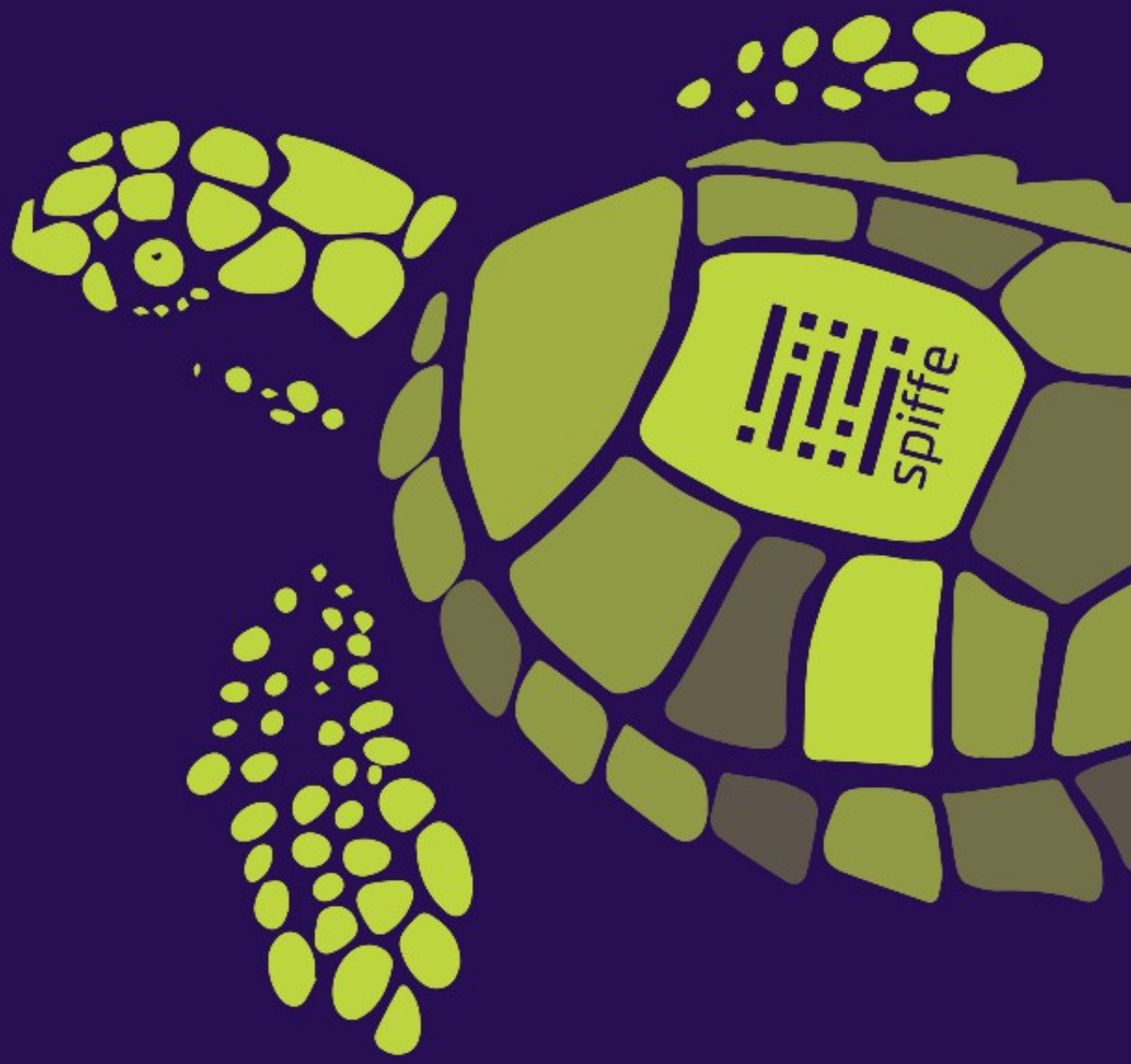
SOLVING THE BOTTOM TURTLE



a SPIFFE Way to Establish Trust in Your Infrastructure via Universal Identity

Contents

Solving the Bottom Turtle—a SPIFFE Way to Establish Trust in Your Infrastructure via Universal Identity

by Daniel Feldman, Emily Fox, Evan Gilman, Ian Haken, Frederick Kautz, Umair Khan, Max Lambrecht, Brandon Lum, Agustín Martínez Fayó, Eli Nesterov, Andres Vega, Michael Wardrop, Volkan Özçelik.

First edition **2020**, `ISBN: 978-0-578-77737-5`—Hewlett Packard Enterprise (HPE) sponsored the Book Sprint of the first edition of this book as a contribution to the open source community.

Converted to Markdown, for portability, and ease of editing, by Volkan Özçelik, **March 2025**.

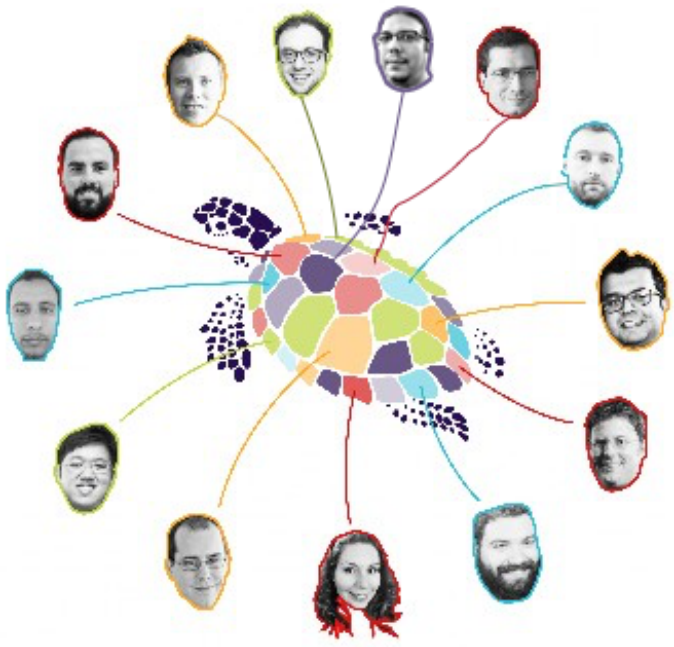
This book was produced using the Book Sprints methodology. Its content was written by the authors during an intensive collaboration process conducted online over two weeks.

- **Book Sprints Facilitation**: Barbara Rühling
- **Copy Editors**: Raewyn Whyte and Christine Davis
- **HTML Book Design**: Manuel Vazquez
- **Illustrations and Cover Design**: Henrik Van Leeuwen

thebottomturtle.io

Participants

- Daniel Feldman is Principal Software Engineer at Hewlett Packard Enterprise
- Emily Fox is the CNCF Special Interest Group for Security (SIG-Security), Co-Chair
- Evan Gilman is the Co-Founder and CEO of SPIRL.
- Ian Haken is Senior Security Software Engineer at Netflix
- Frederick Kautz is Head of Edge Infrastructure at Doc.ai
- Umair Khan is Sr. Product Marketing Manager at Hewlett Packard Enterprise
- Max Lambrecht is Senior Software Engineer at Hewlett Packard Enterprise
- Brandon Lum is Senior Software Engineer at IBM
- Agustín Martínez Fayó is Principal Software Engineer at Hewlett Packard Enterprise
- Eli Nesterov is the Co-Founder and CTO of SPIRL.
- Andres Vega is Product Line Manager at VMware
- Michael Wardrop is Staff Engineer at Cohesity
- Volkan Özçelik is a Lead Principal Engineer at VMware Cloud Foundation (*Broadcom*)



About the Book

This book presents the SPIFFE standard for service identity, and SPIRE, the reference implementation for SPIFFE. These projects provide a uniform identity control plane across modern, heterogeneous infrastructure. Both projects are open source and are part of the CNCF.

As organizations grow their application architectures to make the most of new infrastructure technologies, their security models must also evolve. Software has grown from one monolith on one box, to dozens or hundreds of tightly linked microservices that may be spread across thousands of virtual machines in public clouds or private data centers. In this new infrastructure world, SPIFFE and SPIRE help keep systems secure.

This book strives to distill the experience from the foremost experts on SPIFFE and SPIRE to provide a deep understanding of the identity problem and how to solve it. With these projects, developers and operators can build software using new infrastructure technologies while allowing teams to step back from expensive and time-consuming manual security processes.

About Zero the Turtle

Access control, secrets management, and identity are all dependent on each other. Managing secrets at scale requires effective access control; implementing access control requires identity; proving identity requires possession of a secret. Protecting one secret requires coming up with some way to protect another secret, which then requires protecting that secret, and so on.

This brings to mind the famous anecdote about a woman who interrupted a philosopher's lecture to tell him the world rested on a turtle's back. When the philosopher asked her what the turtle rested on, she said: "It's turtles all the way down!" Finding the bottom turtle, the solid foundation on which all other security rests, is the goal of the SPIFFE and SPIRE projects.

Zero the Turtle, depicted on the cover of this book, is that bottom turtle. Zero represents the foundation for security in the data center and the cloud. Zero is trustworthy and happily supports all the other turtles.

SPIFFE and SPIRE are projects that help you find the bottom turtle for your organization. With the tools in this book, we hope you find a home for Zero the Turtle, too.

Addendum for this Edition (v2.0.0), March 2025

This edition has been converted from the original PDF version of the book into Markdown format. While Markdown is less visually expressive compared to the original format, and certain decorative elements have been omitted during the conversion process, every effort has been made to preserve the integrity and intent of the original content.

Converting the book into Markdown and hosting it in a GitHub repository allows the community to contribute, collaborate, and build upon the work. This transition also provides significant benefits in terms of accessibility and flexibility, making it a worthwhile trade-off for the loss of some visual enhancements.

We are bumping the version number from `1.0.0` to `2.0.0` because this represents a major change in terms of people being able to edit and contribute to the book much more easily thanks to standard tooling and formats. We are also planning to do ongoing improvements after we implement the required automation and tooling in place.

We hope this edition will be an evergreen one, continually enhanced and grown through the support and contributions from the community. Future updates will follow semantic versioning as we make improvements and additions to the content.

1. History and Motivation for SPIFFE

This chapter contextualizes the motivation for SPIFFE and how it came to be.

Overwhelming Motivation and Need

We haven't arrived where we are today without first experiencing some growing pains.

When the internet first became widely available in 1981, it had just 213 different servers, and security was hardly even an afterthought. As the number of interconnected computers increased, security remained a weakness: easily exploited vulnerabilities lead to mass attacks such as the Morris Worm, which took over most Unix servers on the internet in 1988, or the Slammer worm, which spread among hundreds of thousands of Windows servers in 2003.

As the decades have passed, conventional perimeter defense patterns of yesteryear are not well suited to the evolving computing architectures and boundaries of modern technology. Point solutions and technologies have stacked one on to another to cover growing cracks where underlying network security concepts have failed to follow modernization trends.

So why is the perimeter pattern so prevalent and what do we need to do to address the shortcomings?

Over the years, we've observed three considerable trends that highlight conventional perimeter patterns as inhibitors to the future of networking.

- Software no longer runs on individual servers controlled by the organization. Since 2015, new software has typically been built as a collection of microservices that can be individually scaled out or moved to cloud hosting providers. If you can't draw a precise line around the services that need to be secured, it's impossible to build a wall around them.
- You can't trust everything, even the software in the company. Once, we thought software vulnerabilities were like flies that we could swat individually; now they seem more like a swarm of bees. On average, the National Vulnerability Database reports more than 15,000 new software vulnerabilities per year. If you write or buy a piece of software, it will probably have some vulnerabilities at some point.
- You also can't trust people, they make mistakes and get upset, plus they have full access to internal services. First, tens of thousands of successful attacks annually are based on phishing or stealing valid employee credentials. Second, with the advent of cloud applications and mobile workforces, employees may legitimately access resources from many different networks. Building a wall no longer makes sense when people constantly have to cross back and forth over that wall just to do their jobs.

As you can see, perimeter security is no longer a realistic solution for today's organizations. When perimeter security is strictly enforced, it holds organizations back from using microservices and the cloud; when it is lax, it allows intruders in. In 2004, the Jericho Forum recognized the need for a successor to perimeter security. Ten years later in 2014, Google published a case study about the BeyondCorp security architecture.

However, neither reached widespread adoption.

The Network Used to Be Friendly, so Long as We Kept to Ourselves

The original internet use case was focused on academia with the intent to share information, not prevent access. As other organizations began using networked computer systems for business-sensitive use cases, they relied heavily on physical perimeters and physical attestations for assurances that the individuals accessing the network were authorized to do so. The concept of a trusted insider threat did not yet exist. As networks evolved from academic to business-oriented use cases and software evolved from monoliths to microservices, security became a barrier to growth.

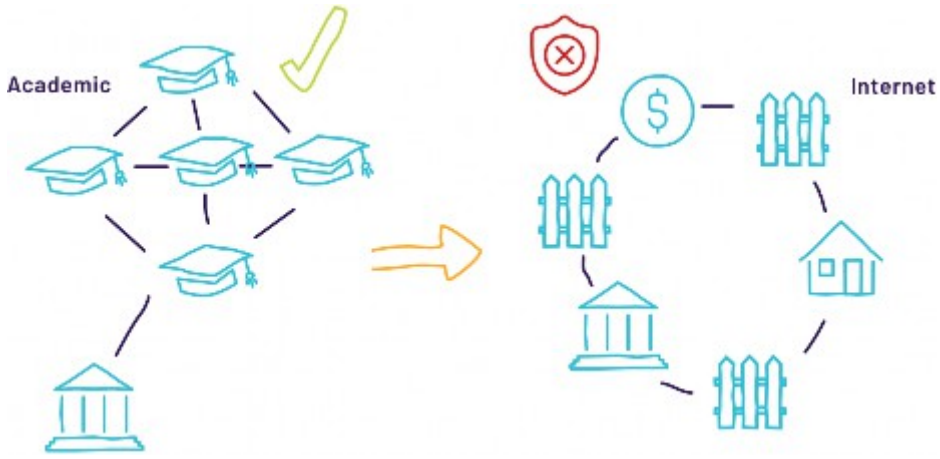


Figure 1: Evolution of networks from the confines of the university campus to the global network of networks.

Initially, conventional methods of protecting physical access to computers with walls and guards were emulated through firewalls, network segmentation, private address space, and ACLs. This made sense at the time, especially when considering the limited number of points needing control.

With the proliferation of networks, and increased access points for users and business partners, physical identity verification (common with walls and guards), became virtualized through securely exchanging and managing keys, credentials, and tokens, all of which become increasingly problematic as technology and needs evolved.

Adopting Public Cloud

The migration from conventional on-premises and data center operations to a public cloud, amplified existing problems of yesteryear.

With the new freedom to create computing resources in the cloud, the development teams and operations teams within organizations began to collaborate more closely and form new teams around the concept of DevOps focused on automated deployment and management of software. The rapidly evolving dynamic environment of the public cloud-enabled teams to deploy far more frequently—changing from one deployment every few months, to many per day. The ability to provision and deprovision resources on-demand enabled the high-velocity creation of suites of specialized, focused, and independently deployable services with a smaller scope of responsibility, colloquially referred to as *microservices*. This, in turn, increased the need for identifying and accessing services across deployment clusters.

This highly dynamic and elastic environment broke accepted perimeter security concepts, requiring better service level interaction that was agnostic of the underlying network. Conventional boundary enforcement used IPs and ports for authentication, authorization, and auditability, which under cloud computing paradigms no longer cleanly mapped to workloads.

Patterns spurred on by public cloud engagement, such as API gateways or managed load balancers for multiservice workloads, underscored the need for identities not dependent on network topology or path.

Protecting the integrity of these service-to-service communications became more important, particularly with teams needing uniformity across workloads.

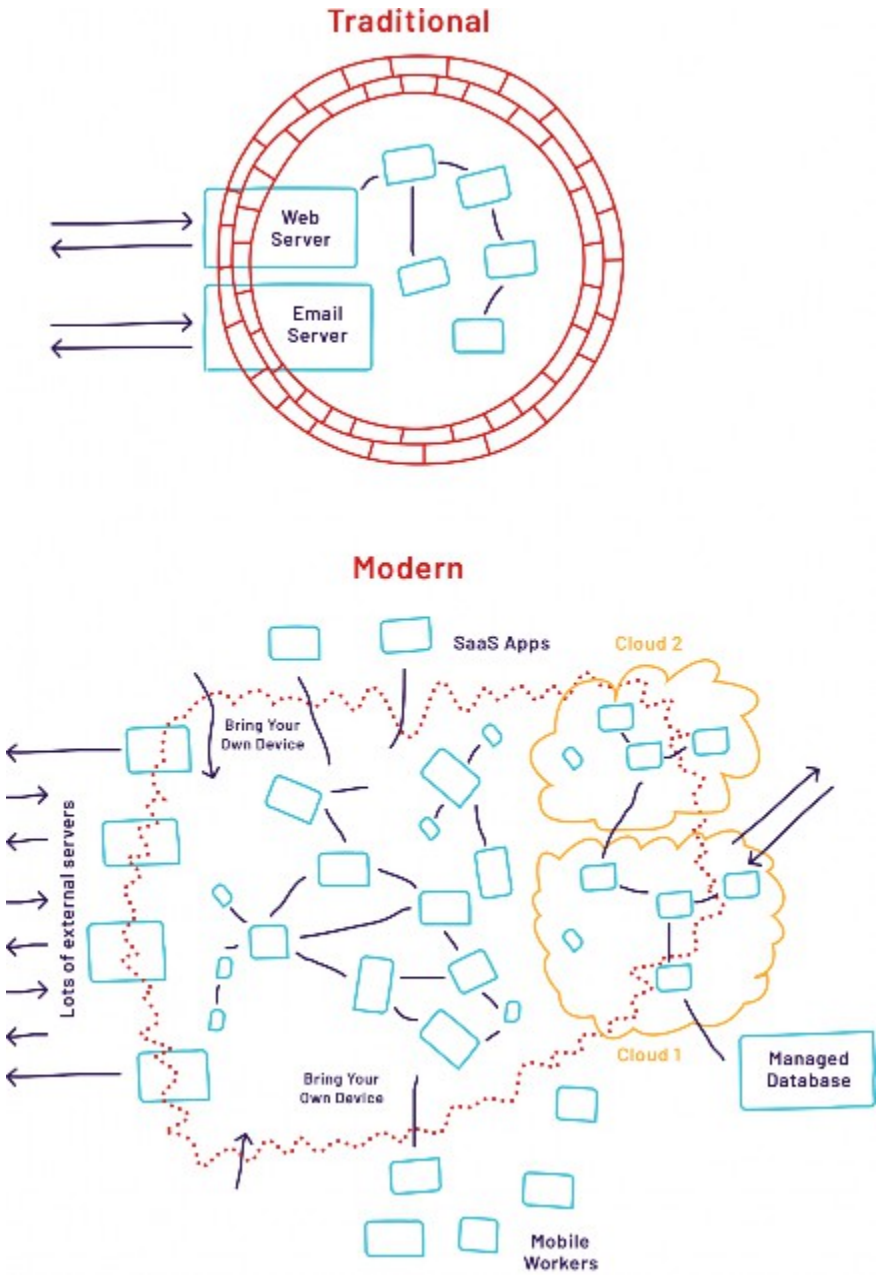


Figure 2: As an organization’s networks increase in complexity, adding cloud, SaaS, and mobile workers, building and maintaining perimeter security becomes increasingly unsustainable.

Houston, We Have a Problem

As organizations adopt new technologies, such as containers, microservices, cloud computing, and serverless functions, one trend is clear: there are a larger number of smaller pieces of software. This both increases the number of potential vulnerabilities that an attacker can exploit, and also makes managing perimeter defenses increasingly impractical.

The push to do more, faster, means increasingly more components are deployed across automated infrastructure, often sacrificing safety and security. Bypassing manual processes such as firewall rule tickets or security group changes are not unheard of. In this new, modern world, network-oriented access controls become rapidly out-of-date and require constant maintenance, regardless of the deployment environment.

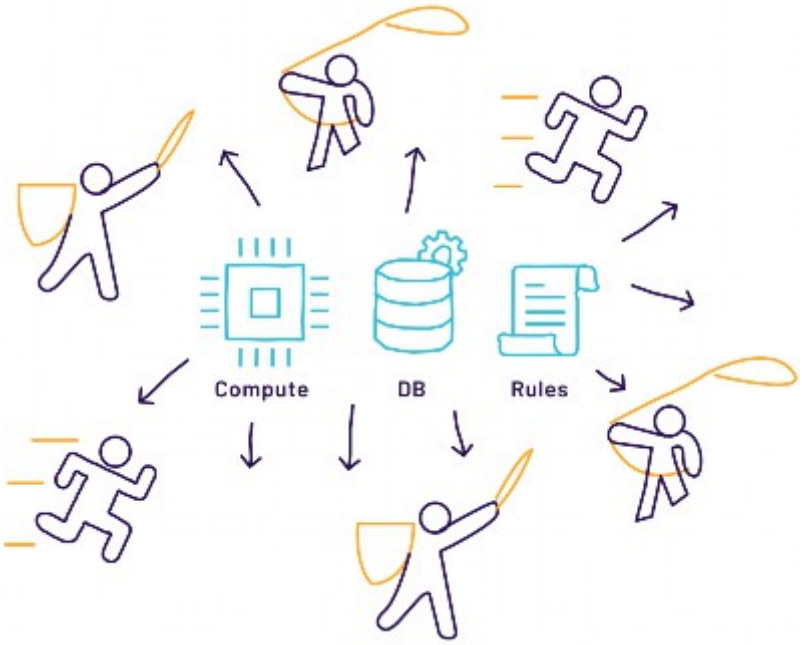


Figure 3: Fig. 1.3: Infrastructure environments and related operation demands are increasingly complex with the proliferation of new technology innovations.

Management of these rules and exceptions can be automated, however, they need to happen expediently, which can be challenging in larger infrastructures. Further, network topology, such as Network Address Translation (NAT), can make this difficult and leaky. As infrastructure becomes larger and more dynamic, human-in-the-loop systems simply won't scale. After all, nobody wants to pay for a team of people to play with firewall rules all day who are still unable to keep up.

Reliance on location-specific details such as server names, DNS names, network interface details has several shortcomings in a world of dynamically scheduled and elastically scaled applications. While there is prevalent usage of networking constructs, the model is an ineffective analog of software identity. Moving towards the application layer, the use of conventional username and password combinations or other hard-coded credentials confers a degree of identity but deals more with authorization than authentication.

Integrating security and introducing feedback earlier in software development lifecycles enables developers more operational control over the mechanisms by which their workloads can be identified and exchange information. With this change, authorization policy decisions may be delegated to individual service or product owners who are best suited to make decisions relevant to the component in question.

Re-imagining Access Control

The ever-expanding list of problems experienced by organizations before public cloud adoption, and painfully forthcoming as a result of this adoption, pushed the concept that conventional perimeters are insufficient and better solutions need to exist. Ultimately, deperimeterization meant that organizations needed to figure out how to identify their software and enable service-to-service access control.

Secrets as a Solution Shared secrets such as passwords and API keys provide a simple option for access control in distributed systems. But this solution brings many problems with it. Passwords and API keys can easily be compromised (try searching GitHub for a phrase such as "client_secret" and see what happens). In large organizations, secrets can be hard to rotate in response to a compromise, since every service needs to be aware of the change in a coordinated fashion (and missing a service could cause an outage).

Tools and secrets repositories such as HashiCorp Vault have been developed to help mitigate the difficulties of secrets management and life cycles. And while many other tools also exist that try to address this problem, they tend to offer an even more limited solution with mediocre results (see Secrets at Scale: Automated Bootstrapping of Secrets & Identity in the Cloud). With all of these options, we ultimately return to the same problem; how should workloads get access to this secret repository? Some API key, password, or other secret is still needed.

All these solutions end up with a "turtles all the way down" problem:

- Enabling access control to a resource such as a database or service requires a secret such as an API key or password.
- That key or password needs to be protected, so you could protect it, e.g., with encryption. But then you still need to worry about a secret decryption key.
- That decryption key could be put into a secrets repository, but then you still need some credential like a password or API key to access the secrets repository.
- Ultimately, protecting access to one secret results in a new secret you need to protect.

To break this loop, we need to find a **bottom turtle**, that is, some secret that provides access to the rest of the secrets we need for authentication and access control. One option is to manually provision secrets on services when they're deployed. However, this does not scale in highly dynamic ecosystems. As organizations have moved to cloud computing with rapid deployment pipelines and automatically scaled resources, it becomes infeasible to manually provision secrets as new compute units are created. And when a secret *is* compromised, invalidating the old credential can pose a risk of bringing down the entire system.

Embedding a secret in the application so that it doesn't need to be manually provisioned has even worse security properties. Secrets embedded in source code have a habit of showing up in public repositories (did you try that GitHub search we suggested?). While the secret could be embedded into machine images at build time, those images can still end up being accidentally pushed to a public image repository or extracted from the internal image repository as a second step in a kill chain.

We want a solution that does not include long-lived secrets (which can be easily compromised and are hard to rotate) and does not require manual provisioning of secrets to workloads. To achieve this, either in hardware or the cloud provider, **there must be a root of trust, upon which an automated solution centered around software (workload) identity is built**. This identity then forms the foundation for all interactions requiring authentication and authorization. To avoid creating another bottom turtle, the workload needs to be able to obtain this identity without a secret or some other credential.

Figure 4: No longer ‘Turtles all the way down’ with a sound identity ‘bedrock’

Steps toward the future Multiple endeavors to solve the software identity problem have been pursued since 2010. Google's Low Overhead Authentication Service (LOAS), later titled Application Layer Transport Security (ALTS), established a new identity format and wire protocol for receiving software identity from the runtime environment and applying it to all network communication. It has been referred to as *dial tone security*.

In another example, Netflix's internally developed solution (codenamed Metatron) identity on an instance-by-instance basis by leveraging cloud APIs to attest the machine image running on instances and CI/CD integration to produce cryptographic bindings between machine images and code identity.

This software identity takes the form of X.509 certificates to perform mutual authentication of service-to-service communication, including access to the *secrets service* developed as part of this solution which enables secrets management on top of this foundation.

Several other efforts in the industry, including from companies such as Facebook, prove the need for a system like this and underscore the difficulty of implementation.

The vision of a Secure Production Identity Framework For Everyone (SPIFFE) The establishment of principles to be encoded in a framework was needed before a generalized solution could exist. Reflecting upon exposure to various technologies in past jobs that made engineering teams lives easier, Kubernetes founding engineer Joe Beda, embarked on a call to arms in 2016 to create a solution for production identity as a purpose-built solution aimed to solve the problem in a common way that could be leveraged across many different types of systems over intermediary solutions of doing PKI the hard way. This massive collaborative effort between companies to develop a new standard for service identity based on PKI was the beginning of SPIFFE.

Beda's paper was presented at GlueCon in 2016 and showcased a hard problem with these parameters:

- Solve secret zero by leveraging kernel-based introspection to obtain information about the caller without the caller having to present a credential,
- Use X.509 since most software is already compatible, and
- Effectively divorcing the concept of identity from network locators.

Following the introduction of the SPIFFE concept, a meeting occurred at the Netflix campus with experts in service identity to discuss the ultimate shape and viability of the original SPIFFE proposal. Many members had implemented, continued to improve upon, and re-solve workload identity, highlighting an opportunity for community collaboration. Members at the meeting desired to obtain interoperability among each other and with others. These experts realized they had implemented similar solutions to solve the same problem, and could work together to build a common standard.

The initial goals of solving the workload identity problem were to establish an open specification and corresponding production implementation. The framework needed to provide interoperability between different implementations and off-the-shelf software, at its core establishing a root of trust in an untrusted environment, exorcising implicit trust. And finally, moving away from network-centric identity to achieve flexibility and better scaling properties.

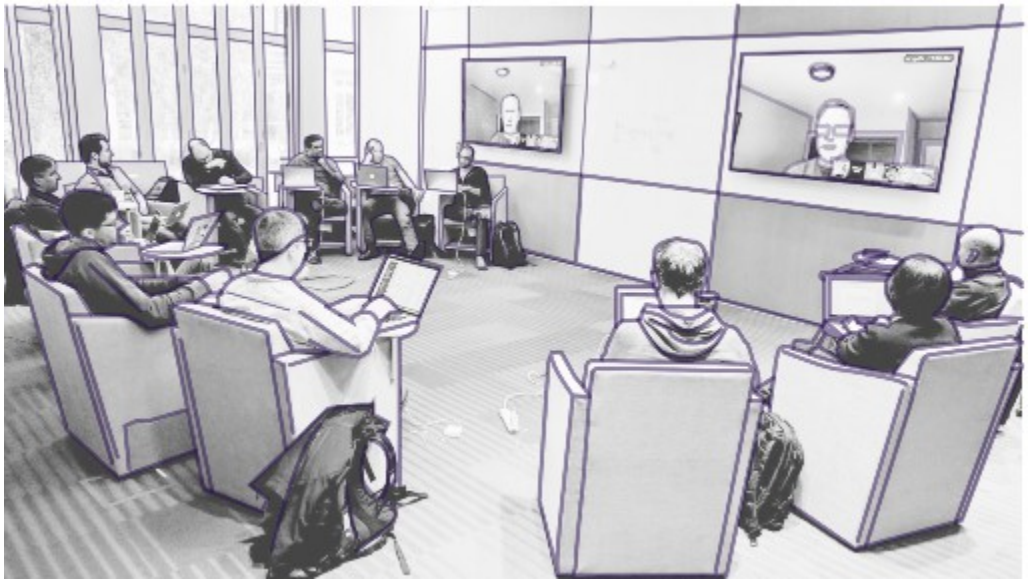


Figure 5: Where it all began: the 2016 meeting at Netflix where security experts started to sketch out the concepts for SPIFFE

2. Benefits

This chapter explains the benefits of deploying SPIFFE and SPIRE across your infrastructure, from a business and technology perspective.

For Everyone, Everywhere

SPIFFE and SPIRE aim to strengthen the identification of software components in a common way that can be leveraged across distributed systems by anyone, anywhere. The modern infrastructure technology landscape is convoluted. Environments grow increasingly disparate with mixes of hardware and software investments.

Maintaining software security by standardizing how systems define, attest, and maintain software identity, regardless of where systems are deployed or who deploys those systems, confers many benefits.

For business leaders focused on improving business expediency and returns, SPIFFE and SPIRE can significantly reduce costs associated with the overhead of managing and issuing cryptographic identity documents (e.g. X.509 certificates), and accelerate development and deployment by removing the need for developers to understand the identity and authentication technologies required to secure service-to- service communication.

For service providers and software vendors focused on delivering a robust, secure, and interoperable product, SPIFFE and SPIRE solve a critical identity problem prevalent when interconnecting many solutions into a final product. For example, SPIFFE can be used as the basis for a product's TLS features and user management/authentication features in one fell swoop. In another example, SPIFFE can replace the need for managing and issuing API tokens for platform access, bringing rotation *for free* and eliminating the customer burden of storing and managing access to said tokens.

For security practitioners looking to not only enhance the security of data in transit but also achieve regulatory compliance and solve their root of trust problem, SPIFFE and SPIRE work to deliver mutual authentication across untrusted environments without the need to exchange secrets. Security and administrative boundaries can be easily delineated, and communication can occur across those boundaries, when and where policy allows.

For developers, operators, and DevOps practitioners in need of identity management abstraction, and interoperability with modern, cloud native services and solutions for their workloads and applications, SPIFFE and SPIRE are compatible with numerous other tools throughout the software development lifecycle to deliver reliable products. Developers can get on with their day, going straight to business logic without worrying about nuisances such as certificates, private keys, and JSON Web Tokens (JWTs).

For Business Leaders

Modern organizations have modern needs In today's business environment, rapidly delivering innovative customer experiences through differentiated applications and services is necessary for competitive advantage. As a result, organizations witness a change in how applications and services are being architected, built, and deployed. New technologies such as cloud and containers help organizations release faster, at scale. Services need to be built at high velocity and deployed on a vast plethora of platforms. As development accelerates, these systems are becoming increasingly interdependent and interconnected to deliver a consistent customer experience.

Organizations may be inhibited in achieving high velocity and gaining market share or mission assurance for such major reasons as compliance, the pool of expertise, and interoperability challenges between teams/organizations and within existing solutions.

The impact of interoperability As systems evolve, the need for interoperability grows indefinitely. Disjointed teams build services that are siloed and unaware of each other, even though they need to eventually be conscious of one another. Acquisitions occur in which new or never-before-seen systems need to be folded into existing systems. Business relationships are established, demanding new communication channels with services that may reside deep in the stack. All of these challenges are centered around the problem of "how do I connect all of these services together, each with its own unique properties and history, in a secure way?".

Technology integration as a result of organizational convergence can be a challenge when different technology stacks must come together and interoperate. Aligning on a common, industry-accepted standard for system-to-system communication with identity and authentication, simplifies the technical aspects of full interoperability and integration across multiple stacks.

SPIFFE brings a shared understanding of what constitutes software identity. By further leveraging SPIFFE Federation, components in disparate systems in different organizations or teams can establish trust to communicate securely without the added overhead of constructs such as VPN tunnels, one-off certificates, or shared credentials for use between such systems.

Compliance and auditability Auditability within the SPIRE implementation provides assurances that identities performing actions cannot be repudiated as a result of enforcing mutual authentication within the environment. Further, the identity documents issued by SPIFFE/SPIRE enable the pervasive use of mutually-authenticated TLS, effectively solving one of the most difficult challenges associated with projects of this nature. Additional benefits to mutually authenticated TLS include native encryption of data in transit between services, not only protecting the integrity of the communications but also assuring the confidentiality of sensitive or proprietary data.

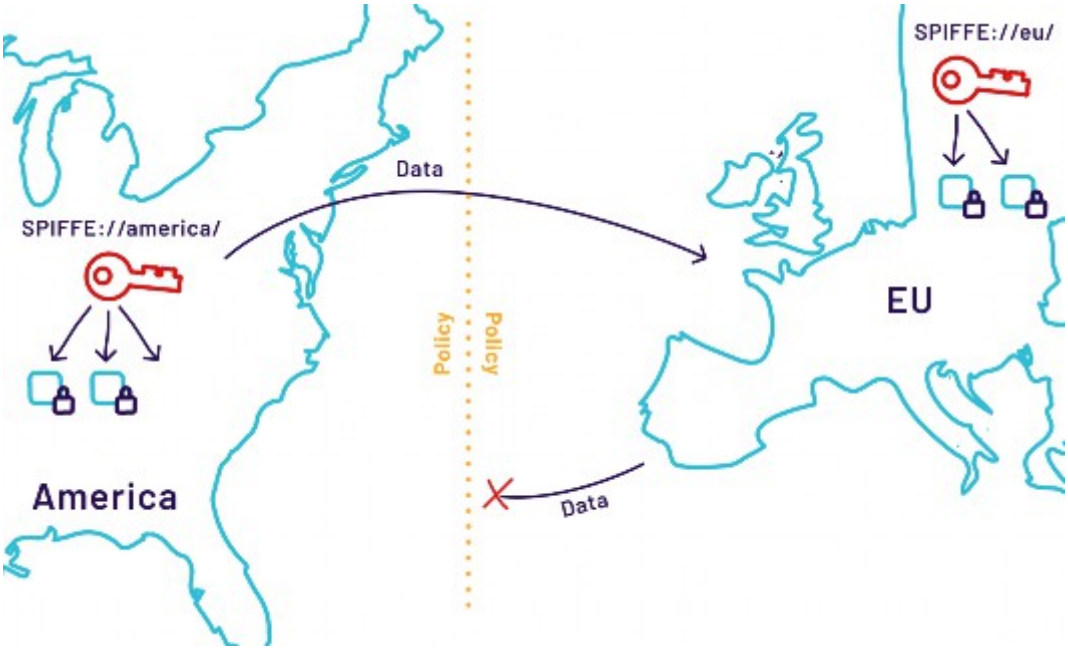


Figure 6: Meeting compliance and regulatory objectives seamlessly using SPIFFE.

Another common compliance requirement is brought on by the General Data Protection Regulation (GDPR)—specifically requiring that European Union (EU) data reside wholly within the EU, not transiting or being processed by entities outside its jurisdiction. With multiple roots of trust, global organizations can ensure that EU entities communicate only with other EU entities.

Pool of expertise Ensuring that development, security, and operations teams are equipped with the right knowledge and experience to handle security-sensitive systems appropriately, remains a significant challenge.

Enterprises need to be able to hire developers with standards-based skill sets to decrease onboarding time and improve time-to-market with reduced risk.

Solving the problem of delivering cryptographic identity to every software instance in an automated fashion, and enabling credential rotation from the root down, poses a major challenge. For security and operations teams, the expertise required to implement such systems is few and far between. Sustaining day-to-day operations without relying on community or industry knowledge exacerbates the problem, leading to outages and finger-pointing.

Developers cannot reasonably be expected to know or gain subject matter expertise in practical

matters of security, especially as it applies to service identity within organizational environments. Further, the pool of security practitioners with the depth of knowledge in development, operations, and workload execution, is minimal. Leveraging an open standard and open specification to solve critical identity problems allows for individuals without personal experience to expand knowledge through a well-supported and growing community of SPIFFE/SPIRE end-users and practitioners.

Savings Adoption of SPIFFE/SPIRE enables cost savings on many fronts, including reduced cloud/platform lock-in, improved developer efficiency, and reduced reliance on specialized expertise, to name a few.

By abstracting cloud provider identity interfaces into a set of well-defined common APIs built on open standards, SPIFFE significantly reduces the burden of developing and maintaining *cloud-aware* apps. Since SPIFFE is platform-agnostic, it can be deployed practically anywhere. This differentiator saves time and money when a platform technology change is necessary, and can even strengthen negotiation positions with existing platform providers. Historically, identity and access management services are the command and control of every organization's deployments—cloud services providers know this and leverage this constraint as the primary lock-in mechanism to fully integrate with their platform.

There are significant savings to be had in the area of improved developer efficiency as well. Two important aspects of SPIFFE/SPIRE unlock these savings: the fully automated issuance and management of cryptographic identity and its associated lifecycle, and the uniformity and offloading of authentication and service-to-service communication encryption. By removing manual processes associated with the former, and time spent in research and trial/error in the latter, developers can better focus on what matters for them: the business logic.

Boost developer productivity	
Avg. time spent by developers in obtaining credentials and configuring authentication/confidentiality protocols per application component (hours)	2
Reduction in time spent by the developer in corresponding for credentials per application component	95%
Avg time spent by the developer in learning and implementing controls for specific API gateways, secret stores, etc. (hours)	1
Reduction in time spent by developers in learning and implementing controls for specific API gateways, secret stores, etc.	75%
Number of new application components developed in the year	200
Projected hours saved due to better developer productivity	**530**

As we've seen historically, Fortune 50 technology organizations employing highly-skilled and specialized engineers took decades to resolve this problem of identity. Adding SPIFFE/SPIRE to an organization's catalog of cloud native solutions allows you to build on top of years of hyper-specialized security and development talent without the corresponding cost.

With a robust community supporting deployments of a few dozen to several hundred thousand nodes, SPIFFE/SPIRE's experience operating in complex, large scale environments can meet the needs of the organization.

For Service Providers and Software Vendors

Reducing customer burden incurred in the course of using a product is always the number one goal of any good product manager. It is important to understand the practical implications of features that seem innocuous on the surface. For example, if a database product needs to support TLS because it is required for customer compliance, it is simple to add a few configurables to the product and call it a day.

Unfortunately, this pushes some significant challenges to the customer. Similar challenges are faced even with seemingly simple user management. Consider the following customer pain points that both of these common features introduce by default:

- Who generates the certificates and passwords, and how?
- How are they securely distributed to the applications that need them?
- How is access to private keys and passwords restricted?
- How are these secrets stored such that they don't leak into backups?
- What happens when a certificate expires, or a password must be changed? Is the process disruptive?
- How many of these tasks necessarily involve a human operator?

Figure 7: Confused turtle.

All of these questions need to be answered before these features are viable from a customer perspective. Often, the solutions invented or installed by the customer are operationally painful.

These customer burdens are very real. Some organizations have entire teams dedicated to managing these burdens. By simply supporting SPIFFE, all of the above concerns are alleviated. The product can snap into existing infrastructure, and grow TLS support for *free*. Further, client (user) identity conferred by SPIFFE can replace the need for directly managing user credentials (e.g. passwords).

Platform access management Accessing a service or platform (e.g. a SaaS service) involves similar challenges. Ultimately, these challenges boil down to the inherent difficulty presented by credential management, particularly so when the credential is a shared secret.

Consider API tokens for a moment—the use of API tokens is prevalent among SaaS providers to authenticate non-human API callers. They are effectively passwords, and each one must be carefully managed by the customer. All of the challenges listed above apply here. Platforms that support SPIFFE authentication greatly alleviate customer burdens associated with accessing the platform, solving storage, issuance, and life cycle problems all at once. Leveraging SPIFFE, the problem is reduced to simply granting a given workload the desired privileges.

For Security Practitioners

Technical innovation cannot be an inhibitor to secure products. Development, distribution, and deployment tools need seamless integration with security products and methods that do not impact the autonomy of software development or create a burden on the organization's success. Organizations need software products that are easy to use and add additional security to existing tools.

SPIRE is not the end-all solution to every security problem. It does not negate the need for robust security practices and defense in depth or layered security. However, leveraging SPIFFE/SPIRE to provide a root of trust across untrusted networks allows organizations to take a meaningful step forward down the path to a zero trust architecture as part of a comprehensive security strategy.

Secure-by-default SPIRE can help mitigate several major OWASP threats. To reduce the likelihood of a breach through credential compromise, SPIRE provides a strongly attested identity for authentication across the entire infrastructure.

The automation that keeps the promise of assurance makes the platform secure-by-default, removing an additional configuration burden by development teams.

For organizations looking to solve the root of trust and identity issues within their products or services, SPIFFE/SPIRE also addresses the customer's security needs by enabling pervasive mutual TLS authentication to securely deliver communications between workloads no matter where they are deployed. Further, a s with every open source product, the community and contributors behind the code base provide multiple sets of eyes to scrutinize code pre-and-post-merge. This implementation of 'Linus' Law' goes beyond the four eyes principle to ensure any potential bugs or known security issues are caught before making their way to distribution.

Policy enforcement SPIRE's APIs provide a mechanism for security teams to enforce consistent authentication policies across platforms and business units in an easy-to-use manner. When coupled with a well-defined policy, interactions between services can be kept to a minimum, ensuring only authorized workloads may communicate with each other. This constrains the potential attack surface by a malicious entity and can trigger alerts in the policy engine's default deny rule.

SPIRE leverages a powerful multi-factor attestation engine that runs in real-time to determine, with certainty, the issuance of cryptographic identities. It also automatically issues, distributes, and renews short-life credentials to ensure that the organization's identity architecture accurately reflects the operational state of the workloads.

Zero trust Adopting a zero trust model in the architecture reduces the blast radius if a breach were to occur. Mutual authentication and trust revocation can block a compromised front end application server from exfiltrating data from unrelated databases that may be available on the network or within a cluster. While not likely to occur in organizations with tight network security, SPIFFE/SPIRE certainly adds additional defense layers to mitigate vulnerabilities and exposure points from misconfigured firewalls or unchanged default logins. It also shifts security decisions away from IP addresses and port numbers (which can be manipulated in undetectable ways) and towards cryptographic identifiers that enjoy integrity protections.

Logging and monitoring SPIRE can help improve the observability of the infrastructure. Critical SPIRE events such as identity requests and issuances are loggable events that help provide a more complete view of the infrastructure. SPIRE will also generate events for a variety of actions, including identity registrations, deregistrations, attestation attempts, identity issuance, and rotations. These events can then be aggregated and sent to the organization's security information and event management (SIEM) solution for single-pane-of-glass monitoring.

For Dev, Ops, and DevOps

Even though you can quantify improvements to the developer or even operational productivity by adopting and supporting SPIFFE/SPIRE regardless of environment, ultimately, it relieves much of the toil teams experience by reintroducing focus, flow, and joy in their daily work.

Focus Security cannot be an inhibitor to technical innovation. Security tools and controls need frictionless integration with modern products and methods that do not impact the autonomy of development or create a burden on the operations team.

SPIFFE and SPIRE provide a uniform service identity control plane that is available through a consistent API across platforms and domains, so the team can focus on delivering applications and products without concern or special configurations for the destination. Each developer can leverage this API to securely and easily authenticate across platforms and domains.

Developers can also request and receive an identity that may then be used to build additional application-specific controls for the supplied identity, while operators and DevOps teams can manage and scale identities in an automated manner, simultaneously implementing and executing policies that consume these identities. Further, teams can use the OIDC Federation to correlate SPIFFE identities with various cloud authentication systems, such as AWS IAM, reducing the need for difficult-to-manage secrets.

Flow Every credential ever generated suffers from the same problem: at some point, it will have to be changed or revoked. When the time comes, the process is often manual and painful — and just as with deploys, the less frequently it occurs the more painful it becomes. Unfamiliarity with the process and outages induced by lack of timeliness or unwieldy update procedures are par for the course.

When rotation is required, it frequently demands expensive context switches for operators and developers alike. SPIFFE/SPIRE addresses this by treating rotation as a critical central function. It is fully automated and occurs regularly without human intervention. The frequency of rotation is an operator choice, and trade-offs are involved; however, it is not uncommon for SPIFFE credentials to rotate hourly. This frequent and automated rotation approach minimizes operator and developer interruptions related to credential lifecycle management.

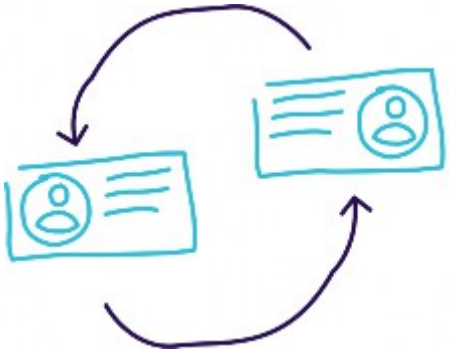


Figure 8: Certificate rotation.

It is important to note that it is not just rotation that is automated. The initial issuance of the credential (most commonly in the form of an X.509 certificate) is also fully automated. This works to streamline developer flow, taking the task of generating or procuring the credential out of the checklist for spinning up a new service.

Interoperability Developers and integrators no longer need to be frustrated by the lack of interoperability in the organization's secure identity and authentication solution. SPIRE provides a plugin model that allows developers and integrators to extend SPIRE to suit their needs. This capability is particularly important if the organization requires a set of proprietary APIs to generate SPIRE's keys, or if the intermediate signing keys of SPIRE should live in a specific proprietary Key Management Service (KMS). Developers also need not worry about developing custom wrappers for new workloads to be brought online because the organization is adhering to an open specification.

Many teams are afraid to change or remove firewall rules that permit traffic between networks because of the potential adverse effect of critical systems availability. Operators may scope identities and their associated policies to applications instead of globally. Locally scoped identities and policies give operators the confidence to enact changes without fear of downstream impact.

Improvement of daily work Without a robust software identity system, service-to-service access management is often accomplished through the use of network-level controls (e.g. IP/Port-based policy). Unfortunately, this approach generates a significant amount of operational toil associated with managing network access control lists (or ACLs). As elastic infrastructure comes up and down, and network topologies change, these ACLs need constant care and feeding. They can even get in the way of turning on new infrastructure, as the existing systems now need to be taught about the existence of the new pieces.

SPIFFE and SPIRE work to reduce this toil, as the concept of software identity is relatively stable when compared with the arrangement of hosts and workloads on a network. Furthermore, they pave the way for delegating authorization decisions to the service owners themselves, who are ultimately in the best position to make such decisions. For example, service owners that want to grant access to a new consumer need not be concerned with network-level details for the creation of the access policy — they may simply declare the name of the service they wish to grant access to, and carry on.

SPIFFE/SPIRE also works to improve observability, monitoring, and ultimately Service Level Objectives (SLO) adherence. By normalizing software identity across many different types of systems (not necessarily just containerized or cloud native), and providing an audit trail of identity issuance and usage, SPIFFE/SPIRE can greatly improve situational awareness before, during, and after an incident occurs.

More mature teams may even find that it improves their ability to predict problems before they impact service availability.

3. General Concepts Behind Identity

This chapter explains what an identity is, as well as the basics of distributing, managing, and using identities. These are concepts you will need to know in order to understand how SPIFFE and SPIRE work.

What Is an Identity?

For humans, identity is complex. Humans are unique individuals who can't be cloned or have their minds replaced with alternate code, and they may carry multiple social identities over the course of their lives. Software services are just as complex.

A single program might scale out to thousands of nodes, or change its code many times a day as a build system pushes new updates. In such a rapidly changing environment, an identity must represent the specific logical purpose of the service (e.g. a *customer billing database*) and an association with established authority or root of trust (e.g. *my* `company.example.org` or *the issuing authority for my production workloads*).

Once identities are issued for all the services in an organization, they can be used for *authentication*: proving that a service is what it says it is. Once services can authenticate to each other, they can use identities for *authorization*, or control who can access those services, and *confidentiality*, or keeping the data they transmit to each other secret. While SPIFFE does not itself include authentication, authorization, or confidentiality, the identities it issues can be used for all of them.

Designating service identities for an organization is similar to designing any other part of the organization's infrastructure: it depends intimately on the organization's needs. When a service scales out, changes code, or moves locations, it may be logical for it to keep the same identity.

Trustworthy identity Now that we've defined identity, how do we represent that identity? How do we know that when a piece of software (or *workload*) claims its identity, that the claim is trustworthy? To start exploring these questions, we must first discuss how an identity is established.

Identity for humans Allow us to explain these concepts with something we all share: our real-world identities.

Identity documents If a name is a person's identity, then the proof of that identity is an identity document. A passport is a document that allows a person to prove their identity, so it is an *Identity Document*. Like passports from different countries, different types of software identity documents can look different and don't always contain the same information. But to be useful, they all usually at least contain some common information such as the user's name.

What is the difference between a passport and a napkin with your name scribbled on it?

The most significant difference is the source. With passports, we trust that the *Issuing Authority* has verified your identity, and we have the ability to verify that the passport was issued by that trusted authority (validation). With that napkin, we don't know where it came from and have no way to validate that it comes from the restaurant you say it does. We also can't trust that the restaurant wrote the correct name on the napkin, or verified the accuracy of your name when you communicated it.

Trusting an issuing authority We trust a passport because we implicitly trust an authority that issues them. We trust the process by which they issue these identity documents: they have records and controls to ensure that they are issuing an identity only to the correct individual. We trust the governance of this process, so we know that the passports that are issued by the authority are a faithful representation of someone's identity.

Verifying the identity document Given this, how can we differentiate between a real passport and a fake one? This is where *verification* comes in. Organizations need a way to determine whether the identity document is issued by the authority we trust. This is typically done through watermarks that are hard to replicate but easy to verify.

Authenticating the person presenting the identity document Passports codify several pieces of information about the person that the identity represents. First, they include a picture of the person that can be used to verify that the presenter is the same person visible on the passport. They may also include other physical attributes of the person—their height, weight, and eye color, for example.

All of these attributes can be used to *authenticate* a person who presents a passport.

To recap, passports are our *identity documents*, and we use them to identify each other because we trust the *Issuing Authority*, and have a way to *verify* that the document originates from that authority. Finally, we can *authenticate* the person presenting the passport by cross-referencing the contents of the passport with the person holding it.

Identity in the Digital World: Cryptographic Identities Circling back to workload identity, how do the above concepts map onto computer systems? Instead of passports, *Digital Identity Documents* are used. X.509 certificates, signed JSON Web Tokens (JWTs), and Kerberos tickets, are examples of digital identity documents. Digital identity documents can be verified using cryptographic techniques. Then, the computer system can be authenticated, much like a person with a passport.

One of the most useful and prevalent techniques to do this is Public Key Infrastructure (PKI). A PKI is defined as a set of roles, policies, hardware, software, and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. With PKI, digital identity documents can be validated locally, ofline, against a small, static set of root trust bundles.

A brief overview of X.509 When the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) initially released the X.509 standard for PKI in 1988, it was extraordinarily ambitious for its time and is still considered to be so. The standard originally envisioned giving certificates to humans, servers, and other devices alike, forming an enormous globally integrated secure communication system.

> **Sidenote**: X.509 was part of the X.500 telecoms standard, which proposed a global directory, in which users could look up data for humans and servers by name and obtain their certificates. No other part of the X.500 standard reached widespread adoption.

While X.509 has never reached its originally intended scope, it is the expected foundation for nearly every secure communications protocol. You can check out RFC 5280 for more information.

How X.509 works with a single authority

1. Bob's computer needs a certificate. He generates a random *private key* , and also a *certificate signing request* (or CSR), which includes basic information on his computer, such as its name; we'll call it *bobsbox*. A CSR is a little bit like a passport application.
2. Bob sends his CSR to a *certificate authority* (or CA). The CA validates that Bob is really Bob. The exact way this validation happens can vary—it may involve a human checking Bob's paperwork, or an automatic check.
3. The CA then creates a *certificate* by encoding the information presented in the CSR, and adding a digital signature, which serves to assert that the CA has verified the information contained within to be true and correct. It sends the certificate back to Bob.
4. When Bob wants to establish secure communications with Alice, his computer can present his certificate, and cryptographically demonstrate that it has Bob's private key (without ever actually sharing the contents of that private key with anyone).
5. Alice's computer can check that Bob's certificate is really Bob's certificate by checking that the Certificate Authority signed it. She trusts that the Certificate Authority properly checked Bob's identity before signing the certificate.

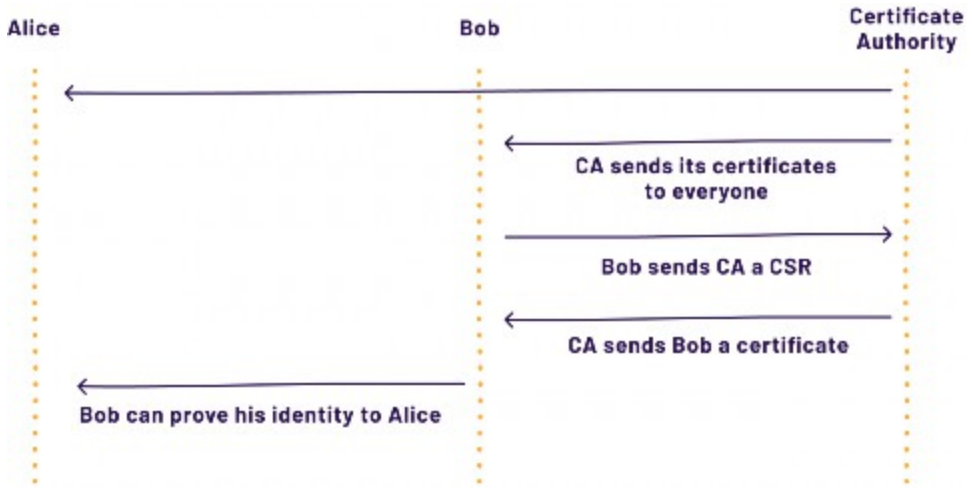


Figure 9: Fig. 3.1: Illustration of Bob requesting a certificate from the Certificate Authority, and using it to prove his identity to Alice

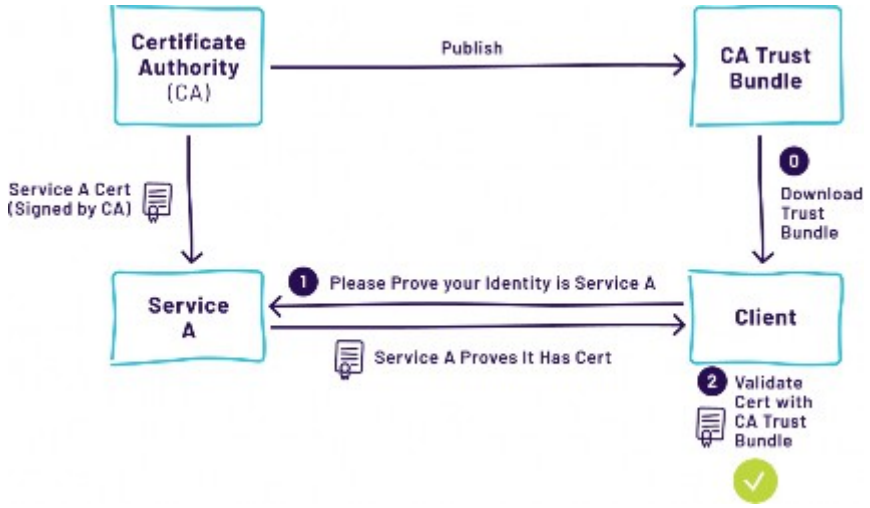


Figure 10: Fig. 3.2: PKI 'in a nutshell

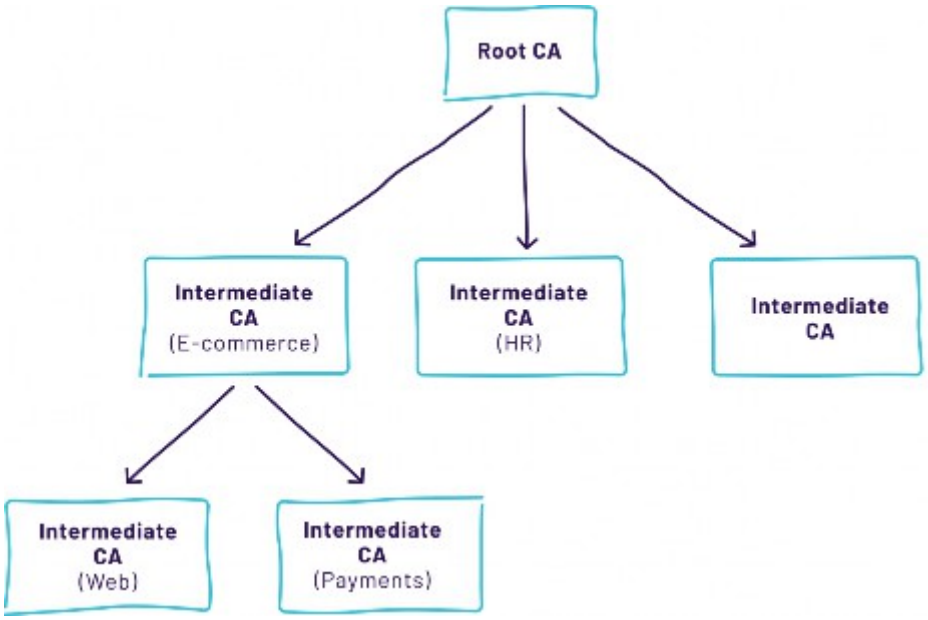


Figure 11: Fig. 3.3: Illustration of intermediate certificate authorities.

X.509 with intermediate certificate authorities In many cases, the CA that has signed a given certificate is *not* well-known. Instead, the CA has its own key and certificate, and that certificate is signed by another CA. By signing this CA certificate, the parent CA is certifying that the lower-order CA is authorized to issue digital identities. This authorization of lower-order CAs by higher-order CAs is known as *delegation*.

Delegation can occur repeatedly, with lower-order CAs further delegating their power, forming an arbitrarily tall tree of certificate authorities. The highest-order CA is known as the *root CA* and must have a well-known certificate. Every other CA in the chain is known as an *intermediate CA*. The benefit of this approach is that fewer keys need to be well-known, allowing the list to change less frequently.

This leads to a key weakness of X.509: *any CA can sign any certificate, with no restrictions*. If a hacker decides to start her own intermediate CA and can get approval from any one existing intermediate CA, then she can effectively issue any identity she wants. It is vital that each well-known CA is *completely* trustworthy, and that each intermediate CA they delegate to is *also completely* trustworthy.

> **Sidenote**: A newer extension called X.509 Name Constraints allows adding restrictions to certificate authorities so they can't issue certificates for outside organizations, but it is not widely adopted.

Certificate and identity lifecycle There are several additional features in PKI that make management and authentication of digital identities easier and more secure.

Authority delegation, identity revocation, and limited identity document lifetimes to name a few.

Identity issuance There is always a point at which a new identity must be issued. Humans are born, new software services are written, and in each of these cases, we must issue an identity where one was not present before.

First, a service needs to request a new identity. For a human, this might be a paper form. For software, it is an X.509 document called a Certificate Signing Request (CSR), which is created with a corresponding private key. The CSR is similar to a certificate, but since it has not been signed by any Certificate Authority, no one will recognize it as valid. The service then sends the CSR securely to the Certificate Authority.

> **Sidenote**: How does the service communicate securely to the Certificate Authority? Since the Certificate Authority's certificate is itself widely known, anyone can establish a secure connection to it.

Next, the Certificate Authority checks every detail of the CSR against the service that requested the certificate. Originally, this was intended to be a manual process: humans checking paperwork and making decisions on an individual basis. Today, the checks and the signing process are often completely automated. If you have used the popular LetsEncrypt certificate authority, then you are familiar with a fully automated certificate authority signing process.

Once satisfied, the Certificate Authority attaches its digital signature to the CSR, turning it into a fully-fledged certificate. It sends the certificate back to the service. Along with the private key that it generated earlier, the service can securely identify itself to the world.

Certificate revocation So, what happens if a service is compromised? What if Bob's laptop is hacked, or Bob leaves the company and should not have access anymore?

This process of undoing trust is called Certificate Revocation. The Certificate Authority maintains a file called the Certificate Revocation List with the unique IDs of the revoked certificates and distributes a signed copy of the file to anyone who asks.

Revocation is tricky for several reasons. First, the CRL must be hosted and served from an endpoint *somewhere*, introducing challenges around making sure that the endpoint is up and reachable. When it isn't... does PKI stop working? In practice, most software will *fail open*, continuing to trust certificates when CRLs are unavailable, making them effectively useless.

Second, CRLs can get large and unwieldy. A revoked certificate must remain in the CRL until it expires, and certificates are generally long-lived (on the order of years). This can lead to performance problems in serving, downloading, and processing the lists themselves.

Several different technologies have been developed to try to make certificate revocation simpler and more reliable, such as the Online Certificate Status Protocol (OCSP). The variety of approaches make certificate revocation a continual challenge. 7

> **Sidenote**: For an introduction to CRLs alongside some of the potential problems, see "Can we eliminate certificate revocation lists?" by Ronald Rivest.

Certificate expiration Every certificate has a built-in expiration date. Expiration dates are a vital part of the security of X.509 for several different reasons: to manage obsolescence, to limit the potential for change in the identity indicated by the certificate, to limit the size of CRLs, and to lessen the possibility that the key will be stolen.

Certificates have been around for a long time. When they were first developed, many CAs used the MD2 hashing algorithm from 1989, which was relatively quickly found to be insecure. If those certificates were still valid, attackers could forge them.

Another important aspect of a limited certificate lifetime is that the CA only has a single chance to validate the identity of the requester, but this information is not guaranteed to remain correct over time. For example, domain names frequently change ownership and are one of the more critical pieces of information generally included in a certificate.

If certificate revocation lists are in use, then each certificate that is still valid has the potential to be revoked. If certificates lasted forever, then the certificate revocation list could grow endlessly. To keep the certificate revocation list small, certificates need to expire.

Finally, the longer a certificate is valid, the greater the risk that the private key for its certificate or any certificate leading to the root could be stolen.

Frequent certificate renewal One compromise in solving the challenges posed by revocation is to rely more heavily on expiration. If the certificate lifetime is very short (perhaps only a few hours), then the CA can frequently re-perform any checks that it originally made. If certificates renew frequently enough, then CRLs might not even be necessary since it may be faster just to wait for the certificate to expire.

The identity lifespan trade-off

Shorter lifespan	Longer lifespan
If a document is stolen, it is valid for a shorter time	Reduced load on certificate authorities (humans and programs)
CRLs are shorter and maybe unnecessary	Reduced load on the network
Fewer outstanding identity documents at a time (easier to keep track of)	Better resiliency in case a node can’t renew its certificate due to a network outage

Another cryptographic identity: JSON Web Tokens (JWT) Another public key identity document, JSON Web Tokens (RFC7519), also behaves as a PKI-like system. Instead of certificates, it uses a JSON token and has a construct called JSON Web Key Set which acts as a CA Bundle to authenticate the JSON tokens. There are important differences between certificates and JWTs that are beyond the scope of this book, but just like X.509 certificates, the authenticity of a JWT can be verified using PKI.

The trustworthiness of foreign identities Whichever kind of identity you use, some trusted authority has to issue it. In many cases, not everyone trusts the same authorities or the process by which they were issued. Alice's New York State driver's license is a valid identification in New York, but it isn't valid in London because London authorities don't trust the state government of New York. However, Bob's US Passport is valid in London, because the UK authorities trust the US government authorities, and the London authorities trust the UK authorities.

The situation is identical in the realm of Digital Identity Documents. Alice and Bob might have certificates signed by completely unrelated CAs, but as long as they both trust those CAs, they can authenticate each other. That doesn't mean Alice *has* to trust Bob, just that she can securely identify him.

How Software Identity Can Be Used

Once a piece of software has a digital identity document, it can be used for many different purposes. We've already discussed using identity documents for authentication. They can also be used for mutual TLS, authorization, observability, and metering.

Authentication The most common use of an identity document is as a basis for authentication. For software identities, several different authentication protocols exist that use X.509 certificates or JWTs to prove the identity of a service to another service.

Confidentiality and integrity Confidentiality means that attackers can't see the contents of a message, while integrity means that they can't alter it in transit. Transport Layer Security (TLS) is a widely used protocol for building secure connections that provides authentication, confidentiality, and message integrity on top of an untrusted network connection using X.509 certificates.

One feature of TLS is that **either side** of the connection can be authenticated using certificates. As an example, when you connect to your bank's web site, your web browser authenticates your bank using an X.509 certificate presented by the bank, but your browser doesn't present a certificate to the bank. (You log in with a username and password, not a certificate.)

When two pieces of software are communicating, it is common for **both sides** of the connection to have X.509 certificates and authenticate each other. This is called *mutually authenticated TLS*.

Authorization Once a digital identity is authenticated, it can be used for authorizing access to services. Typically, each service would have an allowlist of other services that are permitted to make requests against it. Authorization can only occur after authentication.

> **Sidenote**: An identity does not grant authorization, rather the association of particular attributes about the identity (in a separate store) that allow a decision point or enforcement point to determine if the entity is authorized for access.

Observability Identity can also be useful for increasing observability within your organization's infrastructure. In large organizations, it's surprisingly common for old or unmaintained services to communicate in mysterious, undocumented ways. Unique identities for each service can solve this problem in conjunction with observability tooling. For logging, a non-repudiable identity of the requester can be useful if something goes wrong later.

Metering In microservice architectures, a common need is to throttle requests so that a fast microservice doesn't overwhelm a slow one. If each microservice has a unique identity, it can be used to manage a quota of requests per second to solve this problem or to deny access altogether.

Summary

Both humans and pieces of software have identities, and both can use identity documents to prove their identities. For humans, passports are a typical form of an identity document. For software, the most common form of a digital identity document is an X.509 certificate.

Certificates are issued by Certificate Authorities. Certificate Authorities need to take care to properly validate the people or things they are creating certificates for, and manage the lifespan of the certificates.

After the certificate is issued, whoever uses it needs to trust the certificate authority that issued it.

Once trusted digital identity documents are available, they have many different uses. One of the most common is to create a mutually authenticated TLS connection, which includes authentication, confidentiality, and integrity. Another common use is for authorization. With authentication, confidentiality, integrity, and authorization, connections between services are secure.

4. Introduction to SPIFFE and SPIRE

Building upon the concepts introduced in Chapter 3, this chapter illustrates the SPIFFE standard. It explains the components of the SPIRE implementation and how they fit together. Finally, it discusses the threat model and what happens if specific components are compromised.

What is SPIFFE?

The Secure Production Identity Framework For Everyone (or SPIFFE) is a set of open source standards for software identity. To achieve interoperable software identity in an organization- and platform- agnostic way, SPIFFE defines the interfaces and documents necessary to obtain and validate cryptographic identity in a fully automated fashion.

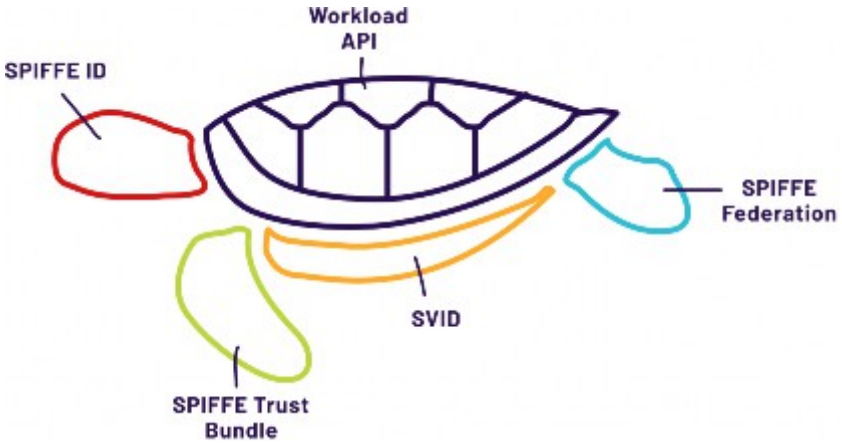


Figure 12: SPIFFE parts.

SPIFFE consists of five parts:

- The SPIFFE ID, how a software service's name (or identity) is represented
- The SPIFFE Verifiable Identity Document (SVID), a cryptographically verifiable document used to prove a service's identity to a peer
- The SPIFFE Workload API, a simple node-local API that services use to obtain their identities without the need for authentication
- The SPIFFE Trust Bundle, a format for representing the collection of public keys in use by a given SPIFFE issuing authority
- SPIFFE Federation, a simple mechanism by which SPIFFE Trust Bundles can be shared

  Sidenote: Lack of authentication on this API is an important distinction as it is one aspect which allows us to solve the bottom turtle problem. SPIFFE implementations are still responsible for positively identifying callers of the API, but they must do so in a manner that denies in-band credentials from being passed from the workloads.

What SPIFFE isn't SPIFFE is intended for identifying servers, services, and other non-human entities communicating over a computer network. What these all have in common is that the identities must be issuable automatically (without a human in the loop). While it may be possible to use SPIFFE for identifying people or other wildlife species, the project has purposely left these use cases out of scope. No special considerations have been other than for robots and machines.

SPIFFE delivers identity and related information to services while managing the lifecycle of this identity, but its role is limited to that of a provider as it does not directly make use of the delivered identities. It's the responsibility of the service to make use of any SPIFFE identities it receives. There are a variety of solutions for using SPIFFE identities that enable authentication layers, such as end-to-end encrypted communication or service-to-service authorization and access control, however, these functions are also considered out of scope for the SPIFFE project and SPIFFE will not solve them directly.

> **Sidenote**: Information provided alongside the identity may include things such as intermediate CA certificates, private keys for proving the identity, and public keys for validating SVIDs.

The SPIFFE ID A SPIFFE ID is a string that functions as the unique name for a service. It is modeled as a URI and is made up of several parts. The prefix `spiffe://` (as the URI's scheme), the name of the trust domain (as the host component), and the name or identity of the specific workload (as the path component).

A simple SPIFFE ID might just be `spiffe://example.com/myservice`.

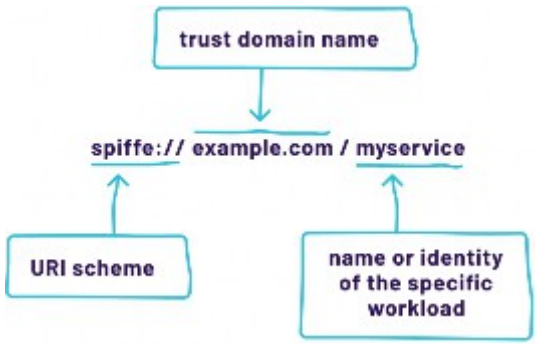


Figure 13: A sample SPIFFE ID, and its composition

The first component of a SPIFFE ID is the `spiffe://` URI scheme. Although mundane, including it is an important detail as it serves to distinguish a SPIFFE ID apart from a URL or other type of network locator.

The second component of a SPIFFE ID is the trust domain name (`example.com`). In some cases, there will simply be one trust domain for an entire organization. In other cases, it might be necessary to have many trust domains. Trust domain semantics are covered later in this chapter.

The final component is the name portion of the workload itself, represented by the URI path. The exact format and composition of this part of the SPIFFE ID is site-specific. Organizations are free to choose a naming scheme that makes the most sense for them. For instance, one might choose a naming scheme that reflects both the organizational location as well as the workload's purpose, such as:

```
spiffe://example.com/bizops/hr/taxrun/withholding
```

> **Sidenote**: The identity itself is inclusive of both the trust domain and name portion. The name portion may be referred to as the 'identity' but is considered incomplete without the trust domain to make it unique.

It is important to note that the primary purpose of a SPIFFE ID is to represent a workload's identity in a flexible way that is easy for both humans and machines to consume. Caution should be exercised when attempting to instill too much meaning in the format of a SPIFFE ID. For example, attempting to codify attributes that are later used as individual pieces of authorization metadata can lead to interoperability and flexibility challenges. Instead, a separate database is recommended.

The SPIFFE Trust Domain The SPIFFE specifications introduce the concept of a *trust domain*. Trust domains are used to manage administrative and security boundaries within and between organizations, and every SPIFFE ID has the name of its trust domain embedded in it, as described above.

Concretely, a trust domain is a portion of the SPIFFE ID namespace over which a specific set of public keys is considered authoritative.

Since different trust domains have different issuing authorities, the compromise of one trust domain does not result in the compromise of another. This is an important property that enables secure communication between parties that may not fully trust each other, for example between staging and production or between one company and another.

The ability to validate SPIFFE identities across multiple trust domains is known as the SPIFFE Federation, introduced later in this chapter.

The SPIFFE Verifiable Identity Document (SVID) The SPIFFE Verifiable Identity Document (SVID) is a cryptographically-verifiable *identity document* that is used to prove a service's identity to a peer. SVIDs include a single SPIFFE ID and are signed by an issuing authority that represents the trust domain that the service resides in.

Rather than invent a new type of document that software must be taught to support, SPIFFE opts to utilize document types that are already in wide use and are well-understood. At the time of this writing, two types of identity documents are defined for use as an SVID by the SPIFFE specifications: X.509 and JWT.

X509-SVID An X509-SVID encodes a SPIFFE identity into a standard X.509 certificate. The corresponding SPIFFE ID is set as a URI type in the Subject Alternative Name (SAN) extension field. While only one URI SAN field is permitted to be set on an X509-SVID, the certificate may contain any number of SAN fields of other types, including DNS SANs.

X509-SVIDs are recommended to be used wherever possible, as they have better security properties than JWT-SVIDs. Specifically, when used in conjunction with TLS, an X.509 certificate can't be recorded and replayed by an intermediary.

Utilization of X509-SVID may have additional requirements, please refer to the X509-SVID portion of the specification.

JWT-SVID A JWT-SVID encodes a SPIFFE identity into a standard JWT—specifically a (https://tools.ietf.org/html/rfc7515) JWS. JWT-SVIDs are used as bearer tokens to prove identity to peers at the application layer. Unlike X509-SVIDs, JWT-SVIDs suffer from a class of attack known as "replay attacks" which a token is obtained and reused by an unauthorized party.

SPIFFE mandates three mechanisms to mitigate this attack vector. First, JWT-SVIDs must be transmitted over secure channels only. Second, the audience claim (or aud claim) must be set to a strict string match of the party the token was intended for. Finally, all JWT-SVIDs must include an expiration, limiting the period that a stolen token is good for.

It is critically important to note that, despite the mitigations, JWT-SVIDs are still fundamentally vulnerable to replay attacks, and should be used with caution and handled carefully. That said, they are an important part of the SPIFFE specification set as they allow SPIFFE authentication to work in scenarios where it is not possible to establish an end-to-end communication channel.

Utilization of JWT-SVID may have additional requirements, please refer to the JWT-SVID portion of the specification.

The SPIFFE Trust Bundle A SPIFFE trust bundle is a document containing a trust domain's public keys. Each SVID type has a specific way that it is represented in this bundle (e.g. for X509-SVID, CA certificates representing the public key(s) are included). Every SPIFFE trust domain has a bundle associated with it and the material in this bundle is used to validate SVIDs that claim to reside in the said trust domain.

Since the trust bundle does not contain any secrets (only public keys), it can be safely shared with the public. Despite this fact, it does need to be distributed securely to protect its contents from unauthorized modification. In other words, confidentiality is not required, but integrity is.

SPIFFE bundles are formatted as a JWK Set (or JWKS document) and are compatible with existing authentication technologies such as OpenID Connect. JWKS is a flexible and widely adopted format for representing various types of cryptographic keys and documents, which provides for some future-proofing in the event that new SVID formats are defined.

SPIFFE Federation It is often desirable to allow secure communication between services that are in different trust domains. In many cases, you can't put all your services in a single trust domain. One common example is two different organizations that need to communicate with each other.

Another might be a single organization that needs to establish security boundaries, perhaps between a less-trusted cloud environment and highly-trusted on-premises services.

To be able to accomplish this, each service must possess the bundle of the foreign trust domain where the remote service hails from. As a result, SPIFFE trust domains must expose or otherwise share their bundle contents, enabling services in foreign trust domains to validate identities from the local trust domain. The mechanism used to share a trust domain's bundle contents is known as the bundle endpoint.

Bundle endpoints are simple TLS-protected HTTP services. Operators wishing to federate with foreign trust domains must configure their SPIFFE implementation with the name of the foreign trust domain and the URL of the bundle endpoint, allowing the contents of the bundle to be periodically fetched.

Figure 14: Illustration of a company architecture with two different trust domains connected through the federation. Each SPIRE Server can only sign SVIDs for its own trust domain.

The SPIFFE Workload API The SPIFFE Workload API is a local, non-networked API that workloads use to get their current identity documents, trust bundles, and related information. Crucially, this API is *unauthenticated*, placing no requirement on the workload to have any pre-existing credential.

Providing this functionality as a local API allows SPIFFE implementations to come up with creative solutions for identifying callers without requiring direct authentication (e.g. by leveraging features provided by the operating system). The workload API is exposed as a gRPC server and uses a bi-directional stream, allowing updates to be pushed into the workload as needed.

The Workload API does not require that a calling workload have any knowledge of its own identity, or possess any credential when calling the API. This avoids the need for deploying any authentication secrets alongside the workload.

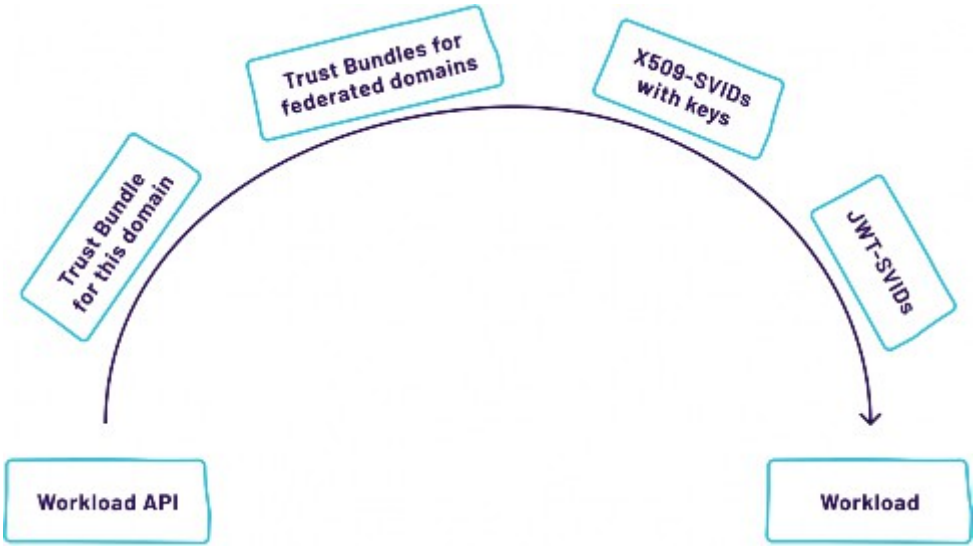


Figure 15: The Workload API provides information and facilities to leverage SPIFFE identities.

The SPIFFE Workload API delivers SVIDs and trust bundles to workloads and rotates them as necessary.

What is SPIRE?

The SPIFFE Runtime Environment (SPIRE) is a production-ready open source implementation of all five pieces of the SPIFFE specification.

The SPIRE project (as well as SPIFFE) is hosted by the Cloud Native Computing Foundation, a group founded by many of the leading infrastructure technology companies to provide a neutral home for open source projects that benefit the cloud native community.

SPIRE has two major components: the server and the agent. The server is responsible for authenticating agents and minting SVIDs, while the agent is responsible for serving the SPIFFE Workload API. Both components are written using a plugin-oriented architecture so they can easily be extended to adapt to a vast array of different configurations and platforms.

SPIRE architecture The architecture of SPIRE consists of two key components, the SPIRE Server and the SPIRE Agent.

SPIRE Server SPIRE Server manages and issues all identities in a SPIFFE trust domain. It uses a data store to hold information about its agents and workloads, among other things. SPIRE Server is informed of the workloads it manages through the use of registration entries, which are flexible rules for assigning SPIFFE IDs to nodes and workloads.

The server can be managed either via API or CLI commands. It is important to note that since the server is in possession of SVID signing keys, it is considered a critical security component. Special consideration should be made when deciding on its placement. This is discussed later in this book.

Data stores

The SPIRE Server uses a *data store* to keep track of its current registration entries as well as the status of the SVIDs it has issued. Currently, several different SQL databases are supported. SPIRE is packed with SQLite, an in-memory embedded database, for development and testing purposes.

Upstream authorities

All SVIDs in a trust domain are signed by the SPIRE Server. By default, the SPIRE Server generates a self-signed certificate (a certificate signed with its own randomly generated private key) to sign SVIDs unless an *Upstream Certificate Authority* plugin interface is configured. The plugin interface called *Upstream Certificate Authorities* allows SPIRE to obtain its signing certificate from another certificate authority.

In many simple cases, it's fine to use a self-signed certificate. However, for larger installations, it may be desirable to take advantage of preexisting Certificate Authorities and the hierarchical nature of X.509 certificates to make multiple SPIRE Servers (and other software that generates X.509 certificates) work together.

In some organizations, the upstream certificate authority might be a central certificate authority that your organization uses for other purposes. This is useful if you have many different kinds of certificates in use, and you want them all to be trusted throughout your infrastructure.

SPIRE Agent SPIRE Agent has just a single function, albeit a very important one: to serve the Workload API. In the course of accomplishing this feat, it solves some related problems such as determining the identity of workloads, calling it, and securely introducing itself to the SPIRE Server. In this arrangement, it is the agent that performs all of the heavy lifting.

Agents do not require active management in the way that SPIRE Servers do. While they do require a configuration file, SPIRE Agents receive information about the local trust domain and the workloads that might call it directly from the SPIRE Server. When defining new workloads in a given trust domain, the records are simply defined or updated in a SPIRE Server, and information about the new workload propagates to the appropriate agents automatically.

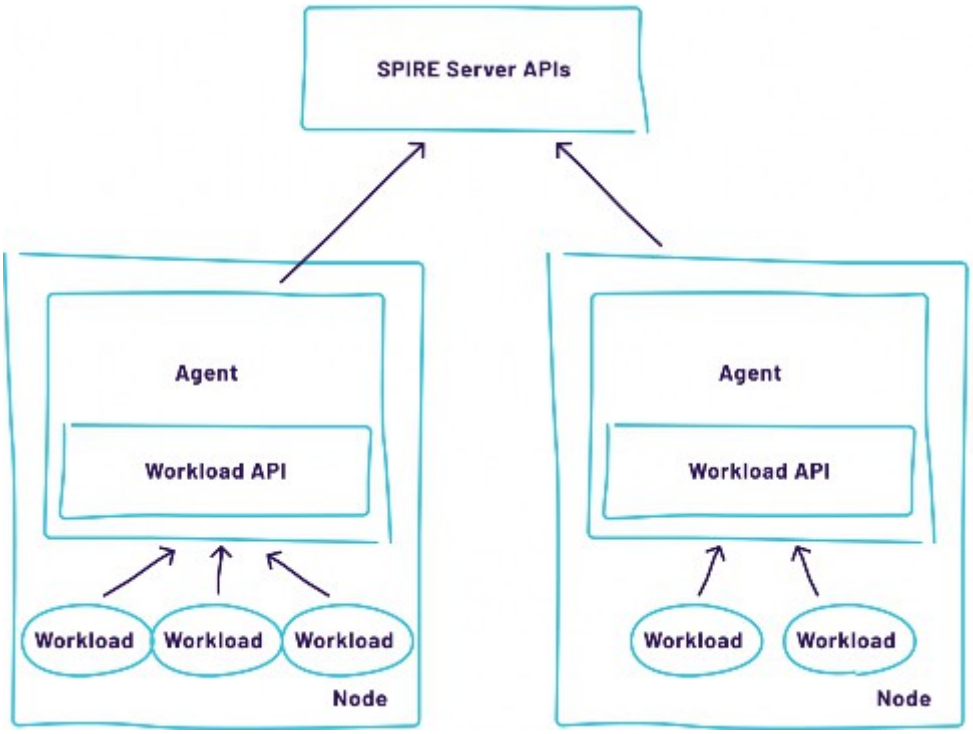


Figure 16: SPIRE Agent exposes the SPIFFE Workload API and works together with SPIRE Servers to issue identities to the workloads calling the agent.

Plugin architecture

SPIRE is built as a set of plugins so that it can easily grow to accommodate new node attestors, workload attestors, and upstream authorities.

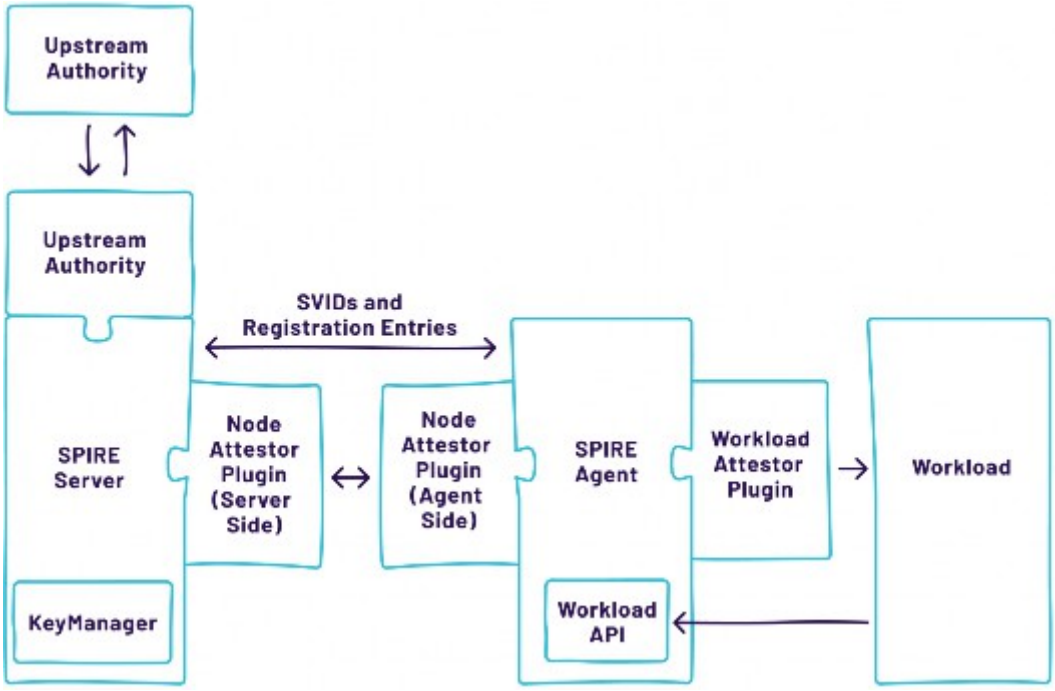


Figure 17: Illustration of the key plugin interfaces supported by SPIRE. The server includes Node Attestor, KeyManager, and Upstream Authority plugins, while the Agent side includes Node Attestor and Workload Attestor plugins.

SVID management SPIRE Agent uses its identity that it obtains during node attestation to authenticate to the SPIRE Server and obtains SVIDs for the workloads it is authorized to manage. Since SVIDs are time-limited, the agent is also responsible for renewing SVIDs as needed and communicating those updates to the relevant workloads. The trust bundle also rotates and receives bundles, and those updates are tracked by the agents and communicated to workloads. The agent maintains an in-memory cache of all this information, so SVIDs can be served even if SPIRE Server is down, and also ensures Workload API responses are performant by negating the need for a roundtrip to the server when someone calls the Workload API.

Attestation Attestation is the process through which information about workloads and their environment is discovered and asserted. In other words, it is the process to prove with certainty the identity of a workload, using available information as evidence.

There are two flavors of attestation in SPIRE: node and workload attestation. Node attestation asserts attributes that describe nodes (e.g. member of a particular AWS auto-scaling group, or which Azure region the node resides in), and workload attestation asserts attributes that describe the workload (e.g. the Kubernetes Service Account it's running in, or the path of the binary on disk). The representation of these attributes in SPIRE is referred to as *selectors*.

SPIRE supports dozens of selector types out of the box, and the list continues to grow. As of the time of this writing, the list of node attestors includes support for bare metal, Kubernetes, Amazon Web Services, Google Cloud Platform, Azure, and more. Workload attestors include support for Docker, Kubernetes, Unix, and others.

Additionally, SPIRE's pluggable architecture allows operators to easily extend the system to support additional selector types as they see fit.

Node Attestation Node attestation occurs when an agent starts for the first time. In node attestation, the agent contacts the SPIRE Server and enters into an exchange in which the server aims to positively identify the node the agent is running on and all its related selectors. To accomplish this, a platform-specific plugin is exercised in both the agent and the server. For example, in the case of AWS, the agent plugin collects information from AWS that only that specific node has access to (a document signed by an AWS key), and passes it to the server. The server plugin then validates the AWS signature and makes further calls to AWS APIs to both assert the accuracy of the claim, as well as gather additional selectors about the node in question.

Successful node attestation results in the issuance of identity to the agent in question. The agent then uses this identity for all further server communication.

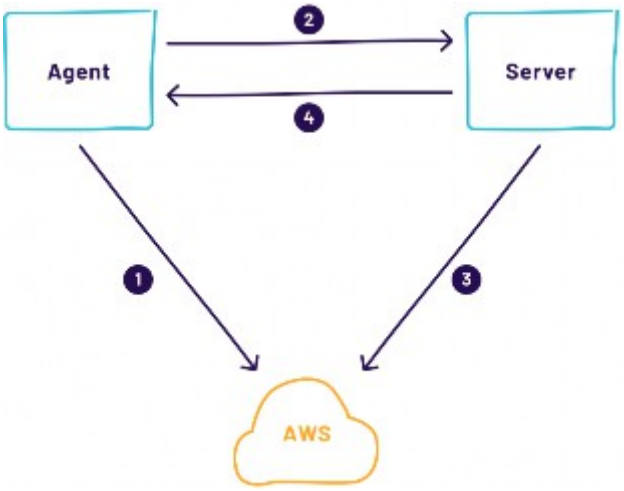


Figure 18: Node attestation of a node running in AWS.

1. The agent gathers proof of the node's identity calling an AWS API.
2. The agent sends this proof of identity to the server.
3. The server validates proof of identity obtained in step 2 by calling out to the AWS API and then creates a SPIFFE ID for the agent.

Workload attestation Workload attestation is the process of determining the workload identity that will result in an identity document being issued and delivered. The attestation occurs any time a workload calls and establishes a connection to the SPIFFE Workload API (on every RPC call a workload makes to the API), and the process from there on is driven by a set of plugins on the SPIRE Agent.

The moment the agent receives a new connection from a calling workload, the agent will leverage operating system features to determine exactly which process has opened the new connection. The operating system features leveraged will be dependent on the operating system the agent is running on. In the case of Linux, the agent will make a system call to retrieve the process ID, the user identifier and the globally unique identifier of the remote system calling on the particular socket. The kernel metadata requested will be different in BSD and Windows. The agent, in turn, will provide the attestor plugins with the ID of the calling workload. From there, attestation fans out across its plugins, providing additional process information about the caller and returning it to the agent in the form of selectors.

Each attestor plugin is responsible for introspecting the caller, generating a set of selectors that describe it. For example, one plugin may look at kernel level details and generate selectors such as the user and group that the process is running as, while another plugin may communicate with Kubernetes and generate selectors such as the namespace and service account that the process is running in. A third plugin may communicate with the Docker daemon and generate selectors for Docker image ID, Docker labels, and container environment variables.

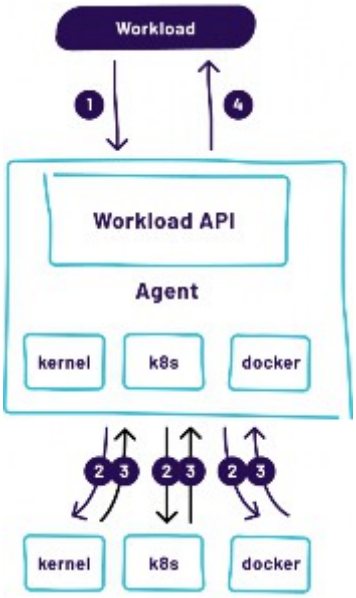


Figure 19: Workload attestation.

1. A workload calls the Workload API to request an SVID.
2. The agent interrogates the node’s kernel to get the attributes of the calling process.
3. The agent gets the discovered selectors.
4. The agent determines the workload’s identity by comparing discovered selectors to registration entries and returns the correct SVID to the workload.

Registration entries For SPIRE to issue workload identities, it must first be taught about the workloads expected or allowed in its environment; what workloads are supposed to run where, what their SPIFFE IDs and general shape should be. SPIRE learns this information via *registration entries*, which are objects that are created and managed using SPIRE APIs that contain the aforementioned information.

For each registration entry, there are three core attributes. The first is known as the Parent ID—this effectively tells SPIRE *where* a particular workload should be running (and, by extension, which agents are authorized to ask for SVIDs on its behalf). The second is a SPIFFE ID—when we see this workload, what SPIFFE ID should we issue it?

And finally, SPIRE needs some information that helps it to identify the workload, which is where the selectors discovered from attestation come in.

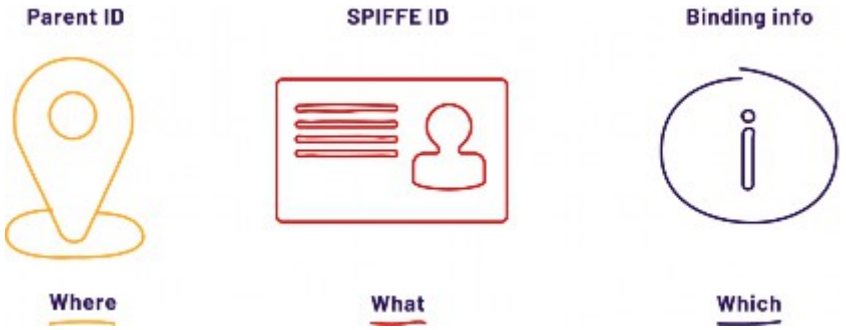


Figure 20: Three core attributes of registration entries.

Registration entries bind SPIFFE IDs to the nodes and workloads that they are meant to represent.

A registration entry can describe either a group of nodes or a workload, where the latter often references the former through the use of a Parent ID.

Node entries Registration entries that describe a node (or a group of nodes) use selectors generated by node attestation to assign a SPIFFE ID, which can be referenced later when registering workloads. A single node may be attested to have a set of selectors that match multiple node entries, allowing it to participate in more than one group. This affords a great deal of flexibility when deciding exactly where a given workload is permitted to run.

SPIRE ships with a variety of node attestors ready to use and each one generates platform-specific selectors. While SPIRE Server supports loading multiple node attestor plugins at once, SPIRE Agent supports loading only one. Some examples of available node selectors are:

- On Google Cloud Platform (GCP),
- On Kubernetes, the name of the Kubernetes cluster the node is part of
- On Amazon Web Services (AWS), the AWS Security Group of the node

Node entries have their Parent ID set to the SPIFFE ID of the SPIRE Server, as it is the server which is performing attestation and asserting that the node in question does indeed match the selectors defined by the entry.

Workload entries Registration entries that describe a workload use selectors generated by workload attestation to assign a SPIFFE ID to workloads when a certain set of conditions are met. When the Parent ID and selectors conditions are met, the workload can receive a SPIFFE ID.

The Parent ID of a workload entry describes *where* this workload is authorized to run. Its value is the SPIFFE ID of a node or set of nodes. SPIRE Agents running on the node(s) receive a copy of this workload entry, including the selectors that must be attested before issuing an SVID for that particular entry.

When a workload calls the agent, the agent performs workload attestation and cross-references the discovered selectors with the selectors defined in the entry. If a workload possesses the entire set of defined selectors, then the conditions are met and the workload is issued an SVID with the defined SPIFFE ID.

Unlike node attestation, SPIRE Agent supports loading many workload attestor plugins simultaneously. This allows mix-and-match selectors in workload entries. For example, a workload entry may require that a workload is in a specific Kubernetes namespace, have a specific label applied to its Docker image, and have a specific SHA sum.

SPIFFE/SPIRE Applied Concepts Threat Model

The specific set of threats that SPIFFE and SPIRE face are situational. Understanding the general threat model of SPIFFE/SPIRE is an important step in asserting that your specific needs can be met, and discovering where further mitigation may be necessary.

In this section, we will describe the security boundaries of both SPIFFE and SPIRE and the impact of compromise of each component in the system. Later in the book, we'll cover specific security considerations imposed by different SPIRE deployment models.

Assumptions SPIFFE and SPIRE are intended to be used as the foundation for distributed identity and authentication that is consistent with cloud native design architectures. SPIRE supports Linux and the BSD family (including MacOS). Windows is not currently supported, though some early prototyping has been done in this area.

SPIRE adheres to the zero trust networking security model in which it is assumed that network communication is hostile or presumably fully compromised. That said, it is also assumed that the hardware on which SPIRE components run, as well as its operators, are trustworthy. If hardware implants or insider threats are part of the threat model, careful considerations should be made around the physical placement of SPIRE Servers and the security of their configuration parameters.

There may further be implied trust in third-party platforms or software, depending on the chosen methods of node and workload attestation. Asserting trust through multiple independent mechanisms provides a greater assertion of trust. For example, leveraging AWS or GCP-based node attestation implies that the compute platform is assumed to be trustworthy, and leveraging Kubernetes for workload attestation implies that the Kubernetes deployment is assumed to be trustworthy. Due to the great variety of ways that attestation can be accomplished, and the fact that the SPIRE architecture is fully pluggable, the security (and associated assumptions) of these processes are not considered in this assessment. Instead, they should be evaluated on a case-by-case basis.

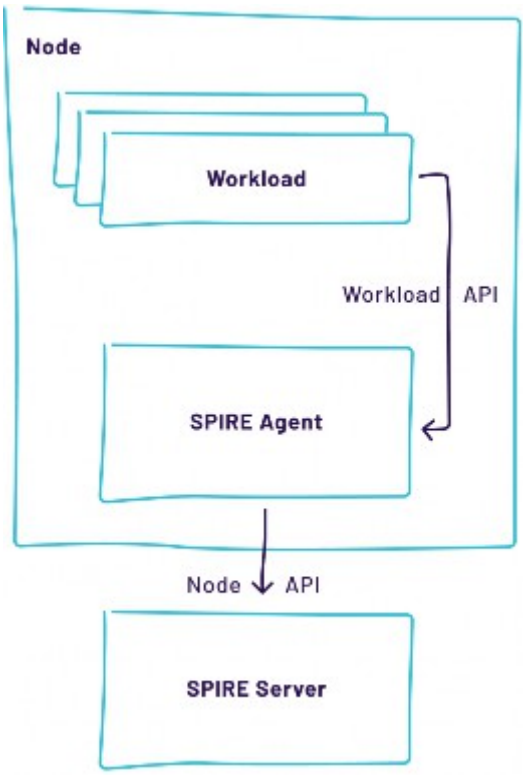


Figure 21: Components considered as part of the threat model.

Security boundaries Security boundaries are formally understood as the line of intersection between two areas of differing levels of trust.

There are three major security boundaries defined by SPIFFE/SPIRE: one between workloads and agents, one between agents and servers, and another between servers in different trust domains. In this model, workloads are fully untrusted, as are servers in other trust domains and, as mentioned previously, network communication is always fully untrusted.

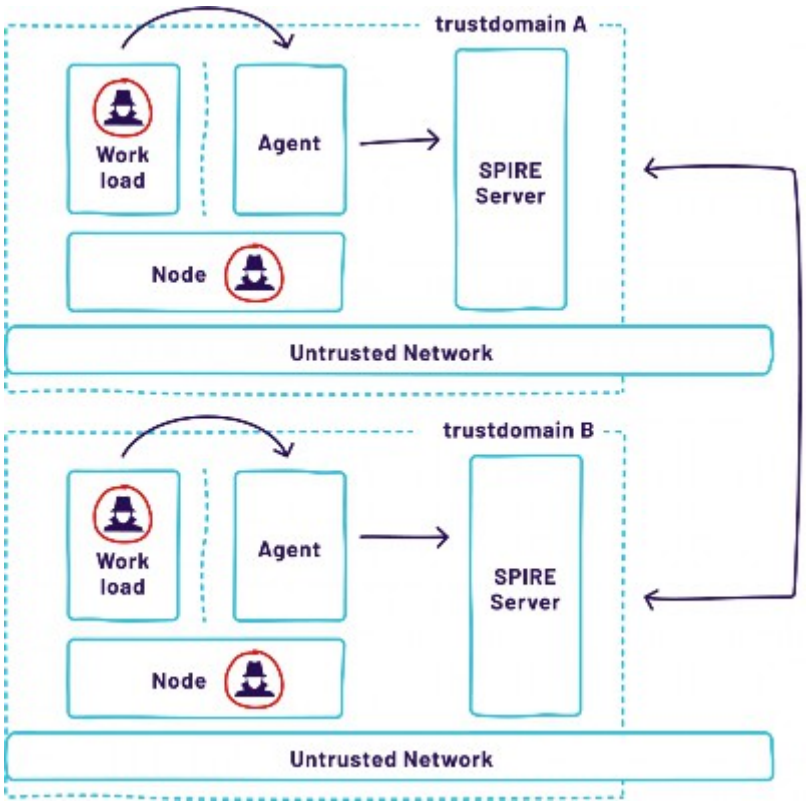


Figure 22: SPIFFE/SPIRE security boundaries.

The workload | agent boundary As one moves through the system and across these boundaries, the level of trust slowly increases. Starting with workloads, we move across a security boundary to the agent. It is generally expected (though not required) that a security mechanism beyond SPIRE design exists between the workload and the agent, for example by leveraging Linux user permissions and/or containerization.

The agent does not trust the workload to give any kind of input. All assertions made by the agent about the workload's identity occur through out-of-band checks. In the context of workload attestation, this is an important detail—any selector whose value can be manipulated by the workload itself is inherently insecure.

The agent | server boundary The next boundary exists between the agent and the servers. Agents are more trustworthy than workloads but less trustworthy than servers. An explicit design goal of SPIRE is that it should be able to survive node compromises. Since workloads are fully untrusted, we are only one or two attacks away from node compromise at any given point in time. Agents have the ability to create and manage identities on the workload's behalf, but it is also necessary to limit the power of any given agent to only what is strictly necessary for it to complete its task (following the principle of least privilege).

To mitigate the impact of node (and agent) compromise, SPIRE requires knowledge of where a particular workload is authorized to run (in the form of a Parent ID). Agents must be able to prove ownership of a registration entry before they can obtain an identity for it. As a result, compromised agents are not able to obtain arbitrary identities—they may only obtain the identities of workloads that should be running on the node in the first place.

It is worth noting that communications between the SPIRE Server and SPIRE Agent can use TLS and mutual TLS at different points in time during the node attestation process depending on whether the node has yet to be attested or if the agent already has a valid SVID and can use it for mutual TLS, at which point all communications between server and agent are secure.

The server | server boundary The final boundary exists between servers in different trust domains. SPIRE Servers are trusted only to mint SVIDs within the trust domain they directly manage. When SPIRE Servers federate with each other and exchange public key information, the keys they receive remain scoped to the trust domain they were received from. Unlike Web PKI, SPIFFE does not simply throw all the public keys in a big mixed bag. The result is that if compromises in foreign trust domains do not result in the ability to mint SVIDs in the local trust domain.

It should be noted that SPIRE Servers do not have any multi-party protection. Every SPIRE Server in a trust domain has access to signing keys with which it can mint SVIDs. The security boundary that exists between servers is strictly limited to servers of different trust domains and does not apply to servers within the same trust domain.

The impact of component compromise While workloads are always considered to be compromised, it is expected that agents are generally not. If an agent is compromised, the attacker will be able to access any identity that the respective agent is authorized to manage. In deployments where there is a 1:1 relationship between workload and agent, this is of less concern. In deployments where agents manage multiple workloads, this is an important point to understand.

Agents are authorized to manage an identity when they are referenced as a parent of that identity. For this reason, it is a good idea to scope registration entry Parent IDs as tightly as is reasonably possible.

In the event of a server compromise, it can be expected that the attacker will be able to mint arbitrary identities within that trust domain. SPIRE Server is undoubtedly the most sensitive component of the entire system. Care should be taken in the management and placement of these servers. For example, SPIRE solves for node compromise as workloads are untrusted, but if SPIRE Servers run on the same host as the untrusted workloads, then the servers no longer enjoy the protection that was once afforded by the agent/server security boundary. Therefore, it is strongly recommended that SPIRE Servers be placed on hardware that is distinct from the untrusted workloads they are meant to manage.

The Agent caveat SPIRE accounts for node compromise by scoping the privileges of an agent to only the identities it is directly authorized to manage… but if an attacker can compromise multiple agents, or perhaps *all* agents, the situation is decidedly much worse.

SPIRE Agents do not have any communication pathway between each other, significantly limiting the possibility of lateral movement between agents. This is an important design decision that is intended to mitigate the impact of a possible agent vulnerability. However, it should be understood that certain configurations or deployment choices may undermine this mitigation in part or whole. For example, SPIRE Agent supports exposing a Prometheus metrics endpoint, however, if all agents expose this endpoint and vulnerability exists there, then lateral movement becomes trivial unless adequate network-level controls are in place. For this reason, exposing the SPIRE agent to incoming network connections is strongly discouraged.

5. Before You Start

This chapter is designed to prepare you for the many decisions you will need to make when rolling out SPIFFE/SPIRE.

Prepare the Humans

If you've read the previous chapters, you must be very excited to get started using SPIRE to manage identity in a way that can be leveraged across many different types of systems and all your organization's services. However, before you begin you need to consider that deploying SPIRE is *a major infrastructure change* that has the potential to affect many different systems. This chapter is about how to start planning a SPIRE deployment: getting buy-in, enabling SPIRE support non-disruptively, and then using it to implement new security controls.

Assemble the crew and identify other stakeholders To deploy SPIRE, you'll need to identify stakeholders from the security, software development, and DevOps teams. Who will maintain the SPIRE Servers themselves? Who will deploy the agents? Who will write the registration entries? Who will integrate SPIFFE functionality into the applications? How will it impact existing CI/CD pipelines? If a service interruption occurs, who will fix it? What are the performance requirements and service level objectives?

In this book, as well as many public blog posts and conference talks, there are examples of organizations that have successfully deployed SPIRE that can serve both as a pattern to follow and as helpful material to proselytize SPIRE to your colleagues.

State your case and get buy-in SPIRE cross-cuts several different traditional information technology silos, so expect to see more cross-organizational collaboration among your DevOps teams, software development teams, and security teams. It is important that they work together to ensure a successful and seamless deployment. Consider that each of these teams has different needs and priorities, that will need to be addressed to get their buy-in.

While planning a SPIRE deployment, you will need to understand what outcomes matter the most to your business and frame these as drivers for the project and the value of the solution you will deliver.

Each team needs to see the benefits of SPIRE to themselves as well as to the business as a whole. Many of the benefits of a SPIRE deployment are described in “Chapter 2: Benefits of this book” and in this section, we will distill some of these benefits down into compelling arguments.

Compelling arguments to security teams Reducing the security team's workload is one very persuasive case for deploying SPIRE: instead of deploying ad hoc security solutions, and managing hundreds or thousands of certificates manually, they can focus on designing the right registration entries to make sure each service gets the right identity.

A more long-term benefit is that SPIRE can increase the overall security posture of the organization, as SPIRE has no credentials that can easily be stolen or misused. A large range of attacks related to misappropriation or misrepresentation of credentials, as well as sensitive data exposure, are mitigated. It is possible to prove to an outside auditor that the right services are communicating securely with each other, with no possibility of accidental oversights. Even if an outsider can compromise one service, their ability to launch attacks on other services is limited.

Compelling arguments to software development teams For application development teams, their ability to move faster by not waiting on tickets or manual workflows to provision certificates is the most compelling case. If they are currently manually deploying secrets alongside their code and getting talked to by their security teams, they no longer have to endure that. They also don't need to manage secrets in a secret store.

A secondary benefit is that software components may be able to directly communicate in ways they couldn't do securely before. If cloud services can't access a critical database or essential cloud service because there's no way to do it securely, it may be possible to use SPIFFE identities to create a secure connection, providing new architecture potential for your teams.

Compelling arguments to DevOps teams The greatest gains of deploying SPIRE are for DevOps teams. If each service has its own secure identity, then services can be deployed anywhere—in any on-premises data center, cloud provider, or region within one cloud provider. This new flexibility allows lowered cost, higher scalability, and improved reliability since deployment decisions can be made independently of security requirements.

Another key benefit for DevOps teams is that incoming requests to each service are all tagged with a SPIFFE ID, which can be logged, measured, and reported to a monitoring system. This is extraordinarily helpful for performance management in large organizations with hundreds or thousands of services.

Create a Plan

The first goal in planning a SPIRE deployment is to determine whether every service needs to be SPIFFE-aware, or whether ‘islands’ of non-SPIFFE services can still satisfy requirements. Moving every service to SPIFFE is the most straightforward option, but it might be challenging to implement all at once, especially in very large organizations.

Planning for islands and bridges Some environments are complex with either multiple organizations represented or a combination of legacy and new development. In this scenario, there is often a desire to make only a subset of the environment SPIFFE-enabled. Two options need to be considered, depending on the level of integration between systems and the complexity across them. Let’s take a look at these two architectures, we’ll call them ‘Independent Islands’ and ‘Bridged Islands’.

Each island is considered its own trust domain and on each island are workloads or ‘residents’.

Independent islands The independent island model allows individual trust domains to operate independently of one another. This is often the easiest option because each island can run SPIRE in a way that makes sense for that island.

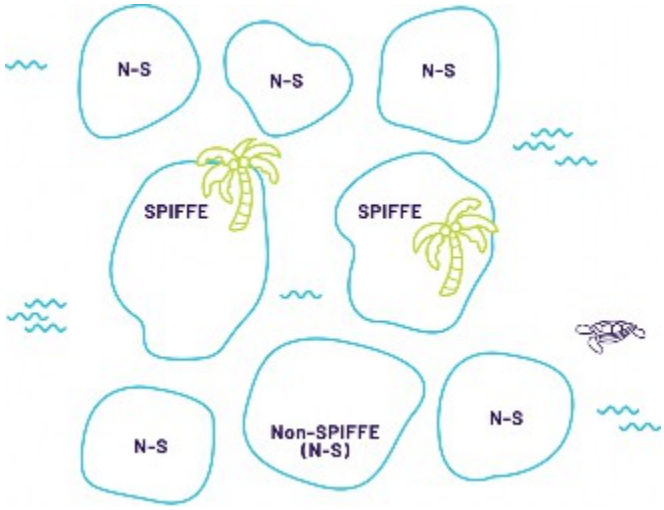


Figure 23: Here there are two independent SPIFFE deployments (Independent Islands).

Bridged islands The bridged islands model allows a non-SPIFFE service on a non-SPIFFE island to talk to a gateway. The gateway then forwards the request on to the SPIFFE-enabled island resident it is intended for, we'll call them Zero. From Zero's perspective, the gateway sent the request. Zero and his friends from the SPIFFE-enabled islands can authenticate to the gateway and send messages to services on the non-SPIFFE island.

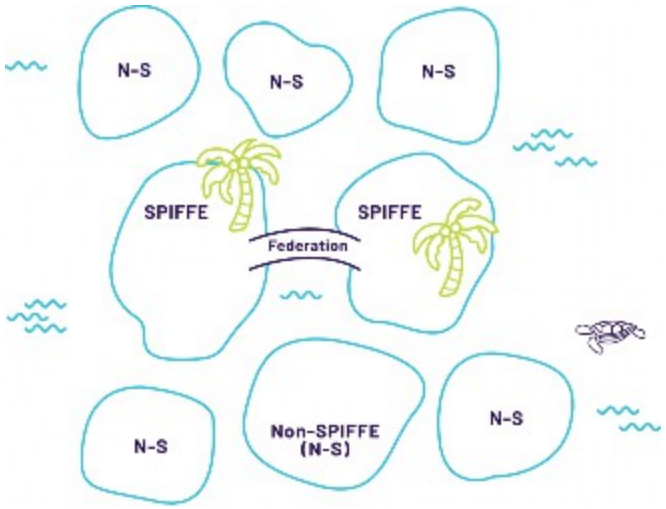


Figure 24: Here we have two independent SPIFFE deployments bridged by Federation, enabling services from each island to trust the other and thereby communicate. There is still no communication between SPIFFE and non-SPIFFE islands.

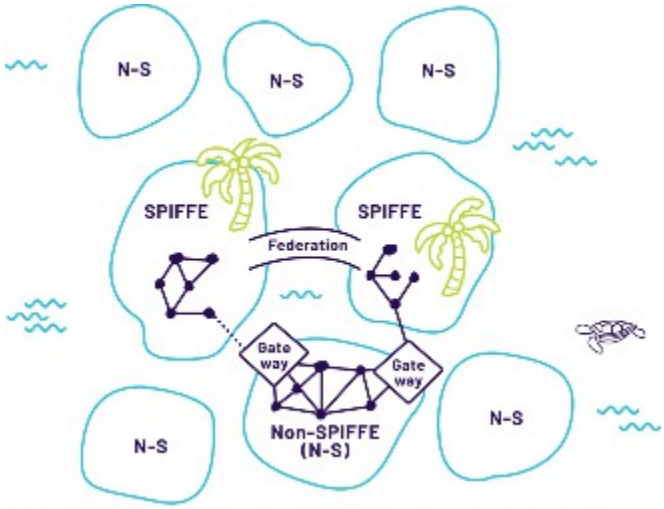


Figure 25: Adding gateways to a non-SPIFFE island is a way to bridge SPIFFE and non-SPIFFE islands.

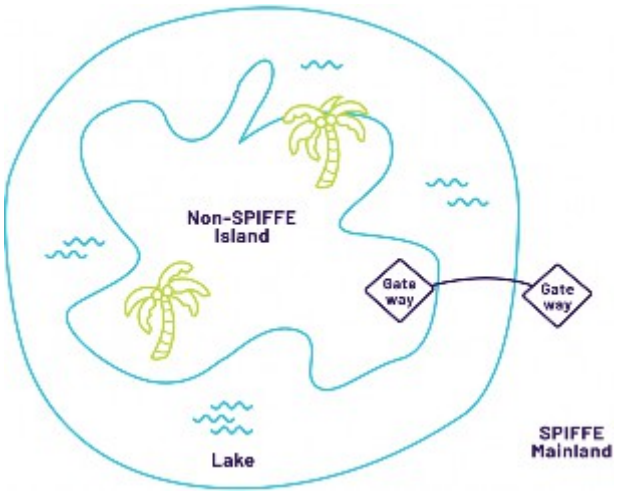


Figure 26: In this diagram, there is a SPIFFE-enabled ecosystem (the mainland), and within that ecosystem, there is a pocket of non-SPIFFE services (the island on the lake). For services on the mainland and island to talk to each other, there needs to be a gateway.

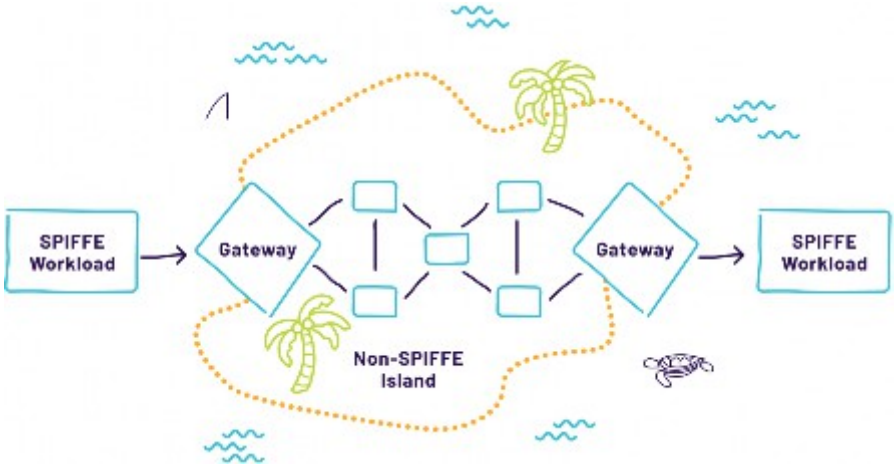


Figure 27: Bridge islands architecture.

With a Bridged Islands architecture, gateways are created on non-SPIFFE-enabled islands. There are many reasons these non-SPIFFE islands may not be able to easily adopt a SPIFFE architecture: there may be legacy software that cannot be easily modified or updated; the island may be using its own identification ecosystem, such as Kerberos or one of the other options described in the “Comparing SPIFFE to Other Technologies” chapter; or the system may be running workloads on technologies not well suited to the models of existing SPIFFE solutions such as SPIRE.

In these cases, it can be useful to use a *gateway* service to bridge the connection between the SPIFFE world and non-SPIFFE island. When a SPIFFE-enabled workload wants to talk to a workload in the non-SPIFFE island, it creates an authenticated connection with the gateway which then creates a connection to the target workload. This connection to the target workload may be unauthenticated or use the non-SPIFFE identity solution for that island. Similarly, when a workload from the non-SPIFFE-enabled island wants to connect to a SPIFFE-enabled workload, the non-SPIFFE workload connects to the gateway which then creates a SPIFFE-authenticated connection to the target SPIFFE-enabled workload.

In this scenario, the authentication that happens between the gateway and the SPIFFE-enabled workload is *terminated* at the gateway. This means that the SPIFFE-enabled workload can verify that it is talking to the appropriate gateway, but cannot verify that it is talking to the correct workload on the other side of it. Similarly, the target workload only knows that the gateway service has sent it a request, but loses the authentication context of the original SPIFFE-enabled workload.

This model allows these complicated organizations to begin adopting SPIFFE without having to convert all at once.

In cases when requests and workflows pass *through* a non-SPIFFE island, it can be useful to utilize JWT-SVIDs for propagating across requests. Even better, you can use X509-SVIDs to sign documents (such as HTTP Message Request signing) rather than only using service-to-service mutually authenticated TLS so that the authenticity of the entire message can be validated by SPIFFE-enabled workloads on the other side. This is especially useful for islands that are known to have weak security properties since it provides confidence that messages passed through the intermediate ecosystem have not been manipulated.

Documentation and instrumentation When preparing to embark on a roll out, it is important to instrument services so that metrics and flow logs are emitted in a way that:

- The people overseeing the roll out know which (and how many) services are SPIFFE-enabled and which (and how many) are not.
- A Client Author knows which services they call are SPIFFE-enabled and which are not.
- A Service Owner knows which and how many of their clients are calling the SPIFFE-enabled endpoint, and which are calling the legacy endpoint.

It is important to prepare for the roll out by creating reference documentation for client and server implementers that anticipates the kind of support requests you will receive.

It is also important to create tooling to assist in common debug and troubleshooting tasks. Recalling the benefits of SPIFFE and SPIRE, introducing SPIFFE to your organization should empower developers and remove roadblocks. Leaving stakeholders with an *impression* that you are adding work or creating friction will ultimately slow or halt broader adoption. To curtail this, and ensure that documentation and tooling cover the appropriate topics, we suggest the following preparatory steps:

Step	
Decide what security features you will need SPIFFE for	SPIFFE identities can be used to create mutual TLS connections, for authorization, or other functionality, such as audit logs
Determine what formats of SVIDs to use and for what purpose.	It is most common to use X509-SVIDs for mutual TLS, but determine whether this applies and whether SVIDs will be used for any other applications.
Determine the number of workloads that will require identities	Not every workload needs identities, especially early on
Determine the number of separate trust domains that are needed	Each trust domain needs its own SPIRE Server deployment. Details on making this decision are in the next chapter.
Determine what languages, frameworks, IPC technologies, etc. are in use at your organization that will need to be SPIFFE-compatible	If using X.509-SVIDs for mutual TLS, determine what web servers are in use in your organization (Apache HTTPD, NGINX, Tomcat, Jetty, etc.) and what client libraries are in use. If client libraries are expecting to perform DNS hostname verification, make sure your SPIFFE deployment is compatible with this expectation.

Understanding performance implications Performance implications should be considered as part of your deployment planning.

As part of your roll out preparation, you should check benchmarks of a range of workloads that are representative of a variety of applications that your organization runs in production. This ensures that you are at least aware of, and hopefully prepared to address, any performance issues that may arise during the rollout.

TLS performance In many organizations, the first concern that developers and operation teams raise is that establishing mutual TLS connections between services will be too slow. On modern hardware, with modern TLS implementations, the performance impact of TLS is minimal:

> "On our production frontend machines, SSL/TLS accounts for less than 1% of the CPU load, less than 10 KB of memory per connection, and less than 2% of network overhead. Many people believe that SSL/TLS takes a lot of CPU time and we hope the preceding numbers will help to dispel that." — Adam Langley, Google, Overclocking SSL, 2010

> "We have deployed TLS at a large scale using both hardware and software load balancers. We have found that modern software-based TLS implementations running on commodity CPUs are fast enough to handle heavy HTTPS traffic load without needing to resort to dedicated cryptographic hardware." — Doug Beaver, Facebook, HTTP/2 Expression of Interest, 2012

In general, performance implication depends on multiple factors, including network topology, API gateways, L4-L7 firewalls, and many others. Also, the protocols you are using and their implementation and certificate and key sizes might affect performance, so it is a pretty broad topic to cover.

The table below provides data points about overhead for two different stages compared to TCP, specifically for handshake and data transfer phases.

TLS Phase	Protocol Overhead	Latency	CPU	Memory
Handshake	2 kB for TLS; 3 kB for mTLS; +1 kB/add'l Cert	12-17 mS	~0.5% more than TCP	<10 kB/conn
Data Transfer	22B/packet	<3 uS	<1% more than TCP	<10 kB/conn

Making the Change to SPIFFE and SPIRE

There is a rich history of study into how organizations react to, effect, and process change. There have also been many interesting studies about the acceptance and adoption of new technologies both within the general public and within organizations. To do any real justice to these topics is beyond the scope of this book, but we would be remiss if we didn't mention these topics due to their relevance to a successful SPIFFE roll out.

Convincing change to occur There are several ways to convince others a change must occur within your organization. The below list outlines ways in which you can pursue this change with SPIFFE and SPIRE.

- Perceived usefulness–how useful does someone think SPIFFE will be in helping them enhance their job performance. The ability to demonstrate tangible results helps to improve perceived usefulness.
- Perceived ease-of-use–how easy to use will one think SPIFFE is. Dedication to the user experience of developers and operators is essential.
  - Peer influence–someone's perception of esteemed other's views on the adoption of SPIFFE and whether they have adopted it. This is where having accumulated political capital within the organization pays off. Often convincing the right people is more important than trying to convince everyone.
  - Image–the degree to which adopting SPIFFE will enhance someone's status within the organization.
  - Voluntariness–the extent to which potential adopters of SPIFFE perceive the adoption to be voluntary or mandatory. The effect this has depends on company culture and individual personalities. Keep this in mind when facing 'Forced' Adopters and Holdouts (covered later in this chapter).

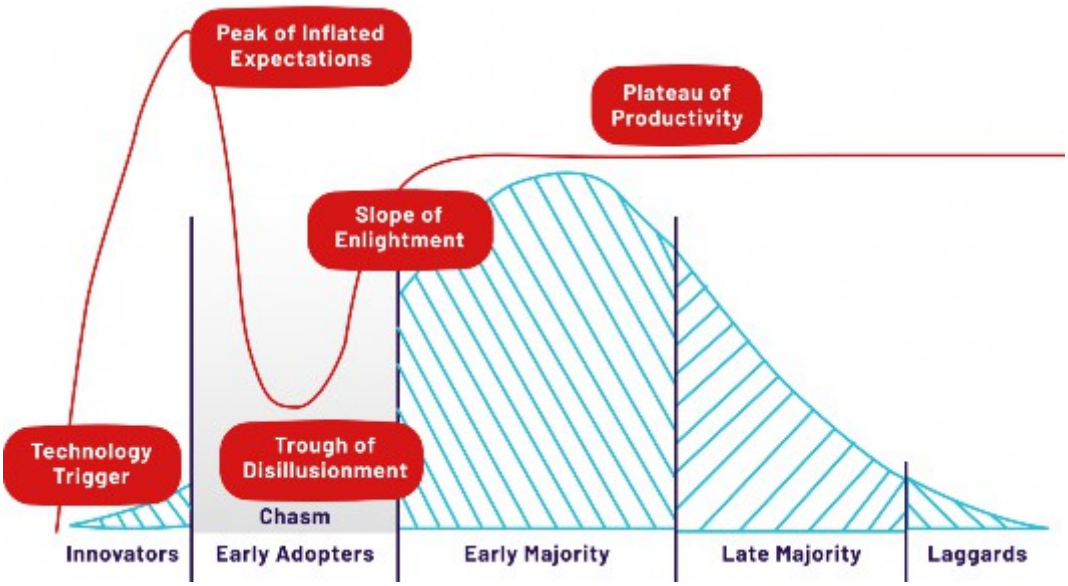


Figure 28: Technology adoption curve (adapted from Roger's bell curve and Gartner's Hype Cycle). The blue area under the graph represents the amount of change and number of SPIFFE adopters. The red line represents enthusiasm and expectations of adopting SPIFFE.

Adoption actors Adoption actors correspond with the technology curve and can help set expectations for how making the change to SPIFFE and SPIRE will happen. Here we've listed adoption actors covered in the technology curve and added two more you are likely to face.

More information on adoption actors in the technology adoption curve is available outside of this book.

> **Sidenote**: An excellent read on adoption actors: Technology Adoption Curve: Traits of Adopters at Each Stage of the Lifecycle.

Innovators Consider yourself the innovator in your organization for taking these steps to read this book, getting this far, and deciding to move forward. You are essentially pioneering the process for adding SPIFFE and SPIRE to your architectures and you need help! A 'white-glove' level of support and hand-holding is recommended so be sure to pick volunteers from the *low-hanging fruit* and precursor categories (covered below) with whom you have a good rapport.

Early adopters It is important to take the lessons learned from the 'white-glove' 'hand-holding' level of support given to the Innovators and distill those lessons into easily accessible and understandable documentation, useful tools, and a scalable support channel. Considerable work may need to be done to SPIFFE-enable 'precursors and enablers' (detailed later in this chapter) so that developers become unblocked and can then SPIFFE-enable their services and clients.

Early and late majority By the time you get to onboarding the early majority of services, the process of SPIFFE-enablement is a well-oiled machine. All commonly used enablers such as CI/CD, workflow engines, orchestrators, container platforms, and service meshes, should be SPIFFE-enabled to ensure that application developers have support through the application lifecycle no matter how the application is run.

Laggards Your organization likely has conservative laggards due to team culture, individual personality, and regulatory or compliance requirements. It is important to not jump to conclusions as to why a service owner falls into this category but to investigate the root cause and address it appropriately.

'Forced' converts The last client to adopt SPIFFE of a SPIFFE-enabled service may feel forced to convert. It is important to be prepared for forced converts to make sure that their experience of adopting SPIFFE is a positive one.

Holdouts They will occur, so make adoption easy and incentivized. Highlight examples of others currently enjoying the Plateau of Productivity. You should expect to provide extra support and hand-holding as holdouts are walked through the process.

Considerations for picking who goes when Keeping maximum compatibility across your organizations is critical to consider as you select who goes when. Services should keep their existing API interfaces and ports as is, and introduce their SPIFFE-enabled APIs on new ports. This enables a smooth transition and facilitates a rollback if needed. Service teams with clients from many other service teams should expect to maintain and support both endpoints for an extended period (> 6 months).

Once all of the clients of a service become SPIFFE-enabled and the non-SPIFFE APIs are no longer used, then the non-SPIFFE APIs can be turned off.

> **Warning**: Be careful that you don't turn off legacy endpoints prematurely. Pay special attention to batch jobs, scheduled tasks, and other kinds of infrequent or irregular call patterns. You don't want to be the person who caused an end-of-quarter or end-of-financial-year reconciliation job to fail.

If your environment is too big or complex to do an all at once approach, it is important to be thoughtful when choosing the order in which services become SPIFFE-enabled. It can be helpful to think about it in terms of *big rocks* , *lowest hanging fruit*, and "precursors and enablers" to accelerate adoption.

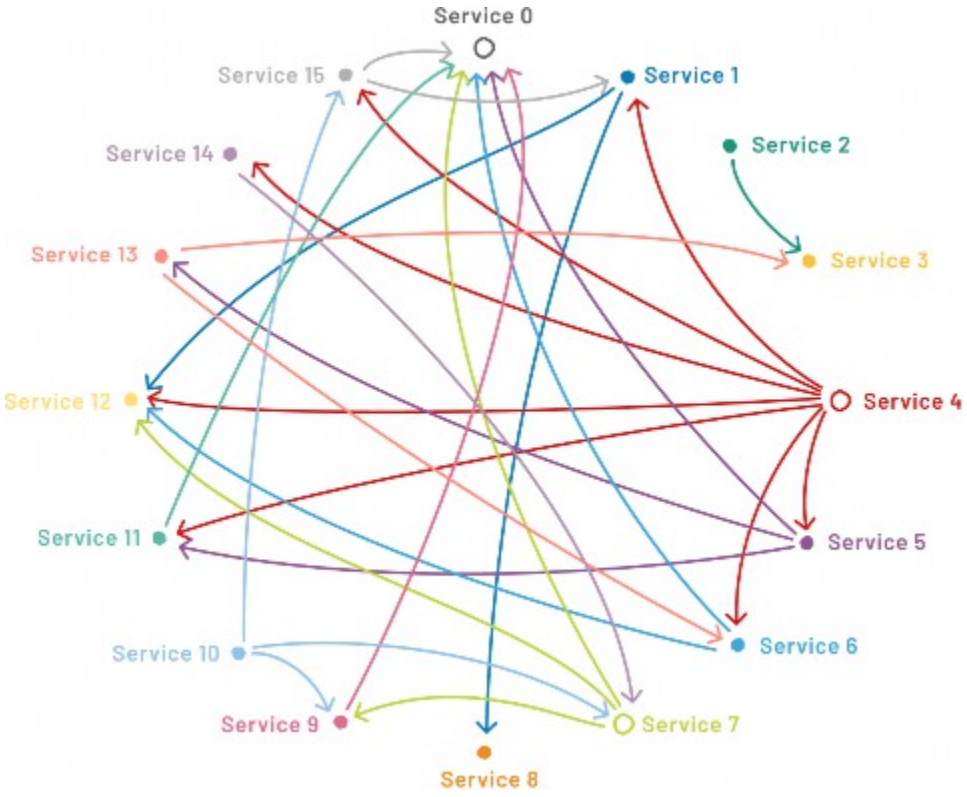


Figure 29: Simplified microservice call graph.

Big rocks Big rocks are the services with the most unique clients and the clients who connect with the highest count of unique services. Tackling big rocks early may be tempting to speed up adoption, but may result in biting off more than you can chew, causing problems and dissuading others from adopting SPIFFE.

Looking at the call graph at Figure 29, the big rocks can be identified by the nodes with the most connections. They may be key services that are called by many clients like Service 0. They may be clients that call many services like Service 4. Big rocks may also include services that are both a client and server, like Service 7. Tackling the migration of these services entails both benefits and risks:

- **Benefits**
  - Attractive choice
  - Potential speedy adoption
  - Wide-reaching benefits
  - Motivate others to adopt
- **Risks and challenges**
  - Maintain 2 endpoints (legacy and SPIFFE-enabled) for an extended period
  - Only turn off the legacy endpoint after all clients have adopted SPIFFE
  - Increase maintenance cost
  - Increases complexity
  - Stretches organizational capacity
  - Forced adoption
  - Disgruntled teams
  - Surprise, there’s a turtle in the corner!

Low hanging fruit Lowest hanging fruit are the services with one to a few clients or clients that connect to one or a few services. These are often the easiest to guide through the transition and make ideal first adopters.

Looking at the same graph above, low hanging fruit are nodes with few connections. These could be services that are a client of a single, other service, like Service 2. Low hanging fruit may also include a service with only one client like Service 8. When choosing which minimally connected services to migrate first, it is wise to choose the ones for which it will be easiest to maintain dual endpoints (legacy and SPIFFE), or those services which will have to maintain a dual-stack for the shortest period.

- **Benefits**
  - Less risk if something goes wrong
  - Easier to completely switch over from legacy to SPIFFE as less coordination and planning are needed
  - Good practice and learning opportunities
- **Risks and challenges**
  - The roll out may be perceived as being slow
  - May not visible or impactful enough to spark the adoption by key services owners

Accelerating adoption Several precursors and enablers can boost the adoption of SPIFFE in complex and heterogeneous environments. Each of them comes with a different set of benefits and challenges to consider. The considerations above apply here too; pick systems that will have the broadest impact and don't turn off non-SPIFFE functionality until you're certain that all non-SPIFFE consumers have been converted.

Precursors include tooling and services (such as CI/CD and workflow engines) that help others adopt SPIFFE. Development and operational tooling should be made available to the first adopters (Innovators) and iteratively improved as the early adopters come on board. The aim is for enabling tools and services to have reached maturity as the early majority comes on board. The late majority and laggards will struggle if there hasn't been enough investment in precursors.

Developer tools Having tools that help improve productivity is essential to a successful SPIFFE roll out. Gather a list of existing tools at your organization that are used during the application lifecycle, from development to operations to end-of-life, and consider which of the existing tools should be SPIFFE-enabled and whether new tools need to be built, bought, or deployed. Time and effort spent on creating, integrating, and improving tools often have a force multiplier effect in saving others time and effort, thereby helping to facilitate a smoother transition.

It is worth noting that tools shouldn't be built or bought in isolation, but in consultation with their intended users, ideally in an incremental and iterative way. Doing this properly can take time.

Choosing when a tool is good enough for the first and early adopters is a judgment call. On rare occasions, the first iteration of tooling is good enough for the early and late majority.

Continuous Integration and Deployment systems The implementation of SPIFFE in CI/CD tools can have a great impact on the adoption of this SPIFFE by the rest of the services in the organization since most teams have regular interaction with CI/CD systems. Conversely, however, this means it is a large task to get all consumers of CI/CD systems to be SPIFFE-aware so it can take a long time to turn off all non-SPIFFE integrations.

Container orchestrators If your organization is already using a container orchestrator, such as Kubernetes, you are halfway there! Orchestrators make it easy to front your workloads with SPIFFE-aware proxies so your developers don't need to be bothered.

Service Mesh Large microservice *service mesh* architectures are particularly relevant as enablers for a SPIFFE deployment because introducing SPIFFE support in the service mesh is a great way to roll out broad support without having to get development teams involved.

The relevance of the service mesh also comes with some risks and challenges. You can imagine that breaking a service mesh can have wide-ranging effects in an environment and may end in a catastrophic failure.

Planning SPIRE Operations

Running SPIRE day in and day out It is recommended that the team responsible for managing and supporting the SPIRE infrastructure get involved as early as possible. Depending on your organization structure, it may very well be the case that your security or platform team will be responsible for the whole lifecycle.

Another aspect to think through is how you split operations that involve any changes that would affect the system's security, performance, and availability. Stricter controls and gates may be needed to change anything related to your PKI, HSM, key rotations, and related operations. You may already have a change management process around it, and if not, this is an excellent time to start implementing it.

Your team needs to create Runbooks for different failure scenarios and test them to know what to do and what essential indicators they need to watch and create monitoring and alerting. You likely already know what monitoring and alerting systems you will use, but understanding the telemetry data and metrics available from SPIRE Server and Agent, and what the data means, would help your teams avoid downtime.

Testing for resilience Failure injection exercises help operators analyze how the system performs under certain failure conditions. You might have certain assumptions about how your system would react based on the architecture. Still, there are multiple potential points of failure in a SPIRE deployment that are worth triggering to test out your assumptions and can serve as good practice for your operations team to make sure they have all the alerts and run books in place.

We've compiled a list of some scenarios you want to include in your failure testing program. It is not a complete guide, just a starting point to build the checklist for your specific environment and deployment model. It would be best to execute all these tests with a different downtime: shorter than half of configured TTL and longer.

1. If a SPIRE deployment is using a single database instance, take down the database.
2. If a SPIRE deployment is using a database in a cluster with a written replica and multiple read replicas, take down the *write* instance.
3. Simulate database loss and test data recovery. What if you cannot recover the data or you can only recover from a month-old data?
4. Take down several of the SPIRE Servers in a HA deployment.
5. Take down the Load Balancer in a HA deployment.
6. Take down agents after they have been attested or simulate SPIRE Server loss completely.
7. If using upstream authority, simulate upstream authority failure.
8. Simulate root and intermediate CA compromise, rotation, and revocation.

Define which metrics are the most useful in each testing scenario and document expected healthy and dangerous ranges for those values and measure them over time.

These scenarios should be well documented, with the expected outputs well-defined, and then implemented through automated tests run automatically and periodically.

Logging Like all systems, logging is an essential part of SPIRE. However, the logs produced by SPIRE also function as evidence for audits and security incidents. The inclusion of identity issuance information, as well as observable attestation details, can be used to prove the state of certain workloads and services. Since the logs can be considered evidence, you may wish to take note of these few considerations when putting together a logging solution:

- Retention of logs should match your organization's legal requirements
- The logging system should have high availability in both admitting logs and storage
- The logs should be tamper-proof and must be able to provide evidence of it
- The logging system should be able to provide a chain of custody

Monitoring In addition to the usual health of SPIRE components to ensure the system is functioning properly, you should set up monitoring of configurations of servers, agents, and trust bundles to detect unauthorized changes, as these components are the foundation of the system's security. Besides, monitoring of issuance of identities and communication between servers and agents can be done to detect anomalies. However, based on the volume of identities issued in the system, you may wish to reconsider the extent of monitoring.

SPIRE offers flexible support for metric reporting through telemetry, allowing metrics collection using multiple collectors. The metrics collectors that are currently supported are Prometheus, Statsd, DogStatsd, and M3. Multiple collectors can be configured simultaneously, both in the servers and the agents.

Many metrics are available on SPIRE, with records that cover all the APIs and functionality:

- Server:
  - Management API operations
  - DB operations per API
  - SVID Issuance API operations
  - Rotation and Key Management
- Agent:
  - Interactions with the Server
  - SVID Rotation and Cache Maintenance
  - Workload Attestation

6. Designing a SPIRE Deployment

> In this chapter, you will learn about the components of a SPIRE deployment, what deployment models are available, and which performance and security considerations to take into account when deploying SPIRE.

The design of your SPIRE deployment should meet the technical requirements of your team and organization. It should also incorporate requirements to support availability, reliability, security, scalability, and performance. This design will serve as the basis for your deployment activities.

Your Identity Naming Scheme

Remember from the previous chapters that a SPIFFE ID is a structured string representing the identity name of a workload, as you saw in Chapter 4. The workload identifier section (the path portion of the URI) appended to the trust domain name (host part of the URI) can be composed to convey meaning about the ownership of a service to denote what platform it runs in, who owns it, its intended purpose, or other conventions. It is purposely flexible and customizable for you to define.

Your naming scheme may be hierarchical, like file system paths. That said, to reduce ambiguity, name schemes should not end with a trailing forward-slash (`/`). Below you will see some different samples following three different conventions you can follow, or come up with your own if you are feeling particularly inspired.

Identifying services directly You may find it useful to identify a service directly by the functionality it presents from an application perspective and the environment it runs in as part of the software development lifecycle. For example, an administrator may dictate that any process running in a particular environment should be able to present itself as a particular identity.

For example:

`spiffe://staging.example.com/payments/mysql`

or,

`spiffe://staging.example.com/payments/web-fe`

The two SPIFFE IDs above refer to two different components—the MySQL database service and a web front end—of a payments service running in a staging environment. The meaning of 'staging' is an environment and 'payments' a high-level service.

The prior two and the following two examples are illustrative and not prescriptive. The implementer should weigh their options and decide their preferred course of action.

Identifying service owners Often higher level orchestrators and platforms have their own identity concepts built-in (such as Kubernetes service accounts, or AWS/GCP service accounts) and it is helpful to be able to directly map SPIFFE identities to those identities. For example:

`spiffe://k8s-workload-cluster.example.com/ns/staging/sa/default`

In this example, the administrator of the trust domain *example.com* is running a Kubernetes cluster *k8s-workload-cluster.example.com* , which has a 'staging' namespace, and within this, a service account (SA) called 'default'.

Opaque SPIFFE identity The SPIFFE path may be opaque, and then the metadata can be kept in a secondary database. That can be queried to retrieve any metadata associated with the SPIFFE identifier. For example:

`spiffe://example.com/9eebccd2-12bf-40a6-b262-65fe0487d4`

SPIRE Deployment Models

We are going to overview the three most common ways to run SPIRE in production. It doesn't mean that we want to limit the available choices here, but for the sake of this book, we are going to limit the scope to these common ways to deploy the SPIRE Server. We will focus on the server deployment architectures only since there is usually one agent installed per node.

How many: big trust domains vs smaller trust domains The number of trust domains are expected to be relatively fixed, only revisited occasionally, and not expected to drift much over time. On the other hand, the number of nodes in a given trust domain, and the number of workloads, are expected to fluctuate frequently according to load and growth.

Choosing whether you centralize into a single root of trust with one big trust domain, or distribute and isolate into multiple trust domains, will be dictated by many factors. The security considerations section in this chapter talks about the use of trust domains for isolation. A few other reasons why you may choose multiple smaller trust domains over one large one include increased availability and isolation of tenants. Variables such as administrative domain boundaries, number of workloads, availability requirements, number of cloud vendors, and authentication requirements will also influence decisions here.

For example, you may choose to have a separate trust domain for every single administrative boundary for autonomy between different groups in the organizations that may have different development practices.

	Single Trust Domain	Nested	Federated
Size of deployment	Large	Very Large	Large
Multi-region	No	Yes	Yes
Multi-cloud	No	Yes	Yes

Table 6.1: Decision table for trust domain sizing

One for one: Single SPIRE cluster in your single trust domain A single SPIRE server, in a high availability configuration, is the best starting point for environments with a single trust domain.

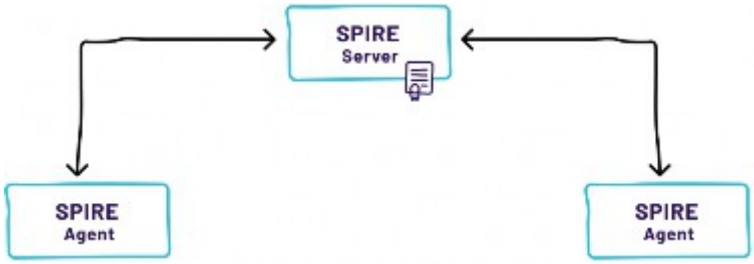


Figure 30: Single trust domain.

However, when deploying a single SPIRE Server to your trust domain that spans regions, platforms, and cloud provider environments, there are potential scaling issues when SPIRE Agents are dependent on a distant SPIRE Server. Under circumstances where a single deployment would span multiple environments, a solution to address the use of a shared data store over a single trust domain is to configure SPIRE Servers in a nested topology.

Nested SPIRE A nested topology for your SPIRE Servers lets you keeps communication between SPIRE Agents and the SPIRE Server as close as possible.

In this configuration, the top-tier SPIRE Servers hold the root certificates and keys, and the downstream servers request an intermediate signing certificate to use as the downstream server's X.509 signing authority. If the top tier goes down, intermediate servers continue to operate, providing resilience to the topology.

The nested topology is well suited for multi-cloud deployments. Due to the ability to mix and match node attestors, the downstream servers can reside and provide identities for workloads and Agents in different cloud provider environments.

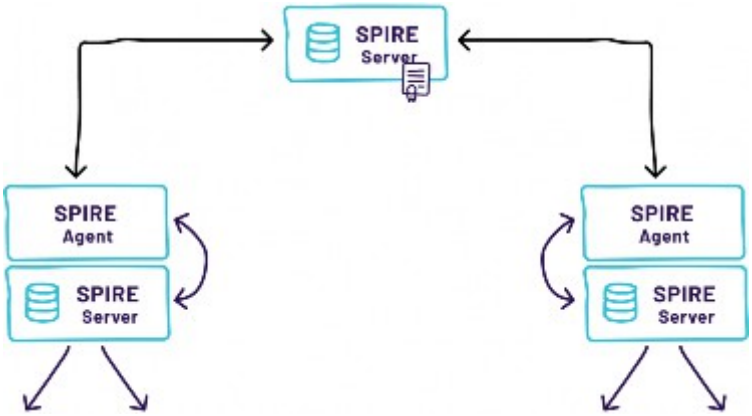


Figure 31: Nested SPIRE topology.

While nested SPIRE is an ideal way to increase the flexibility and scalability of your SPIRE deployment, it doesn't provide any additional security. Since X.509 doesn't provide any way to constrain the powers of intermediate certificate authorities, every SPIRE Server can generate any certificate. Even if your upstream certificate authority is a hardened server in a concrete bunker in your company's basement, if your SPIRE Server is compromised your entire network may be vulnerable. That's why it's important to make sure every SPIRE Server is secure.

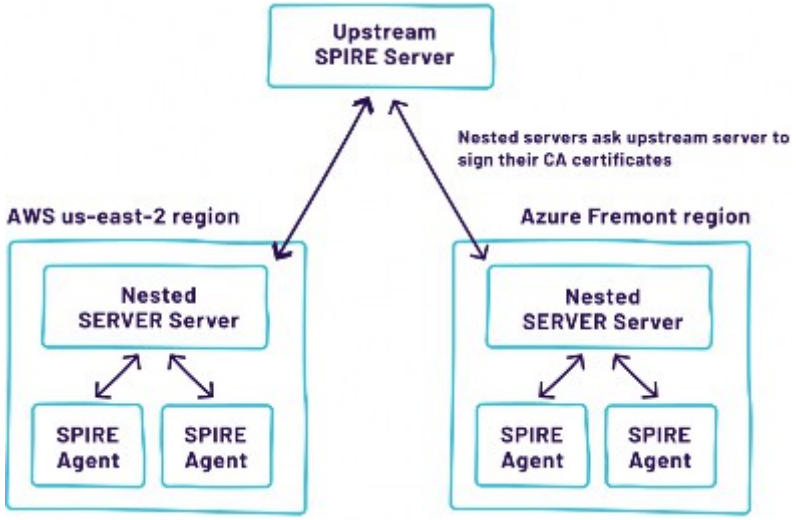


Figure 32: Illustration of a company architecture with one upstream SPIRE Server and two nested SPIRE Servers. Each of the two nested SPIRE Servers can have its own configuration (relevant for AWS and Azure), and if either one of them fails, the other is unaffected.

Federated SPIRE Deployments may require multiple roots of trust, perhaps because an organization has different organizational divisions with different administrators, or because they have separate staging and production environments that occasionally need to communicate.

Another use case is SPIFFE interoperability between organizations, such as between a cloud provider and its customers.

Figure 33: SPIRE Server using Federated trust domains.

These multiple trust domains and interoperability use cases both require a well-defined, interoperable method for a workload in one trust domain to authenticate a workload in a different trust domain. In federated SPIRE, trust between the different trust domains is established by first authenticating the respective bundle endpoint, followed by retrieval of the foreign trust domain bundle via the authenticated endpoint.

Standalone SPIRE Servers The simplest way to run SPIRE is on a dedicated server, especially if there is a single trust domain, and the number of workloads is not large. You can co-host a data store on the same node using SQLite or MySQL as a database in that scenario, simplifying the deployment.

Sidenote: A cluster constitutes more than one identically configured server.

However, when using the co-hosting deployment model, remember to consider database replication or backups. If you lose the node, you can quickly run the SPIRE Server on another node, but all your Agents and workloads need to re-attest to get new identities if you lose the database.

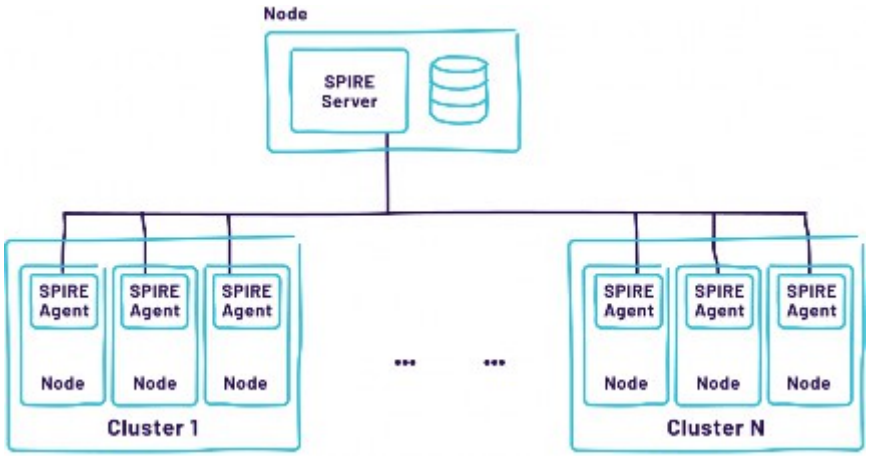


Figure 34: A single dedicated SPIRE Server.

Avoiding a single point of failure The benefit of simplicity always comes with a trade-off. If there is only one SPIRE Server and it is lost, everything is lost and will need to be rebuilt. The system's availability can be improved by having more than one server. There will still be a shared data store and secure connectivity and data replication. We'll talk about the different security effects of such decisions later in the chapter.

To scale the SPIRE Server horizontally, configure all servers in the same trust domain to read and write to the same shared data store.

The data store is where the SPIRE Server persists dynamic configuration information such as registration entries and identity mapping policies. SQLite is bundled with the SPIRE Server and is the default data store.

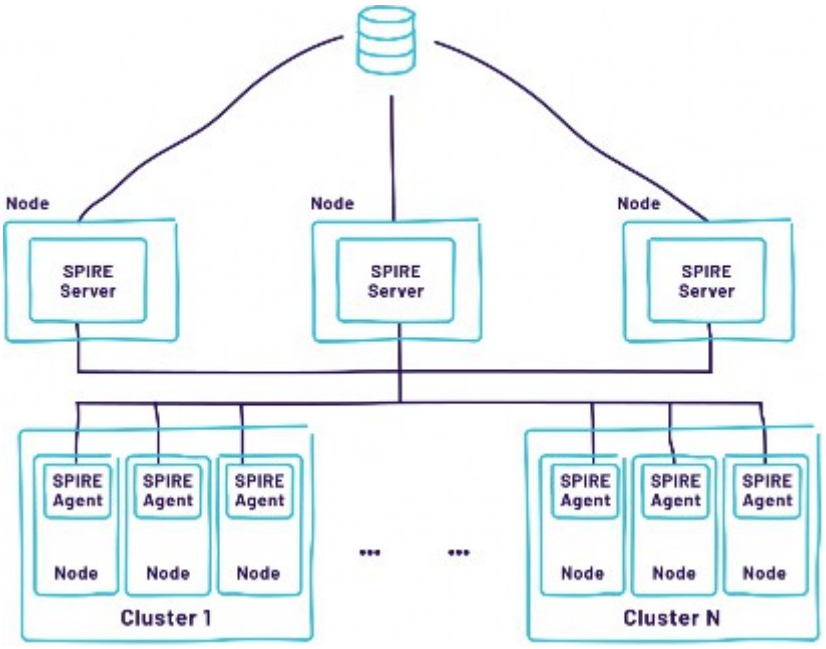


Figure 35: Multiple SPIRE Server instances running on HA.

Data Store Modeling

When working on the data store design, your primary focus should be on redundancy and high availability. You need to determine whether each of the SPIRE Server clusters has a dedicated data store or if there should be a shared one.

The choice of the type of database might be influenced by the entire system availability requirements and your operations team's abilities. For example, if the operations team has experience supporting and scaling MySQL, that should be the primary choice.

Dedicated data store per cluster Multiple data stores allow more independence for each dedicated part of the system. For example, the SPIRE clusters in AWS and GCP clouds might have independent data stores, or each VPC in AWS might have a dedicated data store. The advantage with this choice is if one region or cloud provider fails, SPIRE deployments running in the other regions or cloud providers are unaffected.

The downside of a data store per cluster becomes most apparent during a major failure. If the SPIRE data store (and hence all SPIRE Servers) in a region fails, it would require either restoring the local data store or switching the Agents over to another SPIRE Server cluster in the same trust domain, assuming the trust domain spans regions.

If it becomes necessary to switch agents over to a new cluster, special considerations must be made as the new cluster won't be aware of identities issued by another SPIRE cluster, or the registration entries that cluster contained. Agents will need to re-attest to this new cluster, and the registration entries will need to be restored either via backup or rebuilding.

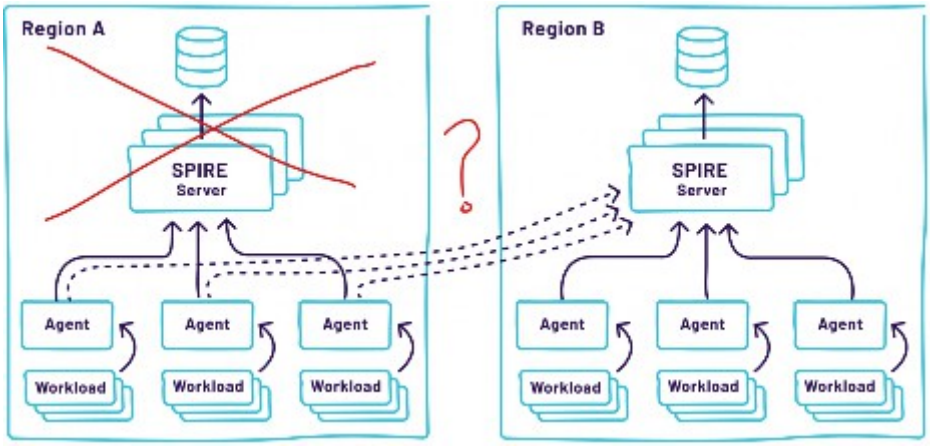


Figure 36: What happens if you need to migrate all the agents in one cluster to another cluster?

Shared data store Having a shared data store solves the issues of having individual data stores described above. However, it may make the design and operations more intricate and rely on other systems to detect outages and update DNS records in the event of a failure. Further, the design still requires pieces of database infrastructure for each SPIRE availability domain, per region or data center depending on the specific infrastructure. Please check the SPIRE documentation for more details.

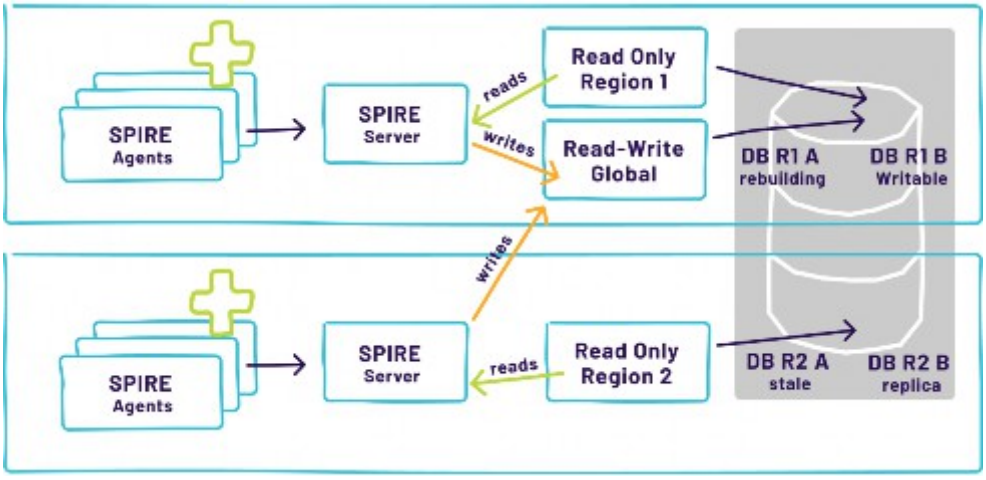


Figure 37: Two clusters using a Global data store scheme.

Managing Failures

When an infrastructure outage occurs, the main concern is how to continue to issue SVIDs to the workloads that need an SVID to operate properly. The Spire Agent's in-memory cache of SVIDs has been designed to be the primary line of defence against a short-term outage.

The SPIRE Agent periodically fetches the SVIDs that are authorized to be issued from the SPIRE Server, to be prepared to deliver them to the workloads when they need them. This process is done in advance of any request for an SVID from a workload.

Performance and reliability There are two advantages of the SVID Cache: performance and reliability. When a workload asks for its SVID, the Agent does not need to request and wait for SPIRE Server to mint the SVID, because it will already have it cached which avoids a round trip to SPIRE Server.

Additionally, if the SPIRE Server is not available at the time that the workload requests its SVID, that will not affect the issuance of the SVID because the Agent will have it cached already.

We need to make a distinction between X509-SVIDs and JWT-SVIDs. JWT-SVIDs cannot be minted in advance, because the Agent does not know the specific *audience* of the JWT-SVID needed by the workload, the Agent only pre-caches X509-SVIDs. However, the SPIRE Agent does maintain a cache of issued JWT-SVIDs that allows it to issue JWT-SVIDs to workloads without contacting the SPIRE Server as long as the cached JWT-SVID is still valid.

Time-to-live One important attribute of an SVID is its time-to-live (TTL). The SPIRE Agent will renew an SVID in the cache if the remaining lifetime is less than one half the TTL. This provides us the indication that SPIRE is conservative in terms of the confidence in the underlying infrastructure to be able to deliver an SVID. It also provides a hint about the role that the SVID TTL has in the resilience against outages. Longer TTLs provide more time to fix and recover any infrastructure outage, but there is a compromise between security and availability when choosing the TTL. A long TTL will provide ample time to remediate outages, but at the cost of exposing SVIDs (and related keys) for a longer period. Short TTLs reduce the time window that a malicious actor can take advantage of a compromised SVID, but will require quicker responses against an outage. Unfortunately, there is no “magic” TTL that is a best choice for all deployments. It has to be chosen while considering what tradeoff you are willing to accept between the time window within which you must solve outages and the acceptable exposure of issued SVIDs.

SPIRE in Kubernetes

This section covers the details of running SPIRE in Kubernetes. Kubernetes is a container orchestrator that can manage software deployment and availability on many different cloud providers, and also on physical hardware. SPIRE includes several different forms of Kubernetes integration.

SPIRE Agents in Kubernetes Kubernetes includes the concept of a *DaemonSet* , which is a container that is automatically deployed across all nodes, with one copy running per node. This is a perfect way to run the SPIRE Agent since there must be one agent per node.

As new Kubernetes nodes come online, the scheduler will automatically spin up new copies of the SPIRE Agent. First, each agent needs a copy of the bootstrap trust bundle. The easiest way to distribute this is through a Kubernetes ConfigMap.

Once an agent has the bootstrap trust bundle, it has to prove its own identity to the server. Kubernetes provides two types of authentication tokens:

1. Service Account Tokens (SATs)
2. Projected Service Account Tokens (PSATs)

Service Account Tokens are not ideal for security, because they remain valid forever and have unlimited scope. Projected Service Account Tokens are much more secure, but they do require a recent version of Kubernetes and a special feature flag to be enabled. SPIRE supports both SATs and PSATs for node attestation.

SPIRE Server in Kubernetes SPIRE Server interacts with Kubernetes in two ways. First, whenever its trust bundle changes, it has to post the trust bundle to a Kubernetes ConfigMap. Second, as agents come online, it has to validate their SAT or PSAT tokens using the *TokenReview* API. Both of these are configured through SPIRE plugins and require relevant Kubernetes API privileges.

The SPIRE Server can run completely in Kubernetes, alongside the workloads. However, for security, it might be desirable to run it on a separate Kubernetes cluster, or standalone hardware. That way, if the primary cluster is compromised, the SPIRE private keys are not at risk.

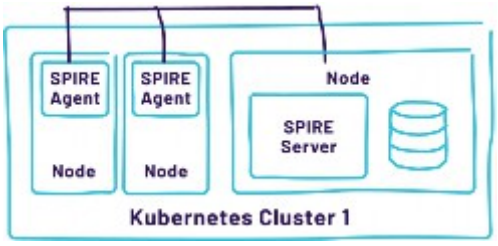


Figure 38: SPIRE Server on the same cluster as workloads.

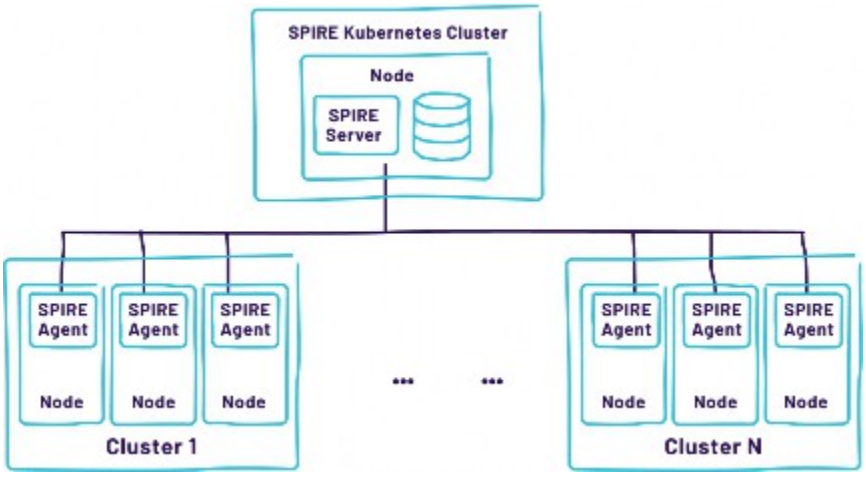


Figure 39: SPIRE Server on a separate cluster for security.

Kubernetes workload attestation The SPIRE Agent includes a Kubernetes Workload Attestor plugin. This plugin first uses system calls to identify the workload's PID. Then, it uses local calls to the Kubelet to identify the workload's pod name, image, and other characteristics. These characteristics can be used as selectors in registration entries.

Automatic Kubernetes registration entries A SPIRE extension called the Kubernetes Workload Registrar can automatically create node and workload registration entries, acting as a bridge between the Kubernetes API server and the SPIRE Server. It supports several different methods of identifying running pods and has some flexibility in the entries it creates.

Adding sidecars For workloads that haven't yet been adapted to use the Workload API (see section "Native SPIFFE support" in "Chapter 7: Integration with others"), Kubernetes makes it easy to add sidecars that do. A sidecar could be a SPIFFE-aware proxy like Envoy. Alternatively, it could be a sidecar developed alongside SPIRE called "SPIFFE Helper" which monitors the Workload API and reconfigures the workload when its SVID changes.

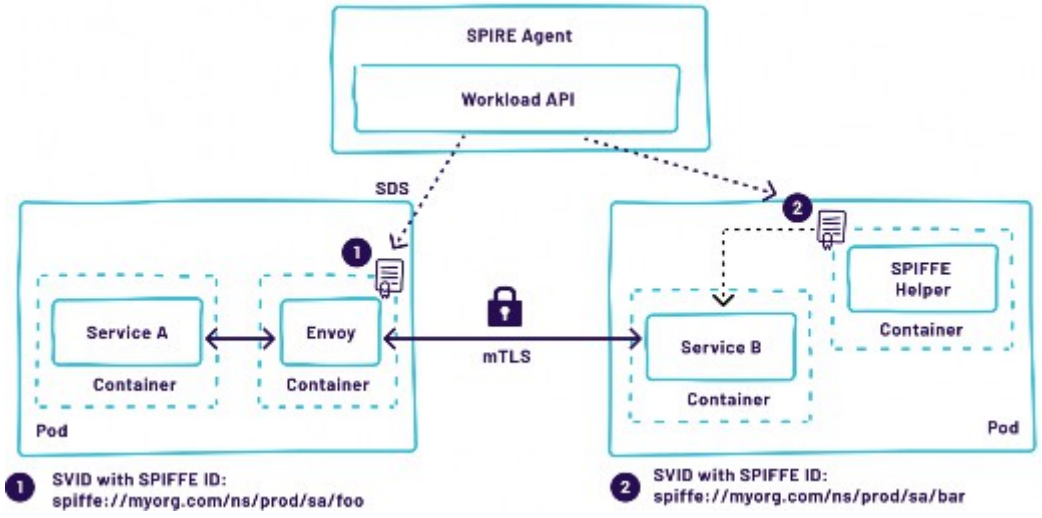


Figure 40: Workloads in a k8s cluster deployed along with sidecar containers.

SPIRE Performance Considerations

We didn't want to put specific performance requirements or recommendations for the number of workloads per agent or the number of agents per server since all the data a) depends on the hardware and network characteristics, and b) changes fast. Just as an example, one of the latest releases improved the data's performance by 30%.

As you've learned in previous chapters, SPIRE Agents continuously communicate with a server to get any new changes such as SVIDs for new workloads or updates to trust bundles. During each synchronization, there are multiple data store operations. By default, synchronization time is 5 seconds, and if that is producing too much pressure on your system, you can increase it to a higher value to address these concerns.

Very short SVID TTLs mitigates security risk, but if you use a very short TTL, be prepared to see additional load to your SPIRE Server as the number of signing operations increases in proportion to rotation frequency.

Another critical factor affecting your system performance could be the number of workloads per node. If you add a new workload to all nodes in your system, that would suddenly produce a spike and a load on the whole system.

If your system relies heavily on JWT-SVID usage, bear in mind that JWT-SVIDs are not preemptively generated on the agent side and need to be signed as requested. This may put extra load on the SPIRE Server and Agent, and increase latency when they are overloaded.

Attestor Plugins

There are various attestor plugins that SPIRE makes available for both node and workload attestation. The choice of which attestor plugin to use depends on the requirement for attestation, as well as the available support that the underlying infrastructure/platform provides.

For workload attestation, this largely depends on the type of workloads being orchestrated. For example, when using a Kubernetes cluster, a Kubernetes workload attestor would be appropriate, and likewise an OpenStack Attestor for an Openstack platform.

For node attestation, it is important to determine the requirements for security and compliance. There are sometimes requirements to perform the geofencing of workloads. In these scenarios, using a node attestor from a cloud provider that can assert that would provide those guarantees.

In highly regulated industries, the use of hardware-based attestation may be required. These mechanisms usually depend on the underlying infrastructure to provide support, such as APIs or Hardware modules like a Trusted Platform Module (TPM). This can include the measurement of the state of the system software, including firmware, kernel version, kernel modules, and even contents of the filesystem.

Designing attestation for different cloud platforms When working in a cloud environment, it is considered a best practice to verify your node's identity against metadata provided by the cloud provider. SPIRE provides a simple way to do this with custom node attestors designed specifically for your cloud. Most cloud providers assign an API that can be used to identify the API caller.

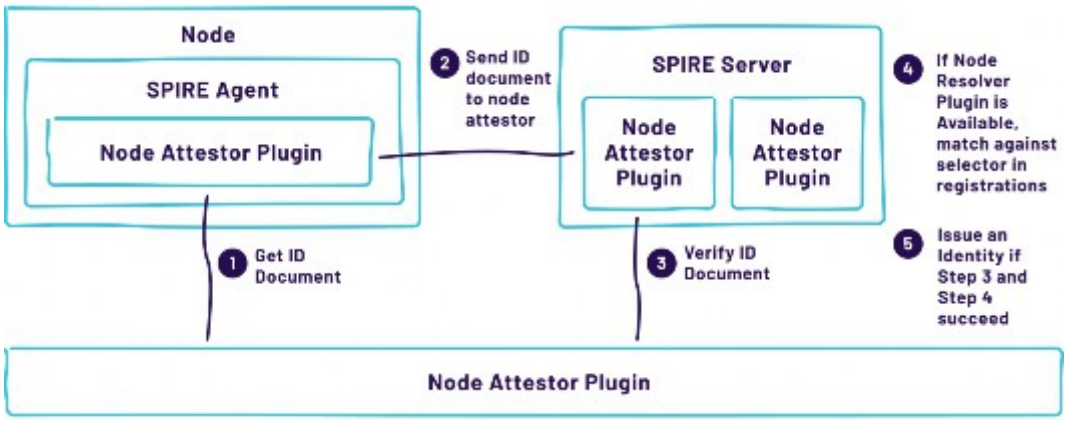


Figure 41: Node Attestor architecture and flow.

Node Attestors and Resolvers are available for Amazon Web Services (AWS), Azure, and Google Cloud Platform (GCP). Node Attestors for cloud environments are specific to that given cloud. The attestor's purpose is to attest the node before issuing an identity to the SPIRE Agent running on that node.

Once an identity has been established, the SPIRE Server may have a Resolver Plugin installed which allows additional selectors to be created that match against the node's metadata. The available metadata is cloud-specific.

On the opposite spectrum, if a cloud provider does not provide the ability to attest the node, it is possible to bootstrap with a join token. However, this provides a very limited set of assurances depending on the process through which this is done.

Management of Registration Entries

SPIRE Server supports two different ways to add registration entries: via a command-line interface or a Registration API (that allows admin-only access). SPIRE needs registration entries to operate. One option is for an administrator to manually create them.

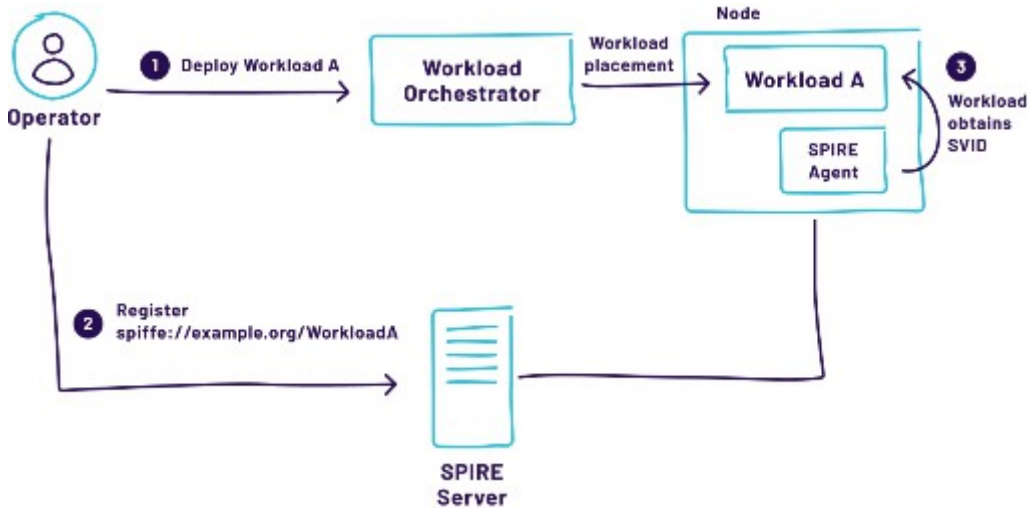


Figure 42: Workload manual registration.

A manual process won't scale in case of large deployments or when the infrastructure is growing fast. Also, any manual process is error-prone and may not be able to track all the changes.

Using an automated process to create registration entries using the SPIRE API is a better choice for deployments with a large number of registration entries.

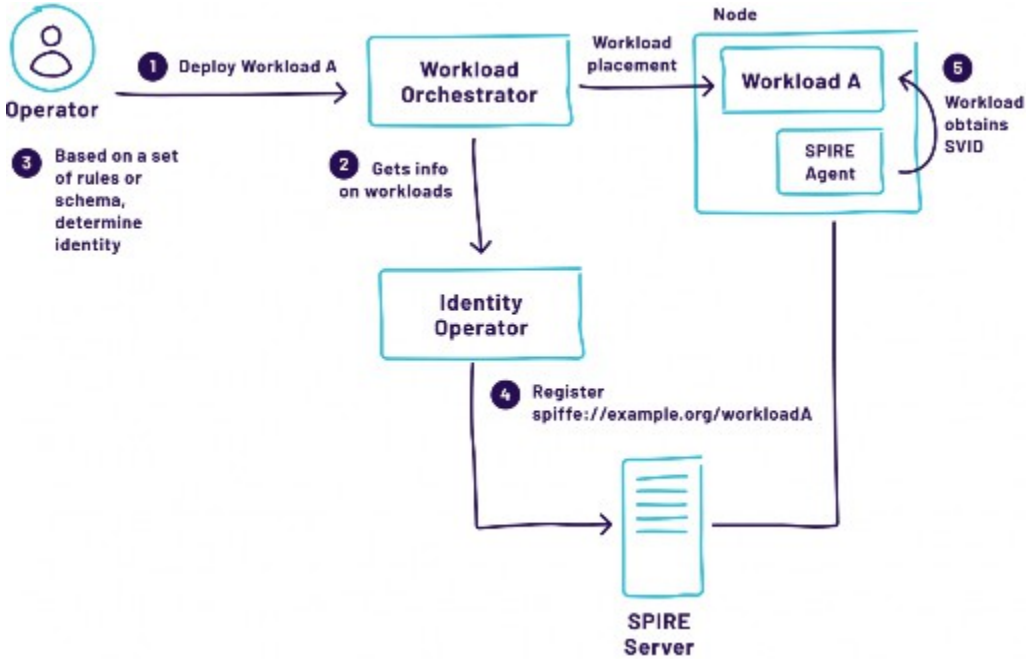


Figure 43: An example of automatically creating workload registration entries using an “Identity Operator” that communicates with the Workload Orchestrator.

Factoring Security Considerations and Threat Modeling

Whatever design and architectural decisions you make will affect the threat model of the whole system, and possibly other systems that interact with it.

Here are some important security considerations and the security implications you should consider when you are in the design stage.

Public Key Infrastructure (PKI) design What is the structure of your PKI, how you define your trust domains to establish security boundaries, where you keep your private keys, and how often they are rotated, are key questions you need to ask yourself at this point.

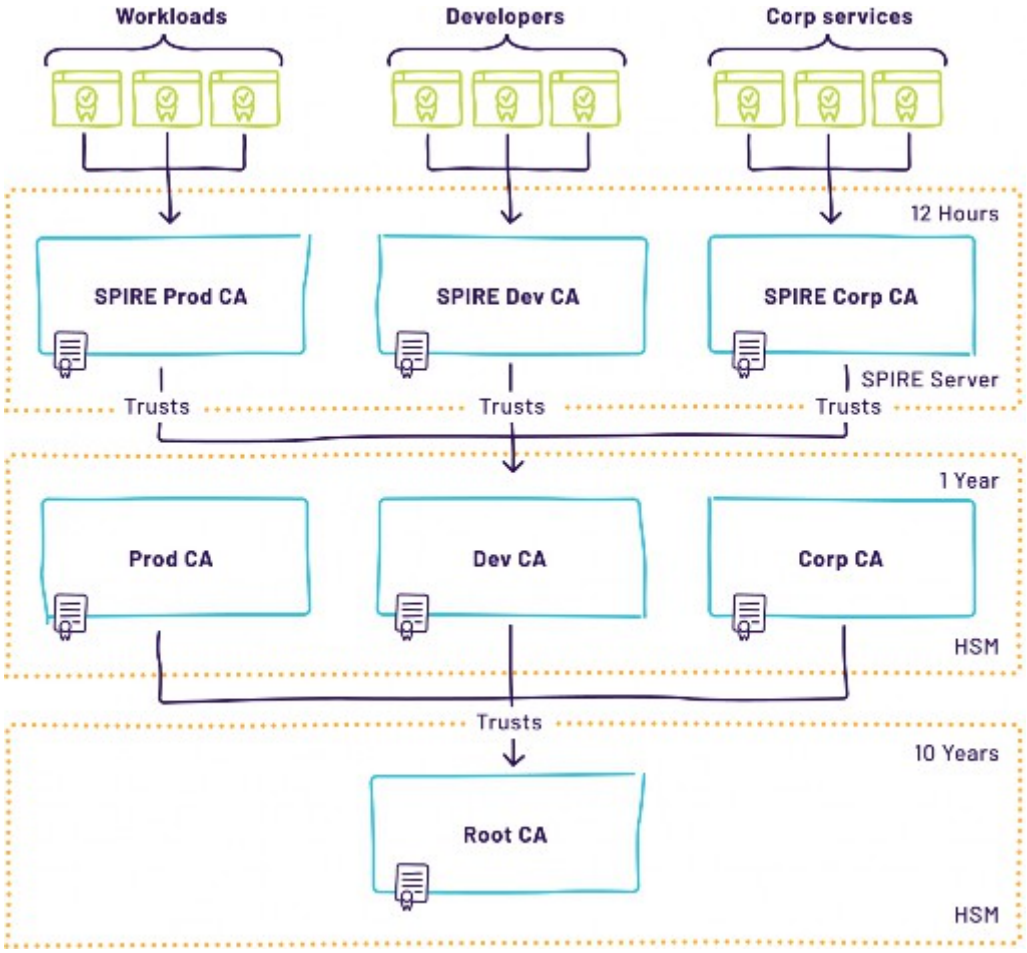


Figure 44: An example SPIRE deployment with three trust domains, each of which uses a different corporate Certificate Authority, each of which uses the same Root Certificate Authority. In each layer, certificates have a shorter TTL.

Each organization will have a different certificate hierarchy because each organization has different requirements. The diagram above represents one potential certificate hierarchy.

TTL, revocation, and renewal When dealing with PKI, questions around certificate expiry, re-issuance, and revocation always surface. Several considerations can influence the decisions made here. These include:

- Performance overhead for document expiry/re-issuance—how much performance overhead can be tolerated. The shorter the TTL, the higher the performance overhead.
- The latency of delivering documents—The TTL must be longer than the expected delivery latency of the identity documents to ensure that services do not have gaps in authenticating themselves.
- PKI Ecosystem Maturity—Are there revocation mechanisms in place? Are they maintained and kept up to date?
- Risk Appetite of the organization—If the revocation is not enabled, what is the acceptable amount of time where an identity can be valid if it has been compromised and detected.
- The expected lifetime of objects—TTL should not be set to too long a time, based on the expected lifetime of objects.

Blast radius During the PKI design phase, it is very important to consider how the compromise of one of the components will affect the rest of the infrastructure. For instance, if your SPIRE Server keeps keys in memory and the server gets compromised, all the downstream SVIDs need to be canceled and reissued. To minimize the impact of such an attack, you may design SPIRE infrastructure to have multiple trust domains for different network segments, Virtual Private Clouds, or cloud providers.

Keep your private keys secret What is important is where you keep your keys. As you might have learned earlier, SPIRE has a notion of a Key Manager which manages CA keys. If you are planning to make SPIRE Server a root in your PKI, you probably want to have persistence of your root key, but storing it on the disk is not a good idea.

A solution for storing the SPIRE keys might be a software or hardware Key Management Service (KMS). There are standalone products that function as a KMS, as well as built-in services for each of the major cloud providers.

Another possible design strategy to integrate SPIRE with existing PKI is to use the SPIRE Upstream Authority plugin interface. In this case, the SPIRE Server signs its intermediate CA certificate by communicating with existing PKI using one of the supported plugins.

SPIRE data store security considerations We intentionally removed the SPIRE Server's data store from our threat model in Chapter 4. The data store is where SPIRE Server persists dynamic configuration such as registration entries and identity mapping policies that are retrieved from the SPIRE Server APIs. SPIRE Server data store supports different database systems which it can use as the data store. The compromise of the data store would allow the attacker to register the workload on any node and potentially the node itself. Attackers would also be able to add keys to the trust bundle and get into the trust chain of downstream infrastructure.

Another possible surface for attackers is a denial of service attack on the database or SPIRE Server connectivity to the database, which would lead to denial of service to the rest of the infrastructure.

When you consider a design with any database for SPIRE Server infrastructure in production, you won't likely use the model where the database process coexists on the same host with the server. Though the model with limited access to the database, and co-hosting it with the server significantly limits the attack surface, it is very hard to scale in a production environment.

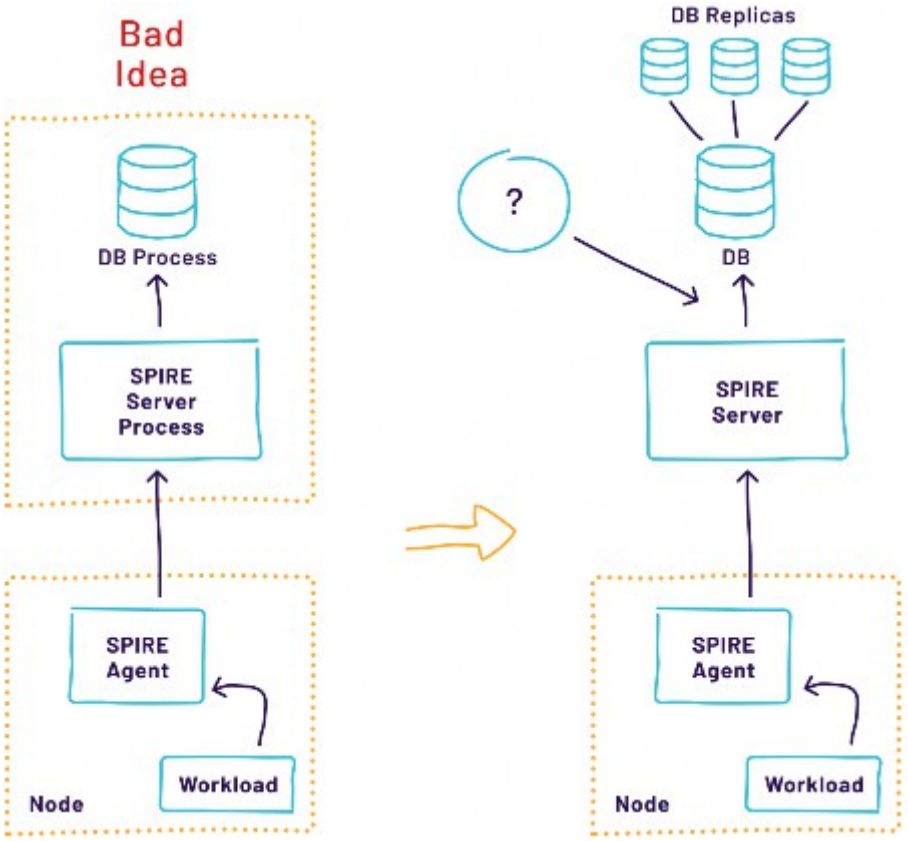


Figure 45: Typically the SPIRE Server data store runs on remotely over a network connection for availability and performance reasons, but this presents a security challenge.

For availability and performance reasons, the SPIRE data store will typically be a network connected database. But you should consider the following:

- If this is a shared database with other services, who else has access to it and manages it?
- How will the SPIRE Server authenticate to the database?
- Does the database connection allow TLS-protected secure communication?

These are relevant questions that need to be taken into consideration since how the SPIRE Server connects to the database determines greatly how secure the whole deployment is. In the case of using TLS and password-based authentication, the SPIRE Server deployment should rely on a secrets manager or KMS to keep the data secure.

In some deployments, you may need to add another lower-level *meta PKI* infrastructure that will allow you to secure communication with all low-level dependencies for the SPIRE Server, including your configuration management or deployment software.

SPIRE Agent configuration and trust bundle The way you distribute and deploy components of your SPIRE ecosystem, and its configuration in your environment might have severe consequences for your threat model and the whole system's security model. It is the low-level dependency not only for SPIRE but for all security systems you have, so here we'll focus only on what is specific to SPIFFE and SPIRE.

Trust bundle There are different ways to deliver the agent's *bootstrap trust bundle*. This is the trust bundle that the agent uses when initially starting up, in order to authenticate the SPIRE Server. If an attacker can add keys to the initial trust bundle and perform a *monster-in-the-middle* attack, it will perform the same attack on workloads because they receive SVID and trust bundle from the victim agent.

Configuration The SPIRE Agent configuration also needs to be kept secure. If an attacker can modify this configuration file, then they could point it at a compromised SPIRE Server and control the agent.

Effects of node attestor plugins Asserting trust through multiple independent mechanisms provides a greater assertion of trust. The node attestation you choose might significantly impact your SPIRE deployment's security and shift the root of trust for it toward another system. When deciding what type of attestation to use, you should incorporate it into your threat model and review the model every time something changes.

For example, any other proof-of-possession-based attestation will shift the root of trust, so you want to make sure the system you have as a lower dependency meets your organization's security and availability standards.

When designing a system with an attestation model using a join token, carefully evaluate operation procedures of adding and using tokens, whether by operator or provisioning system.

Telemetry and health checks Both SPIRE Server and Agent support health checks and different types of telemetry. It might not be immediately apparent that enabling or misconfiguration of health checks and telemetry may increase the attack surface for the SPIRE infrastructure. The SPIFFE and SPIRE threat models assume that the agent only exposes the Workload API interface over the local Unix socket. The model doesn't consider that misconfigured (or intentionally configured) health check service listening not on the localhost might expose the agent to potential attacks such as DoS, RCE, and memory leak. It would be best to take similar precautions when choosing the telemetry integration model because some of the telemetry plugins (e.g. Prometheus) might expose additional ports.

7. Integrating with Others

This chapter explores how SPIFFE and SPIRE integrate with your environment.

SPIFFE is designed from the ground-up to be pluggable and extensible, so the topic of integrating SPIFFE and SPIRE with other software systems is a broad one. The architecture of a given integration is beyond the scope of this book. Instead, this chapter intends to capture some common integrations that are possible along with a high-level overview, as well as a strategy for conducting integration work.

Enabling Software to Use SVIDs

There are many options available when considering how to adapt the software to use SVIDs. This section describes a few of those options, and considerations that go along with them.

Native SPIFFE support This approach requires modifying existing services to make them SPIFFE aware. It is the preferred choice when the modifications required are minimal, or can be introduced in a common library or framework used across application services. Native integration is the best approach for data plane services that are sensitive to latency, or services that want to utilize identity at the application layer. SPIFFE provides libraries such as GO-SPIFFE for Go programming language and JAVA-SPIFFE for the Java programming language that facilitate the development of SPIFFE-enabled workloads.

When building software in languages with supported SPIFFE libraries, this is often the most straightforward way to utilize SPIFFE. The Go and Java libraries mentioned above have examples using SPIFFE with gRPC and HTTPS clients and servers.

That said, it should be noted that you are not limited to the choices of Java and Go languages. These libraries are implemented on top of the open specification. At the time of writing, community development efforts are underway for SPIFFE libraries in Python, Rust, and C programming languages.

SPIFFE-aware proxies Often, the refactoring cost is deemed too high, or the service is running a third-party code that can not be modified. In these situations, it is often a pragmatic choice to front the application with a proxy that supports SPIFFE. Depending on the application deployment model, it could be a standalone proxy or a set of colocated proxies.

Colocated proxies have the advantage that traffic between the proxy and the unsecured service remains local—if standalone proxies are in use, security between the proxy and the application must still be accounted for.

Envoy is a popular choice for this, and Ghostunnel is another good option. While other proxies such as NGINX and Apache can also work, their SPIFFE-related functionality is limited.

Ghostunnel is an L3/L4 proxy that enjoys fully native support of the entire SPIFFE specification set, including support for the SPIFFE Workload API and Federation. For applications that require L7 features, Envoy is recommended. While Envoy does not natively support the SPIFFE Workload API, SPIRE implements the Secret Discovery Service API (or SDS API) which is an Envoy API used to retrieve and maintain certificates and private keys.

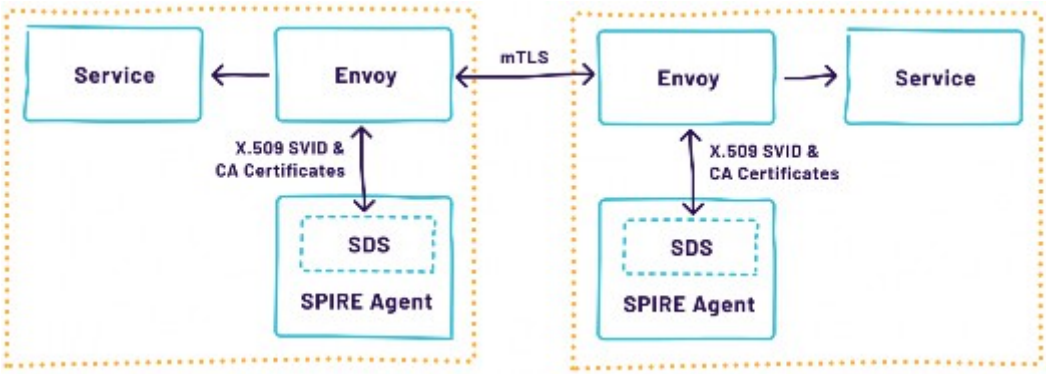


Figure 46: high-level diagram of two Envoy proxies sitting between two services using the SPIRE Agent SDS implementation to establish mutual TLS.

By implementing the SDS API, SPIRE can push TLS certificates, private keys, and trusted CA certificates directly into Envoy. SPIRE then takes care of rotating the short-lived keys and certificates as required, pushing updates to Envoy without the need for a restart.

Service mesh L7 proxies such as Envoy can perform many functions above and beyond SPIFFE security. For example, service discovery, request authorization, and load balancing are all features that Envoy brings to the table. It can be particularly attractive to offload this kind of functionality to a proxy in environments where using a shared library is prohibitive (e.g. when applications are written in many different languages, or are not able to be modified). This approach also pushes proxy deployment towards the colocated model, in which every application instance has a dedicated proxy running next to it.

This, however, creates an additional problem: how to manage all of these proxies.

A service mesh is an opinionated deployment of a proxy fleet and an associated proxy control plane. They typically allow for automatic injection and configuration of colocated proxies as workloads are deployed, and provide ongoing management of these proxies. By offloading many platform concerns to a service mesh, applications can be made agnostic of these functions.

Most service mesh implementations, to date, leverage SPIFFE authentication for service-to-service traffic. Some use SPIRE to accomplish this, and others have implemented product-specific SPIFFE identity providers.

Helper programs For cases where workloads do not natively support the SPIFFE Workload API, but still support using certificates for authentication, a helper program running alongside the workloads can work to bridge the gap. One example of such a helper program is the SPIFFE Helper. The SPIFFE Helper fetches SVIDs from the SPIFFE Workload API and writes them to disk where they can be picked up by the application. The SPIFFE Helper can continue to run, ensuring that certificates on disk are continually updated as they rotate. When an update does occur, the helper can signal the application (or run a configurable command) such that the changes are picked up by the running application.

Many off-the-shelf applications that support TLS can be configured to use SPIFFE this way. The SPIFFE Helper repository has examples of configuring MySQL and PostgreSQL. Many web servers such as Apache HTTPD and NGINX can be similarly configured. That is also useful for client software, which can only be configured to utilize certificates on disk.

- `openssl`
- `x509curl`
- `grpcurl`

It is important to note that this gives less flexibility than native SPIFFE integration since, in particular, it may not allow the same granularity of trust configuration. For example, when using SPIFFE Helper to configure Apache HTTPD for mutual TLS, it is not possible to configure `mod_ssl` to only accept clients with particular SPIFFE IDs.

Using SVIDs with Software That Is Not SPIFFE-aware

Since SVIDs are based on well-known document types, it is relatively common to encounter software that supports the document type but isn't necessarily SPIFFE-aware *per se*. The good news is that this is a relatively expected condition and that SPIFFE/SPIRE have been designed to handle this case well.

X509-SVID dual-use Many non-SPIFFE systems support using TLS (or mutual TLS) but rely on certificates having identity information in either the common name (CN) of the certificate subject or a DNS name of the Subject Alternative Name (SAN) extension. SPIRE supports issuing X.509 certificates with specific CN and DNS SAN values that can be specified on a workload-by-workload basis (as part of a registration entry).

This functionality is an important detail, as it allows for the use of X509-SVIDs with software that doesn't directly understand how to use SPIFFE IDs. For example, HTTPS clients often expect the presented certificate to match the DNS name of the request. In another example, MySQL and PostgreSQL can use the Common Name for identifying an mutual TLS client. By leveraging this SPIRE feature, and the flexibility afforded by SPIFFE in general, these use cases can be accommodated with the very same SVIDs that are used for SPIFFE use cases as well.

JWT-SVID dual-use Similar to the way that X509-SVIDs can be used for SPIFFE authentication as well as for more traditional X.509 use cases, JWT-SVIDs also support this kind of duality. While JWT-SVIDs *do* use the standard subject (or sub) claim to store the SPIFFE ID, the validation methodology is similar to and compatible with OpenID Connect (or OIDC).

More specifically, the SPIFFE Federation API exposes public keys via a JWKS document served by an HTTPS endpoint, which is the same mechanism used to obtain public keys for OIDC validation. As such, any technology that supports the OIDC Identity Federation will also support accepting JWT-SVIDs, regardless of whether or not they're SPIFFE-aware.

One example of an integration supporting this identity federation is Amazon Web Services (AWS) Identity and Access Management (IAM). By configuring IAM in an AWS account to accept identities from SPIRE as an OIDC identity provider, it becomes possible to use JWT-SVIDs from the SPIFFE Workload API to assume AWS IAM roles. This is particularly powerful when the workload which needs to access AWS resources is not running in AWS, effectively negating the need to store, share, and manage long-lived AWS access keys. For a detailed example of how to accomplish this, please see the AWS OIDC Authentication tutorial on the SPIFFE website.

Ideas for What You Can Build on Top of SPIFFE

Once SPIFFE exists as a universal identity foundation in your ecosystem and is integrated with your applications, it might be a good time to think about what to build on top. In this section, we want to cover what is possible to build on top of SPIFFE and SPIRE. It is not necessarily that the project has all the building blocks to make everything work right out of the box. Some integration pieces will need to be implemented to make it happen, and the details on exactly how it can be accomplished will vary from deployment to deployment.

Logging, monitoring, observability, and SPIFFE SPIFFE can provide verifiable proof of identity to other systems which is an advantage to the following components:

- Metrics infrastructure
- Logging infrastructure
- Observability
- Metering
- Distributed tracing

You can use SVIDs to ensure secure client-server communication for these systems. However, you can also extend all these components to enriching the data with the SPIFFE ID. Doing so comes with a variety of advantages, such as the ability to correlate events across multiple types of platforms and runtimes. It can even help to identify applications and services that still do not use SPIFFE identities, or spot operational anomalies and attacks regardless of which corner of the infrastructure they may be occurring in.

Auditing For any security system, like the one you build on top of SPIRE, logs are not just information that help developers and operators understand what happened with the system. Any security systems logs are evidence of what is happening, so having a centralized location in which logs are stored is a good idea. In case of any security incident, such information is highly valuable for forensics analysis.

SPIFFE can help augment audit data by providing non-repudiation through the use of authenticated calls to centralized auditing systems. For example, by using an X509-SVID and mutual TLS when establishing sessions to auditing systems, we can be sure of where the log line came from—attackers cannot simply manipulate labels or other data that is being sent across.

Certificate transparency Certificate Transparency (CT) helps spot attacks on certificate infrastructure by providing an open framework for monitoring and auditing X.509 certificates in nearly real-time. Certificate Transparency allows detecting certificates that have been maliciously acquired from a compromised certificate authority. It also makes it possible to identify certificate authorities that have gone rogue and are maliciously issuing certificates. To learn more about Certificate Transparency, read the introductory documentation.

There are different possibilities for the integration of SPIRE with Certificate Transparency. With this integration, it is possible to log information about all certificates issued by your system and protect it with a special cryptographic mechanism known as Merkle Tree Hashes to prevent tampering and misbehavior.

Another method you might consider is to enforce Certificate Transparency across all your systems. That would prevent establishing TLS and mutual TLS connection with applications and services that do not have certificate information logged in the certificate transparency server.

Integration with CT is beyond the scope of the book. Check the SPIFFE/SPIRE communities for more information and the latest updates.

Supply chain security Most of the coverage on the intended use of SPIFFE has been given to securing the communications between software systems at runtime.

However, it is also critical to protect software during the stages before it is deployed. Supply chain compromises a potential attack vector. For that reason, it is desirable to protect the integrity of the software supply chain to prevent malicious actors from introducing backdoors or vulnerable libraries into the code. Verification of the provenance of software artifacts and the set of steps performed during the pipeline is a way to verify that software has not been tampered with.

You can consider using SPIFFE to provide the root of trust for signing. It can also be used for issuing identities to the software components of supply chain systems. There are several ways it can work in conjunction with complementary software like The Update Framework (TUF) or an artifact signing service like Notary, or leveraged along with supply chain logs like In-Toto.

It is possible to integrate SPIRE with supply chain components at two levels.

First, you may use it to identify the different elements of this supply chain system to secure the machinery and the control plane. Second, to ensure that only binaries of known provenance are issued identities by customizing selectors. As an example of the latter, at a very rudimentary level, such attributes can be passed as labels into a container image using existing Docker selectors or by developing a workload attestor that can check for supply chain metadata.

Ideas to Integrate SPIFFE for Users

The primary focus of SPIFFE and SPIRE architecture is on software identity. It doesn't take the users' identity into account because the problem is already considered to be well-solved, and significant differences exist in how identity is issued to humans vs software. That said, it doesn't mean you cannot distribute SPIFFE identities to users.

Verifiable identity for users How should users interact in a SPIFFE-enabled ecosystem? Remember that SPIFFE stands for Secure Production Identity Framework for *Everyone*. While most of this book focuses on identity for software, it is equally valid and even desirable for SPIFFE verifiable identities (SVIDs) to be given to users. This way, everything that workloads can do with SVIDs can be done by people as well, such as mutual TLS access to services. This can be especially useful for developers who are building software and need access to the same resources that their software will be using once deployed.

Just as the SPIFFE spec is open-ended about the scheme of SPIFFE IDs, it is up to you how you would like to represent humans. It may be enough to have your username as the SPIFFE ID path, e.g. `spiffe://example.com/users/zero_the_turtle`. Alternatively, you could create a distinct trust domain for users versus workloads, e.g. `spiffe://users.example.com/zero_the_turtle`.

In an ideal scenario, your existing SSO provider is capable of producing JWTs for your users, as is the case for OIDC identity providers. In this case, if you can configure your SSO provider to use a SPIFFE ID for the sub claim, you might not need to do any extra work to produce SVIDs for your users.

If you are unable to get SPIFFE JWTs from your identity provider directly but you do have the means to get a verifiable identity token, you could instead utilize a custom SPIRE attestor that accepts an identity token from your provider as a rudimentary means of attestation.

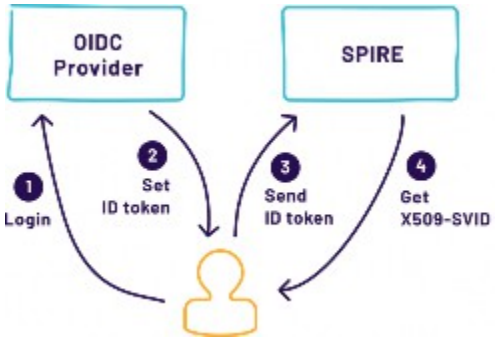


Figure 47: An example of using an OIDC ID token for SPIRE authentication.

If none of the above situations apply, you can always build a distinct service integrated into your existing SSO solution which can produce SVIDs for users based on their authenticated session. Check out the SPIFFE website for sample projects.

Using SPIFFE and SPIRE with SSH OpenSSH supports authentication with Certificate Authorities (CAs) and certificates. Although the format of OpenSSH certificates is different from X.509, one can build a service for creating SSH certificates using SVIDs as authentication. This allows you to utilize your SPIFFE identities for SSH as well.

For users that need SSH access to workloads in your ecosystem, this model provides ephemeral, short-lived, auditable credentials for SSH access and also provides a single control-point with which you enforce access control policies or multi-factor authentication.

This also allows the workloads to retrieve server-side (aka "host") SSH certificates that allow the workload to authenticate itself to the user. Using this certificate, users no longer need to have their SSH connection interrupted with a question about trusting a server's host key on the first connection.

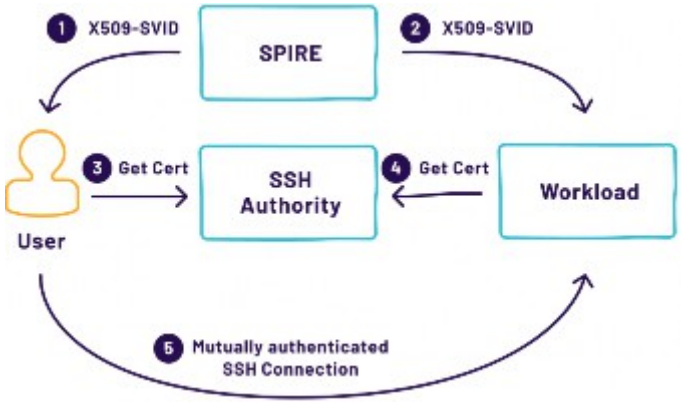


Figure 48: Using SVIDs to bootstrap SSH certificates

Microservice UIs While most of this book is concerned with workload-to-workload authentication, there is often also a need for users to authenticate to workloads. If the user is doing so via a CLI or other desktop tool then mutual TLS with a user's SVIDs can be used. However, many microservices will also want to host some sort of browser-based user interface. This may be because developers are accessing a purpose-built administrative or management interface for their service, or consumers may be using a tool like Swagger UI to explore and experiment with a service's API.

Providing a good experience for services with a browser-based user interface requires bridging a browser-friendly form of authentication and SPIFFE mutual TLS authentication. The easiest way to achieve this is to have one API port that uses mutual TLS and another API port that accepts a browser-friendly authentication method such as an existing web-based SSO mechanism or OAuth2.0/OIDC.

A post-authentication filter for requests on the secondary port should provide a translation layer between the browser-based authentication principal and a corresponding SPIFFE ID. If you have set up a mechanism for users to get SVIDs directly, as described above, then the same translation should be used here. This way, the underlying application is agnostic to the specific authentication mechanism used, so web-based requests made by a given user are functionally equivalent to the same request made via mutual TLS using that user’s SVID.

8. Using SPIFFE Identities to Inform Authorization

This chapter explains how to implement authorization policies using SPIFFE identities.

Building Authorization on Top of SPIFFE

SPIFFE focuses on the issuance and interoperability of secure cryptographic identity for software, but as mentioned previously in this book, it does not directly solve for the use or consumption of these identities.

SPIFFE frequently acts as the cornerstone of a strong authorization system, and SPIFFE IDs themselves play an important part in this story. In this section, we discuss options for using SPIFFE to build authorization.

Authentication Vs Authorization (AuthN Vs AuthZ)

Once a workload has a secure cryptographic identity, it can prove its identity to other services. Proving identity to an outside service is called *authentication*. Once authenticated, that service can choose what actions are permissible. This process is called *authorization*.

In some systems, any entity that is authenticated is also authorized. Because SPIFFE automatically grants identities to services as they startup, it is vital to clearly understand that not every entity that can authenticate itself should be authorized.

Authorization Types

There are many ways authorization can be modeled. The simplest solution is to simply have an *allowlist* of authorized identities attached to each resource. However, as we explore this, we will notice several limitations with the allowlist approach when dealing with the scale and complexity of an ecosystem. We will look at two more sophisticated models, Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC).

Allowlists In small ecosystems, or when just getting started with SPIFFE and SPIRE, it's sometimes best to keep things simple. For example, if you only have a dozen identities in your ecosystem, access to each resource (i.e. service, database) can be managed through maintaining a list of identities with access.

```
ghostunnel server --allow-uri spiffe://example.com/blog/web
```

Here the ghostunnel server is explicitly authorizing access based on the identity of clients alone.

The advantage of this model is that it's easy to understand. As long as you have a limited number of identities that don't change, it is easy to define and update access control on resources. However, scalability can become a hurdle. If an organization has hundreds or thousands of identities, maintaining allowlists quickly becomes unmanageable. For instance, every time a new service is added, it might require the operations team to update many allowlists.

Role-Based Access Control (RBAC) In Role-Based Access Control (RBAC), services are assigned to roles, and then access control is designated based on roles. Then, as new services are added, only a relatively small set of roles need to be edited.

While it is possible to encode a service's role into its SPIFFE ID, this is generally a poor practice because the SPIFFE ID is static, while the roles it is assigned to might have to change. Instead, it's best to use an external mapping of SPIFFE IDs to roles.

Attribute-Based Access Control (ABAC) Attributed-Based Access Control (ABAC) is a model where authorization decisions are based on attributes that are associated with a service. In conjunction with RBAC, ABAC can be a powerful tool for strengthening authorization policies. For example, to meet legal requirements it may be necessary to limit access to a database to services from a particular region. The region information can be an attribute in an ABAC model that is used for authorization and encoded in a SPIFFE ID scheme.

Designing SPIFFE ID Schemes for Authorization

The SPIFFE specification doesn't specify or limit what information you can or should encode into a SPIFFE ID. The only limitations you need to be aware of come from the maximum length SAN extension and what characters you're are allowed to use.

> **A Word of Warning**: Use extreme care when encoding authorization metadata into your organization's SPIFFE ID format. The examples below illustrate how to do this, since we did not want to introduce additional authorization concepts.

SPIFFE scheme examples To make an authorization decision on the SPIFFE identity sub-strings, we must define what each part of the identity means. You can design your scheme in the format where you encode information by the order. In this case, the first part might represent a region, the second environment, and so on.

```
spiffe://trust.domain.org/<region>/ ↵
  <dev,stage,prod>/<organization>/<workload_name>
```

Below is an example of the scheme and identity:

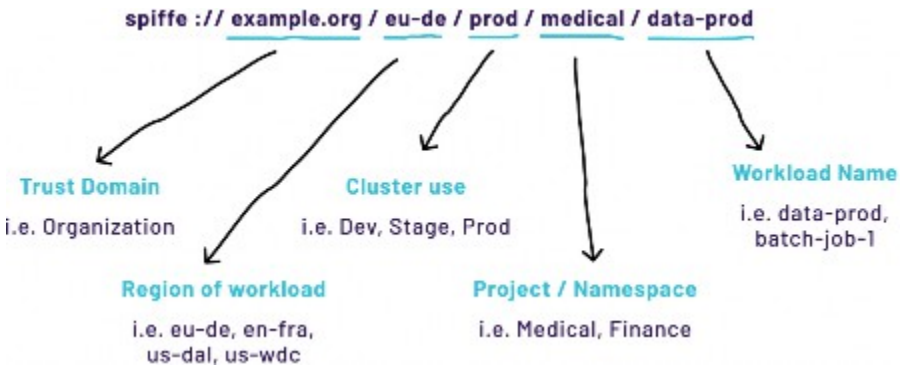


Figure 49: Components of a SPIFFE ID and potential meanings at one organization.

An identity scheme can not only take the shape of a series of fixed fields but can take a more complex structure, depending on the needs of an organization. A common example we can look at is workload identities across different orchestration systems. For example, in Kubernetes and OpenShift, the naming conventions of workloads are different. The following illustration shows this as an example. You may notice that the fields not only refer to different attributes and objects but that the structure of the SPIFFE ID also depends on the context.

The consumer can distinguish the structure of the scheme by observing the prefix of the identity. For example, an identity with the prefix `spiffe://trust.domain.org/Kubernetes/…` would be parsed as a Kubernetes identity according to the scheme structure in the following figure.

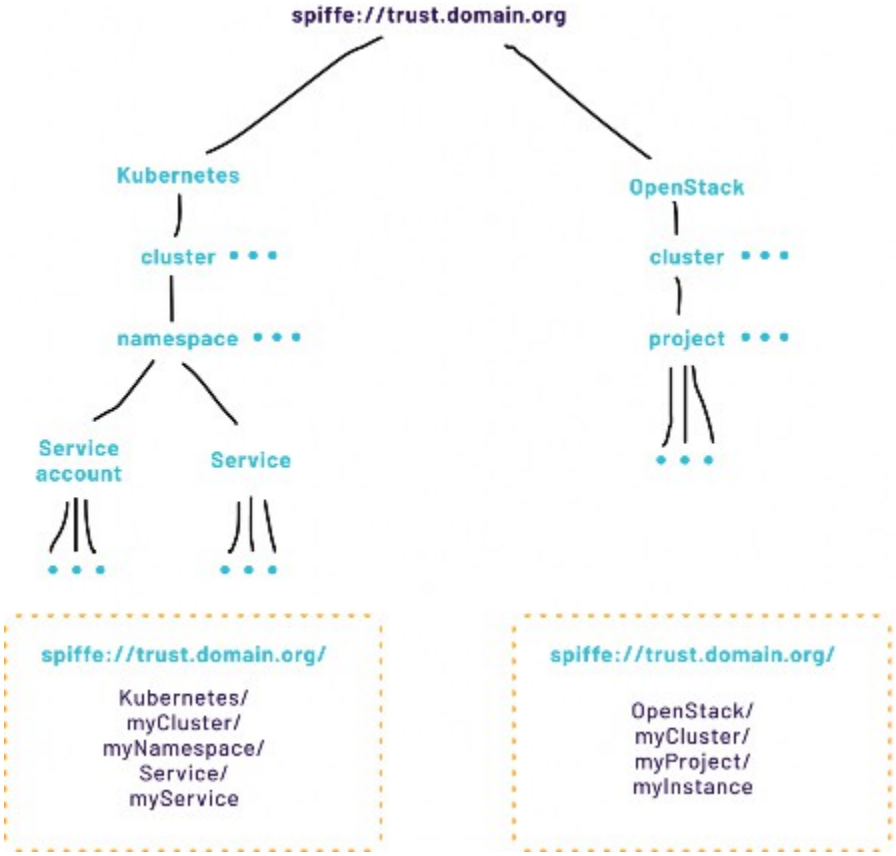


Figure 50: Illustration of another potential SPIFFE ID scheme.

Changing schemes More often than not, organizations change, and so will the requirements of the identity scheme. This could be due to the organizational restructuring, or even a shift in the technology stack. It might be hard to predict how different your environment could be in a few years from now. Therefore, when designing a SPIFFE ID scheme, it is crucial to think about potential changes in the future and how these changes would affect the rest of the systems based on the SPIFFE identity. You should think about how to incorporate backward and forward compatibility into the scheme. As we already mentioned before, with an ordered scheme, you only need to add new entities to the end of your SPIFFE ID; but what if you need to add something in the middle?

One method is with a scheme based on key-value pairs, and another method is one that we're all too familiar with: versioning!

Scheme based on key-value pairs We notice that the above scheme designs are all ordered. The scheme is evaluated by looking at the prefix of the identity and determining how to evaluate the suffix that follows. However, we note that because of this ordering, it is difficult to easily add new fields to the scheme.

```
spiffe://trust.domain.org/environment:dev/ ↵
  region:us/organization:zero/name: turtle
```

Key-value pairs, by their nature, are unordered, and this is one way to easily extend fields into an identity scheme without much change. For example, you can use key-value pairs with a known delimiter, e.g. column (`:`) character within the identity. In this case, the identity above might be encoded in the following way:

Because consumers of the identity process it into a set of key-value pairs, more keys can be added without changing the underlying structure of the scheme. There is also the possibility that SPIFFE can support the inclusion of key-value pairs into SVIDs in future.

As usual, trade-offs between structured and unstructured data types should be considered.

Versioning One of the possible solutions here is to incorporate versioning into the scheme. The version could be the first item and the most critical part of your scheme. The rest of the systems need to follow the mapping between versions and encoded entities when dealing with SPIFFE IDs data.

```
spiffe://trust.domain.org/v1/region/env/org/workload
v1 scheme:
0 = version
1 = region
2 = environment
3 = organization
4 = workload

spiffe://trust.domain.org/v2/region/datacenter/env/org/workload
v2 scheme:
0 = version
1 = region
2 = datacenter
3 = environment
4 = organization
5 = workload
```

In SPIFFE, a single workload can have multiple identities. However, it’s the responsibility of your workload to decide which identity to use. To keep authorization simple, it’s best to first have one identity per workload and add more if necessary.

Authorization Examples with HashiCorp Vault

Let's go through an example of a service that workloads may wish to talk to, Hashicorp Vault. We will go through an RBAC example and an ABAC example, and cover some of the gotchas and considerations when performing authorization with SPIFFE/SPIRE.

```
spiffe://example.org/<region>/<dev,stage,prod>/ ↵
  <organization>/<workload_name>
```

Vault is a *secret store*: administrators can use it to safely store secrets such as passwords, API keys, and private keys that services might need. Since many organizations still need to store secrets securely, even after using SPIFFE to provide secure identities, using SPIFFE to access Vault is a common request.

Configuring Vault for SPIFFE identities Vault handles both authentication and authorization tasks for identity when dealing with client requests. Like many other applications that handle the management of resources (in this case, secrets), it has a pluggable interface into various mechanisms for authentication and authorization.

In Vault, this is through the TLS Certificate Auth Method or JWT/OIDC Auth Method which can be configured to recognize and validate JWTs and X509-SVIDs generated from SPIFFE. To enable Vault to use SPIFFE identities to be used, the trust bundle needs to be configured with these pluggable interfaces so that it can authenticate the SVIDs.

This takes care of authentication, but we still need to configure it to perform authorization. To do that, a set of authorization rules need to be put into place for Vault to decide which identities can access secrets.

A SPIFFE RBAC example For the following examples, we will assume that we are using the X509-SVID. Vault allows the creation of rules, which can express which identities can access which secrets. This usually consists of creating a set of access permissions and creating a rule that ties it to access.

For example, a simple RBAC policy:

```
{
  "display_name": "medical-access-role",
  "allowed_common_names":
    ["spiffe://example.org/eu-de/prod/medical/data-proc-1",
     "spiffe://example.org/eu-de/prod/medical/data-proc-2"
  ],
  "token_policies": "medical-use",
}
```

This encodes a rule that states that if a client with identities `spiffe://example.org/eu-de/prod/medical/data-proc-1` , or `spiffe://example.org/eu-de/prod/medical/data-proc-2` can gain access to a set of permissions (`medical-use`), it will grant access to medical data.

In this scenario, we have granted these two identities access to the secret. Vault takes care of mapping two different SPIFFE IDs to the same access control policy, which makes this RBAC rather than an allowlist.

A SPIFFE ABAC example In some cases, it is easier to design authorization policies based on attributes, rather than roles. Typically, this is needed when there are multiple different sets of attributes that could individually match policies, and it is challenging to create enough unique roles to match each situation.

```
{
  "display_name": "medical-access-role",
  "allowed_common_names":
    ["spiffe://example.org/eu/prod/medical/batch-job\*"],
  "token_policies": "medical-use",
}
```

In line with the above example, we can create a policy that authorizes workloads with a certain SPIFFE ID prefix:

This policy states that all workloads with the prefix `spiffe://example.org/eu/prod/medical/batch-job` would be authorized to access the secret. This may be useful as batch jobs are ephemeral and may be given a randomly assigned suffix.

Another example would be a policy with the following:

```
{
  "display_name": "medical-access-role",
  "allowed_common_names":
    ["spiffe://example.org/eu-*/prod/medical/data-proc"],
  "token_policies": "medical-use",
}
```

The desired effect of this policy is to state that only data-proc workloads in any EU data center can access the medical secret. Thus, if a new workload is started in a new data center in the EU, any data-proc workload would be authorized to access the medical secrets.

Open Policy Agent Open Policy Agent (OPA) is a Cloud Native Computing Foundation (CNCF) project that performs advanced authorization. Using a domain-specific programming language called Rego, it efficiently evaluates the properties of an incoming request and determines what resources it should be allowed to access. With Rego, it is possible to design elaborate authorization policies and rules including ABAC and RBAC. It can also take into account the properties of the connection that are unrelated to SPIFFE, such as the user ID of an incoming request. The Rego policies are stored in text files so they can be centrally maintained and deployed through a continuous integration system, and even unit tested.

Here is an example which encodes access to a certain database service which should only be allowed by a certain SPIFFE ID.

```
# allow Backend service to access DB service
allow {
  http_request.path == "/good/db"
  http_request.method == "GET"
  svc_spiffe_id == "spiffe://domain.test/eu-du/backend-server"
}
```

If more elaborate authorization policies need to be implemented, then OPA is a great choice. The Envoy proxy integrates both with SPIRE and OPA, so it is possible to get started right away without changing service codes. To read more details about using OPA for authorization please consult with OPA documentation.

Summary

Authorization is an enormous and complex topic in its own right, far beyond the scope of this book. However, like many other aspects of the ecosystem that interacts with identity, it is useful to understand the relationship of identity with authorization (and more broadly, policy).

In this chapter, we've introduced several ways to think about authorization using SPIFFE identities, as well as design considerations related to identity. This will help better inform the design of your identity solution to cater to the authorization and policy needs of your organization.

9. Comparing SPIFFE to Other Technologies

This chapter compares SPIFFE to other technologies that solve similar problems.

Introduction

The problems that SPIFFE and SPIRE solve are not new. Every distributed system has to have some form of identity to be secure. Web Public Key Infrastructure, Kerberos/Active Directory, OAuth, secret stores, and service meshes are examples.

However, these existing forms of identity are not a good fit for identifying internal services within an organization. Web PKI is challenging to implement and also insecure for typical internal deployments. Kerberos, the authentication component of Active Directory, requires an always-online Ticket Granting Server and doesn't have any equivalent attestation. Service meshes, secrets managers, and overlay networks all solve portions of the service identity puzzle but are incomplete. SPIFFE and SPIRE are currently the only complete solution to the service identity problem.

Web Public Key Infrastructure

Web Public Key Infrastructure (Web PKI) is the widely used method to connect from our web browsers to secure web sites. It utilizes X.509 certificates to assert that the user is connecting to the website they intend to visit. Since you are probably familiar with this model, it's reasonable to ask: why can't we use Web PKI for service identity within our organization?

In traditional Web PKI, certificate issuance and renewal were entirely manual processes. These manual processes are a poor fit for modern infrastructure, where service instances may dynamically grow and shrink at any time. However, in the last few years, Web PKI has shifted to an automatic certificate issuance and renewal process called Domain Validation (DV).

In DV, the certificate authority sends a token to the certificate requester. The certificate requester shares this token using an HTTP server. The certificate authority accesses the token, verifies it, and then signs the certificate.

The first problem with this arrangement is that internal services frequently don't have individual DNS names or IP addresses. If you wanted to do mutual TLS between all services, then even the **clients** would need DNS names to get certificates, which would be challenging to configure. Assigning identities to multiple services running on one host requires separate DNS names, which would also be challenging to configure.

A more subtle problem is that anyone who could successfully respond to requests for the token could successfully acquire a certificate. This might be a different service running on the same server, or even on a different server that can tamper with the local Layer 2 network.

In general, while Web PKI works great for secure web sites on the internet, it is not a good fit for service identity. Many internal services that need certificates don't have DNS names. The currently available process for doing certificate validation is easily compromised if attackers successfully infiltrate any services on local network.

Active Directory (AD) and Kerberos

Kerberos is an authentication protocol initially developed at MIT in the late 1980s. Originally it was designed to allow for human-to-service authentication using a centralized user database. Kerberos was later expanded to support service-to-service authentication and the use of *machine accounts* in addition to user accounts. The Kerberos protocol itself is agnostic to the account database. However, Kerberos's most common usage is authentication in a Windows domain, using Active Directory (AD) as the account database.

The core credential of Kerberos is called a *ticket*. A ticket is a credential that can be used by a single client to access a single resource. Clients get tickets by calling a Ticket Granting Service (TGS). Clients need a new ticket for every resource it accesses. This design leads to much more chatty protocols and decreases reliability.

All services have a trust relationship with the TGS. When a service registers with the TGS, it shares key material, such as a symmetric secret or a public key, with the TGS. TGS uses key material to create tickets that authenticate access to the service. Rotating the key material requires coordination between the service and the TGS. The service must accept previous key material and remain aware of it so that existing tickets remain valid.

How SPIRE mitigates the Kerberos and AD drawbacks In SPIRE, each client and resource will call the SPIRE Server once to get its credentials (SVID), and all resources can authenticate those credentials in the trust domain (and federated trust domains) without any further calls to the SPIRE Server. SPIRE's architecture avoids all the overhead of getting a new credential for every resource that needs to be accessed.

Authentication mechanisms based on PKI, such as SPIRE, make credential rotation simpler: This coordination of key material between services and a centralized authenticator does not exist.

Finally, it is worth noting that the Kerberos protocol tightly couples services with hostnames, which complicates multiple services per host and clusters. On the other hand, SPIRE easily supports multiple SVIDs per workload and clusters. It is also possible to assign the same SVID to multiple workloads. These properties provide a robust and highly scalable approach to identity.

OAuth and OpenID Connect (OIDC)

OAuth is a protocol designed to enable access *delegation*, and not necessarily as a protocol for enabling authentication on its own. OIDC's primary purpose is to allow humans to allow a secondary website (or mobile app) to act on their behalf against a different primary website. In practice, this protocol enables authentication of users on the secondary website since the delegated access credential (an *access token* in the OAuth protocol) is proof from the primary website that the user authenticated against that website.

If the primary website includes user information or provides a way to retrieve user information using an access token, a secondary website may authenticate the user using the primary website's token. OpenID Connect, an opinionated flavor of OAuth, is a great example.

OAuth is designed for people and not for non-person entities. The OAuth login process requires a browser redirect with interactive passwords. OAuth 2.0 is similar to its predecessor and includes support for non-person entities, usually by creating *service* accounts (i.e. user identities representing workloads instead of people). When a workload wants to obtain an OAuth access token to access a remote system, it must use an OAuth client secret, password, or a refresh token to authenticate to the OAuth provider and receive the access token. Workloads should all have independent credentials to enable a high degree of granularity of workload identities. The management of these credentials quickly becomes complicated and difficult for elastic compute since each workload and identity must register with the OAuth provider. Long living secrets introduce further complexities when they must be revoked. Propagation of secrets in your environment, due to rotation, reduces infrastructure mobility and, in some cases, may present a vector of attack if developers manage secrets manually.

How SPIFFE and SPIRE can mitigate OAuth and OIDC complexity The reliance on a pre-existing credential to identify a workload, such as an OAuth client secret or refresh token, fails to resolve the bottom turtle problem (as explained in Chapter 1). Leveraging SPIRE as the identity provider in these instances permits the *bootstrap credential* or bottom turtle's issuance before contacting the OAuth infrastructure.

SPIRE significantly improves security since no long-lived static credential needs to be co-deployed with the workload itself. SPIFFE can be complimentary to OAuth. It removes the need to manage OAuth client credentials directly—apps may use their SPIFFE ID to authenticate to the OAuth provider as needed. Indeed, OAuth access tokens can be SVIDs themselves, allowing users to authenticate to services in a SPIFFE ecosystem in the same way as workloads. See the integration with OIDC for more.

Secrets Managers

Secrets managers are tools (e.g. HashiCorp Vault, OpenBao, Square Keywhiz) that securely store, manage, and distribute sensitive information** like passwords, API keys, certificates, and encryption keys. They typically offer a central "*vault*" where secret data is encrypted at rest, along with features for access control, auditing, and sometimes even on-demand cryptographic operations (e.g. encrypting or decrypting data on behalf of workloads). In practice, an application or service must authenticate to the secrets manager (the vault) before it can retrieve or use a secret. This introduces a classic chicken-and-egg problem: **how does the workload securely obtain the credential needed to authenticate to the secrets manager in the first place?**

This initial credential is often called "**secret zero**" or the *bootstrap credential*, referring to the foundational secret at the bottom of the trust stack (the proverbial "bottom turtle"). Securely provisioning this bootstrap secret is non-trivial–if it's stored insecurely or hard-coded, it can become a single point of failure for the entire security model. This challenge of **secure introduction** has long been a pain point in secrets management.

How SPIFFE and SPIRE can be used to mitigate secrets managers challenges Using a secret manager dramatically improves the security posture of systems that rely on shared secrets by providing a secure location by which those secrets can be stored, retrieved, rotated, and revoked. Heavy use, however, perpetuates the use of shared secrets rather than using strong identities.

Traditional secrets managers can integrate with SPIFFE (for example, OpenBao supports X.509 certificate auth, which can be tied to SPIFFE IDs), allowing a SPIRE-issued certificate to bootstrap trust without any human-injected secret. However, *SPIRE itself is not a secrets store*: it doesn't hold application passwords or keys for you. You still need a secrets management system–but SPIFFE can drastically improve how you authenticate to it.

SPIKE–A SPIFFE-First Secrets Manager

In earlier chapters, we discussed how SPIFFE can eliminate the need for a static bootstrap secret by using *workload attestation and SVIDs*. Recall that SPIRE (a SPIFFE runtime environment) issues each workload an **SVID** (SPIFFE Verifiable Identity Document), which is a short-lived X.509 certificate or JWT that serves as the workload's identity. Workloads can use these SVIDs to authenticate to other systems, including secret stores, *in lieu of presenting a pre-shared password or token*.

In other words, the SPIFFE identity itself becomes the "credential" for secrets retrieval. This dramatically reduces the "secret zero" problem–instead of embedding a long-lived credential with the application, the workload obtains a strong identity via attestation, and that identity is recognized by the secrets manager.

With the emergence of SPIFFE, a new class of secrets managers has been designed **from the ground up** around SPIFFE identities. SPIKE (Secure Production Identity for Key Encryption) is a prime example: a modern, SPIFFE-native secrets management system built to leverage SPIRE for authentication and trust. In essence, SPIKE integrates an identity control plane (SPIFFE) directly into its architecture, making **workload SVIDs the primary method of authentication and authorization** rather than traditional static tokens or credentials. This means that every SPIKE component and every client workload uses strong identities (SVIDs) to mutually authenticate over mTLS for any secret operations.

By doing so, SPIKE addresses the common challenges of secrets management–especially the bootstrap problem–in a *clean, "zero trust" way* where no component trusts another unless it presents a valid SPIFFE-issued identity.

SPIKE architecture uses SPIFFE identities for every interaction. An application with an SVID retrieves its secrets over an mTLS connection to SPIKE Nexus, while administrators use the SPIKE Pilot CLI (*with an admin SVID*) to manage secrets. Under the hood, SPIKE Nexus stores secrets securely and relies on SPIKE Keeper instances (each with its own SVID) for distributed root key shards. All connections between components–app, Nexus, Pilot, and Keepers–are mutually authenticated via SPIFFE SVIDs, eliminating the need for any static passwords or "secret zero" in the system.

SPIKE's design and components The SPIKE system is composed of three core components:

- **SPIKE Nexus:** a central service that acts as the **secure secrets store** (the equivalent of the vault). All secret data is handled through Nexus, which keeps secrets in memory and encrypts any persisted data on disk.
- **SPIKE Pilot:** a CLI and administrative API used to manage secrets and policies. Operators (or automation) use Pilot to add, update, or revoke secrets in SPIKE Nexus. Pilot itself is a SPIFFE-aware client; it authenticates to SPIKE Nexus with its SVID over mTLS.
- **SPIKE Keeper:** one of potentially several lightweight helper processes responsible for *holding encrypted shards of the Nexus root key* (more on this below). Keepers provide high availability and recovery: they enable Nexus to reconstruct its master encryption key if it restarts, without human intervention.

All these components register with the SPIRE server (or another SPIFFE Identity Provider) to obtain their SVIDs, establishing a common identity trust domain. The application workloads that need to fetch secrets also get their own SVIDs from SPIRE. As illustrated above, an application connects to SPIKE Nexus using mTLS–the client presents its workload SVID and the Nexus presents its service SVID, mutually verifying each other. **No username, API key, or static token is required**; trust is built entirely on SPIFFE certificates.

This not only solves the bootstrap authentication problem (the app's identity is its auth credential), but also means every request to the secrets manager carries a strong attested identity that can be logged and authorized.

In SPIKE, **access control policies** are tied to SPIFFE IDs: by default no workload can access any secret until an admin grants permissions for that workload's SPIFFE ID to perform specific actions on specific secrets. This approach aligns with the Zero Trust model–every actor must prove its identity cryptographically on each interaction.

From a security perspective, SPIKE ensures that even if an attacker somehow intercepts the storage layer or network traffic, they cannot extract secrets without possessing a valid SVID. All inter-component communications are encrypted and mutually authenticated via SPIFFE mTLS. Additionally, SPIKE treats its persistent storage as untrusted–any secret material written to disk is encrypted with a *root key* that only SPIKE Nexus (in memory) can decrypt. This brings us to one of SPIKE's most notable design elements: *how it handles the root encryption key*.

No More Manual Unsealing A hallmark of legacy secrets managers is the "**unseal**" process. When the secrets store crashes and restarts, or is locked for any reason, an unseal key is needed to decrypt its data store at startup–an operation often gated by human operators holding pieces of the key. This is another facet of the bootstrap problem: Bringing a secrets manager online securely can require a secret of its own (the unseal key) and human coordination.

SPIKE eliminates the need for a manual unseal step by leveraging its SPIFFE identities and an automated **root key sharding** mechanism.

When SPIKE Nexus first initializes, it generates a strong random *root key* to encrypt the backing data store. Instead of storing this root key on disk (which would be dangerous) or asking an operator to manually supply and split a key (*because human intervention could equally be risky and error-prone*), SPIKE Nexus automatically *splits the root key into shards* using Shamir's Secret Sharing algorithm.

It then securely distributes these *shards to the SPIKE Keeper instances*, each of which holds one shard in memory. The number of Keepers and the threshold of shards needed for recovery are configurable, allowing a trade-off between redundancy and security paranoia (e.g. you could require 2 of 3 Keepers or 4 of 5, etc., to reconstruct the key).

SPIKE Nexus **never** writes its root encryption key to disk. Instead, it splits the key into multiple **shards** (using Shamir's algorithm) and hands these out to SPIKE Keepers. Only if enough shards are recombined (e.g. any 2 out of 3) can the root key be reconstructed. This means an attacker can't recover the root secret unless they compromise a threshold of Keeper instances. SPIKE Nexus will automatically reassemble the root key at startup by retrieving shards over mTLS from the Keepers, avoiding any manual "unseal" procedure.

This design provides **hands-off resilience**. If SPIKE Nexus crashes or restarts, it can automatically *recover its root key* by contacting a quorum of SPIKE Keepers, each sending back its in-memory shard (over an authenticated channel). As long as the required threshold of Keeper instances are running, Nexus can restore the key and resume serving secrets without operator intervention. There's no need for a human to input an unlock key or for the system to store a master key in an insecure location.

Even in a disaster scenario where the entire system goes down, the shards can be combined offline by administrators (each admin might hold an encrypted copy of a shard for "break-glass" recovery), but under normal conditions, **SPIKE essentially self-unseals**. Both the root key and the individual shards live only in memory and are never persisted, greatly limiting the window of exposure.

By removing the manual unseal step, SPIKE not only streamlines operations but also reduces risk: there is no static unseal key to guard or accidentally leak, and no need to trust humans with combining key parts except in true emergencies.

This is a clear example of SPIKE's overarching philosophy–use platform-provided identity and cryptography to automate away "secret-of-the-day" problems that have historically plagued secure infrastructure.

Bridging Identity and Secrets in Practice It's worth emphasizing why a system like SPIKE is needed at all, given a strong SPIFFE-based identity infrastructure. In an ideal world, every service-to-service authentication could rely solely on SPIFFE identities, and secrets (like passwords or API tokens) might seem unnecessary. **Reality, however, is more complex**. Applications still often need to consume secrets for various reasons: connecting to databases or third-party APIs that don't (yet) accept SPIFFE authentication, handling data encryption keys, or managing certificates and keys for external systems.

Having a central secrets manager greatly simplifies secret distribution and rotation. SPIKE fills this need while **complementing the SPIFFE ecosystem**–it ensures that managing those inevitable secrets does not reintroduce the very static credentials and ad-hoc trust that SPIFFE set out to eliminate.

In a real-world deployment, SPIRE and SPIKE work in tandem. For example, imagine a microservice that needs to access a legacy database using a username/password. Without SPIFFE, you might have baked that password into a config file or used a traditional secrets store with a manually provisioned token.

With SPIRE and SPIKE, the microservice simply presents its SPIFFE SVID to SPIKE Nexus over mTLS; if authorized, it receives the database password secret. The secret was stored encrypted in SPIKE Nexus, protected by the SPIKE root key, and only released to workloads with the right SPIFFE ID and privileges. No static passwords were ever floating around–the only long-lived trust anchor is the SPIFFE federation (e.g. the root CA of the trust domain), which is managed by SPIRE.

This model demonstrates **the natural evolution of a SPIFFE-based security architecture**: once identities are in place everywhere, they can be used to secure everything else, including the distribution of traditional secrets. SPIKE acts as a logical extension of SPIRE's identity platform, providing a secure store for the things you can't eliminate but can now tightly control with identity.

Sysadmins `#sleepmore`: They can use their own SPIFFE identities (or a SPIFFE-authenticated CLI like SPIKE Pilot) to perform secret management, instead of maintaining separate admin credentials for the secrets store. And because SPIKE is built on SPIFFE, it inherits properties like short-lived credentials and automated rotation. For instance, if an operator's or workload's SVID expires or is revoked, their access to SPIKE automatically expires too–aligning secret access with dynamic trust.

In summary, SPIKE brings secret management into the fold of cloud-native identity, so that adding secrets to your system doesn't break your Zero Trust posture.

VMware Secrets Manager – An Alternative SPIFFE-Enabled Approach SPIKE is not the only project marrying SPIFFE to secrets management. **VMware Secrets Manager (VSecM)** is another SPIFFE-compatible secret store that arrived around the same time, aiming to improve secrets management in cloud-native environments.

It's encouraging to see multiple implementations (SPIKE, VSecM, and others) exploring the integration of SPIFFE into secrets management. These innovations demonstrate that the concept of an identity-first secrets manager is broadly applicable: regardless of the specific implementation details, the core idea is to replace the "authenticate-with-secret-to-get-secret" cycle with a cleaner "authenticate-with-identity-to-get-secret" approach.

The Future of Secrets Management

As organizations adopt SPIFFE for workload identity, it's a natural next step to modernize secrets management in parallel. Solutions like SPIKE illustrate what **the next generation of secrets managers** looks like when designed for a SPIFFE world: they are more secure, easier to operate, and seamlessly fit into a Cloud Native workflow.

Secrets managers aren't going away–even in an identity-rich environment, we still need to handle actual secret material. But with SPIFFE and projects like SPIKE, we can handle that secret material more safely and elegantly.

SPIKE exemplifies the evolution of the "bottom turtle" in security architecture: the base of your trust stack is no longer a static secret, but a robust identity system that in turn safeguards all other secrets.

This is a profound shift, and it cements the role of SPIFFE not just in authentication and encryption, but as a foundation for **practical, real-world secret management** in modern systems. With SPIKE (and its kins like VSecM), the SPIFFE community is effectively saying: *let's solve secret management with the same principles of identity, automation, and least privilege that we apply elsewhere*–bringing us one step closer to a world where infrastructure is secure by default.

Service Meshes

A service mesh aims to simplify communication between workloads by providing automatic authentication, authorization, and enforcing mutual TLS between workloads. A service mesh typically provides integrated tooling that:

- Identifies workloads
- Mediates communication between workloads, usually through a proxy deployed adjacent to each workload (sidecar pattern)
- Ensures each adjacent proxy enforces a consistent authentication and authorization policy (generally through an authorization policy engine)

All the major service mesh packages include a native platform-specific service authentication mechanism.

While a service mesh can function without a cryptographic identity plane, weak forms of identity are inevitably created to permit service-to-service communication and discovery. Service mesh in this implementation does not provide a security function and also does not resolve the existing root of trust identity issue discussed previously.

Many service mesh offerings implement their cryptographic identity planes or integrate with an existing identity solution to provide both transit communication security and root of trust resolution. Most service mesh offerings implement SPIFFE or parts of it. Many service mesh implementations have adopted partial implementations of the SPIFFE specification (including Istio and Consul) and can be considered SPIFFE identity providers. Some incorporate SPIRE as a component of their solution (i.e. Grey Matter or Network Service Mesh).

For example, Istio uses SPIFFE for node identification, but its identity model is tightly coupled to and solely based on Kubernetes specific primitives. There is no way to identify services in Istio based on attributes outside of Kubernetes. IBM explains Why the current Istio mechanism is not enough. This presents a constraint on Istio compared to a universal identity control plane like SPIRE when desiring a richer attestation mechanism, or when a service needs to authenticate off-mesh outside of Istio using a common identity system. An additional advantage of using SPIRE for workload identities is that it may secure communication not controlled by a service mesh. For such reasons, organizations sometimes integrate SPIRE with Istio and use the SPIFFE identity instead of the built-in Istio identity. IBM published an example located at: IBM/istio-spire: Istio identity with SPIFFE/SPIRE.

Service meshes are not direct alternatives to SPIFFE/SPIRE—instead, they are complementary, with SPIFFE/SPIRE acting as the identity solution for higher-level abstractions within the mesh.

Service mesh solutions that specifically implement the SPIFFE Workload API support any software that expects this API to be available. Service mesh solutions that can deliver SVIDs to their workloads and support the SPIFFE Federation API can establish trust automatically between mesh-identified workloads and workloads running SPIRE or running on different mesh implementations.

Overlay Networks

Overlay networks simulate a single unified network for services across multiple platforms. Unlike a service mesh, an overlay network uses standard networking concepts such as IP addresses and routing tables to connect services. The data is encapsulated and routed across other networks, creating a virtual network of nodes and logical links built on top of an existing network.

While the most common overlay networks have no authentication features, the latest ones do. However, they still don’t attest the identity of services before allowing them to connect. Typically, they rely on a pre-existing certificate. SPIFFE is a good fit for providing certificates for overlay network nodes.

10. Practitioners' Stories

This chapter includes five stories from practitioners who are engineers at real-world businesses that have deployed SPIFFE and SPIRE.

Uber: Securing Next-gen and Legacy Infrastructure Alike with Cryptographic Identity

Ryan Turner, Software Engineer 2, Uber

Over the last decade, Uber has become the poster child for explosive growth. As the number of software services and the geographic scale we operate at grew, so did complexity and risk. To meet the growing needs, we started building our next-generation infrastructure platform. Simultaneously, a couple of years back, we saw some early traction with the open source projects SPIFFE and SPIRE.

We immediately saw the value SPIFFE could bring by enabling us to strengthen our next-generation infrastructure security posture. We have rolled out SPIRE at Uber and are now using it to establish trust across various workload environments using cryptographically verifiable identities. We started with a few application services and internal services, such as a workflow engine that spins multiple dynamic workloads to complete specific tasks by accessing data across the platform. SPIRE provides SPIFFE identities to our workloads across our application cycle. SPIFFE is used to authenticate services and helps us avoid misconfigurations that might result in production issues.

Retrofitting SPIRE into the legacy stack SPIRE is now a key component of Uber's next infrastructure, but we are also using a sidecar approach to retrofit authentication into legacy infrastructure. While SPIFFE and SPIRE are commonly known to work in modern, cloud native architectures, we can adapt the projects to our proprietary legacy stack quickly. SPIRE can provide a critical bridge of trust within Uber's next-gen and legacy infrastructure and positively impact internal security and developer efficiency.

Along our journey, the SPIFFE community has been very supportive in helping us find solutions. As a result, our engineers have actively been making code contributions to the projects as well.

Security, development, and audit teams are benefiting from SPIFFE SPIFFE gives our security team more confidence in the back-end infrastructure and less reliance on network-based security controls. As we deal with financial data and operate across geographic boundaries, we have to control access to financial and customer data. With SPIRE, we can provide a strongly attested identity for access control. It helps us meet these requirements and reduces the burden on audit teams in the process.

Our development teams at Uber use consistent client libraries to create AuthZ policies using SPIFFE-based identities. The projects have enabled development teams to take advantage of workload identity primitives such as X.509 and JWT without needing a deep understanding of complex topics such as trust bootstrap, secure introduction, credential provisioning, or rotation.

Pinterest: Overcoming the Identity Crisis with SPIFFE

Jeremy Krach, Senior Security Engineer, Pinterest

In 2015, Pinterest was having an identity crisis. The infrastructure at the company was growing in diverse and divergent directions. Each new system solved authentication—the identity problem—in its unique way. Developers were spending hours each month in meetings and security reviews to design, threat-model, and implement their customized identity solutions or integrate their new service with disparate identity models of its dependencies. It became clear that the security team needed to build a common infrastructure to provide identity in a generic way that could be used across our heterogeneous services.

The initial draft of this system delegated identity to machines as X.509 certificates based on hostnames. It was heavily used for secrets management (see Knox), but broader adoption had not yet occurred. As we continued to scale, in particular with multi-tenant systems such as Kubernetes, we needed more finely tuned identities that weren't tied to specific hosts in our infrastructure, but rather to the identity of the service itself.

Enter SPIFFE.

Flattening the complexities with SPIFFE SPIFFE now provides uniform identity across most of our infrastructure. We initially started with Kubernetes, as the need was the most explicit in that multi-tenant environment. Later we moved the rest of the infrastructure to SPIFFE as its primary form of identity. As a result, nearly every service at Pinterest has a standardized name that we can use and there's no obscure conventions or disjointed schemes. It has helped us unify and standardize our identity conventions, which aligned with other internal projects to identify service attributes such as service ownership.

We leverage SPIFFE as the identity in ACLs for secrets management, mutual TLS service-to-service communication, and even generic authorization policies (via OPA, another CNCF project). Knox, Pinterest's open source secrets management service, uses SPIFFE X.509 identity documents as one supported authentication method. See our blog post about adding SPIFFE support into Knox.

Dev, Sec, and Ops are in harmony again. SPIFFE makes it easier for the security team to write authorization policies. Developer velocity is significantly better as our engineers don't have to worry about custom schemes or disparate integrations for authentication. As we now have a standard way to interpret identity across our infrastructure it is much easier for billing and ownership teams to determine who owns a service. Having a strong sense of identity is also convenient for logging and tracing consistency. We're excited about the future of the SPIFFE project and thankful for its ability to help us solve our identity crisis!

ByteDance: Providing Dial Tone Authentication for Web-scale Services

Eli Nesterov, Security Engineering Manager, ByteDance

Bytedance, the company behind TikTok, has built and deployed large-scale Internet services worldwide that serve millions of users. Our infrastructure supporting these services is a mix of private data centers and public cloud providers. Our applications run on multiple Kubernetes clusters and dedicated nodes across platforms in the form of thousands of microservices.

As we grew in scale and size, we had multiple authentication mechanisms across our platforms using everything from PKIs, JWT tokens, Kerberos, OAuth, and custom frameworks. Add numerous programming languages to these authentication mechanisms, and the operational complexity and risk increased even more. Operationally it became complex for our security and operations team to manage these authentication schemes. In the case of a known vulnerability in an authentication framework, we could not move swiftly as each framework had to be dealt with separately. In some cases, they had code level dependencies, which made it even harder to change. Audit and compliance across geographic boundaries were challenging as each platform-specific authentication approach had to be reviewed and governed separately.

Figure 51: Heterogeneous architecture.

A move towards zero trust-based architecture in general, and an effort to improve our developer productivity, forced us to build a unified identity management plane for our services that would scale to our growing needs.

Building web-scale PKI with SPIRE It is hard to build an identity system that can work across different islands of infrastructure or platforms like ours. We could have created our own, but it would have required a significant amount of effort. We went ahead with SPIRE as it provided the scale and flexibility in supporting a wide variety of platforms we needed and at web-scale.

Since it offers cryptographic identity based on standard X.509 certificates, it helps us easily enable mutual TLS, which, by default, meets a lot of the compliance requirements. Extensibility and being open source were another plus as we could easily integrate it with our existing control plane and data stacks.

Transparent authentication has simplified operations With SPIRE we can deploy a consistent, “dial-tone” authentication across all our platforms. The burden of authentication and security is now encapsulated from the developers so they can focus on business or application logic. This has improved our deployment velocity overall. We are also less likely to get “production errors” due to configuration issues such as using development credentials in production.

Standardized authentication with SPIRE has also simplified compliance and audit since we have mutual TLS across trust domains and platforms. SPIRE also has allowed us to move to a more semi-decentralized model in terms of identity distribution where the identity system is local to say a data center. This improves our overall availability and positions us well for recovery.

We are pretty much “future proof” with SPIRE since it can scale and adapt to meet our growing business needs.

Anthem: Securing Cloud Native Healthcare Applications with SPIFFE

Bobby Samuel, VP AI Engineering, Anthem

Rising healthcare costs in the industry are compelling organizations like Anthem to rapidly innovate and rethink how we interact with providers, employer groups, and individuals. As part of this initiative, we are developing a host of applications that will help us drive costs down by securely opening healthcare data access. We have started building the supporting next-generation infrastructure based on cloud native technologies such as Kubernetes. This new infrastructure will drive rapid innovation and engage a broader ecosystem of organizations and developers. An example of this would be our HealthOS platform. HealthOS will enable third parties to build HealthApp capabilities to deliver into front-end interfaces, leveraging an ocean of de-identified health data.

But at nearly every major enterprise, especially healthcare organizations, someone is trying to get their data with malicious intent. Protected Healthcare Information (PHI) sells for much higher than financial information; thus, malicious actors such as hackers and script kiddies, find healthcare systems and the corresponding health information highly lucrative. With the adoption of cloud native architectures, the risk and complexity rise further. The risk of a breach is even higher since the threat radius increases significantly while manual security reviews and processes become *cloud-scale* inhibitors.

Building a foundation for zero trust architecture We could not rely on traditional parameter-based security tools and processes to secure our next-generation applications and infrastructure. Zero trust, a fine-grained, automated approach to security, made a lot of sense to us, especially in the future, as we plan to operate across organizational boundaries and cloud providers.

Identity and authentication for both users and services are among the zero trust security model's core principles. Zero trust allows us to rely less on network-based controls than authenticating every system or workload. SPIFFE and SPIRE have enabled a foundational authentication layer for our zero trust security architecture. They allow each workload to cryptographically prove "who they are" before they start communicating.

A shift away from secret management Typically when you think of authentication, you think of usernames, passwords, and bearer tokens. Unfortunately, these types of credentials were becoming a risk and source of complexity at Anthem. They tend to be long-lived, and management or rotation of these was tricky. We wanted to shift away from this type of secret management practices in general. Instead of asking a service, "what do you have," we want to ask, "who are you." In short, we wanted to move to cryptographic identities, such as SPIFFE. We can see additional benefits of using a strongly attested identity in the future, such as establishing mutual TLS between workloads and bubbling identity up to the applications.

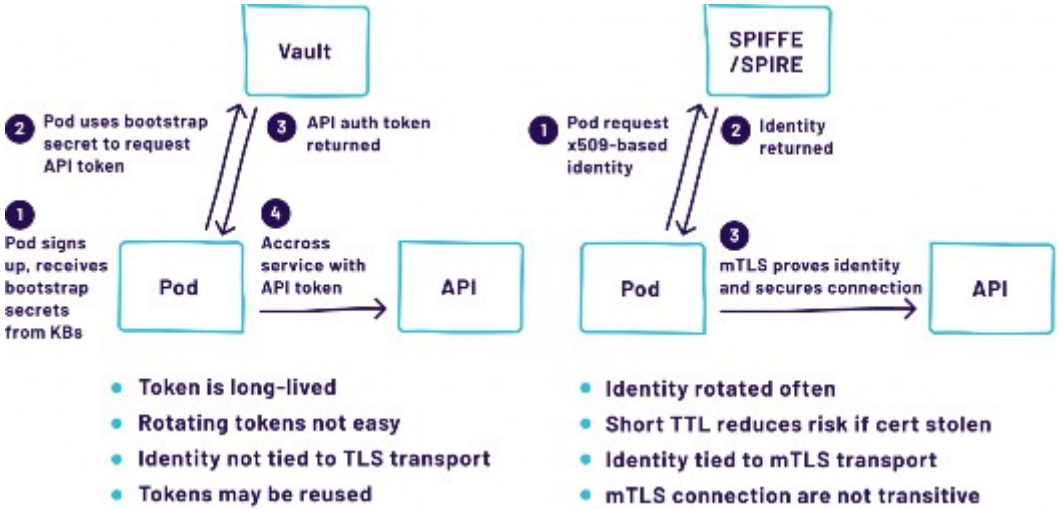


Figure 52: Shifting away from secrets.

Building security as part of the infrastructure with SPIFFE Security is often considered an inhibitor to deployment by the development teams. DevOps teams want to deploy new innovative features faster. However, they have to go through manual tickets, processes, integrations, and reviews related to security controls. At Anthem, we doubled down on removing obstacles for our development teams by making security a function of the infrastructure. With the adoption of technologies like SPIFFE, we can abstract the complexity of security controls away from development teams and provide consistent rules across various platforms. SPIFFE, along with other zero-trust-based technologies, will help us drive our system provision time from three months to less than two weeks in most scenarios.

Security is becoming an enabler at Anthem with SPIFFE leading the charge.

Square: Extending Trust to the Cloud

Michael Weissbacher and Mat Byczkowski, Senior Security Engineers, Square

Square provides a wide variety of financial services. Over its lifetime the company grew new business units from within the company, such as Capital and Cash, but also acquired companies like Weebly and Stitch Labs. Different business units use different technologies and can operate from different data centers and clouds while still needing to communicate seamlessly.

Our internally developed service identity system needed to scale beyond the internal architecture that Square developed for its data centers. We wanted to expand the system to the cloud and we wanted to provide an equally secure system that would also serve us well for years to come. We were ideally looking for an open standard-based tool that would also seamlessly integrate with Envoy proxy. Both SPIFFE and SPIRE supported our goals of growth and independent platforms working with multiple clouds and deployment tools.

An open standard that works with popular open source projects Since SPIFFE was based on existing open standards such as X.509 Certificates, it provided a clear upgrade path from how we did service identity. Envoy is the foundational building block of how apps communicate at Square. Since SPIRE supports Envoy's Secrets Discovery API, getting X509-SVIDs was easy. Envoy has access controls built-in that can use SPIFFE identities to decide what apps are allowed to communicate.

We deployed SPIRE architecture in parallel with the existing service identity system, then made changes to various internal tooling and frameworks to support both systems. Next, we integrated SPIRE with the deployment system to register all services in SPIRE. This meant we could stress test SPIRE's frequent SVID rotation. Finally, we used feature flags to slowly opt-in services to start using SVIDs in their service-to-service calls.

Seamless, secure connectivity across cloud and data centers SPIFFE and SPIRE are enabling our Security Infrastructure team to provide a critical bridge to securely connect different platforms and technologies. We're still in the early migration stage of moving to SPIRE, but the changes we put in place allowed us to seamlessly connect our production AWS EKS infrastructure with services deployed to Square's data centers. We are now working on an automated federation between our trust domains, as we have only federated manually before. We use SPIFFE identity as a standard even for the custom identity work we do in the company.

We are also really happy to get involved with the SPIFFE community, everyone has been friendly and helpful during our journey. The community has provided an added benefit of a good place to bounce off system design ideas for zero trust systems in general.

Glossary

- **Access Control**: A method of regulating access to resources. It includes both authentication and authorization.
- **ACLs (Access Control Lists)**: A set of rules that specifies which users or systems are granted access to objects, as well as what operations are allowed on given objects.
- **Active Directory**: A directory service developed by Microsoft for Windows domain networks.
- **Agile**: A set of software development practices that helps deliver applications faster through iterative processes instead of traditional waterfall processes.
- **API (Application Programming Interface)**: A set of functions that a service or application provides to be used by other services as an abstraction layer.
- **API Token**: A general term that refers to a unique identifier of a service used for authenticating or accessing an API.
- **Attestation**: Asserting certain properties of an entity. In the context of SPIRE, is referred to as the process of asserting properties of a Node or Workload.
- **Auditing**: A set of procedures for inspecting and examining a process or system, to ensure compliance to requirements.
- **Authentication**: The process of verifying the identity of a human user or a system. It's focused on answering "Who are you?"
- **Authorization**: The process of determining the permissions a human user or a system has on a resource. It's focused on answering "What can you access?
- **Blast Radius**: The reach that a compromised secret or faulty configuration problem might have.
- **BSD**: Berkeley Software Distribution.
- **Certificate Authority (CA)**: An entity that issues digital certificates.
- **Certificate Transparency**: A internet security standard and open source framework for monitoring and auditing digital certificates.
- **CI/CD (Continuous Integration/Continuous Deployment)**: The set of tools and practices that enable development teams to make frequent changes to applications and systems introducing automation into the development pipelines.
- **CLI**: Command Line Interface; a program that accepts text input to execute operating

system functions.

- **Cloud (also Cloud computing)**: A model of on-demand availability of computing power and storage that allows the users to consume resources, based on their needs, from a pool of resources provided by a vendor. In the process of doing so, this allows the delegation of certain management responsibilities to the vendor.
- **Cloud-aware or Cloud native**: An application designed to run on a cloud platform. Cloud native design patterns can scale horizontally, easily migratable, and resilient in distributed systems environments.
- **Common Name (CN)**: A field in certificates generally used to specify the host or server identity.
- **Container**: A unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.
- **Credential provisioning**: The process of generating and/or issuing credentials to a service or a system.
- **Cryptographic identities**: A piece of data that an entity can use to prove its identity using a cryptographic authentication protocol, e.g. An X.509 certificate.
- **Data store**: A repository for storing and managing collections of data.
- **Denial of Service (DoS)**: A type of cyber-attack in which the attacker tries to make a service or network unavailable for its intended users.
- **DevOps**: Combination of tools and practices that enable organizations to deliver applications at a high velocity, empowering teams to *own* the whole lifecycle of a service from **Dev**elopment to **Op**erations.
- **DNS (Domain Name Service)**: A hierarchical and decentralized naming system for computers, services, or other resources.
- **Encryption**: The process of encoding information, converting an original *plain-text* into another form known as *ciphertext*, through the use of algorithms to protect the confidentiality of the information from unauthorized parties.
- **Federation**: The union of self-governed domains. When two domains are federated it means a user or service can authenticate to one domain and then access resources in other domains without having to go through domain-specific authentication again.
- **Firewall**: A hardware/software appliance that governs network traffic. Most commonly used to create allow/deny rules to manage connections between entities.

- **Four eyes principle**: The mechanism that requires that the actions done by an individual must be reviewed by a second, independent individual.
- **GDPR (General Data Protection Regulation)**: Regulation in the European Union covering data protection and privacy.
- **gRPC**: A modern open source high-performance Remote Procedure Call (RPC) framework developed by Google.
- **Hardened**: The state of a system after being put through a set of steps to strengthen security. This can include several steps such as patching and configuration.
- **Hardware Security Module (HSM)**: A dedicated cryptographic processor designed to manage and safeguard sensitive keys, that provides encryption and decryption functions for digital signatures and other cryptographic functions.
- **High Availability**: The characteristic of systems that are resilient to failure, providing close to 100% uptime of service. This is achieved through the use of software and hardware redundancy making them available despite failures.
- **HTTPS**: Hypertext Transfer Protocol Secure is an extension of the HTTP protocol used for secure communication. It uses TLS to establish secure connections.
- **Human-in-the-loop systems**: A system that requires humans to make decisions and actions, e.g. manual workflow or approvals.
- **Interoperability**: The ability of a system to interact with another system even if it is deployed on a different platform or developed in a different language.
- **IP (Internet Protocol)**: Protocol, or set of rules, for sending data, from one computer to another on a network.
- **IPC (Inter-process Communication)**: A mechanism provided by the operating system to allow processes to communicate with each other.
- **JWT (JSON Web Tokens)**: An open and industry-standard (RFC 7519) method for representing claims securely between two parties.
- **JWT-SVID**: The token-based SVID in the SPIFFE specification, aimed at providing identity assertion across L7 boundaries. JWT-SVIDs are standard JWTs tokens with some extra restrictions applied.
- **Kerberos**: An authentication protocol based on tickets that allow computers to communicate on an untrusted network to prove their identity to one another in a secure way.

- **Kernel**: The central part of an operating system, which manages the memory, CPU, and device operations.
- **Key Management Service (KMS)**: A system that manages cryptographic keys. It deals with generating, exchange, storing, and revocation of keys.
- **Kill chain**: A concept that refers to the different phases of a cyber attack. It is a series of steps an attack takes to penetrate a system.
- **Kubernetes**: An orchestration system for Containers.
- **L3/L4**: The network (L3) and transport (L4) layers of the OSI networking model.
- **L7**: The application layer of the OSI networking model.
- **Lead time reduction**: The time taken to develop, test, and deploy an application into production.
- **Legacy**: Technology and infrastructure which is dated and is generally hard to maintain, or make work with modern technology.
- **Linux**: A family of open source operating systems based on the Linux kernel.
- **Logging**: The process of keeping a register of the events that happen during the functioning of a system.
- **Man-in-the-middle**: An attack where an attacker secretly intercepts the communication between two parties, stealing or altering sensitive information.
- **Memory Leak*: This occurs when the memory that is no longer needed is not released.
- **Metering**: The process of collecting usage information about services or microservice in terms of resource utilization, usually to enforce a quota or for resource management.
- **Merkle Tree**: A hash-based tree structure in which each leaf node is a hash of a block of data, and each non-leaf node is a hash of its children. Merkle trees are used in distributed systems for efficient data verification.
- **Microservices**: A modern architectural pattern in which applications are split into distinct, independently managed services for better scalability, re-usability, and rapid deployment.
- **Mint SVIDs**: To create a new cryptographically verifiable identity document.
- **Multi-cloud**: A system architecture that uses multiple cloud computing and storage vendors.
- **Multiregion**: A system deployment that crosses multiple regions in a cloud platform.
- **Mutual TLS (mTLS)**: Cryptographic protocol that ensures that traffic is both secure and trusted in both directions between a client and server.

- **NAT (Network Address Translation)**: A method for converting IP addresses across different network address spaces. The typical use case allows multiple devices to access the internet using a single public IP address.
- **Node/Worker**: A logical or physical entity that runs computational workloads in a computer systems context. A Node is the underlying compute system that a workload runs on.
- **Observability**: The ability to infer a system's internal state based on its external outputs.
- **OAuth**: OAuth is an open standard for access delegation, commonly used as a way for internet users to grant websites or applications access to their information on other websites but without giving them the passwords.
- **OIDC (OpenID Connect)**: Identity layer built on top of OAuth 2.0.
- **OpenStack**: An orchestration system for Virtual Machines.
- **OpenSSH**: A set of tools used to secure network communications, based on the Secure Shell protocol (SSH).
- **OWASP (Open Web Application Security Project)**: A non-profit foundation that is focused on improving the security of software. It provides tools, resources, education, and training for developers and security practitioners.
- **Perimeter-based security**: A security approach that establishes a strong perimeter, e.g. firewall, and then trusts activities or entities within that perimeter by default.
- **PKI (Public Key Infrastructure)**: Set of policies, procedures, and tools used to create, manage, and distribute digital certificates.
- **Proxy**: An application or appliance that sits in front of an application and intercepts connections being made to the application.
- **Public-key cryptography (also asymmetric cryptography)**: Cryptographic system that uses pairs of keys (private keys and public keys) created with special mathematical properties; one key to encrypt a piece of information and then the other key to decode it. This type of cryptography is used for creating digital certificates.
- **Remote Code Execution (RCE)**: A security vulnerability that allows an attacker to execute code from a remote server.
- **Root of trust**: A source in which another trust is built on. Usually referring to a cryptographic mechanism provided by hardware or by a trusted entity.
- **SaaS (Software as a Service)**: A model of software consumption where instead of software being bought and run by a user, the software is hosted and maintained by a

service provider.

- **Scheme**: When relating to identity, a convention that defines how identity can be parsed and understood.
- **Secure introduction**: The challenge of getting the first secret from a secrets distribution system.
- **Security group**: In the context of AWS, a Security Group acts as a virtual firewall to control incoming and outgoing traffic.
- **Selector**: A selector is an attribute or a property of a software entity that can be used to assert who that entity is. It can be a Node Selector or a Workload Selector.
- **Serverless**: A cloud computing execution model that allows developers to deploy code without worrying about the underlying infrastructure and resource allocation.
- **Service mesh**: A configurable infrastructure layer used for facilitating and managing communication between services,
- **Sidecar**: A process that runs alongside a workload to provide additional functionality to a system with minimal intrusion.
- **Signing**: The addition of a digital signature for verifying the authenticity of a message or document.
- **SIEM (Security Information and Event Management)**: A concept that enables organizations to collect and analyze data from applications, systems, and networks to detect potential malicious activities or attacks.
- **SLA (Service Level Agreement)**: An agreement between a service provider and client on the various aspects of a service such as quality and availability. SLO (Service Level Objective) is an agreement with the SLA about a particular metric.
- **SOX (Sarbanes-Oxley Act)**: An act passed in the United States to protect shareholders and the general public from accounting errors and fraudulent practices, resulting in additional compliance requirements that enterprises need to meet.
- **SPIFFE ID**: The string that uniquely identifies a workload or a service, e.g. spiffe://uk-inc.com/billing/payments
- **SSH (Secure Shell)**: Protocol that allows encrypted connections between a client and a server.
- **SAN (Subject Alternative Name)**: Alternate or additional hostnames to be protected by a single SSL or certificate.

- **SVID (SPIFFE Verifiable Identity Document)**: A document with which a workload proves its identity. It includes a SPIFFE ID in two currently supported formats: an X.509 certificate or a JWT token.
- **Telemetry**: Automated process of collecting data points and measurements for monitoring the functioning of a system.
- **TLS (Transport Layer Security)**: Cryptographic protocol that enables secure communications between two services by authenticating the identity of a service over a computer network. It is typically used by websites for secure communication between their web servers and web browsers.
- **URI (Uniform Resource Identifier)**: A string that unambiguously identifies a resource following a standard set of syntax rules.
- **Trusted Platform Module (TPM)**: A piece of hardware that can securely store cryptographic material.
- **Trust bootstrap**: Establishing trust on a service or system.
- **Trust Domain**: Corresponds to the trust root of a system, which could be an individual, organization, environment, or department. A SPIFFE trust domain is an identity namespace that is backed by an issuing authority.
- **Web Public Key Infrastructure (Web PKI)**: The set of systems and procedures to enable confidentiality, integrity, and authenticity to communications between Web browsers and Web servers.
- **Workload**: A workload is a single piece of software, deployed with a particular configuration for a single purpose that can consist of multiple running instances, where all of them perform the same task. The term *workload* may include a wide range of different definitions of a software system, including: “A webserver running a Python web application, running on a cluster of virtual machines with a load-balancer in front of it.”, “A MySQL database instance.”, “A worker program processing items in a queue.”, “A collection of independently deployed systems that work together, such as a web application that uses a database service. The web application and database could also individually be considered workloads.”
- **Windows**: An operating system developed by Microsoft
- **X.509**: A widely used standard format for public-key certificates. These certificates are used in many internet protocols, including TLS/SSL.

- **X509-SVID**: The X.509 representation of a SPIFFE Verifiable Identity Document (SVID).
- **Zero trust**: A security concept centered on the idea that organizations should not automatically trust anything based on whether a service is running inside or outside a perimeter and instead must verify everything trying to connect to its systems before granting access.

Epilogue

Before SPIFFE and SPIRE, organizations faced an immense challenge: adopting modern infrastructure practices caused their old security model to break down. Perimeters became increasingly porous and at the same time network properties like IP addresses became useless for access control.

SPIFFE and SPIRE help organizations rise to that challenge by providing unique, cryptographically verifiable identities for all of their services.

Once they have identities, services can implement higher-level security functionality including authentication, encryption, observability, and authorization.

Providing secure identities to services first requires a secure way to determine the identities of the servers, cloud instances, and clusters that they run on. That is the bottom turtle problem we referenced in the prologue. SPIFFE and SPIRE solve the bottom turtle problem by leveraging multiple factors to provide a strong proof of identity.

SPIFFE and SPIRE are open source projects and could always use more help! Visit the web site at spiffe.io and join the SPIFFE Slack workspace to engage with the growing community of users.

Trust in Zero.

Daniel Feldman, Emily Fox, Evan Gilman,
Ian Haken, Frederick Kautz, Umair Khan,
Max Lambrecht, Brandon Lum, Eli Nesterov,
Agustín Martínez Fayó, Andres Vega,
Michael Wardrop, Volkan Özçelik

spiffe

slack.spiffe.io spiffe.io