Binary Authorization for Borg: how Google verifies code provenance and implements code identity

Google has written several whitepapers explaining projects our security team have developed internally to help improve security, including BeyondCorp (https://cloud.google.com/beyondcorp) *and BeyondProd* (https://cloud.google.com/security/beyondprod). *For an overview of Google's security, see our Security Infrastructure Design* (https://cloud.google.com/security/infrastructure/design/) *whitepaper.*

In this whitepaper we describe Google's code review process, its provenance[1] (#footnote-1)*, and the need for enforcement mechanisms. We focus on the development of a specific enforcement check - Binary Authorization for Borg (BAB). The goal of BAB is to reduce insider risk by ensuring that production software deployed at Google is properly reviewed and authorized, particularly if that code has the ability to access user data. For all Google products, we value the protection of user data and we strive to be as transparent as possible about the security measures we take.*

The content of this whitepaper is correct as of December 2019. This document 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 customers.

CIO-level summary

- Insider risk represents a threat to the security of user data. At Google, we want to do as much as we can to minimize the potential for Google personnel to use their organizational knowledge or access to user data in an unauthorized way⍰this includes running an unauthorized job.
- Binary Authorization for Borg, or BAB, is an internal deploy-time enforcement check that minimizes insider risk by ensuring that production software and configuration deployed at Google is properly reviewed and authorized, particularly if that code has the ability to access user data.
- BAB ensures that code and configuration deployments meet certain minimum standards.

- In addition to enforcement, BAB can also be used in a non-enforcing auditing mode to warn when certain requirements are not met.
- Adopting BAB helps Google reduce insider risk, prevent possible attacks, and support the uniformity of Google's production systems.

Minimizing insider risk at Google

As a company where security is a top priority, we have put in place measures to limit our **insider risk**⍰the potential for Google personnel (or any other person in the organization) to use their organizational knowledge or access to perform malicious acts. Insider risk also covers the scenario where an attacker has compromised the credentials of someone at Google to facilitate their attack. We have devoted significant resources to innovation in the area of insider risk protection. At Google, safeguarding user data is paramount and we strive to protect it comprehensively. Our goal is to prevent a Google employee from accessing user data without authorization.

Authorization for access to user data

At Google, there are times when our services and personnel have to access user data. We interact with user data in the following ways:

1. **As an end user**: An employee using a service authenticates to the service directly and the service returns their own data. For example, an employee logging into their Gmail account will see their own emails.
2. **As part of their role**: The majority of the work done by Google personnel can be successfully completed using anonymized user data. However, on rare occasions, our personnel require access to user data (https://cloud.google.com/logging/docs/audit/reading-access-transparency-logs#justification_reason_codes) as part of their job (for example, support or debugging). We aim to provide as much transparency about user data accesses as possible. One of the ways we do this is with Access Transparency (https://cloud.google.com/access-transparency/)⍰a feature that enables Google Cloud customers to see eligible accesses of their data through real-time logs.
3. **As part of a service, programmatically**: In order to accomplish a task, a service may need to access user data programmatically on a large scale. For example, a data pipeline will query thousands of users' data at once in order to generate aggregated usage statistics.

When this type of need arises, access is granted for a data set rather than for an individual user's data. Access to each individual user's data is not logged.

This whitepaper focuses on the threat model presented in the third scenario. We want to have confidence that the administrators who run the systems that access user data cannot abuse their powers.

Threat model

The controls we discuss in this paper were built to protect user data by preventing unilateral access. We want to stop Google personnel, acting alone, from directly or indirectly accessing or otherwise affecting user data without proper authorization and justification. Our controls prevent these actions regardless of whether the actor has malicious intent, their account has been compromised, or they have unintentionally been granted authorization.

Google's containerized infrastructure

Our infrastructure is containerized, using a cluster management system called Borg (https://research.google.com/pubs/pub43438.html?hl=es). We run hundreds of thousands of jobs from many different applications, across multiple clusters, each with up to tens of thousands of machines. Despite this scale, our production environment is fairly homogeneous. As a result, the touchpoints for access to user data can be more easily controlled and audited.

Furthermore, by using containers, we have gained notable security benefits. Containers are meant to be used immutably, with frequent re-deployments from a complete image rebuild. This property enables us to review a code change in context, and provides a single choke point for all changes that get deployed into our infrastructure.

To understand how we developed a solution that limits programmatic access to user data by a service, it's important to first understand how a service goes into production at Google.

Step 1: Code review

Our source code is stored in a monolithic central repository (https://ai.google/research/pubs/pub45424) that enables thousands of employees to check code into a single location. Using a single code base simplifies source code management, in particular our dependencies on third party code. A monolithic code base also allows for the enforcement of a single choke point for code reviews. At Google, our code reviews include inspection and approval from at least one engineer other than the author. Our code review process enforces a rule that, at a minimum, code modifications to any system must be approved by the owners of that system. Once the code is checked in, it is built.

When importing changes from third party or open source code, we verify that the change is appropriate (for example, the latest version). However, we often don't have the same review controls in place for every change made by external developers to the third party or open source code we use.

Step 2: Verifiable builds

We use a build system very similar to Bazel (https://bazel.build/), which builds and compiles source code, creating a binary for deployment. Our build system runs in an isolated and locked-down environment that is separated from the employees performing the builds. For each build, the system produces a **verifiable build manifest**⍰a signed certificate fully describing the sources that went into the build, the cryptographic hashes of any binaries or other build artifacts, and the full build parameters. This manifest enables us to trace a binary to the source code that was used in its creation, and by extension, to the process around the creation and submission of the source code it describes. In addition, the manifest enables us to verify that the binary wasn't modified as any changes to the file would automatically invalidate its signature.

Since build actions can in theory be arbitrary code, our build system has been hardened for multi-tenancy. In other words, our build system is designed to prevent one build from influencing any other builds. The system prevents builds from making changes that could compromise the integrity of the verifiable build manifests or of the system itself. Once the build is complete, the change is deployed using Borg.

Step 3: Deployment jobs

Once built, a binary can be deployed in our containerized infrastructure as a Borg job. These jobs use packages that may contain binaries or data, whose installation is managed by Borg. A Borg configuration specifies the requirements for the job to be deployed: the packages, the runtime parameters, arguments, and flags. Borg schedules the job taking into account its constraints, priority, quota, and any other requirements listed in the configuration. Once deployed, the Borg job can interact with other jobs in production.

Step 4: Job identity

A Borg job runs as an identity; it is this identity which is used to access data stores or remote procedure call (RPC) methods of other services. An identity can be either a **role account** (like a service) or a **user account** (like an employee's individual account). Multiple jobs may run as the same identity. We restrict the ability to deploy or modify jobs with a particular identity to those responsible for running the service⍰generally our Site Reliability Engineers (SREs) (https://landing.google.com/sre/).

When Borg starts a job, it provisions the job with cryptographic credentials. The job uses these credentials to prove its identity when making requests of other services (using Application Layer Transport Security (https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/)). For a service to access certain data or another service, its identity must have the necessary permissions. Consider the example of a service such as Google Cloud's Cloud Data Loss Prevention (Cloud DLP) API. In order for the DLP API to access data for scanning, it needs two things. First, the DLP API needs to be configured to run with a distinct identity, in this case a role account. Second, the permissions on the data the DLP API is scanning need to include that role account. Without a valid identity and the correct permissions, the service would be unable to perform the scan.

Our policies require that any role account with access to user data (or any other sensitive information) be tightly controlled by BAB and other security systems. Quality assurance and development jobs which don't have access to sensitive data or resources are permitted to run with fewer controls.

Putting it all together: Life of a job

In summary, the steps to running a job on our infrastructure are as follows:

1. A Google developer authors a change to code. They then send it to one or more other Google engineers for review. The review includes checks for proper authorization and logging. Once approved, the code is checked in.
2. Triggered by the developer, the build system verifiably builds and packages the binaries in a sandboxed environment. This build operation produces a package which the build system then signs for verification purposes.
3. The job is deployed on Borg by an engineer who is specifically authorized to manage jobs that run under the appropriate secure identity.

4. When a Borg job tries to access privileged resources like user data, the job’s identity is verified for authorization

Now that you know how jobs are run in our production environment, let's examine the threat model that BAB addresses⍰preventing a potential malicious insider from running an unauthorized job. To achieve this, BAB verifies that all necessary security checks have taken place before a binary is deployed.

This section provided an overview of our containerized infrastructure. A basic grasp of our production environment is a necessary prerequisite for you to understand the controls we have in place for programmatic user access to data. These controls are described in more detail in the next section.

Binary Authorization for Borg (BAB)

For several years, we have had significant efforts underway to protect user data from being accessed manually. These efforts include limiting data and job management access to the set of Google personnel who needed it to do their jobs.

BAB’s mission is to ensure that all production software and configuration deployed at Google is properly reviewed and authorized, particularly if that code has the ability to access user data. To achieve its mission, BAB provides a deploy-time enforcement service to prevent unauthorized jobs from starting, as well as an audit trail of the code and configuration used in BAB-enabled jobs.

Verification

BAB verifies that binaries meet certain requirements when they are deployed. For example, a service owner could require that the binary for their service must be built from code that's reviewed, checked in, tested, and authorized. We refer to these kinds of checks as **deploy-time checks**. BAB can also perform checks once a binary has been deployed⍰we call these **post-deploy checks**. For more information on these post deployment verifications, see our Enforcement modes (#enforcement-modes) section.

A code change creates a new binary. In order for the changes in the new binary to take effect, it must be deployed. BAB allows many kinds of deploy-time checks. Some examples of these checks include:

- **Is the binary built from checked in code**?

  Is the code submitted and checked into Google's source code repository? For code to be submitted, it must generally have been reviewed by a second Google engineer.
- **Is the binary built verifiably**?

  Can this binary be traced back to auditable sources and was it built by an approved build system?
- **Is the binary built from tested code**?

  Does the binary meet test requirements?
- **Is the binary built from code intended to be used in the deployment**?

  Has the code been submitted into the appropriate area of our source code repository that corresponds to the relevant project or team for that particular Borg job?

Though these are specific to our production environment, similar requirements could be enforced in your CI/CD (Continuous Integration/Continuous Deployment) environments based on your own security, compliance, or reliability requirements.

Service-specific policy

Jobs that access sensitive data generally require code to be submitted, with some exceptions for valid business reasons. Sensitive data includes user data, employment and financial data, and any other need-to-know proprietary or business data. All jobs at Google are required to have a policy⍰even a Borg job needing no access to user data will have a policy defined, but no specific requirements listed.

When service owners onboard to BAB, they define a policy with the security requirements for their service. All role accounts used to implement their service must specify a BAB Policy. For each role account, the BAB Policy defines the intended jobs to be launched and the job's code and configuration requirements. Defining or modifying a policy is itself a code change that must be reviewed.

Requirements that can be enforced in a BAB Policy include:

- **Code is built verifiably:** Code that is built verifiably is auditable, however it doesn't necessarily mean that the code has undergone a code review⍰there are even cases where code is unsubmitted (#emergency-response-procedures). Code used in verifiable builds is auditable for at least 18 months, even if it is not submitted.

- **Code is submitted**: The code is built from a specified, intended, location in our source repository. This generally implies that the code has undergone a code review.
- **Configurations are submitted:** Any configurations provided during deployment go through the same review and submission process as regular code. Consequently, command line flag values, arguments, and parameters can't be modified without review.

The systems and components that enforce BAB need to be tightly controlled. This is achieved by implementing the strictest possible requirements, plus additional manual controls.

Enforcement modes

BAB takes different actions based on the policy specified by the Borg job. We refer to these actions as enforcement modes, of which there are three: deploy-time enforcement, deploy-time audit, and continuous verification. When defining a policy, the service owner must choose either deploy-time enforcement or deploy-time auditing. Continuous verification mode is enabled by default. The next sections provide more details on each mode.

Deploy-time enforcement mode

When a new job is submitted, Borgmaster[2] (#footnote-2) consults BAB to verify that the job meets the necessary requirements as laid out in the BAB Policy. This check acts as an admission controller⍰if requirements are met, then Borgmaster will launch the job. If not, the Borgmaster will reject the request even if the user making the request is otherwise authorized.

In enforcement mode, BAB will block a Borg job from being deployed if it doesn't meet the requirements set out in the BAB Policy for the Borg job. Services that are new to BAB typically start out in audit mode (#deploy-time-audit-mode) (described in the next section), then graduate to enforcement mode.

Emergency response procedures

In the case of an incident or outage, our first priority is to restore the affected service as quickly as possible. In an emergency situation, it may be necessary to run code that hasn't been reviewed or checked in. As a result, enforcement mode can be overridden using an **emergency response** flag. Emergency response procedures also act as a backup in case there is a failure of BAB that would otherwise block a deployment. A developer deploying a job using the emergency response procedure must submit a justification as part of their request.

Within seconds of the emergency response procedure getting used, BAB logs details about the associated Borg job. The log includes the code that was used and the user-provided justification. A few seconds later, an audit trail is sent to our centralized security team. Within hours, the audit trail is sent to the team which owns the role account. Emergency response procedures are only meant to be used as a last resort.

Deploy-time audit mode

In audit mode, BAB logs when a Borg job doesn't meet the policy requirements but won't block its deployment. This policy breach triggers an alert to the team which owns the role account.

At Google, we require certain services, such as those accessing user data, to use enforcement mode. We strongly encourage all service owners to use enforcement mode, and only use audit mode when onboarding a new service. To use audit mode, service owners must provide a justification to get an exception. For example, a service whose reliability SLO (Service Level Objective) is significantly higher than what BAB provides, would use audit mode.

Although audit mode is useful when fine-tuning an initial policy, it's not a practical steady state for most services. When using audit mode, the service owner isn't notified of any policy violations until several hours after the violation has occurred. This can lead to a noisy stream of notifications, causing true security issues to be hidden by mistakes or policy misconfigurations that were introduced by the service owner. With enforcement mode, the service owner gets immediate feedback when they attempt to launch a job that doesn't adhere to the policy. As a result, their stream of notifications is much cleaner. Additionally, enforcement mode in BAB catches inadvertent errors, such as accidentally launching a job into a role other than the one it is designated to run in.

Continuous verification

Once a job is deployed, regardless of its enforcement at deployment time, it's continuously verified for its lifetime. A BAB process runs at least once a day to check that any jobs that were started (and may still be running) conform to any updates to their policies. For example, continuous verification mode is constantly checking for jobs that are running with outdated policies or with an identity that was deployed using emergency response procedures. If a job is found that doesn't adhere to the updated policy, BAB will notify the service owners so they can mitigate the risk.

Globally allowed packages

At Google, there are some packages that are widely used by many of our services. Rather than forcing each service to maintain its own version, these packages are centrally allowed⍰we call them **globally managed packages**. When writing their BAB Policy, a service owner can specify a globally allowed package for a given job. There is also a global default mechanism for widely used packages, so that they don't need to be individually listed as part of each service's policy. This allows Google to maintain explicit control over common system components used across the organization, and ensures these are properly reviewed and updated without involving individual teams. Although an individual service owner could allow these packages explicitly as part of their service's BAB Policy, this makes the recommended and supported path easy for owners to use.

Edge cases

Google implements robust security controls for code deployed in production, including code reviews (#step-1-code-review) and verifiable builds (#step-2-verifiable-builds). BAB acts as an additional control and enforcement point to ensure that these security controls are in fact properly implemented.

However, BAB cannot be effectively used in all cases. BAB does not support the following edge cases: code built using non-standard build systems; code deployed in environments other than Borg; and data analysis and machine learning code, which doesn't lend itself well to human code reviews before the final production parameters are chosen. In these cases, a variety of other security mitigations are in place, including other code provenance systems. This code is nonetheless still subject to Google's general security controls.

Implementing Binary Authorization for Borg

To implement BAB, the BAB team developed several new features, including emergency response procedures and audit mode. These tools made it as easy as possible for service owners to try BAB for themselves. If you're considering deploying something like BAB in your organization, you may need to do some additional work to facilitate this transition. This section describes the organizational and change management work that we did as part of our implementation plan.

Benefits

BAB has three benefits which helped build the business case for its development and adoption at Google�namely, reduced insider risk, robust code identity, and simplified compliance.

Reduced insider risk

BAB was primarily developed to prevent any single individual from obtaining unauthorized programmatic access to user data. BAB makes it more difficult for a single engineer, or an attacker who gains an engineer's credentials to access sensitive data inappropriately and without detection.

Robust code identity

As described in the Containerized infrastructure (#containerized-infrastructure) section, Borg jobs run as an identity, typically a role account. This identity is used by other services to verify proper authorization before granting access to any data. However, other services can't enforce the use of that data and so must trust that the job identity is not abusing the data it received. BAB ties a job's identity to specific code, ensuring that only the specified code can be used to exercise the job identity's privileges. This allows for a transition from a job identity—trusting an identity and any of its privileged human users transitively, to a code identity—trusting a specific piece of code that was reviewed to have a specific semantics and which cannot be modified without approval processes.

Simplified compliance

BAB simplified what were previously manual compliance tasks. Certain processes at Google require tighter controls on how they deal with data. For example, our financial reporting systems must comply with the Sarbanes-Oxley Act (SOX). Prior to BAB, we had a system that helped us manually perform verifications to ensure our compliance. Post BAB, many of these checks were automated based on the services' BAB Policies. Automating these checks enabled the compliance team to increase both the scope of services covered and the adoption of appropriate controls on these services.

Onboarding a service

The BAB team had to ensure that the onboarding process was simple and straightforward. Initially, service owners at Google had two main concerns about adopting BAB:

- If BAB wasn't sufficiently reliable, its implementation could block changes in critical situations.
- BAB could slow the development of a service with frequent code check-ins and iterative processes.

To address these initial concerns, the BAB team developed audit mode (#deploy-time-audit-mode). Using this mode, the BAB team was able to prove to some key early adopters that BAB worked. To further enforce its reliability, the BAB team developed an availability SLO and introduced emergency response procedures (#emergency-response-procedures) for enforcement mode (#deploy-time-enforcement-mode).

When onboarding an **existing service** to BAB, the service owner typically enables audit-only mode first. This helps them to identify and address any issues before turning on enforcement mode. The default policy for any job using BAB in production is enforcement mode. To deploy their job, the service owner must submit a policy that, at a minimum, requires code to be submitted and built verifiably. A service owner may deploy their job without meeting this minimum standard, but they must be granted an exception. If the service needs access to more sensitive data and/or services, the owner can move to stricter requirements. Defining an initial policy can be difficult, so the BAB team created automated tooling to help service owners write their policies. The tools look at which parts of the source repository are used for an existing job, and suggest an appropriate policy.

When onboarding a **new service** to BAB, the service owner enables enforcement mode from the beginning. The service owner drafts an initial policy and rapidly iterates to add additional requirements. The policies themselves are managed as code changes and so require a second Google engineer to review any updates.

Impact

Adopting BAB and a containerized deployment model provides many benefits to Google in terms of security and supportability:

- **BAB helps reduce overall insider risk**: By requiring code to meet certain standards and change management practices before accessing user data, BAB reduces the potential for a Googler acting alone (or whose account has been compromised) from accessing user data programmatically.

- **BAB supports uniformity of production systems**: By using containerized systems and verifying their BAB requirements prior to deployment, our systems are easier to debug, more reliable, and have clearer change management. BAB requirements provide a common language for production system requirements.
- **BAB dictates a common language for data protection**: BAB tracks conformance across our systems. Data about this conformance is published internally and is queryable by other teams. Publishing BAB data in this way enables teams to use common terms when communicating with each other about their data protection. This common language reduces the back-and-forth work needed when working with data across teams.
- **BAB allows programmatic tracking of compliance requirements**: Certain processes, such as those for financial reporting, need to meet certain change management requirements for compliance purposes. Using BAB, these checks can be automated, saving time and increasing the scope of coverage.

BAB is one of several technologies used at Google to mitigate insider risk.

Adopting similar controls in your organization

We learned many important lessons when implementing BAB at Google. Making such a large change can seem like a daunting task. In this section we share the lessons we learned along the way in the hope that you can apply the principles of BAB to your organization.

Work towards a more homogeneous containerized CI/CD pipeline.

Adoption of BAB at Google was made possible by the consistency and integration of the tooling we use in our code development. This work included code reviews, verifiable builds, containerized deployments, and service-based identity for access control. Verifiable builds allow you to check how your binaries are built; and containers allow you to separate binaries from data and give you an enforcement choke point to ensure these meet requirements for use. This approach simplified the adoption of BAB and strengthened the guarantees that a solution like BAB can provide.

The introduction of microservices allowed the adoption of service-based identity (like a service account), rather than host-based identity (like an IP address). Making the shift towards a service-based identity will enable you to change how you manage authentication and authorization between services. For example, if you’re using an identity token baked into a container image to attest identity, the guarantees provided by code provenance will not be as strong. If you're not able to directly adopt a service identity, you could try more strongly protecting identity tokens as an interim step.

Determine your goals, and define your policies based on your requirements.

Build your policy-driven release process one piece at a time. You may need to implement certain changes earlier than others in your CI/CD pipeline. For example, you may need to start conducting formal code reviews before you can enforce them at deployment time.

A great motivator for a policy-driven release process is compliance⍰if you can encode at least some of your compliance requirements in a policy, it can help automate your tests and ensure they are always in effect. Start with a base set of requirements and codify more advanced requirements as you go.

Enforce policies early in development.

It's hard to define comprehensive policies on a piece of software without first knowing where it will run and what data it will access. This is why BAB Policy enforcement is done when code is deployed and when it accesses data, not when it's built. A BAB Policy is defined in terms of runtime identity, so the same code may run in different environments and be subject to different policies.

We use BAB in addition to other access mechanisms to limit access to user data. Using BAB in this way, service owners can further ensure that data is only accessed by a job meeting particular BAB requirements, effectively treating the code as identity.

Determine how to onboard existing service owners.

Identify a handful of service owners who will see immediate benefits from enforcement and are willing to provide feedback. One way to do this might be to ask owners to volunteer before making any changes mandatory.

If possible, require enforcement mode over audit mode⍰or be forceful by limiting the grace period for audit mode. Audit mode allows owners to quickly onboard and better understand insider risk. The drawback of audit mode is that it can take time to see a tangible reduction in insider risk. This delay can make it hard to show value and convince others to adopt enforcement. When the BAB team provided emergency response procedures for enforcement, service owners were more willing to adopt enforcement, giving them an escape hatch if they needed it.

Enlist change agents across teams.

When we created a Google-wide mandate for BAB deployment, what most affected our success rate was finding owners to drive the change in each product group. Once we had their help, we set up a formal change management team to track ongoing changes. We then identified accountable owners in each product team to implement the changes.

Figure out how to manage third party code.

Many of the CI/CD controls we describe in this paper are placed where your code is developed, reviewed, and maintained by one organization. If you are in this situation, consider how you will include third party code as part of your policy requirements. For example, you could initially exempt the code, while you move towards an ideal state of keeping a repository of all third party code used, and regularly vet that code against your security requirements.

Conclusion

Implementing a deploy-time enforcement check as part of Google's containerized infrastructure and CI/CD process has enabled us to verify that the code and configuration we deploy meet certain minimum standards for security. This is a critical control used to limit the ability of a potentially malicious insider to run an unauthorized job that could access user data. Adopting BAB has allowed Google to reduce insider risk, prevent possible attacks, and also support the uniformity of our production systems.

Further references

To learn more about Google's overall security and containerized infrastructure, check out these resources:

- Google's infrastructure
  - Borg: Large-scale cluster management at Google with Borg (https://research.google.com/pubs/pub43438.html?hl=es)
  - Monolithic repository: Why Google Stores Billions of Lines of Code in a Single Repository (https://ai.google/research/pubs/pub45424)
  - Bazel (https://bazel.build/)
- Google's security
  - BeyondProd (https://cloud.google.com/security/beyondprod)

- Infrastructure Security Design whitepaper (https://cloud.google.com/security/infrastructure/design/)
- Application Layer Transport Security (https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/)
- Encryption at Rest whitepaper (https://cloud.google.com/security/encryption-at-rest/default-encryption/)

- For Google Cloud users
  - Binary Authorization (https://cloud.google.com/binary-authorization/)
  - Container security (https://cloud.google.com/containers/security)
  - Access Transparency (https://cloud.google.com/access-transparency/)

Notes

1 (#top_of_page) Provenance describes the components, changes made to the components, and their origination. See https://csrc.nist.gov/glossary/term/Provenance (https://csrc.nist.gov/glossary/term/Provenance).

2 (#deploy-time-enforcement-mode) 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.