SECURITY

BeyondCorp Part III
The Access Proxy

LUCA CITTADINI, BATZ SPEAR, BETSY BEYER, AND MAX SALTONSTALL

Luca Cittadini is a Site Reliability Engineer at Google in Dublin. He previously worked as a Network Engineer at the Italian Central Bank in Rome. He holds a PhD in computer science from Roma Tre University. lucacittadini@google.com

Batz Spear is a Software Engineer at Google in Mountain View. He holds a PhD in computer science from UC Davis. batz@google.com

Betsy Beyer is a Technical Writer for Google Site Reliability Engineering in NYC. She has previously provided documentation for Google Data Center and Hardware Operations teams. Before moving to New York, Betsy was a lecturer in technical writing at Stanford University. She holds degrees from Stanford and Tulane. bbeyer@google.com

Max Saltonstall is a Program Manager for Google Corporate Engineering in New York. Since joining Google in 2011 he has worked on video products, internal change management, IT externalization, and coding puzzles. He has a degree in computer science and psychology from Yale. maxsaltonstall@google.com

This article details the implementation of BeyondCorp's front-end infrastructure. It focuses on the Access Proxy, the challenges we encountered in its implementation and the lessons we learned in its design and rollout. We also touch on some of the projects we're currently undertaking to improve the overall user experience for employees accessing internal applications.

In migrating to the BeyondCorp model (previously discussed in "BeyondCorp: A New Approach to Enterprise Security" [1] and "BeyondCorp: Design to Deployment at Google" [2]), Google had to solve a number of problems. Figuring out how to enforce company policy across all our internal-only services was one notable challenge. A conventional approach might integrate each back end with the device Trust Inferer in order to evaluate applicable policies; however, this approach would significantly slow the rate at which we're able to launch and change products.

To address this challenge, Google implemented a centralized policy enforcement front-end Access Proxy (AP) to handle coarse-grained company policies. Our implementation of the AP is generic enough to let us implement logically different gateways using the same AP codebase. At the moment, the Access Proxy implements both the Web Proxy and the SSH gateway components discussed in [2]. As the AP was the only mechanism that allowed employees to access internal HTTP services, we required all internal services to migrate behind the AP.

Unsurprisingly, initial attempts that dealt only with HTTP requests proved inadequate, so we had to provide solutions for additional protocols, many of which required end-to-end encryption (e.g., SSH). These additional protocols necessitated a number of client-side changes to ensure that the device was properly identified to the AP.

The combination of the AP and an Access Control Engine (a shared ACL evaluator) for all entry points provided two main benefits. By supplying a common logging point for all requests, it allowed us to perform forensic analysis more effectively. We were also able to make changes to enforcement policies much more quickly and consistently than before.

BeyondCorp's Front-End Infrastructure

Any modern Web application deployed at scale employs front-end infrastructure, which is typically a combination of load balancers and/or reverse HTTP proxies. Enterprise Web applications are no exception, and the front-end infrastructure provides the ideal place to deploy policy enforcement points. As such, Google's front-end infrastructure occupies a critical position in BeyondCorp's enforcement of access policies.

The main components of Google's front-end infrastructure are a fleet of HTTP/HTTPS reverse proxies called Google Front Ends (GFEs [3]). GFEs provide a number of benefits, such as load balancing and TLS handling "as a service." As a result, Web application back ends can focus on serving requests and largely ignore the details of how requests are routed.

BeyondCorp leverages the GFE as a logically centralized point of access policy enforcement. Funneling requests in this manner led us to naturally extend the GFE to provide other features, including self-service provisioning, authentication, authorization, and centralized logging. The resulting extended GFE is called the ***Access Proxy*** (AP). The following section details the specifics of the services the Access Proxy offers.

Features of the Extended GFE: Product Requirements

The GFE provides some built-in benefits that weren't designed specifically for BeyondCorp: it both provides load balancing for the back ends and addresses TLS handling by delegating TLS management to the GFE. The AP extends the GFE by introducing authentication and authorization policies.

Authentication

In order to properly authorize a request, the AP needs to identify the user and the device making the request. Authenticating the device poses a number of challenges in a multi-platform context, which we address in a later section, "Challenges with Multi-Platform Authentication." This section focuses on user authentication.

The AP verifies user identities by integrating with Google's Identity Provider (IdP). Because it isn't scalable to require back-end services to change their authentication mechanisms in order to use the AP mechanism, the AP needs to support a range of authentication options: OpenID Connect, OAuth, and some custom protocols.

The AP also needs to handle requests made without user credentials, e.g., a software management system attempting to download the latest security updates. In these cases, the AP can disable user authentication.

When the AP authenticates the user, it strips the credential before sending the request to the back end. Doing so is essential for two reasons:

- The back end can't replay the request (or the credential) through the Access Proxy.
- The proxy is transparent to the back ends. As a result, the back ends can implement their own authentication flows on top of the Access Proxy's flow, and won't observe any unexpected cookies or credentials.

Authorization

Two design choices drove our implementation of the authorization mechanism in a BeyondCorp world:

- A centralized Access Control List (ACL) Engine queryable via Remote Procedure Calls (RPCs)
- A domain-specific language to express the ACLs that is both readable and extensible

Providing ACL evaluation as a service enables us to guarantee consistency across multiple front-end gateways (e.g., the RADIUS network access control infrastructure, the AP, and SSH proxies).

Providing centralized authorization has both benefits and drawbacks. On the one hand, an authorizing front end frees back-end developers from dealing with the details of authorization by promoting consistency and a centralized point of policy enforcement. On the other hand, the proxy might not be able to enforce fine-grained policies that are better handled at the back end (e.g., "user A is authorized to modify resource B").

In our experience, combining coarse-grained, centralized authorization at the AP with fine-grained authorization at the back end provides the best of both worlds. This approach doesn't result in much duplication of effort, since the application-specific fine-grained policies tend to be largely orthogonal to the enterprise-wide policies enforced by the front-end infrastructure.

Mutual authentication between the proxy and the back end

Because the back end delegates access control to the front end, it's imperative that the back end can trust that the traffic it receives has been authenticated and authorized by the front end. This is especially important since the AP terminates the TLS handshake, and the back end receives an HTTP request over an encrypted channel.

Meeting this condition requires a mutual authentication scheme capable of establishing encrypted channels—for example, you might implement mutual TLS authentication and a corporate public key infrastructure. Our solution is an internally developed authentication and encryption framework called LOAS (Low Overhead Authentication System) that bi-directionally authenticates and encrypts all communication from the proxy to the back ends.

One benefit of mutual authentication and encryption between the front end and back end is that the back end can trust any additional metadata inserted by the AP (usually in the form of extra HTTP headers). While adding metadata and using a custom protocol between the reverse proxy and the back ends isn't a novel approach (for example, see Apache JServe Protocol [4]), the mutual authentication scheme between the AP ensures that the metadata is not spoofable.

As an added benefit, we can also incrementally deploy new features at the AP, which means that consenting back ends can opt in by simply parsing the corresponding headers. We use this functionality to propagate the device trust level to the back ends, which can then adjust the level of detail served in the response.

The ACL language

Implementing a domain-specific language for ACLs was key in tackling challenges of centralized authorization. The language both allows us to compile ACLs statically (which aids performance and testability) and helps reduce the logical gap between the policy and its implementation. This strategy promotes separation of duties among the following parties:

- **The team that owns the security policy:** Responsible for the abstract and statically compiled specification of access decisions
- **The team that owns the inventory pipeline:** Responsible for the concrete instantiation of a decision about granting access to a resource based on the specific device and user requesting access (see [2] for more details about the inventory pipeline)
- **The team that owns the Access Control Engine:** Responsible for evaluating and executing the security policy

The ACL language works using first-match semantics, which is similar to traditional firewall rules. While this model creates well-studied corner cases (for example, rules which shadow each other), we've found that the security team can reason about this model relatively easily. The structure of the ACLs we currently enforce consists of two macro-sections:

- **Global rules:** Usually coarse-grained and affect all services and resources. For example, "Devices at a low tier are not allowed to submit source code."
- **Service-specific rules:** Specific to each service or hostname; usually involve assertions about the user. For example, "Vendors in group G are allowed access to Web application A."

The above assumes that service owners can identify the sections of their URL space that need policies. Service owners should almost always be able to identify these sections, except for some cases in which the differentiation occurs in the request body (although the AP could be modified to handle this scenario). The portion of the ACL dealing with service-specific rules inevitably grows in size as the Access Proxy accounts for more and more services with a business need for a specialized ACL.

The set of global rules is very handy during security escalations (e.g., employee exit) and incident response (e.g., browser exploits or stolen devices). For example, these rules helped us successfully mitigate a zero-day vulnerability in third-party plugins shipped with our Chrome browser. We created a new high-priority rule that redirected out-of-date Chrome versions to a page with update instructions, which was deployed and enforced across the entire company within 30 minutes. As a result, the observed population of vulnerable browsers dropped very quickly.

Centralized Logging

In order to conduct proper incident response and forensic analysis, it's essential that all requests are logged to persistent storage. The AP provides an ideal logging point. We log a subset of the request headers, the HTTP response code, and metadata relevant to debugging or reconstructing the access decision and the ACL evaluation process. This metadata includes the device identifier and the user identity associated with the request.

Features of the Access Proxy: Operational Scalability

Self-Service Provisioning

Once the Access Proxy infrastructure is in place, developers and owners of enterprise applications have an incentive to configure their services to be accessible via the proxy.

When Google began gradually limiting users' network-level access into corporate resources, most internal application owners looked to the Access Proxy as the fastest solution to keep their service available as the migration proceeded. It was immediately clear that a single team couldn't scale to handle all changes to the AP's configuration, so we structured the AP's configuration to facilitate self-service additions. Users retain ownership of their fragment of the configuration, while the team that owns the AP owns the build system that collates, tests, canaries, and rolls out configurations.

This setup has a few main benefits:

- Frees the AP owners from continuously modifying the configuration per user requests
- Encourages service owners to own their configuration fragment (and write tests for it)
- Ensures a reasonable compromise between development velocity and system stability

The time it takes to set up a service behind the AP has effectively been reduced to minutes, while users are also able to iterate on their configuration fragments without requesting support from the team that owns the AP.

Challenges with Multi-Platform Authentication

Now that we've described the server side of BeyondCorp's front end—its implementation and the resulting challenges and complications—we'll take a similar view into the client side of this model.

At minimum, performing proper device identification requires two components:

- Some form of device identifier
- An inventory database tracking the latest known state of any given device

One goal of BeyondCorp is to replace trust in the network with an appropriate level of trust in the device. Each device must have a consistent, non-clonable identifier, while information about the software, users, and location of the device must be integrated in the inventory database. As discussed in the previous BeyondCorp papers, building and maintaining a device inventory can be quite challenging. The following subsections describe the challenges and solutions related to device identification in more detail.

Desktops and Laptops

Desktops and laptops use an X.509 machine certificate and a corresponding private key stored in the system certificate store. Key storage, a standard feature of modern operating systems, ensures that command-line tools (and daemons) that communicate with servers via the AP can be consistently matched against the correct device identifier. Since TLS requires the client to present a cryptographic proof of private key possession, this implementation makes the identifier non-spoofable and non-clonable, assuming it's stored in secure hardware such as a Trusted Platform Module (TPM).

This implementation has one main drawback: certificate prompts tend to frustrate users. Thankfully, most browsers support automatic certificate submission via policy or extension. Users might also be frustrated if the server rejects the TLS handshake when the client presents an invalid certificate. A failed TLS handshake results in a browser-specific error message that can't be customized. To mitigate this user experience issue, the AP accepts TLS sessions that don't have valid client certificates, and presents an HTML deny page when required.

Mobile Devices

The policies to suppress certificate prompts discussed above don't exist on major mobile platforms. Instead of relying on certificates, we use a strong device identifier natively provided by the mobile operating systems. For iOS devices, we use the identifier ForVendor, while Android devices use the device ID reported by the Enterprise Mobility Management application.

Special Cases and Exceptions

While we've been able to transition the vast majority of Web applications to the Access Proxy over the past few years, some special use cases either don't naturally fit the model or need some sort of special handling.

Non-HTTP Protocols

A number of enterprise applications at Google employ non-HTTP protocols that require end-to-end encryption. In order to serve these protocols through the AP, we wrap them in HTTP requests.

Wrapping SSH traffic in HTTP over TLS is easy thanks to the existing ProxyCommand facility. We developed a local proxy which looks a lot like Corkscrew, except the bytes are wrapped into WebSockets. While both WebSockets and HTTP CONNECT requests allow the AP to apply the ACLs, we opted to use WebSockets over CONNECT because WebSockets natively inherit user and device credentials from the browser.

In the cases of gRPC and TLS traffic, we wrapped the bytes in an HTTP CONNECT request. Wrapping has the obvious downside of imposing a (negligible) performance penalty on the transported protocol. However, it has the important advantage of separating device identification and user identification at different layers of the protocol stack. Inventory-based access control is a relatively new concept, so we frequently find that existing protocols have native support for user authentication (e.g., both LOAS and SSH provide this), but extending them with device credentials is non-trivial.

Because we perform device identification on the TLS layer in the wrapping CONNECT request, we don't need to rewrite applications to make them aware of the device certificate. Consider the SSH use case: the client and server can use SSH certificates to perform user authentication, but SSH doesn't natively support device authentication. Furthermore, it would be impossible to modify the SSH certificate to also convey device identification, because SSH client certificates are portable by design: they are expected to be used on multiple devices. Similar to how we handle HTTP, the CONNECT wrapping ensures we properly separate user and device authentication. While we use the TLS client certificate to authenticate the device, we might use the username and password to authenticate the user.

Remote Desktop

Chrome Remote Desktop, which is publicly available in the Chrome code base [5], is the predominant remote desktop solution at Google. While wrapping protocols in HTTP works in many cases, some protocols, like those powering remote desktop, are especially sensitive to the additional latency imposed by being routed through the AP.

In order to ensure that requests are properly authorized, Chrome Remote Desktop introduces an HTTP-based authorization server into the connection establishment flow. The server acts as an authorizing third party between the Chromoting client and the Chromoting host, while also helping the two entities share a secret, operating similarly to Kerberos.

We implemented the authorization server as a simple back end of the AP with a custom ACL. This solution has proven to work very well: the extra latency of going through the AP is only paid once per remote desktop session, and the Access Proxy can apply the ACLs on each session creation request.

Third-Party Software

Third-party software has frequently proved troublesome, as sometimes it can't present TLS certificates, and sometimes it assumes direct connectivity. In order to support these tools, we developed a solution to automatically establish encrypted point-to-point tunnels (using a TUN device). The software is unaware of the tunnel, and behaves as if it's directly connected to the server. The tunnel setup mechanism is conceptually similar to the solution for remote desktop:

- The client runs a helper to set up the tunnel.
- The server also runs a helper that acts as a back end of the AP.
- The AP enforces access control policies and facilitates the exchange of session information and encryption keys between the client and server helpers.

Lessons Learned

ACLs Are Complicated

We recommend the following best practices to mitigate the difficulties associated with ACLs:

- **Ensure the language is generic.** The AP's ACL has changed numerous times, and we've had to add new feeds (e.g., user and group sources). Expect that you'll need to regularly change the available features, and ensure that the language itself won't hamper these changes.
- **Launch ACLs as early as possible.** The reasons for doing so are twofold:
  - Ensures that users become trained on the ACLs and potential reasons for denial sooner rather than later.
  - Ensures that developers begin to adjust their code to meet the requirements of the AP. For example, we had to implement a cURL replacement to handle user and device authentication.
- **Make modifications self-service.** As previously mentioned, a single team that manages service-specific configuration doesn't scale to support multiple teams.
- **Create a mechanism to pass data from the AP to the back ends.** As mentioned above, the AP can securely pass additional data to the back end to allow it to perform fine-grained access controls. Plan for this required functionality early.

Emergencies Happen

Have well-tested plans in place to handle inevitable emergencies. Be sure to consider two major categories of emergencies:

- **Production emergencies:** Caused by outages or malfunctions of critical components in the request serving path
- **Security emergencies:** Caused by urgent needs to grant/revoke access to specific users and/or resources

Production Emergencies

In order to ensure the AP survives most outages, design and operate it according to SRE best practices [3]. To survive potential data source outages, all of our data is periodically snapshotted and available locally. We also designed AP repair paths that don't depend on the AP.

Security Emergencies

Security emergencies are more subtle than production emergencies, as they're easy to overlook when designing the access infrastructure. Be sure to factor ACL push frequency and TLS issues into user/device/session revocation.

User revocation is relatively straightforward: revoked users are automatically added to a special group as part of the revocation process, and one of the early global rules (see "The ACL language," above) in the ACL guarantees that these users are denied access to any resource. Similarly, session tokens (e.g., OAuth and OpenID Connect tokens) and certificates are sometimes leaked or lost and therefore need to be revoked.

As discussed in the first BeyondCorp article [1], device identifiers are untrusted until the device inventory pipeline reports otherwise. This means that even losing the certificate authority (CA) key (which implies inability to revoke certificates) doesn't imply losing control, because new certificates aren't trusted until they are properly catalogued in the inventory pipeline.

Given this ability, we decided to ignore certificate revocation altogether: instead of publishing a certificate revocation list (CRL), we treat certificates as immutable and simply downgrade the inventory trust tier if we suspect the corresponding private key is lost or leaked. Essentially, the inventory acts as a whitelist of the accepted device identifiers, and there is no live dependency on the CRL. The major downside of this approach is that it might introduce additional delay. However, this delay is relatively easy to solve by engineering fast-track propagation between the inventory and the Access Proxy.

You need a standard, rapid-push process for ACLs in order to ensure timely policy enforcement. Beyond a certain scale, you must delegate at least part of the ACL definition process to service owners, which leads to inevitable mistakes. While unit tests and smoke tests can usually catch obvious mistakes, logic errors will trickle through safeguards and make their way to production. It's important that engineers can quickly roll back ACL changes to restore lost access or to lock down unintended broad access. To cite our earlier zero-day vulnerability plugin example, our ability to push ACLs rapidly was key to our incident response team, as we could quickly create a custom ACL to force users to update.

Engineers Need Support

Transitioning to the BeyondCorp world does not happen overnight and requires coordination and interaction among multiple teams. At large enterprise scale, it's impossible to delegate the entire transition to a single team. The migration will likely involve some backwards-incompatible changes that need sufficient management support.

The success of the transition largely depends on how easy it is for teams to successfully set up their service behind the Access Proxy. Making the lives of developers easier should be a primary goal, so keep the number of surprises to a minimum. Provide sane defaults, create walkthrough guides for the most common use cases, and invest in documentation. Provide sandboxes for the more advanced and complicated changes—for example, you can set up separate instances of the Access Proxy that the load balancer intentionally ignores but that developers can reach (e.g., temporarily overriding their DNS configuration). Sandboxes have proven extremely useful in numerous cases, like when we needed to make sure that clients would be able to handle TLS connections after major changes to the X.509 certificates or to the underlying TLS library.

Looking Forward

While our front-end implementation of BeyondCorp has been largely quite successful, we still have a few pain points. Perhaps most obvious, desktops and laptops use certificates to authenticate, while mobile devices use a different device identifier. Certificate rotations are still painful, as presenting a new certificate requires a browser restart to ensure that existing sockets are closed.

To address both of these issues, we're planning to migrate desktops and laptops to the mobile model, which will remove the need for certificates. To carry out the migration, we plan to build a desktop device manager, which will look quite similar to the mobile device manager. It will provide a common identifier in the form of a Device-User-Session-ID (DUSI) that's shared across all browsers and tools using a common OAuth token-granting daemon. Once the migration is complete, we'll no longer need to authenticate desktops and laptops via a certificate, and all controls can migrate to use the DUSI consistently across all OSes.

Conclusion

Google's implementation of the Access Proxy as a core component of BeyondCorp is specific to our infrastructure and use cases. The design we ultimately implemented is well aligned with common SRE best practices and has proven to be very stable and scalable—the AP has grown by a number of orders of magnitude over the course of its deployment.

Any organization seeking to implement a similar security model can apply the same fundamental principles of creating and deploying a solution similar to the AP. We hope that by sharing our solutions to challenges like multi-platform authentication and special cases and exceptions, and the lessons we learned during this project, our experience can help other organizations to undertake similar solutions with minimal pain.

References

[1] R. Ward and B. Beyer, "BeyondCorp: A New Approach to Enterprise Security," *;login:*, vol. 39, no. 6 (December 2014): https://www.usenix.org/system/files/login/articles/login_dec14_02_ward.pdf.

[2] B. Osborn, J. McWilliams, B. Beyer, and M. Saltonstall, "BeyondCorp: Design to Deployment at Google," *;login:*, vol. 41, no. 1 (Spring 2016): https://www.usenix.org/system/files/login/articles/login_spring16_06_osborn.pdf.

[3] B. Beyer, C. Jones, J. Petoff, and N. Murphy, eds., *Site Reliability Engineering* (O'Reilly Media, 2016).

[4] Apache JServer Protocol: https://tomcat.apache.org/connectors-doc/ajp/ajpv13ext.html.

[5] https://src.chromium.org/viewvc/chrome/trunk/src/remoting/.