BeyondProd: A new approach to cloud-native security

Google has written several whitepapers explaining projects developed internally that help improve security. BeyondProd purposefully calls back to the concepts of BeyondCorp just as a perimeter security model no longer works for end users, it also no longer works for microservices. Adapting the original BeyondCorp (http://cloud.google.com/beyondcorp) *paper, "Key assumptions of this model no longer hold: The perimeter is no longer just the physical location of the ~~enterprise~~ [data center], and what lies inside the perimeter is no longer a blessed and safe place to host ~~personal computing devices and enterprise applications~~ [microservices]."*

In this whitepaper, we provide details on how several pieces of Google's infrastructure work together to protect workloads in an architecture that is now known as "cloud-native". For an overview of Google's security, see the Security Infrastructure Design whitepaper (https://cloud.google.com/security/infrastructure/design/).

The content contained herein is correct as of December 2019. This whitepaper represents the status quo as of the time it was written. Google Cloud's security policies and systems may change going forward, as we continually improve protection for our users.

Glossary

The following definitions are used in this document:

- A **microservice** separates the individual tasks an application needs to perform into separate services, each of which can be developed and managed independently, with their own API, rollout, scaling and quota management. In a more modern architecture, an application, such as a website, can be run as a collection of microservices instead of as a single monolithic service. Microservices are independent, modular, dynamic, and ephemeral. They can be distributed across many hosts, clusters, or even clouds.
- A **workload** is a unique task that an application completes. In a microservice architecture, a workload may be one or multiple microservices.
- A **job** is a single instance of a microservice, running some part of an application.

- A microservice uses a **service identity** to authenticate itself to other services running in the infrastructure.
- A **service mesh** is an infrastructure layer for service-to-service communication, which can control traffic, apply policies, and provide centralized monitoring for service calls. When using a microservice architecture, this removes the burden on individual services to implement these controls, and allows for simpler, centralized management across many microservices.

CIO-level summary

- Google's infrastructure deploys workloads as individual microservices in containers, and manages these workloads using Borg□our container orchestration system. This is an inspiration and template for what's widely known today as a "cloud-native" architecture.
- Google's infrastructure has been purposefully designed with security in mind; not added later as an afterthought. Our infrastructure assumes no trust between its services.
- Google protects its microservices with an initiative called BeyondProd. This protection includes how code is changed and how user data in microservices is accessed. BeyondProd applies concepts like: mutually authenticated service endpoints, transport security, edge termination with global load balancing and denial of service protection, end-to-end code provenance, and runtime sandboxing.
- Moving from a traditional security model to a cloud-native security model required us to make changes to two main areas, namely our infrastructure and our development process. Building shared components into a shared fabric enveloping and connecting all microservices, also known as a service mesh, made it easier to roll out changes and achieve consistent security across services.

Motivation

Google moved to containers and container orchestration to achieve higher resource utilization, build highly-available applications, and simplify work for Google developers. We had another motivation to move to a containerized infrastructure□to align our security controls with our architecture. It had become clear to us that a perimeter-based security model wasn't secure enough. If an attacker were to breach the perimeter, they would have free movement within the network. While we realized we needed stronger security controls throughout our infrastructure,

we also wanted to make it easy for Google developers to write and deploy secure applications without having to implement security features themselves.

Moving from monolithic applications to distributed microservices deployed from containers using an orchestration system had tangible operational benefits: simpler management and scalability. This cloud-native architecture required a different security model with different tools to protect deployments aligned with the management and scalability benefits of microservices.

This document describes how cloud-native security is implemented at Google, termed here as **BeyondProd**: what the change to cloud-native means for security, security principles for cloud-native security, systems built to address these requirements, and some guidance on how to tackle a similar change yourself.

Cloud-native security at Google

Containerized microservices

From its earliest days, Google made a conscious decision to build its data center capacity from low cost commodity servers, rather than invest in more expensive highly-available hardware. The guiding philosophy for our reliability was, and continues to be, that any individual part of a system should be able to fail without affecting the availability of user visible services. Achieving this availability required running redundant instances of services so that single failures would not lead to outages. A result of this philosophy was the development of containers, microservices, and container orchestration to scalably manage deployment of these highly redundant and distributed systems.

A containerized infrastructure means that each workload is deployed as its own set of immutable, moveable, scheduleable containers. To manage these containers internally, we developed a container orchestration system called Borg (https://research.google.com/pubs/pub43438.html?hl=es)[1] (#footnote-1), which we still use today to deploy several billion containers a week.

Containers made workloads easier to bin pack and reschedule across machines. Microservices made it easier to develop and debug different parts of an application. Used in combination, microservices and containers enable workloads to be split into smaller, more manageable units for maintenance and discovery. Moving to a containerized infrastructure with a microservice architecture is known as going "cloud-native (https://github.com/cncf/toc/blob/master/DEFINITION.md)". Services are run inside containers

deployed by Borg. This architecture scales workloads as needed—if there is high demand for a particular workload, there may be multiple machines running copies of the same service to handle the required scale of the workload.

Where Google stands out is that we have taken security into account as part of every evolution in our architecture. The more recent concept of cloud-native security is comparable to what Google has used for many years to secure our infrastructure. Our goal with this microservice architecture and development process is to address security issues as early in the development and deployment lifecycle as possible—when addressing issues is less costly—and do so in a way that is standardized and consistent. The end result is that developers spend less time on security while still achieving more secure outcomes.

Migrating to a cloud-native architecture

Modern security architectures have moved beyond a traditional perimeter-based security model where a wall protects the perimeter and any users or services on the inside are fully trusted. BeyondCorp was a response to a change in the way the modern corporate user works. Today, users are mobile and commonly operate outside an organization's traditional security perimeter such as from a coffee shop, from an airplane, or anywhere in between. In BeyondCorp, we dispensed with the idea of a privileged corporate network and authorized access based solely on device and user credentials and attributes regardless of a user's network location.

Cloud-native security addresses the same concern for services as it does for users—that in a cloud-native world, we can't simply rely on a firewall to protect the production network, just as we can't rely on a firewall to protect the corporate network. In the same way that users aren't all using the same physical location or device, developers are not all deploying code to the same environment. With BeyondProd, microservices may be running not only within a firewalled data center, but in public clouds, private clouds, or third-party hosted services, and they need to be secure everywhere.

Just like users move, use different devices, and connect from different locations; microservices also move and are deployed in different environments, across heterogeneous hosts. Where BeyondCorp states that "***user*** *trust should be dependent on characteristics like* ***the context-aware state of devices*** *and not the ability to connect to the corp network*", BeyondProd states that "***service trust*** *should be dependent on characteristics like* ***code provenance and service identity****, not the location in the production network, such as IP or hostname identity*".

Cloud-native and application development

A more traditional security model, focused on perimeter-based security, can't singularly protect a cloud-native architecture. Consider this example, a monolithic application using a three-tier architecture is deployed to a private corporate data center which has enough capacity to handle peak load for critical events. Applications with specific hardware or network requirements are purposefully deployed onto specific machines which typically maintain fixed IP addresses. Rollouts are infrequent, large, and hard to coordinate as the resulting changes simultaneously affect many parts of the application. This leads to very long-lived applications that are updated less frequently and where security patches are typically less frequently applied.

However, in a cloud-native model, containers decouple the binaries needed by your application from the underlying host operating system, and make applications more portable. Containers are meant to be used immutably, meaning they don't change once they're deployed—so they are rebuilt and redeployed frequently. Jobs are scaled to handle load, with new jobs deployed when the load increases, and existing jobs killed when the load diminishes. With containers being restarted, killed, or rescheduled often, there is more frequent reuse and sharing of hardware and networking. With a common standardized build and distribution process, the development process is more consistent and uniform between teams, even though teams independently manage the development of their microservices. As a result, security considerations (e.g. security reviews, code scanning, and vulnerability management), can take place earlier in the development cycle.

Implications for security

We've talked a lot about how the model of an untrusted interior, with users in BeyondCorp, can also apply to microservices in BeyondProd—but what does that change look like? Table 1 provides a comparison between aspects of traditional infrastructure security and their counterpoints in a cloud native architecture. The table also shows the requirements needed to move from one to the other. The remainder of this section provides more details on each row of the table.

Traditional infrastructure security	Cloud native security	Implied requirements for cloud-native security
Perimeter-based security (i.e. firewall), with internal communications considered trusted.	Zero-trust security with service-to-service communication verified, and no implicit trust for services in the environment.	Protection of the network at the edge (remains applicable) and no inherent mutual trust between services.
Fixed IPs and hardware for certain	Greater resource utilization, reuse, and	Trusted machines running code

applications.	sharing, including of IPs and hardware.	with known provenance.
IP address-based identity.	Service based identity.	
Services run in a known, expected location.	Services can run anywhere in the environment, including hybrid deployments across the public cloud and private data centers.	
Security-specific requirements built into each application and enforced separately.	Shared security requirements integrated into service stacks following a centralized enforcement policy.	Choke points for consistent policy enforcement across services.
Limited restrictions on how services are built and reviewed.	Security requirements applied consistently to all services.	
Limited oversight of security components.	Centralized view of security policies and adherence to policies.	
Specialized and infrequent rollouts.	Standardized build and rollout process with more frequent changes to individual microservices.	Simple, automated and standardized change rollout.
Workloads are typically deployed as VMs or to physical hosts and use physical machine or hypervisor to provide isolation.	Bin-packed workloads and their processes run in a shared operating system, requiring a mechanism to isolate workloads.	Isolation between workloads sharing an operating system.

Table 1: Implied requirements for security in moving to a cloud-native architecture

From perimeter-based security to zero-trust security

In a traditional security model, an organization's applications could depend on an external firewall around its private data center to protect against incoming traffic. In a cloud-native environment, although the network perimeter still needs to be protected just as in the BeyondCorp model, a perimeter-based security model is no longer enough. This doesn't introduce a new security problem, but rather recognizes the reality that if a firewall can't fully protect the corporate network, it can't fully protect the production network. With a zero-trust security model, you can no longer implicitly trust internal traffic—it requires other security controls, like authentication and encryption. At the same time, the shift towards microservices provides an opportunity to re-think the traditional security model. As you remove your

dependence on a single network perimeter (for example, a firewall), you can further segment the network by service. To take this idea one step further, you could implement microservice-level segmentation, with no inherent trust between services. With microservices, traffic can have varying levels of trust with different controls—you are no longer comparing just internal versus external traffic.

From fixed IPs and hardware to greater shared resources

In a traditional security model, an organization's applications were deployed to specific machines, and the IP addresses of those machines changed infrequently. This meant that security tools could rely on a relatively static architecture map that linked applications in a predictable way—security policies in tools like firewalls could use IP addresses as identifiers.

However, in the cloud-native world, with shared hosts and frequently changing jobs, using a firewall to control access between microservices doesn't work. You can't rely on the fact that a specific IP address is tied to a particular service. As a result, instead of basing identity on an IP address or hostname, you base it on a service.

From application-specific security implementations to shared security requirements integrated into service stacks

In a traditional security model, individual applications were each responsible for meeting their own security requirements independently of other services. Such requirements included identity management, SSL/TLS termination, and data access management. This often led to inconsistent implementations or unaddressed security issues as these issues had to be fixed in many places, making fixes harder to apply.

In the cloud-native world, components are much more frequently re-used between services and there are choke points that allow for policies to be consistently enforced across services. Different policies can be enforced using different security services. Rather than requiring every application to implement critical security services separately, you can split out the various policies into separate microservices (for example, one policy to ensure authorized access to user data, and another to ensure the use of up-to-date TLS cipher suites).

From specialized and infrequent rollout processes to standardized processes with more frequent rollouts

In a traditional security model, there were limited shared services. Code that is more distributed, coupled with local development, meant that it was difficult to ascertain the impact of a change

which affected many parts of an application—as a result, rollouts were infrequent and difficult to coordinate. To make a change, developers might have to update each component directly (for example, SSHing into a virtual machine to update a configuration). Overall, this led to extremely long-lived applications. From a security perspective, as code was more distributed, it was more difficult to review, and even more challenging to ensure that when a vulnerability was fixed, it was fixed everywhere. Moving to cloud-native where rollouts are frequent and standardized enables security to shift left"[2] (#footnote-2) in the software development lifecycle. This enables simpler and more consistent enforcement of security hygiene, including regular application of security patches.

From workloads isolated using physical machines or hypervisors to bin-packed workloads running on the same machine requiring stronger isolation

In a traditional security model, workloads were scheduled on their own instances, with no shared resources. An application was effectively delimited by its machine and network boundary, and workload isolation was enforced solely by relying on physical host separation, hypervisors, and traditional firewalls.

In a cloud-native world, workloads are containerized and bin-packed onto shared hosts, and shared resources. As a result, you need to have stronger isolation between your workloads. Workloads can be separated into microservices which are isolated from one another in part using network controls and sandboxing technologies.

Security principles

In developing a cloud-native architecture, we wanted to concurrently strengthen our security—and so we developed and optimized for the following security principles:

- **Protection of network at the edge**, so that workloads are isolated from network attacks and unauthorized traffic from the Internet. Although a wall-based approach is not a concept new to cloud-native, it remains a security best practice. In a cloud-native world, a perimeter approach is used to protect as much infrastructure as possible against unauthorized traffic and potential attacks from the Internet, for example, volume-based Denial of Service attacks.
- **No inherent mutual trust between services**, so that only known, trusted, and specifically authorized callers can utilize a service. This stops attackers from using untrusted code to access a service. If a service does get compromised, it prevents the attacker from

performing actions that allow them to expand their reach. This mutual distrust helps to limit the blast radius of a compromise.

- **Trusted machines running code with known provenance,** so that service identities are constrained to use only authorized code and configurations, and run only in authorized, verified environments.
- **Choke points for consistent policy enforcement across services.** For example, a choke point to verify requests for access to user data, such that a service's access is derived from a validated request from an authorized end user, and an administrator's access requires business justification.
- **Simple, automated, and standardized change rollout**, so that infrastructure changes can be easily reviewed for their impact on security, and security patches can be rolled out with little impact on production.
- **Isolation between workloads sharing an operating system**, so that if a service is compromised, it can't affect the security of another workload running on the same host. This limits the "blast radius" of a potential compromise.

Across our entire infrastructure, our goal is to have automated security that doesn't depend on individuals. Security should scale in the same way that services scale. Services should be secure by default and insecure by exception— human actions should be by exception, not routine, and auditable when they occur. We can then authenticate a service based on the code and configuration deployed for the service, instead of the people who deployed the service.

Taken together, the implementation of these security principles mean that containers and the microservices running inside can be deployed, communicate with each other, and run next to each other without weakening the properties of a cloud-native architecture (i.e. simple workload management, no-ops scaling, and effective bin-packing). All of this can be achieved without burdening individual microservice developers with the security and implementation details of the underlying infrastructure.

Google's internal security services

To protect Google's cloud-native infrastructure, we designed and developed several internal tools and services. The security services listed below work together to address the security principles defined in the Security Principles (#security-principles) section:

- **Google Front End (GFE)** (https://cloud.google.com/security/encryption-in-transit/#user_to_google_front_end_encryption):

Terminates the connection from the end user, and provides a central point for enforcing TLS best practices. Even though our emphasis is no longer on perimeter-based security, the GFE is still an important part of our strategy for protecting internal services against denial of service attacks. GFE is the first point of entry for a user connection to Google; once within our infrastructure, the GFE is also responsible for load balancing and rerouting traffic between regions as needed. In our infrastructure, GFE is the edge proxy that routes traffic to the right microservice.

- **Application Layer Transport Security (ALTS)** (https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/): Used for RPC authentication, integrity, and encryption. ALTS is a mutual authentication and transport encryption system for services in Google's infrastructure. Identities are in general bound to services instead of to a specific server name or host. This facilitates seamless microservice replication, load balancing, and rescheduling across hosts.
- **Binary Authorization for Borg** and **Host Integrity** are used for microservice and machine integrity verification, respectively:
  - **Binary Authorization for Borg (BAB)** (https://cloud.google.com/security/binary-authorization-for-borg): A deploy-time enforcement check that ensures that code meets internal security requirements before it is deployed. BAB checks include changes getting reviewed by a second engineer, code being submitted to our source code repository, and binaries being verifiably built on dedicated infrastructure. In our infrastructure, BAB restricts the deployment of unauthorized microservices.
  - **Host Integrity (HINT):** Verifies the integrity of the host system software through a secure boot process and is backed by secure microcontroller hardware where supported. HINT checks include the verification of digital signatures on the BIOS, BMC, bootloader and OS kernel.
- **Service Access Policy** and **End user context tickets** are used to restrict access to data:
  - **Service Access Policy** (https://cloud.google.com/security/infrastructure/design/#inter-service_access_management) : Limits how data is accessed between services. When an RPC is sent from one service to another, the Service Access Policy defines the authentication, authorization, and auditing policies required to access the receiving service's data. This limits how data is accessed, grants the minimal level of access needed, and specifies how that access can be audited. In Google's infrastructure, the Service

Access Policy limits one microservice's access to another microservice's data, and allows for global analyses of access controls.

- **End user context (EUC) tickets** (https://cloud.google.com/security/infrastructure/design/#access_management_of_end_user_data)
  : These tickets are issued by an End User Authentication service, and provide services with a user identity, separate from their service identity. These are integrity-protected, centrally-issued, forwardable credentials that attest to the identity of an end user who made a request of the service. This reduces the need for trust between services, as peer identity via ALTS is often insufficient to grant access, with such authorization decisions typically also based on the end user's identity.

- **Borg tooling for blue/green deployments** [3] (#footnote-3): This tooling is responsible for migrating running workloads when performing maintenance tasks. A new Borg job is deployed in addition to the existing Borg job, and a load balancer gradually moves traffic from one to the other. This allows a microservice to be updated with no downtime and without the user noticing. This tooling is used to apply service upgrades when we add new features, as well as to apply critical security updates with no downtime (for example, Heartbleed and Spectre/Meltdown (https://cloud.google.com/blog/topics/inside-google-cloud/answering-your-questions-about-meltdown-and-spectre)
). For changes affecting Google Cloud infrastructure, we use live migration (https://cloud.google.com/compute/docs/instances/live-migration) to ensure VM workloads are not impacted.

- **gVisor** (https://gvisor.dev/), for workload isolation. gVisor uses a user space kernel to intercept and handle syscalls, reducing the interaction with the host and the potential attack surface. This kernel provides most of the functionality required to run an application, and limits the host kernel surface that is accessible to the application. In Google's infrastructure, gVisor is one of several important tools used to isolate both internal and Google Cloud customer workloads running on the same host from each other.

Table 2 maps each principle we described in the Security Principles section to a corresponding tool we use at Google to implement that principle.

Security principle	Google's internal security tool/service
Protection of network at the edge	**Google Front End (GFE)**, for managing TLS termination and policies for incoming traffic

No inherent mutual trust between services	**Application Layer Transport Security (ALTS)**, for RPC authentication, integrity, encryption, and service identities
Trusted machines running code with known provenance	**Binary Authorization for Borg (BAB)**, for code provenance verification **Host Integrity (HINT)**, for machine integrity verification
Choke points for consistent policy enforcement across services	**Service Access Policy**, for limiting how data is accessed between services **End user context (EUC) tickets,** for attesting the identity of the original requester
Simple, automated, and standardized change rollout	**Borg tooling**, for blue/green deployments
Isolation between workloads sharing an operating system	**gVisor**, for workload isolation

Table 2: Principles and security tools for implementing cloud-native security at Google

Putting it all together

In this section we describe how the components we have discussed so far fit together to serve user requests in a cloud-native world. We use two examples: first, we trace a typical user data request from its creation to delivery at its destination and, second, we trace a code change from development to production. Not all of the technologies listed here are used in all parts of Google's infrastructure—it depends on the services and workloads.

Accessing user data

As shown in Figure 1, when the GFE receives a user's request (step 1), it terminates the TLS connection and forwards the request to the appropriate service's frontend over ALTS[4] (#footnote-4) (step 2). The application frontend authenticates the user's request using a central End User Authentication (EUA) Service and, if successful, receives a short lived cryptographic end user context ticket (EUC) (step 3).

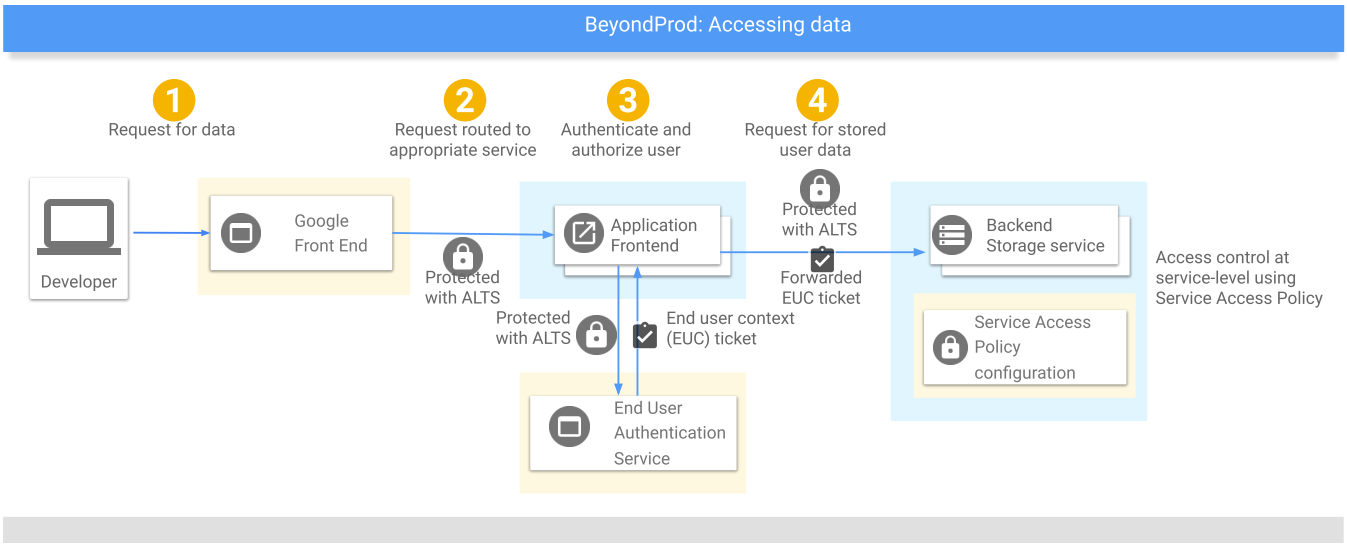

Figure 1: Google's cloud-native architecture security controls accessing user data

The application frontend then makes an RPC over ALTS to a storage backend service, forwarding the EUC ticket in the backend request (step 4). The backend service uses Service Access Policy to ensure that:

1. the frontend service's ALTS identity is authorized to make requests to the backend service and present an EUC ticket,
2. the frontend's identity is protected by our Binary Authorization for Borg (BAB), and
3. the EUC ticket is valid.

The backend service then checks that the user in the EUC ticket is authorized to access the requested data. If any of these checks fail, the request is denied. In many cases, there is a chain of backend calls and every intermediary service does a Service Access Policy check on inbound RPCs, and the EUC ticket is forwarded on outbound RPCs. If these checks pass, then the data is returned to the authorized application frontend, and served to the authorized user.

Each machine has an ALTS credential that is provisioned via the HINT system, and can only be decrypted if HINT has verified that the machine boot was successful. Most Google services run as microservices on top of Borg, and these microservices each have their own ALTS identity. Borgmaster [5](#footnote-5) grants these ALTS microservice credentials to workloads based on the microservice's identity, as described in Figure 1. The machine-level ALTS credentials form the secure channel for provisioning microservice credentials, so that only machines that have successfully passed HINT verified boot can actually host microservice workloads.

Making a code change

As shown in Figure 2, when a Googler makes a change to a microservice protected by a suitably strong BAB, the change must be submitted to our central code repository which enforces a code review. Once approved, the change is submitted to the central, trusted build system which produces a package with a signed verifiable build manifest certificate (step 1). At deployment time, BAB verifies this process was followed by validating the signed certificate from the build pipeline (step 2).

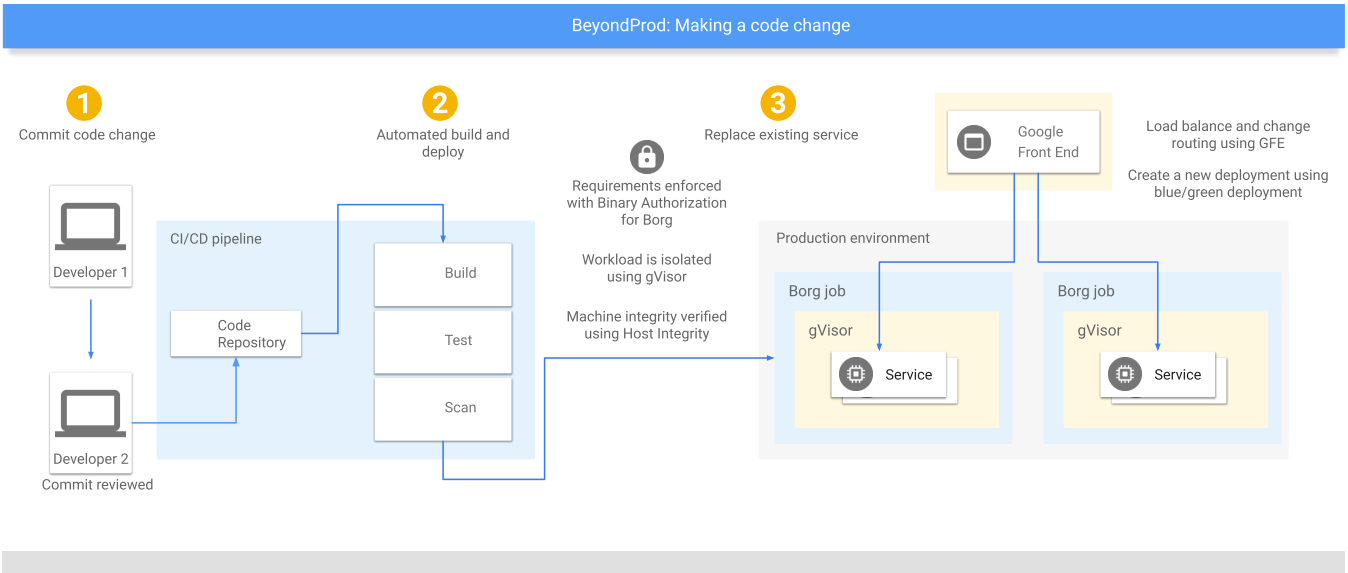

Figure 2: Google's cloud-native architecture security controls making a code change

All workload updates are handled through blue/green deployments, whether it's a routine rollout or emergency security patch (step 3). GFE load balances traffic over to the new deployment to ensure continuity of operations.

All workloads require isolation. If the workload is less trusted, for example, the workload is multi-tenant or the source code originates from outside of Google, it may be deployed into a gVisor-protected environment, or use other layers of isolation. This isolation ensures that if one instance of the application is compromised, none of the other instances are affected.

Applying BeyondProd

Going all in

By going cloud-native, and appropriately securing that infrastructure, Google can offer very strong security properties for its internal and external (Google Cloud) workloads.

By building shared components, the burden on individual developers to meet common security requirements is minimal. Ideally, security functionality should require little to no integration into each individual application, and is instead provided as a fabric enveloping and connecting all microservices. This is commonly called a service mesh. This also means that security can be managed and implemented separately from regular development or deployment activities.

Making the change to cloud-native

Google's transition to cloud-native required changes in two main areas: in our infrastructure and in our development process. We tackled these changes simultaneously, but they could be decoupled and addressed independently.

Changing our infrastructure

We started by building a strong foundation of service identity, authentication, and authorization. Having a foundation of trusted service identities in place enabled us to implement higher-level security capabilities dependent on validating these service identities, such as Service Access Policies and EUC tickets. To make this transition simple for both new and existing services, ALTS was first provided as a library with a single helper daemon. This daemon ran on the host called by every service, and evolved over time into a library using service credentials. The ALTS library was integrated seamlessly into the core RPC library—this made it easier to gain wide adoption, without significant burden on individual development teams. ALTS rollout was a prerequisite to rolling out Service Access Policies and EUC tickets.

Changing our development processes

It was crucial for Google to establish a robust build and code review process. This allowed us to ensure both the integrity of services that are running, and that the identities used by ALTS are meaningful. We established a central build process where we were able to begin enforcing requirements such as a two-person code review and automated testing at build and deployment time. (See the Binary Authorization for Borg (https://cloud.google.com/security/binary-authorization-for-borg) whitepaper for more details on deployment.)

Once we had the basics in place, we started to address the need to run external, untrusted code in our environments. To achieve this goal, we started sandboxing—first with ptrace, then later using gVisor. Similarly, blue/green deployments provided significant benefit in terms of security (e.g., patching) as well as reliability.

We quickly discovered that it was easier if a service started out by logging policy violations rather than blocking violations. The benefit of this approach was two-fold. First, it gave the service owners a chance to test the change and gauge the impact (if any) that moving to a cloud-native environment would have on their service. Second, it enabled us to fix any bugs as well as identify any additional functionality that we might need to provide to service teams. For example, when a service is onboarded to BAB, the service owners enable audit-only mode. This helps them identify code or workflows that don't meet their requirements. Once they address the issues flagged by audit-only mode, the service owners switch to enforcement mode. In gVisor, we did this by first sandboxing workloads, even with compatibility gaps in the sandboxing capabilities, and then addressing these gaps systematically to improve the sandbox.

Benefits of making the change

In the same way that BeyondCorp helped us to evolve beyond a perimeter based security model, BeyondProd represents a similar leap forward in our approach to production security. The BeyondProd approach describes a cloud-native security architecture that assumes no trust between services, provides isolation between workloads, verifies that only centrally built applications are deployed, automates vulnerability management, and enforces strong access controls to critical data. The BeyondProd architecture led Google to innovate several new systems in order to meet these requirements.

All too often, security is ‘called in’ last?when the decision to migrate to a new architecture has already been made. By involving your security team early and focusing on the benefits of the new security model like simpler patch management and tighter access controls, a cloud-native architecture can provide significant benefits to both application development and security teams. When applying the security principles outlined in this paper to your cloud-native infrastructure, you can strengthen the deployment of your workloads, how your workloads’ communications are secured, and how they affect other workloads.

Notes

1 (#anchor-1) Borg (https://research.google.com/pubs/pub43438.html?hl=es) is Google’s cluster management system for scheduling and running workloads at scale. Borg was Google’s first unified container management system (https://queue.acm.org/detail.cfm?id=2898444), and the inspiration for Kubernetes (http://kubernetes.io).

2 (#anchor-2) "Shifting left" refers to moving steps earlier in the software development lifecycle, which may include steps like code, build, test, validate, and deploy. Lifecycle diagrams are frequently drawn from left to right, so left means at an earlier step.

3 (#anchor-3) A blue/green deployment is a way to roll out a change to a workload without affecting incoming traffic, so that end users don't experience any downtime in accessing the application.

4 (#anchor-4) To better understand how traffic is routed inside Google's infrastructure from the GFE to a service, see the How traffic gets routed (https://cloud.google.com/security/encryption-in-transit/#how_traffic_gets_routed) section of our Encryption in Transit whitepaper.

5 (#anchor-5) Borgmaster is Borg (https://ai.google/research/pubs/pub43438)'s centralized controller. It manages scheduling of jobs, and communicates with running jobs on their status.

Last updated December 17, 2019.