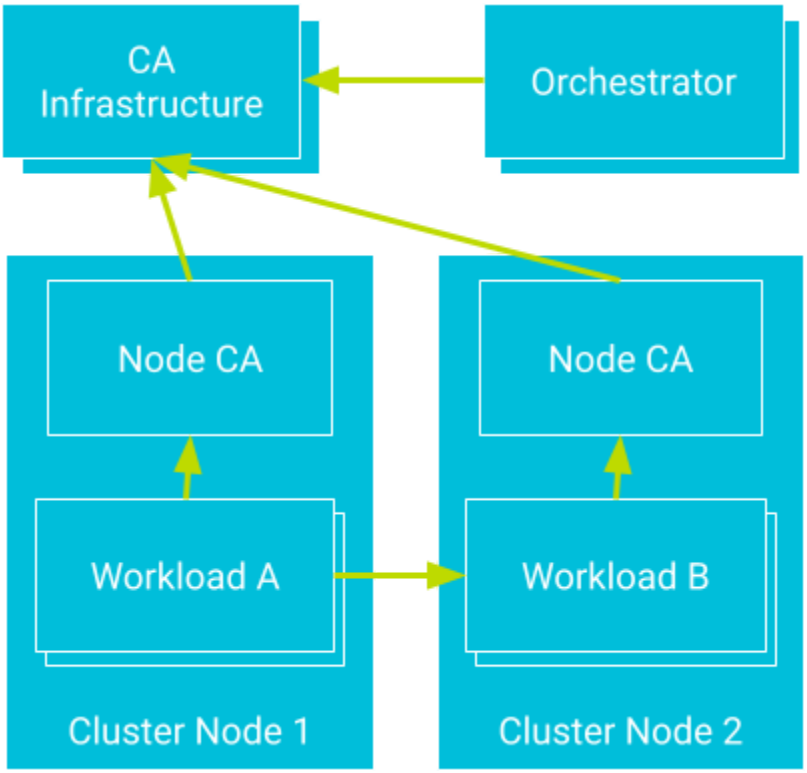
Design Sketch

At its core, SPIFFE will be a set of conventions around how to get and use x.509 certificates. Public key encryption is a great fit for the challenges both inside a datacenter and out. But the expertise to specify and deploy a certificate system is outside the capabilities of most organizations. In addition, integration with other automated systems is difficult as APIs are lacking in this space.

Moving to an automated API driven system will force clean "certificate lifecycle" management. If all certificates are short lived, then rotation must be automated. In addition, the same API mechanisms can be used for managing revocation.

Let's break SPIFFE down into some separable components. This will make it easier to understand and easier to implement. We'll go into detail bellow on these.

1. SPIFFE ID and the x.509 certificate mapping.
2. SPIFFE workload API. This is how workloads get and verify certificates
3. Open source reference implementation of SPIFFE.



SPIFFE ID and x.509 Certs

To start with, we will define an identity namespace. This is a simple structured string that will be the key part of the certificate. While it is possible that we could borrow from existing types of names (such as email or x.509 Distinguished names) it may make sense to have a bit of a clean break here. I'd suggest we define a new type of URI for this name.

```
urn:spiffe:example.com:caribou:frontend
```

This has the benefit of being based on the DNS namespace but not implying that there is anything reachable over, say, HTTP. Let's break it into parts:

- `urn` -- this explicitly states that this isn't a resource but just a name
- `spiffe` -- this name is part of the SPIFFE framework
- `example.com` -- the organization that this name belongs to.
- `caribou:frontend` -- a list of ':' delimited strings that are up to the organization to define. Best practices would be to not put too much structure into this part of the namespace as organizations change over time.

This scheme is a **straw man** and is worth investigating other approaches deeply. However, it will suffice for the discussions here.

Why not email addresses or URLs or LDAP distinguished names? Email addresses and URLs imply that the names can be used in a certain way that may not be supported. They probably won't be an email box for each of the names in question. Similarly, using URLs makes it easy to confuse where to talk to a service with the identity of a service. LDAP style hierarchy maps very easily to x.509 certificates but is alien[2] to many users and has concepts that don't map easily.

It is an open question on how to encode this into x.509 certificates[3]. **I'm still exploring the options in this area and would appreciate input.** Regardless of how the names are encoded into certificates, they will be represented as a simple string that can be compared easily[4].

It will be common for organizations to map employees into identities also so that their actions can be represented and tracked in this system. While production services wouldn't be run with an employee identity, one off jobs and tests may be. In this case, we may see `urn:spiffe:example.com:eng:jbeda`.

Hostname Certificates

This is still a bit of an open question.

All we've talked about so far is around abstract identity. However, most TLS libraries expect to verify against a hostname when making a connection. The actual host name used is generally going to be a function of the service discovery system used and may not even be DNS based. In addition a single identity may offer multiple services.

For Kubernetes, as an example, the discovery DNS scheme for services is `<service>.<namespace>.svc.<cluster-base>`. `<cluster-base>` defaults to `cluster.local`. If we map each Kubernetes namespace to a SPIFFE ID, then we could also make those SPIFFE certs good for every service in that namespace by issuing a certificate with a `subjectAltName` of `*.<namspace>.svc.<cluster-base>`.

The domain to put in the certificate will be highly dependent on the granularity of SPIFFE IDs and the services that those SPIFFE IDs expose. It will be easier for users if there is a predictable mapping and no need for explicitly configuring which services which IDs own.

---

[2] LDAP DNs are actually a list of sets of KV pairs. The double nested nature is very confusing. In fact, the Go implementation (and I assume many others) over simplifies and throws away data when parsing these. See https://golang.org/src/crypto/x509/pkix/pkix.go?s=1953:2006#L52.

[3] Options include trying to squeeze it into the CN, breaking it apart into standard DN components, inventing new extensions to the DN, or perhaps the URI type in the SubjectAltName extension. Furthermore, we must consider if the certificate duplicates information found in the root. Specifically, if the root cert specifies that it handles `urn:spiffe:example.com:*` but it signs a cert for `urn:spiffe:example.net:id` then it would be up to the client to catch and verify this. Perhaps the root cert should include the namespace and the identity cert should include the relative ID? Then it is up to the client to reconstruct the FQSID (Fully Qualified SPIFFE ID).

[4] We'll have to be specific about what characters are allowed and how these are compared. Ideally we can also easily define globbing type structure for specifying a space of SPIFFE IDs. See RFC 6943.

SPIFFE Workload API

This is the protocol that the workload (usually through a microservices/RPC framework) will use to communicate with the Node CA.

It assumed that workloads have been enlightened to know about SPIFFE and speak this protocol. If they have not been, they should instead run behind a SPIFFE aware proxy.

A scenario that requires more study is a workload that knows how to use x.509 certificates but doesn't know about SPIFFE. These workloads won't be able to deal with short lived certificates and manage their own rotation.

When a workload runs, it should be able to get a key/certificate with very little work. There is an environment variable that is passed into the workload that specifies the Node CA that the workload should look to and trust. It is expected that this environment variable will specify either a UNIX domain socket, localhost, or a link-local address (maybe?).

As it is expected that this address will be to a Node CA on the same machine, the workload can communicate without the overhead of authentication or encryption. This makes it very easy to use.

Services provided over the SPIFFE Client Protocol:

- Get the root certificates it should trust. (Those certificates have the namespaces that they cover embedded in them).
- List the identities that are available to the workload, including a "default" identity.
- Get a key pair and certificate for each of those identities.
    - This certificate is very short lived[5] (1-2 hours?)
    - It may further be scoped[6] to specify that it can only be used from a specific set of IP addresses.
- Process a CSR for a pre-existing public key.
- Sign a blob of data so that the workload never sees the key.

Question: Can/should this be a variant of ACME?

SPIFFE Proxies

While new code can talk the SPIFFE client protocol directly, it is likely that existing workloads will want to take advantage of SPIFFE with no or few changes. It will also be a challenge for polyglot[7] multi-service applications to have consistent libraries and behavior across languages.

---

[5] We need to think through what happens when a certificate expires while being used for an active connection. Most likely we would keep this connection alive and grandfather the auth for the lifetime of the connection. New connections would have to use an updated certificate.

[6] The certs the Node CA would get from the root/intermediate would be locked to its IP. That would trickle down to the certificates it creates/signs. Workload developers wouldn't be able to use those certs outside of that IP. (NAT obviously creates problems here and so we'll want to be careful if this system is used with NAT.)

[7] Polyglot here means applications built up of services implemented with different languages/runtimes/platforms.

To accomplish this, SPIFFE can be integrated into a "sidecar" proxy. This proxy would run in a very trusted context with the workload (localhost, domain socket) and would handle all communication that goes off the machine for that workload. In effect, the workload would only talk to the proxy over localhost (or a domain socket, etc.). The proxy, in turn, will encrypt and handle connections to the rest of the cluster.

- The proxy would handle both incoming and outgoing connections. Outgoing connections are a little more complicated and may require some client changes.
- The proxy may inspect/modify the request. For instance, for outgoing connections, it may key off of the 'Host' header to determine the ultimate destination. Or, for incoming connections, it may add an additional header to communicate the SPIFFE ID of the caller.
- The proxy can implement an ACL mechanism based on the ID.

The sidecar proxy could operate at either (or both) the TCP and HTTP layers. However, HTTP will be much richer as it can do advanced things like inject identity into the headers, do SPIFFE ID based flow control, or implement ACLs on HTTP paths/verbs. Similar things can be done with raw TCP[8] but standards are lacking and not widely supported.

As we see smarter microservice oriented proxies come to market (linkerd, Weave Flux, CoreOS jwtproxy) it will become more and more natural to put authentication and authorization into those proxies. Note that there is a tradeoff here between making authorization rich and built in to the application versus putting it in a proxy and more generic.

SPIFFE Cert Sync Sidecar

This optional component will help "legacy" servers use SPIFFE certificates that also cover hostnames.

If we put DNS hostnames into SPIFFE certificates those certificates can be used directly by "legacy" servers. However, those servers won't know how to get and rotate the certificates. This is where the "cert sync sidecar" (better name welcome!) comes in. This helper will talk the SPIFFE Workload API and write certificates to disk. It will also take some action (such as sending a HUP signal to a process) so to cause the certificate to be reloaded.

Open Reference SPIFFE Implementation

Certificate Authority Infrastructure

It is expected that each organization will have a root certificate authority and a (rotatable) Root certificate. This can either be run via a dedicated CA for just their organization or via a shared service. While it is possible for this CA to sign certificates that go beyond the scenarios listed here, we won't specify those.

[8] See the HAProxy PROXY protocol as a direction to take. Note that only the very new version 2 of the protocol supports extensibility for other "headers". Headers defined so far are not rich enough for SPIFFE so more extensions would be needed.

Unlike web browsers, there will be no large set of universally trusted root certificates. It is expected that the location of the root CA and an initial root certificate will be installed into client machines as part of a machine provision process. The Node CA will be responsible for any necessary rotation and surfacing the trust roots to client workloads.

Federation between organizations is largely left as an option question for now. For SPIFFE instance A to trust instance B, A will have to be configured with B's root certificate and namespace. It is possible that DNSSec may be used but that shifts the burden to the trust roots for the DNS system. HTTPS could be used to retrieve SPIFFE root certificates if an organization desires to trust HTTPS. Or that machines can get the organization's root cert as part of the machine provisioning process.

Signing other certificates by root certificate should be API driven -- potentially via the ACME protocol (with necessary extensions[9]). There may be other APIs into the CA for actions that aren't specified for ACME. Specifically, many implementations will have a manual approval flow for certain entities.

Simple configurations of SPIFFE will have a root CA that is always online. More complex configurations will have offline roots with intermediate CAs that handle the day to day workload.

Root certificates should have long expiration times -- on the order of years. If intermediate certificates are used, they should have shorter lifetimes. There is a tradeoff between the manual labor of rotating the intermediate certificates vs. the risk of long term compromise.

Node Certificate Authority

A trusted agent running on each node, the "Node CA," plays a unique and critical for SPIFFE. It has two main jobs: mediating certificate issuance for workloads and distributing SPIFFE configuration information (mainly trust roots). This works for both dynamically scheduled systems (Kubernetes, Swarm, Mesos, etc) and for more statically provisioned workloads.

The Node CA has a trust relationship (via a public key) with the CA infrastructure. It then plays traffic cop for all certificates issued to workloads on that particular node. The CA infrastructure will ensure that the Node CA cannot get certificates for workloads that are not assigned to it. In this way a compromised Node will only compromise identities for workloads that are assigned to run on that node.

When a workload asks the Node CA for a certificate, the Node CA can map that workload to the set of identities that it should have access too. This mapping is supplied as part of the configuration information provided by the CA infrastructure. This mapping can be flexible and extensible. Starting out, the mapping could be open (any workload on a machine gets access to an identity), pinned to a local UID or to a specific container.

[9] The DNS based challenges defined as part of ACME now probably aren't appropriate. But they are extensible. I'm thinking we could (a) use a pre-existing key pair that is on the machine as the ACME account key (TPM?) and thus no challenge, (b) use a time limited one time code that is communicated in a slightly insecure way during machine provisioning or (c) have a manual approval process similar to many existing systems. I'm sure there are other ways we could bootstrap trust in a low risk way.

Because the Node CA is running on the same kernel as the workload, it has the unique ability to trace networking connections back to the actual calling process. For example, it can look at the other end of a UNIX domain socket to determine the UID of that user. Or, in a containerized environment, it can offer certificates to containers by tracing the networking connection in a trusted way.

Open question: is the Node CA a true CA or a trusted forwarder? In other words, does the Node CA have longer lived certificates for each identity that it can use to sign short lived certificates for a workload or does it forward all certificate requests to the CA infrastructure. There are pros and cons to each approach around complexity and reliability.

Other interesting aspects of the Node CA:

- Its certificates have a relatively short expiration time -- on the order of hours[10].
- Beyond providing keys/certs to the workload, it could act as a "software TPM" and sign data without exposing the key to the workload at all. Similarly, it could provide a trusted implementation for verifying an incoming certificate chain.
- Assuming the Node CA is a true CA, colocating the Node CA with the workload will provide for increased robustness. The workload and the "Node CA" share fate in that they are on the same hardware.
- The Node CA would provide other limited restrictions on the certificates that it hands to the workload. Specifically, the certificates could be locked down to the specific node. This further limits the damage when a node is compromised.

Orchestrator Integration

The Node CA needs to be configured to only be able to author certificates for the workloads that are running on that machine. It also needs to know which identities map to which workloads. This should be coordinated through any Orchestrator[11] that happens to be active in the production environment.

The Orchestrator will configure the CA infrastructure with information about which Node CAs should have access to which identities. Along with this will be mapping information that will be made available to the Node CAs.

In order to scope down the power of the Orchestrator, the CA infrastructure may have further policy to limit which identities the orchestrator can control for which Node CAs. For instance, a user may have a set of dedicated machines for doing payment processing. The Orchestrator may have policy as to which workloads gets scheduled on to those machines. The root CA could provide an extra policy to ensure that the identities for payment processing will *only* ever be given to that set of machines. This could be done in the way where the root CA has policy that supersedes the orchestrator and so the orchestrator is not fully trusted.

---

[10] This assumes that workloads using these certificates are written with SPIFFE in mind and support automatic key rotation. This should be true for both microservice frameworks and SPIFFE proxies. If "legacy" workloads want to use these certificates things get more complicated.

[11] Orchestrator here could be a system such as Kubernetes, Docker Swarm or Mesos. Or it could be a more monolithic PaaS. Or it could be a traditional config management system such as Puppet, Chef, Salt or Ansible.

Beyond v0

Federation

SPIFFE supports wide and loose federation. In order for OrgA to call OrgB, OrgB must be configured with OrgA's root certificate and the SPIFFE ID namespace that corresponds to that root cert.

A practical example: the current state of the art for authenticating to a web service is to get a relatively static token (usually via a web page) and use that as part of an HMAC signature or just simply embed it in a header over an HTTPS connection. The biggest problem with a shared secret approach like this is that it is up to the user to responsibly manage storing that token. It is all too likely that the token will either be lost or leaked.

With SPIFFE we can do better. Instead of getting a token from the web service, the user instead provides the root certificate for the calling code along with the SPIFFE IDs to trust for that certificate. This is much preferred as (a) the caller has no need to store a separate secret and (b) all configuration information could be made public with no compromise.

Why X.509?

Maik Zumstrull says:

> I would recommend reconsidering [using X.509 certificates].
>
> It seems to me the only thing you want from X.509 here is "raw PKI": the ability to make a tree out of keypairs and signed blobs.
>
> But with X.509, you are also getting:
>
> - A family of file formats widely understood to be the worst.
> - A hypercomplex identity model that you don't plan to use at all, but can't entirely avoid because it's deeply tied into the format.
> - A collection of existing implementations that are all terrible.
> - Unfortunate technology choices around how to make cryptographic signatures that can never be fixed because compatibility.
> - Difficulty building flexible PKI models because each cert encodes the identity of the cert it expects to be signed by (Issuer Name).
> - Zero hint of any plan for migration to post-quantum crypto primitives.

> I get that I'm basically recommending "invent your own crypto" now, which is not good, but I feel like if you have someone on the team (or can get someone) who is qualified to get this right, this might be situation where you want to.

While I agree that x.509 (and ASN) is a very complex specification that is hard to understand, reinventing it is out of scope for this effort. While the libraries for dealing with x.509 certs may be lacking, they exist. Using an alternate certificate format would require much more code to be written to drive to wide acceptance.

There is prior art here in avoiding X.509. Both OpenPGP and OpenSSH use public key encryption with alternate certificate formats. Since these formats are only supported for these specific projects adopting them will provide yet another barrier to entry for SPIFFE. OpenPGP, specifically, has a trust model that is less adaptable to a business organization and will likely result in a hard coded “root”.

Even if we had an alternate certificate format, we would then have to adapt TLS to use this new certificate type. RFC 6091 starts this for OpenPGP but it is not widely implemented.

However, X.509 is not a critical part of this framework. The general flow and protocols can be used with alternate certificate formats. To this end, we’ll ensure that all APIs specify that they are looking for X.509 certificates to leave room for alternate formats in the future.

Open Questions

Cluster Bootstrapping

With a new cluster, it is necessary to establish a trust relationship between the Node CAs and the root CA. Similarly, the orchestrator will have a privileged role and that trust relationship needs to be set up. What techniques can be used to do that in a painless and secure way?

Ideas to enable easy bootstrapping:

- A manual approval queue with tooling. Administrators would look at pending requests and approve/deny those requests based on out of band knowledge.
- One use time limited tokens. The root CA could provide an API to get tokens for “pre-provisioned” accounts. These would be good for a short amount of time and would be single use to bootstrap longer term trust relationships.
- Automated “cloud” verification. It is possible to use lower level APIs (such as cloud APIs) to collect enough information to confidently automatically approve trusting a new machine.
- Hardware encryption. With TPMs and HSMs it is possible to tie this to unique aspects of the machine that cannot be counterfeited.

Delegation

If TLS becomes the common mechanism for authentication, how do we implement delegation -- that is trust a third party to act on behalf of other identities?

Ideas here include impersonation at the application and authorization layer. The immediate caller is clear but other solutions are used to identify indirect callers. This could be implemented via capability tokens that are signed by SPIFFE keys.

The "blessing" mechanism from Vanadium may provide some additional inspiration here.

Grouping

Many times when doing authorization, users want to specify a dynamic group of principals. Resolving a "group" name to the transitive group of individual identities is a subtle problem. There are concerns around security, caching and performance.

Down-scoping

What ways can we further limit the power of certificates in appropriate ways. Can more be done beyond locking those certs to a machine and having a short expiration?

Authorization

How is a client's SPIFFE ID mapped by a server to a role with associated authorization policy? Part of the power of SPIFFE is that this is left undefined. For some situations authorization may be as simple as a command line flag and a an "if" statement. Other scenarios may call for centralized access management ACL systems with centralized auditing. However, extending SPIFFE with optional conventions and examples would be helpful.

Alternatives

I'm sure that there are existing systems that echo the design decisions and principles of SPIFFE. I'm not, however, aware of any system that is in wide use and open for this type of scenario.

TODO: Add lots more detail here.

- Kerberos --
- Active Directory -- AD combines a direoctory, kerberos infrastructure and, optionally, x.509. While it may be usable for the use cases described here, it is brings in too many assumptions for the simplest applications. In other words, any system that starts with "first install AD" will see limited uptake.
- Vanadium Security Model -- while both the Vanadium project and SPIFFE are both inspired by Google systems such as LOAS, the Vanadium security model assumes that users are taking on the entire Vanadium model. It also avoids existing crypto implementations (x.509 certs, TLS) in favor of optimal and less widely deployed technologies. We should look to borrow/merge ideas in the future as applicable.
- SDN w/ 802.1X
- VPN mesh

TODO: list alternatives and analysis. Kerberos?

Thank You!

Many of the use cases for SPIFFE are inspired by an internal system at Google called LOAS. I never looked under the covers there so I can't say how much the implementation overlaps.

Lots of folks have given feedback and advice here. Note that a name listed here does not imply endorsement.

- William Morgan, Buoyant
- Brendan Burns, Google
- Brandon Philips, CoreOS
- Jimmy Zelinskie, CoreOS
- Jake Moshenko, CoreOS
- Maik Zumstrull, SysEleven GmbH
- Sunil James, BVP
- Erica Lan, Salesforce
- Eric Tune, Google
- David Strauss, Pantheon
- Solomon Boulos, Google
- Vijay Gill, SalesForce
- Asim Aslam, Micro
- Sasha Klizhentas, Gravataional
- Christopher Liljenstolpe, Tigera

Also lots of folks on the Xoogler Slack.

If I forgot you here, please reach out and I'll happily add you.

Changelog

June 14, 2016. Broke out design into parts. SPIFFE ID and certs, workload API and reference implementation. Documented thoughts on including hostnames in certs and how those might be used (SPIFFE cert sync sidecar). Also explicitly state what is out of scope in intro (authz, directory).