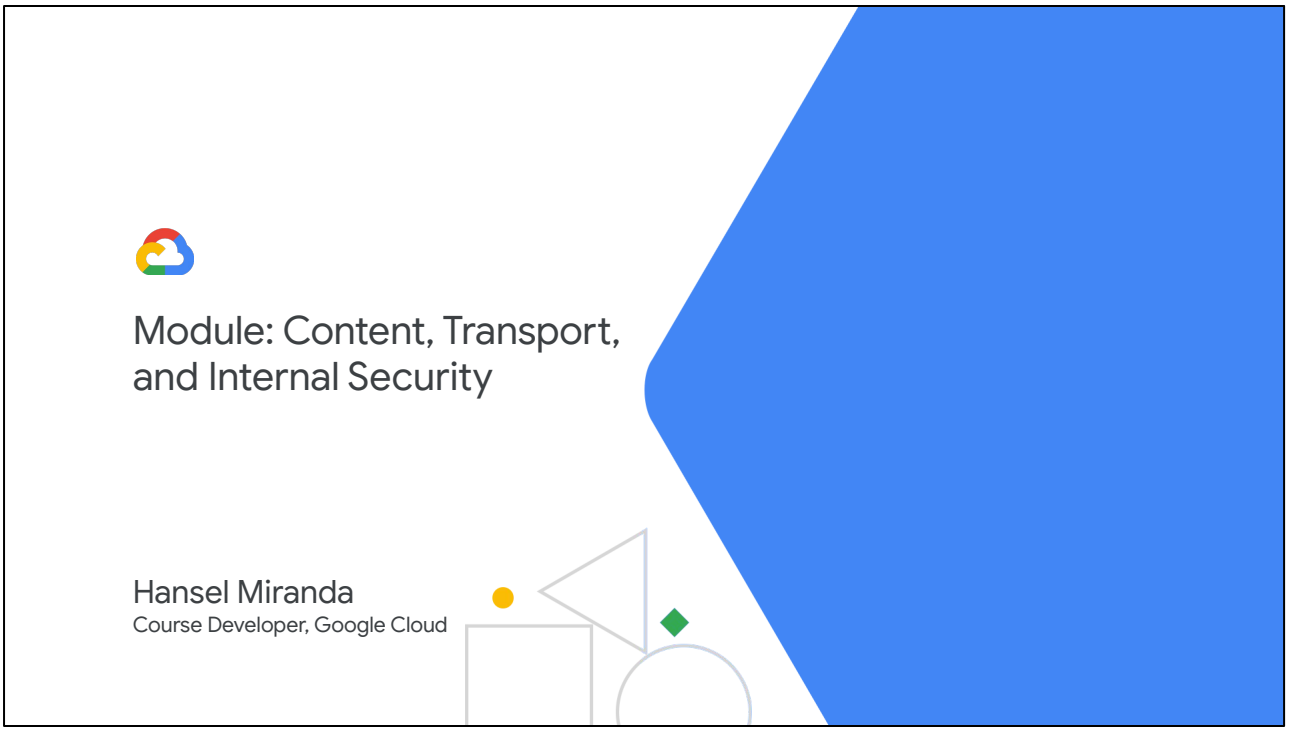


In the last module, you learned about authentication and authorization.

This module will teach you about content-based attacks, and the Apigee policies that can help protect your APIs against these attacks.

We will learn about transport security, and about features of Apigee that keep users of the Apigee Management UI from seeing sensitive data.

You will add JSON threat protection to your retail API proxy, and add an encrypted key value map to store backend credentials.

You will also complete a lab that uses the regular expression threat protection policy, and a lab that uses the management API to create a data mask to mask variables while tracing an API proxy.

In this lecture you'll learn about content-based attacks against APIs and how Apigee can help you protect your APIs against them.

Content-based API attacks use malformed API requests to cause issues with APIs and backend services.

Some attacks use text fields to compromise data in the backend, either retrieving data that the user should not be permitted to get, or destroying data in backend databases.

Other attacks use JSON or XML payloads that are crafted specifically to cause problems for parsers. This type of attack can cause an application-level denial of service, where the API or backend service stops responding to requests.

Legacy services are often susceptible to these attacks. When services are designed to receive traffic only from inside the company network, developers may not pay much attention to content-based attacks. When you start using your legacy services to build your external APIs, though, all of a sudden your services are exposed to traffic from the internet.

It is a good practice to add protection against these attacks in your API proxies so that you can block the malicious data from reaching your backend services.

Apigee has policies that can help you create this level of protection in your proxies.

Tips for input validation

- Extract fields from payloads using JSONPath/XPath in the ExtractVariables policy.
- Validate required fields, field interactions, and regular expression patterns by using proxy conditions or JavaScript policies.
- Rewrite validation error messages from the backend.

You may need to use many different types of input validation for your APIs, and most of this validation can be done in the API proxy, reducing traffic to the backend that would be rejected anyway.

API payloads are often in JSON or XML format. Fields within your JSON and XML can be extracted using the JSONPath and XPath configuration within an ExtractVariables policy.

Required fields and field dependencies can be validated using boolean conditions.

Complex formatting patterns can also be checked by using regular expressions in conditions or by using a JavaScript policy.

If you choose not to validate all input at the proxy layer, make sure that you know all of the different error messages that might be returned by your backends and have a method for remapping each one to your consistent error format for your API.

Malicious input

- Malicious consumers may send inputs that contain dangerous strings.
  - Example: SQL injection code
    - You can perform malicious input checks in the proxy even if you expect the backend to check these also.
      - You still need to use best practices like parameterized queries and escaping/encoding data.
- The RegularExpressionProtection policy can be used to detect these types of attacks.

Blocking dangerous input at the API proxy layer helps protect your backend services from intentional attacks.

For example, a common attack on an API is a SQL injection attack, where the attacker creates inputs that have SQL code or comment characters embedded in them. These inputs can cause leakage or destruction of your backend database data if you do not block them.

The correct way to block SQL injection is to escape dangerous characters and not build SQL queries using concatenation. You should protect against SQL injection in your backend services.

However, blocking known bad requests will protect your APIs even if the backend is not inherently protected.

You can use the RegularExpressionProtection policy to block dangerous string patterns.

RegularExpressionProtection policy

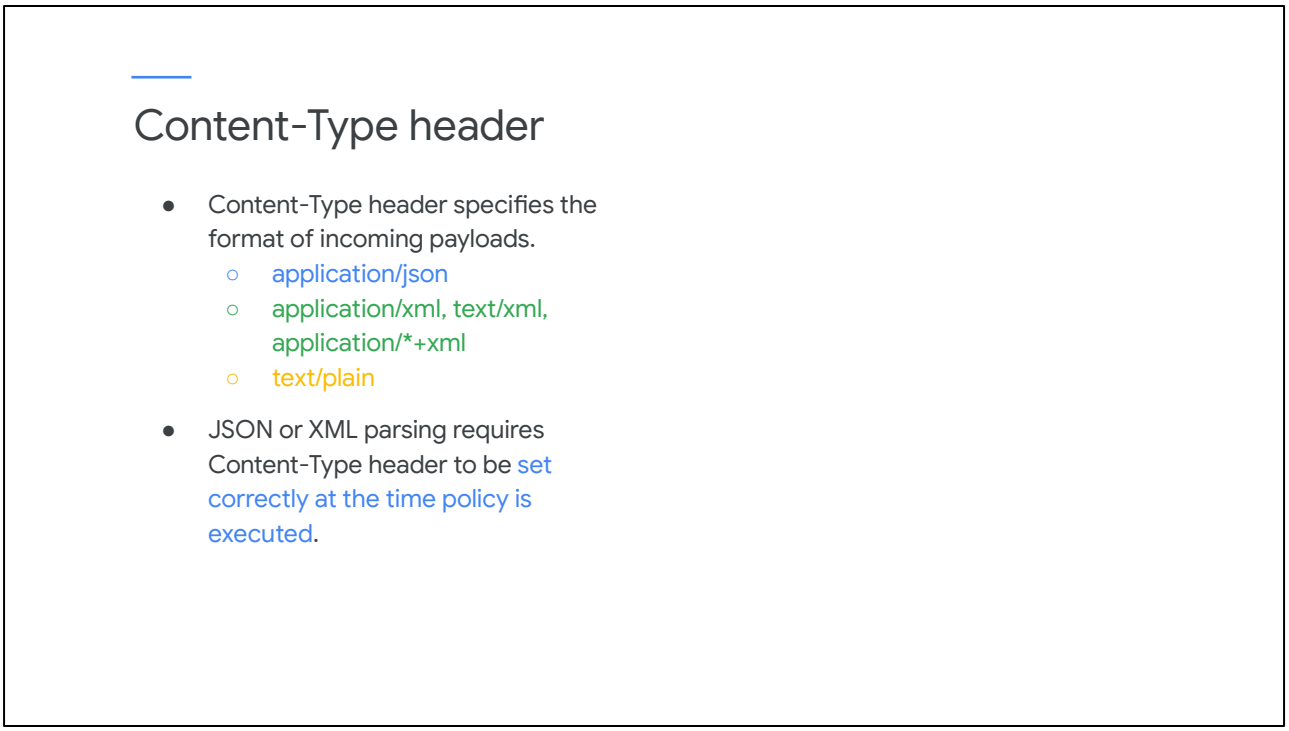
- Search for blocked patterns in any location (URI, payload, headers, query params, variables).
- If any specified regular expression matches the corresponding input field or variable, the message is considered a threat and is rejected.

```
<RegularExpressionProtection continueOnError="false"
 enabled="true" name="REP-InputProtection">
  <Source>request</Source>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
  <Variable name="request.content">
    <Pattern>[\s]*((delete)|(exec)|(drop\s*table)|(insert)|(shutdown)
|(update)|(\bor\b))</Pattern>
  </Variable>
  <JSONPayload>
    <JSONPath>
      <Expression>$.comment</Expression>
      <Pattern>[\s]*((delete)|(exec)|(drop\s*table)|(insert)
            |(shutdown)|(update)|(\bor\b))</Pattern>
    </JSONPath>
  </JSONPayload>
  <XMLPayload>
    ⋮
  </XMLPayload>
</RegularExpressionProtection>
```

The RegularExpressionProtection policy can be used to search for blocked patterns in the input request.

You can search inside variables or inside specific fields in an XML or JSON payload using XPath and JSONPath.

If any regular expression pattern in the policy matches the value in the corresponding location, the message is considered a threat, and the policy raises a fault. The API request should be rejected as a bad request.



Let's take a quick but important detour.

We've talked about using JSONPath and XPath to find specific fields inside a JSON or XML payload. How do your proxy's policies know whether the payload is JSON or XML?

You might say: if you see curly braces it's JSON, and angle brackets are XML. That's how humans tell the difference.

The proxy actually uses the Content-Type header.

The Content-Type header specifies which format is being used for the message. Apigee recognizes application/json as a JSON payload.

There are multiple formats for XML, including application/xml.

Text/plain indicates a plaintext payload.

As you use some of the powerful JSON and XML parsing functions in Apigee policies, remember that the Content-Type header must be set correctly at the time the policy is executed. For example, if you are trying to use JSONPath in a policy, and the Content-Type header is not application/json, that part of the policy will be silently skipped.

Content-Type header

- Content-Type header specifies the format of incoming payloads.
  - application/json
  - application/xml, text/xml, application/*+xml
  - text/plain
- JSON or XML parsing requires Content-Type header to be set correctly at the time policy is executed.

```
Payload:
{
  "message": "What format is this?"
}
```

Look at this example payload from an HTTP message. What format is it?

If you think it is a trick question, you're right.

It certainly looks like JSON, but we don't really know what the format is until we know what the content type is.

Content-Type header

- Content-Type header specifies the format of incoming payloads.
  - application/json
  - application/xml, text/xml, application/*+xml
  - text/plain
- JSON or XML parsing requires Content-Type header to be set correctly at the time policy is executed.

```
Content-Type: application/xml
Payload:
{
  "message": "This is invalid XML, NOT JSON!"
}
```

If the Content-Type is set to application/xml, this payload is invalid XML, not JSON.

If the Content-Type header is not set, the payload is also not considered JSON.

When creating an API that only uses JSON, you might want the default content type to be application/json when no Content-Type header is provided.

Content-Type header

- Content-Type header specifies the format of incoming payloads.
  - application/json
  - application/xml, text/xml, application/*+xml
  - text/plain
- JSON or XML parsing requires Content-Type header to be set correctly at the time policy is executed.

```
Content-Type: application/xml
Payload:
{
  "message": "This is invalid XML, NOT JSON!"
}
```

```
<Step>
  <Name>RF-ContentTypeInvalid</Name>
  <Condition>request.header.Content-Type != null AND
    request.header.Content-Type!="application/json"
  </Condition>
</Step>
<Step>
  <Name>AM-SetContentTypeToJSON</Name>
  <Condition>request.header.Content-Type == null
  </Condition>
</Step>
<!-- can now use JSON parsing -->
<Step>
  <Name>EV-ExtractFromJSON</Name>
</Step>
```

You can do this with a series of policies in a proxy.

The first condition is true if the content type header has a value and the content type is set to something other than application/json.

In this case, the caller has set a content type that is not allowed for this API.

The policy named RF-ContentTypeInvalid would raise a fault and cause an error to be returned for invalid content type.

Content-Type header

- Content-Type header specifies the format of incoming payloads.
  - application/json
  - application/xml, text/xml, application/*+xml
  - text/plain
- JSON or XML parsing requires Content-Type header to be set correctly at the time policy is executed.

```
Content-Type: application/xml
Payload:
{
  "message": "This is invalid XML, NOT JSON!"
}
```

```
<Step>
  <Name>RF-ContentTypeInvalid</Name>
  <Condition>request.header.Content-Type != null AND
    request.header.Content-Type!="application/json"
  </Condition>
</Step>
<Step>
  <Name>AM-SetContentTypeToJSON</Name>
  <Condition>request.header.Content-Type == null
  </Condition>
</Step>
<!-- can now use JSON parsing -->
<Step>
  <Name>EV-ExtractFromJSON</Name>
</Step>
```

The second condition is true if the Content-Type header is not set.

In this case, you could use the policy AM-SetContentTypeToJSON to set the Content-Type header in the request to application/json.

Content-Type header

- Content-Type header specifies the format of incoming payloads.
  - application/json
  - application/xml, text/xml, application/*+xml
  - text/plain
- JSON or XML parsing requires Content-Type header to be set correctly at the time policy is executed.

```
Content-Type: application/xml
Payload:
{
  "message": "This is invalid XML, NOT JSON!"
}
```

```
<Step>
  <Name>RF-ContentTypeInvalid</Name>
  <Condition>request.header.Content-Type != null AND
    request.header.Content-Type!="application/json"
  </Condition>
</Step>
<Step>
  <Name>AM-SetContentTypeToJSON</Name>
  <Condition>request.header.Content-Type == null
  </Condition>
</Step>
<!-- can now use JSON parsing -->
<Step>
  <Name>EV-ExtractFromJSON</Name>
</Step>
```

Now that you have a valid content type, the EV-ExtractFromJSON ExtractVariables policy will be able to parse JSON.

Parsing attacks

- XML and JSON payloads can be crafted to overwhelm parsers.
- JSONThreatProtection and XMLThreatProtection policies use string-based evaluation of the payload instead of parsing the payload.
- These policies should be run before doing any JSON/XML parsing.

XML and JSON payloads can be crafted to overwhelm parsers, causing application-level denial of service attacks.

The JSON and XMLThreatProtection policies use string-based evaluation of the payload instead of loading it into a parser.

By using string-based evaluation of the payload, you can detect XML or JSON payloads that exceed configured limits and reject the request before you ever load it into a parser.

It is important to run these protection policies before you attempt to use JSON or XML parsing at all.

For example, you may want to validate fields within your JSON by using JSONPath in an ExtractVariables policy.

You should use the JSONThreatProtection policy to confirm that the payload is safe before parsing it.

JSONThreatProtection policy

- Specify limits on JSON structures and fields.
- These limits are checked without the use of a parser.

```
<JSONThreatProtection continueOnError="false" enabled="true"
name="JTP-ValidateJSON">
  <Source>request</Source>
  <ArrayElementCount>20</ArrayElementCount>
  <ContainerDepth>10</ContainerDepth>
  <ObjectEntryCount>15</ObjectEntryCount>
  <ObjectEntryNameLength>50</ObjectEntryNameLength>
  <StringValueLength>500</StringValueLength>
</JSONThreatProtection>
```

The JSONThreatProtection policy allows you to specify limits on your JSON, like how deep your JSON can be nested, or how many elements can be in an array.

The policy will check for these structural limits by scanning through the text of the JSON object, without parsing the JSON.

JSONThreatProtection policy

- Specify limits on JSON structures and fields.
- These limits are checked without the use of a parser.
- The first exceeded limit results in an error.
- Craft limits to allow all good requests through.

```
<JSONThreatProtection continueOnError="false" enabled="true"
name="JTP-ValidateJSON">
  <Source>request</Source>
  <ArrayElementCount>20</ArrayElementCount>
  <ContainerDepth>10</ContainerDepth>
  <ObjectEntryCount>15</ObjectEntryCount>
  <ObjectEntryNameLength>50</ObjectEntryNameLength>
  <StringValueLength>500</StringValueLength>
</JSONThreatProtection>
```

```
{
  "fault": {
    "faultstring": "JSONThreatProtection[JTP-ValidateJSON]: Execution failed.
reason: JSONThreatProtection[JTP-ValidateJSON]: Exceeded object entry name
length at line 2",
    "detail": {
      "errorcode": "steps.jsonthreatprotection.ExecutionFailed"
    }
  }
}
```

The first time a limit is exceeded, the policy will raise a fault, allowing you to reject the request.

Only the first error encountered in the JSON will be mentioned in the response.

It is important to set limits that allow legitimate JSON payloads through, so you don't reject requests as false positives.

XMLThreatProtection policy

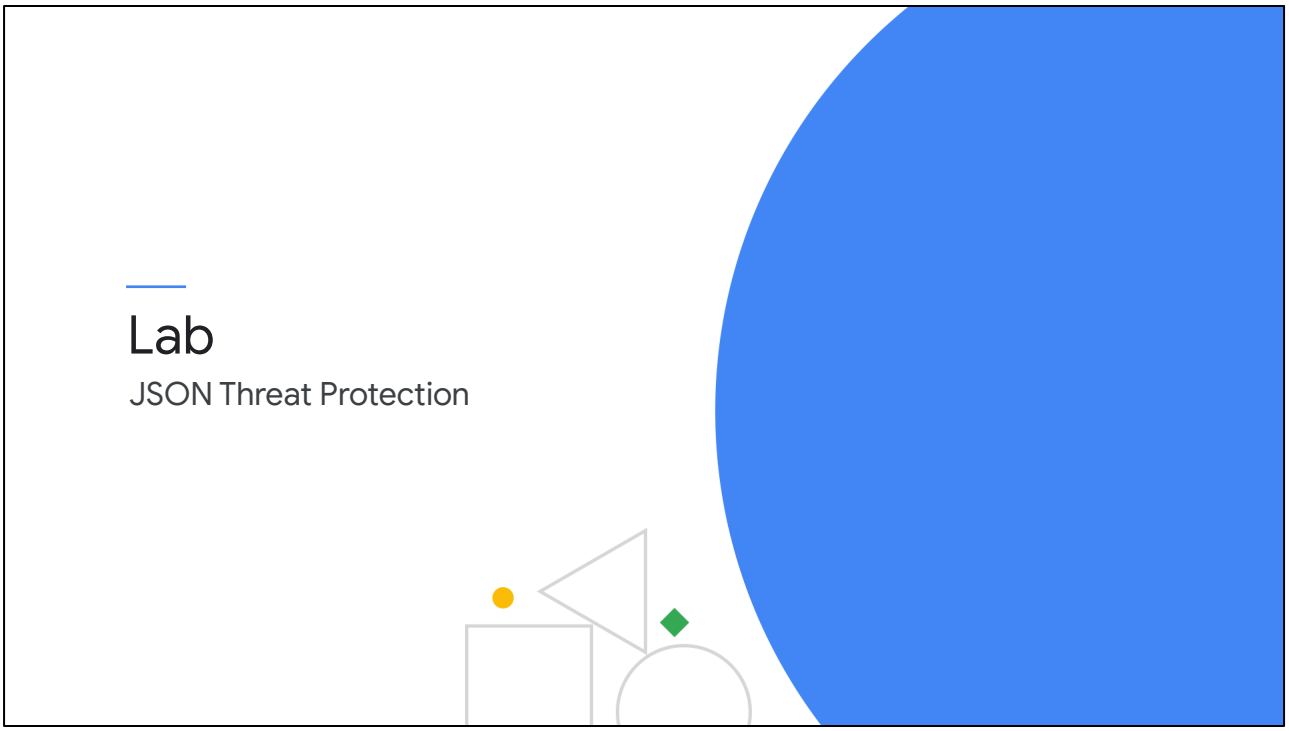
```
<XMLThreatProtection continueOnError="false" enabled="true" name="XTP-ValidateXML">
  <Source>request</Source>
  <NameLimits>
    <Element>10</Element>
    <Attribute>10</Attribute>
    <NamespacePrefix>10</NamespacePrefix>
    <ProcessingInstructionTarget>10</ProcessingInstructionTarget>
  </NameLimits>
  <StructureLimits>
    <NodeDepth>5</NodeDepth>
    <AttributeCountPerElement>2</AttributeCountPerElement>
    <NamespaceCountPerElement>3</NamespaceCountPerElement>
    <ChildCount includeComment="true" includeElement="true"
includeProcessingInstruction="true" includeText="true">3</ChildCount>
  </StructureLimits>
  <ValueLimits>
    <Text>15</Text>
    <Attribute>10</Attribute>
    <NamespaceURI>10</NamespaceURI>
    <Comment>10</Comment>
    <ProcessingInstructionData>10</ProcessingInstructionData>
  </ValueLimits>
</XMLThreatProtection>
```

- Functionality similar to JSONThreatProtection policy

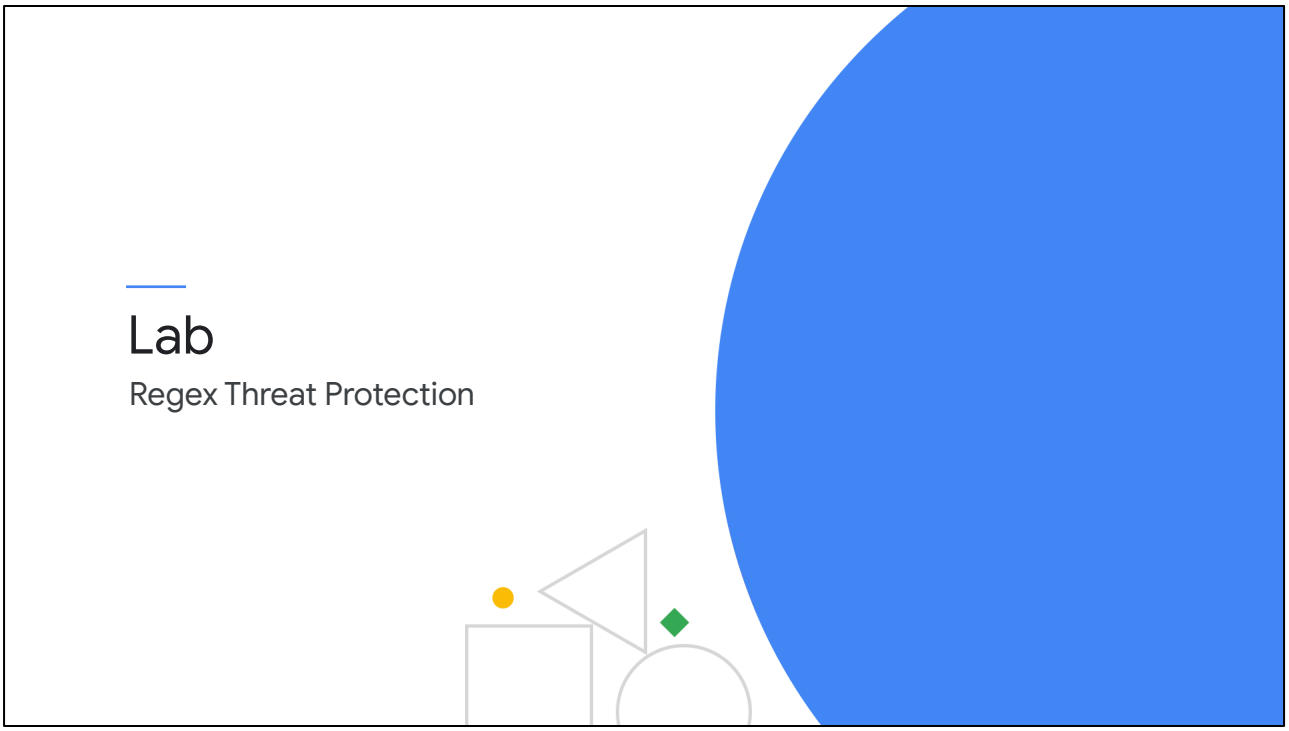
The XMLThreatProtection policy works exactly the same way as the JSONThreatProtection policy.

The XML payload is validated against configured limits without using a parser.

Note that the configuration elements are very different for XML, and there are many more options. This is because the XML and JSON formats are very different. The JSON format is simple, and the XML format is significantly more complex.



In this lab you add JSON threat protection to your retail API proxy. You will test with different payloads to see how the JSONThreatProtection policy's limits work.



In this lab you add a RegularExpressionThreatProtection policy to a new API proxy and test that the RegularExpressionThreatProtection policy rejects requests with data that matches the configured regular expressions.

During this lecture you will learn about Transport Layer Security, or TLS, which is the primary method of securing API requests and responses when sent across a network.

You'll learn about how TLS works, and how it is supported on Apigee. You'll also learn how to allow or deny traffic based on source IP address.

Transport layer security (TLS)

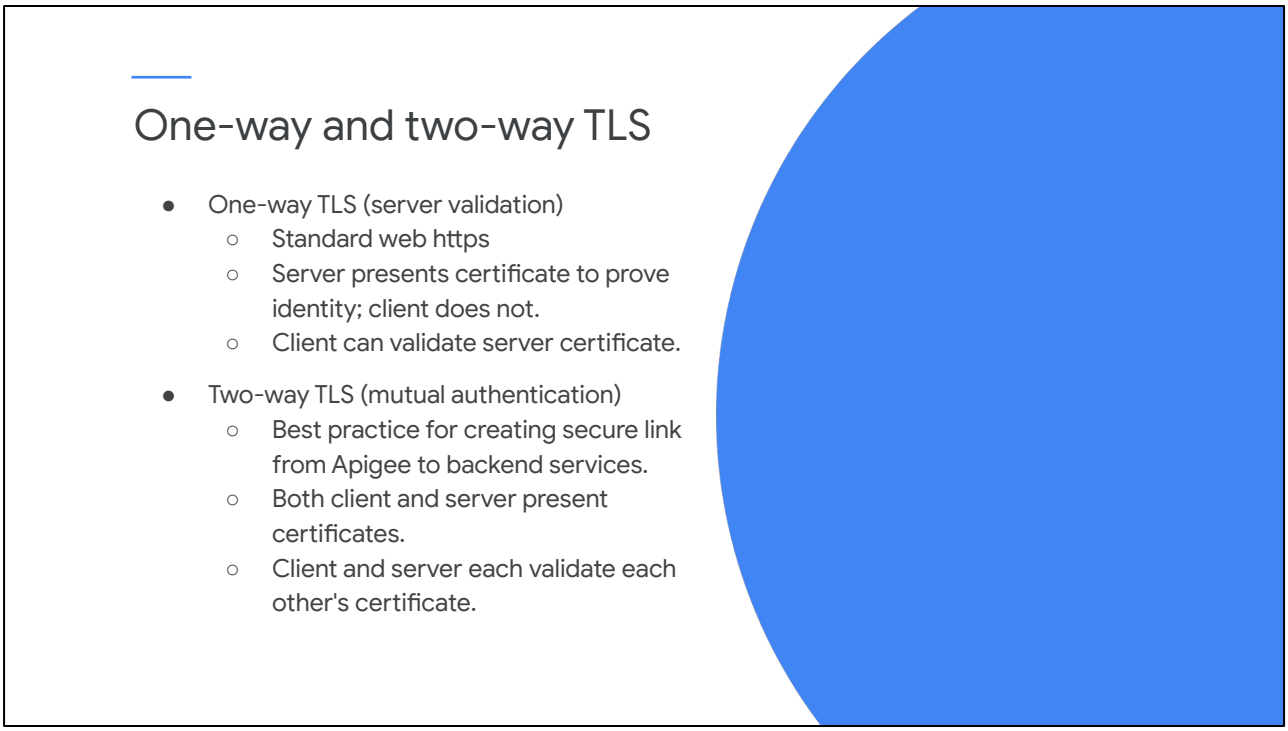
- Successor to Secure Sockets Layer (SSL).
  - SSL is deprecated
- Establishes encrypted http link between client and server (https).
- Certificates prove identity.
- All OAuth traffic must use TLS; recommended for all API traffic.

The first time a limit is exceeded, the policy will raise a fault, allowing you to reject the request. Only the first error encountered in the JSON will be mentioned in the response.

TLS establishes an encrypted link over http between a client and server. When you connect to a web page or API over https, you are using TLS.

TLS uses certificates with public and private keys to prove identity.

When we discussed OAuth, you learned that one of the firm requirements is that all OAuth traffic be sent using TLS, and, in fact, it is recommended to send all API traffic over TLS.



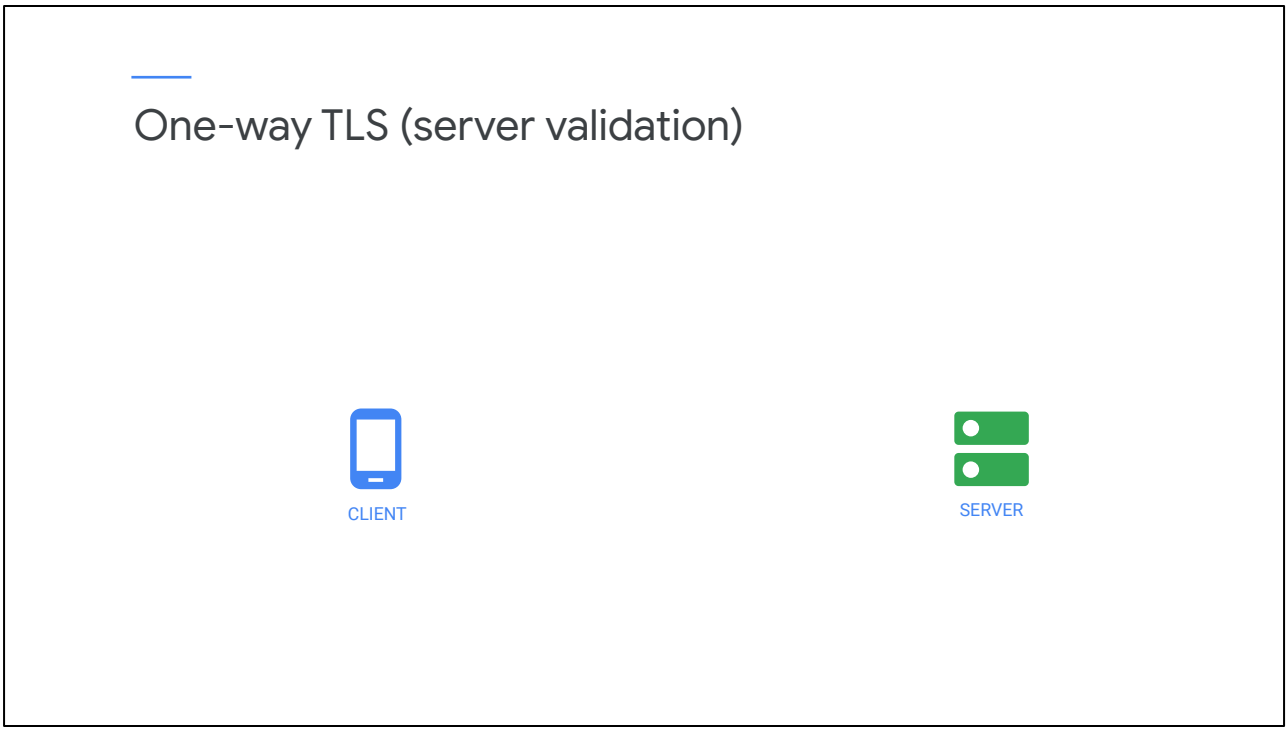
Two types of TLS connections can be made. They are commonly called one-way and two-way TLS.

One-way TLS, also called server validation, is probably very familiar to you. When you open a web page in your browser, and the request URL starts with https://, you are using one-way TLS. The server being contacted is required to present a certificate to the client to prove its identity.

For an API, we typically use something called a truststore on the client side to validate the certificate. For web requests, we trust the certificate because it has been signed by a trusted third-party certificate authority. Certificate authorities are outside the scope of this course.

Two-way TLS, also called mutual authentication, is the recommended method for securing the network connection between an Apigee proxy and the backend services it uses. Both the client and server present certificates, and each can validate the other's certificate.

Because your backend typically needs to let traffic through the firewall, it is important to validate that backend requests are coming from a legitimate source.



Let's look at one-way TLS, and how a client can trust a server.

We are calling the participants the client and the server.

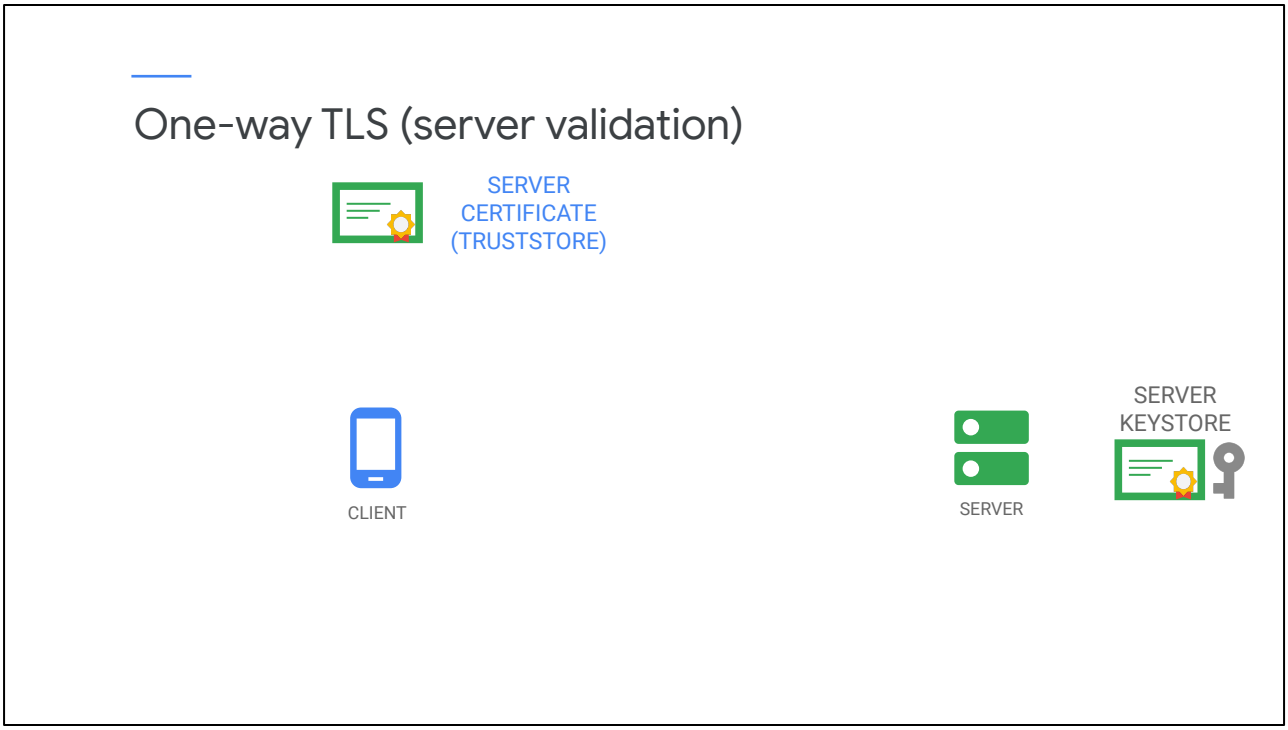
The client will initiate the request. Even though the client is represented in this diagram with a mobile device icon, the client could also be an Apigee proxy, with the server being a backend or third-party service.

Both one-way and two-way TLS can be used for http traffic between any client and server.

One-way TLS (server validation)

For one-way TLS, the server will need to present a certificate to the client. This certificate, which contains a public key, will be stored in a keystore on the server side.

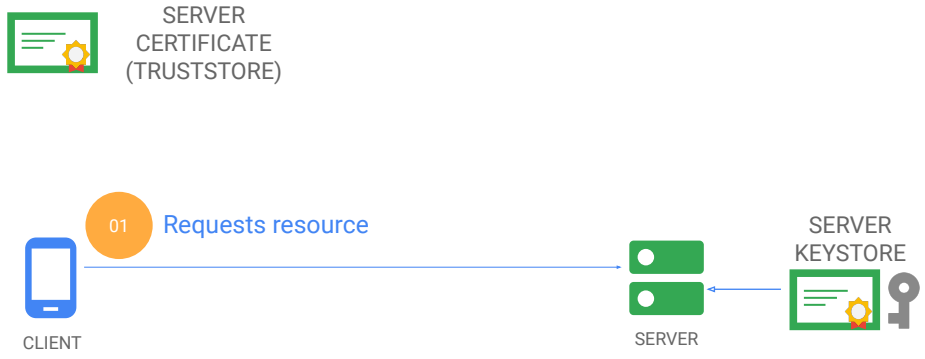
In addition, the keystore will contain the private key associated with the server certificate.



The client will need to validate the certificate from the server.

The server's certificate or chain of certificates can be stored in a truststore.

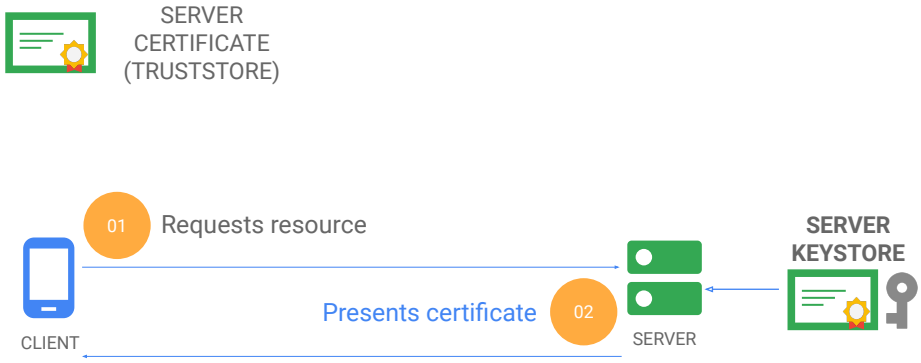
One-way TLS (server validation)



First, the client sends a request to access a resource on the server.

Before creating the encrypted connection, the server must prove its identity to the client.

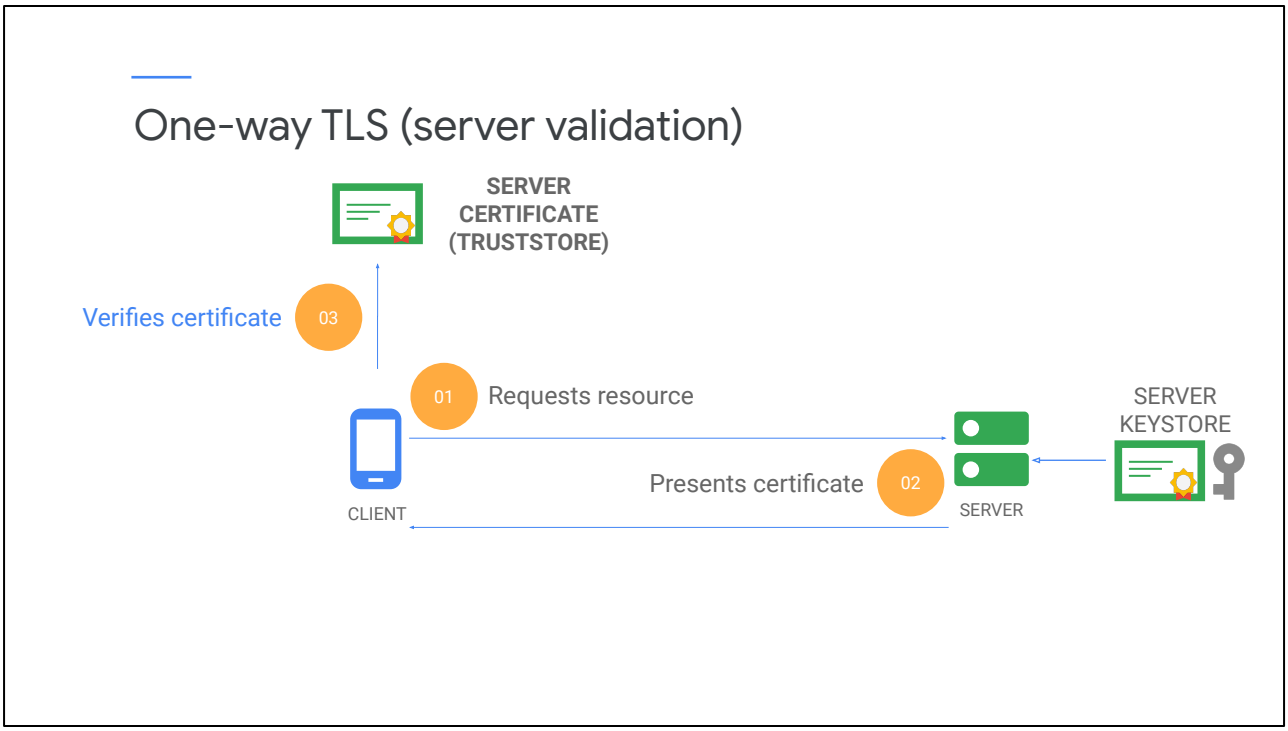
One-way TLS (server validation)



The server will present its certificate containing its public key to the client.

The certificate and public key are presented to any client connecting to the server, so the certificate is not a secret.

However, the server is the only entity that should have the associated private key.



The client will confirm that the certificate is the expected one by validating it against the truststore.

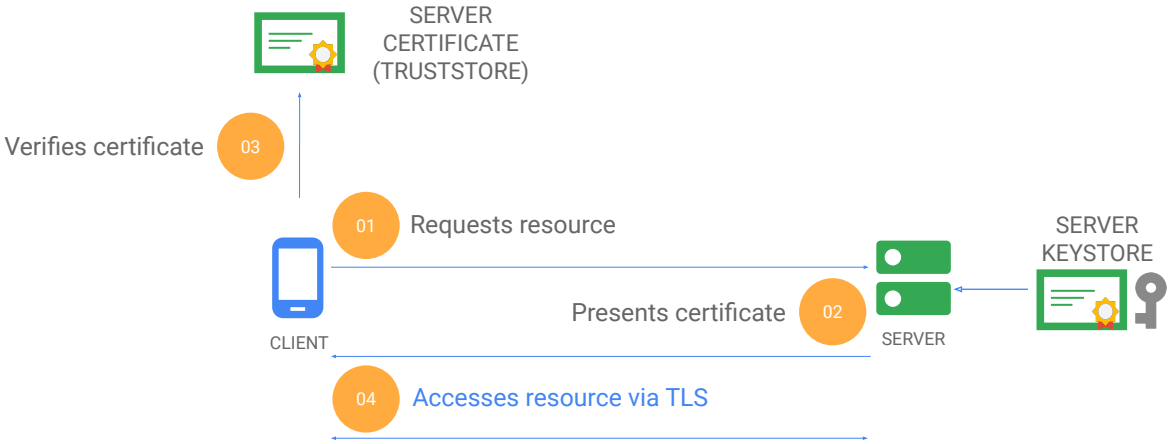
Note that a simplified, logical version of the communication between client and server is shown here; the actual TLS handshake is a bit more complex.

During this handshake, the client will typically encrypt some random data using the public key from the server certificate and send it to the server.

When the server proves to the client that it can decrypt the data, the client knows that the server must have the private key, and is therefore the legitimate holder of the certificate.

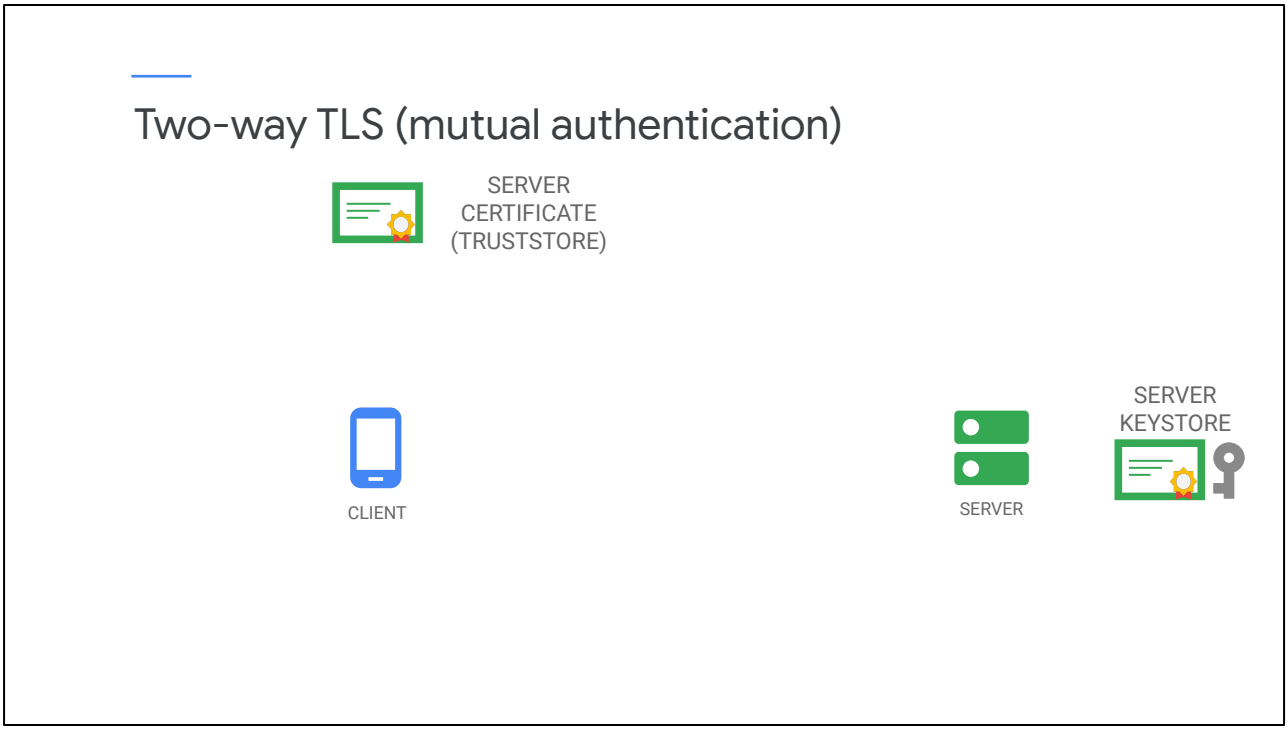
In addition to proving that the server has the private key, the two sides also negotiate the encryption cypher for the handshake and the symmetric keys that will be used to encrypt traffic across the network.

One-way TLS (server validation)



If the client is confident that the server is legitimate, the encrypted connection setup will be completed, and the client can access the resource via TLS.

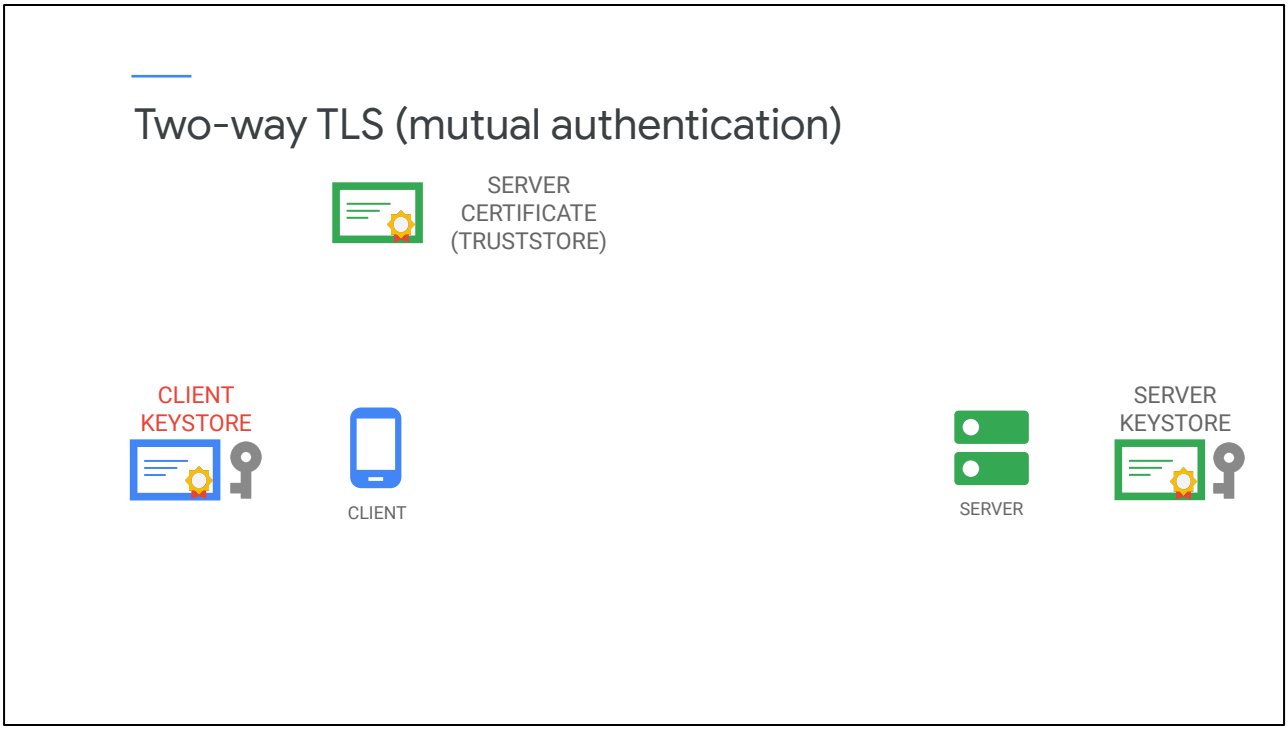
The TLS connection between client and server can remain up for future communication between client and server.



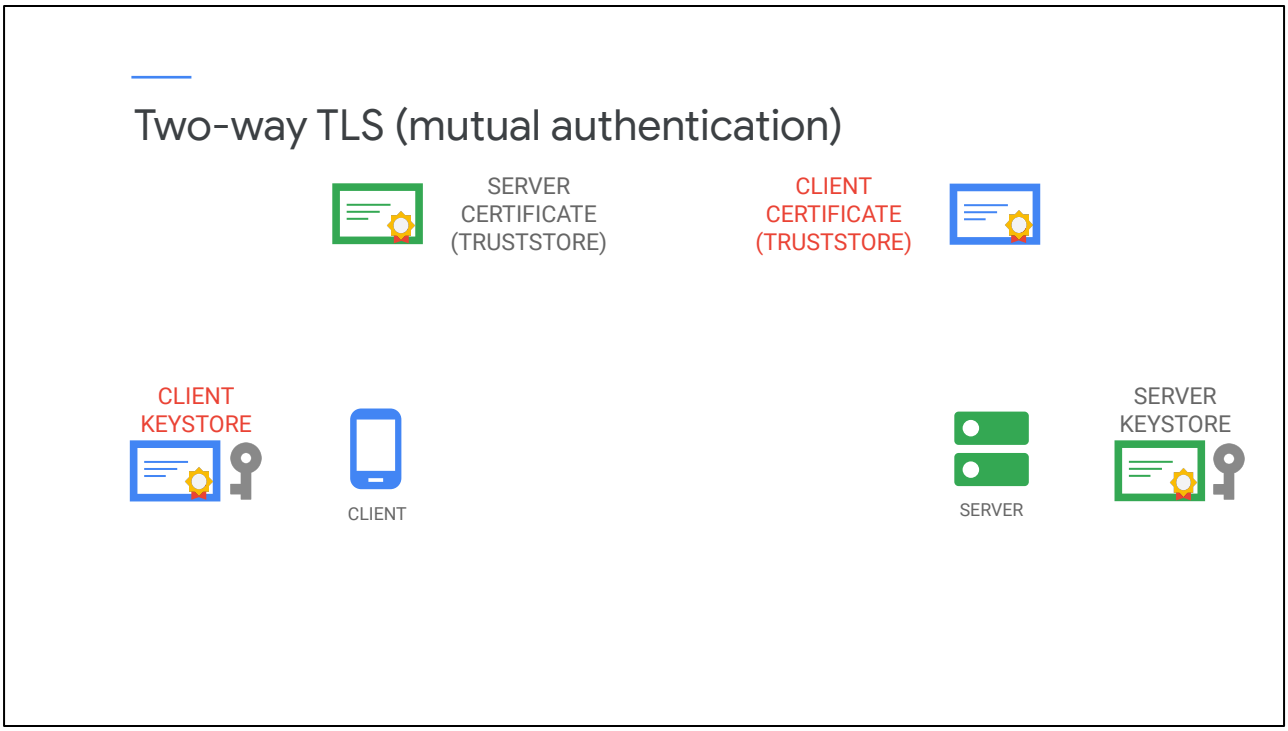
Let's see how two-way TLS is set up between client and server.

This two-way TLS is client validation layered on top of one-way TLS.

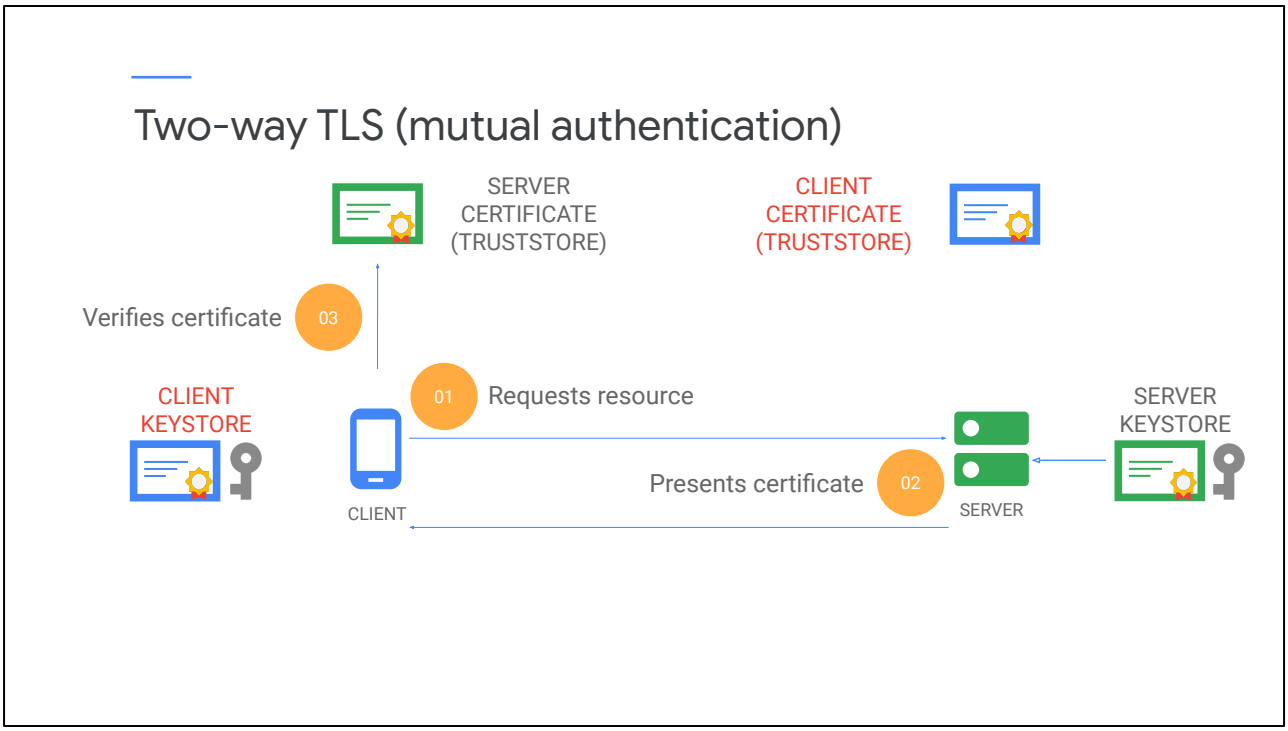
We still have the server's keystore and the client's truststore as before.



In addition, the client will need to present a certificate and encrypt using its private key, so there will be a client keystore.

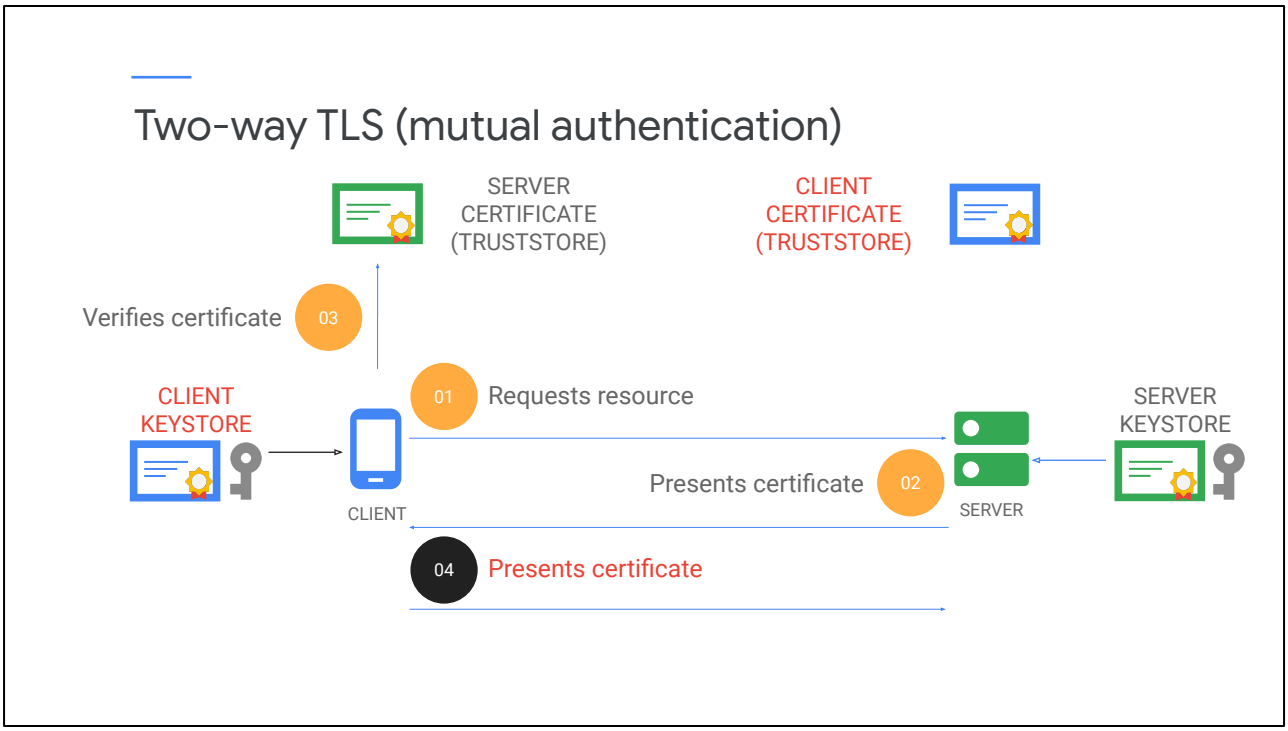


The server will need to validate that client certificate, so it will have a truststore.



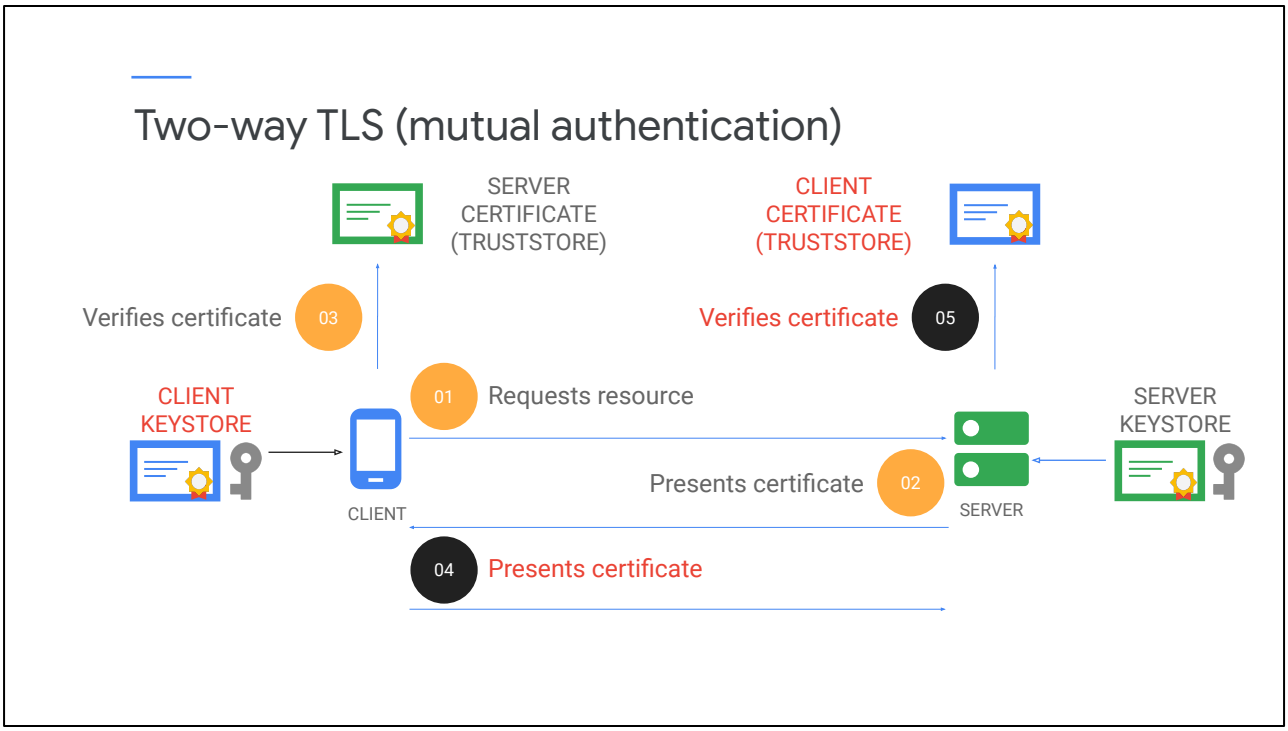
Steps 1 through 3 of the interaction are the same as before.

The client requests a resource, and the server presents its certificate and proves it has the private key.



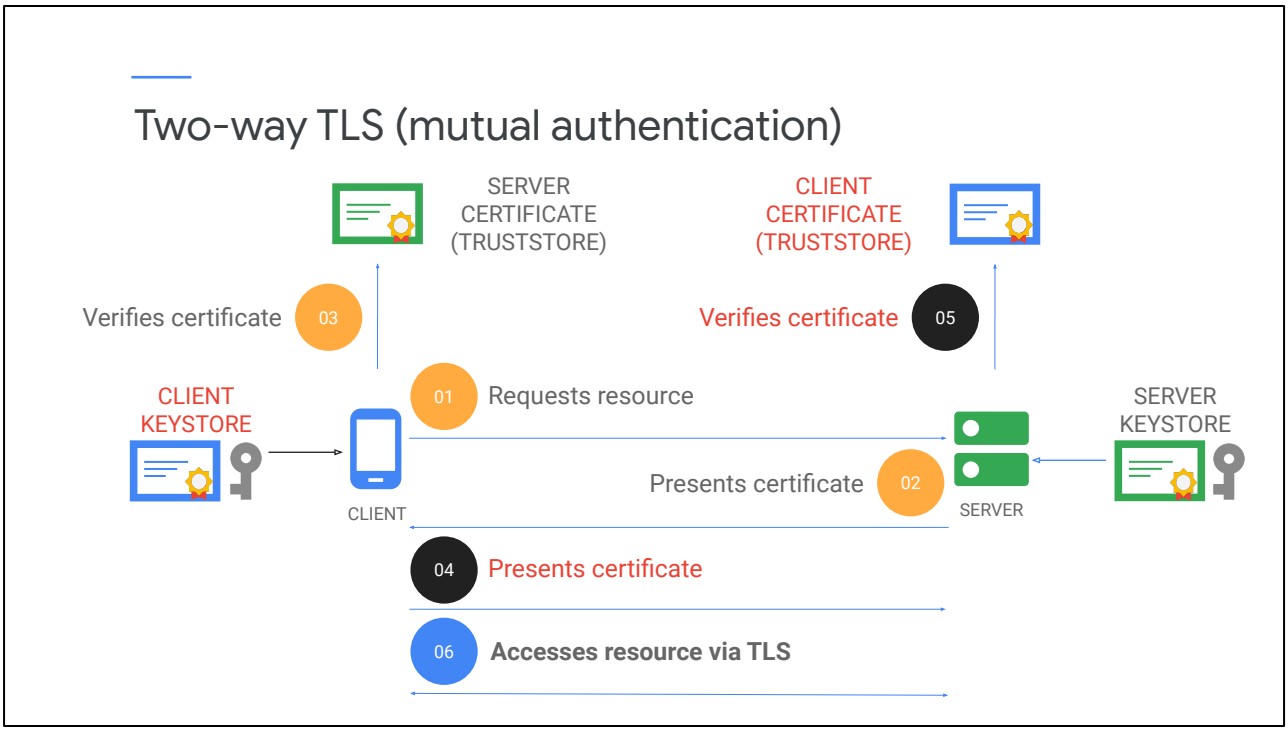
After the client is convinced of the authenticity of the server, the client needs to prove its identity.

The client takes its certificate from the keystore and presents it to the server.



The server will validate the client's certificate against its trust store.

During the handshake, the validity of the associated private key will also be proven.



Once each side is convinced of the other's identity, the encrypted TLS connection setup can complete, and the client can securely access resources on the server.

Keystores and truststores

- Keystore:
  - Stores certificates to be presented to remote server.
  - Also stores private key to prove identity.
- Truststore:
  - Stores certificates to compare with remote certificates.

On Apigee, you'll create keystores to store certificates and private keys to prove the Apigee proxy's identity over secure connections.

You'll also create truststores to store certificates that are expected from remote participants, so the proxy can verify that it is communicating with the correct participant.

Keystores and truststores

- Keystore:
  - Stores certificates to be presented to remote server.
  - Also stores private key to prove identity.
- Truststore:
  - Stores certificates to compare with remote certificates.
- Either one-way or two-way TLS may be used for each type of connection:
  - App ➔ Apigee (configured via virtual hosts)
    - May also be configured on load balancer in front of Apigee.
  - Apigee ➔ Backend (configured in target endpoints/target servers)

Apigee can create one-way or two-way connections for both incoming and outgoing traffic.

One-way or two-way TLS connections between the app, acting as the client, and Apigee, acting as the server, are configured using virtual hosts.

Note that the incoming TLS connections are often terminated at the load balancer that is fronting Apigee instead.

For Apigee communication with a backend, the TLS configuration will be set up in the TargetEndpoint in the proxy or in a TargetServer configuration referenced by the proxy.

References

```
<TrustStore>ref://truststore-reference</TrustStore>
<TrustStore>truststore-name</TrustStore>
```

- A keystore or truststore may be specified by using a keystore or truststore name or a reference.
- References are variables containing the name of the keystore or truststore.
- If a reference is not used, a proxy must be undeployed and redeployed to change the keystore or truststore for a target.
  - To change a keystore or truststore for a virtual host, servers must be rebooted.
- References allow key and certificate rotation without downtime.

When you configure a keystore or truststore on Apigee, you have two choices: configure using the keystore or truststore name, or use a reference.

A reference is a variable containing the name of a keystore or truststore. References are configured within the scope of an environment.

If you use a name in your proxies instead of a reference, changes to the certificates and private keys will only be picked up after the proxy is undeployed and redeployed. For virtual hosts not using a reference, it is even worse: the Apigee API gateway would need to be rebooted.

With a reference, a new keystore or truststore with the new certificates and keys can be created, and the reference can be updated on the fly without downtime.

Rotating keys and certificates is a best practice, so we recommend that you always use references.

Configuring one-way or two-way TLS for targets

- By default, backend certificate will not be validated.
- To validate a backend certificate, include truststore with target certs.

```
<HTTPTargetConnection>
  <URL>https://internal-test.apigee.org/store</URL>
  <SSLInfo>
    <Enabled>true</Enabled>
    <TrustStore>ref://backend-truststore</TrustStore>
  </SSLInfo>
</HTTPTargetConnection>
```

When you create a target definition, either in a proxy or in a target server, the backend certificate will not be validated by default. This means that Apigee will not validate that the certificate has expired, that the certificate's common name matches the hostname in the URL, or that it has been signed by a trusted certificate authority.

In order to validate the backend certificate, you must configure a truststore, preferably using a reference. This truststore must contain the server's certificate or a certificate chain. If the server's certificate is signed by a third party, you would need to upload the entire certificate chain, up to the root certificate authority, or CA, certificate.

Unlike a web browser, Apigee does not implicitly trust any certificate authorities.

Configuring one-way or two-way TLS for targets

- By default, backend certificate will not be validated.
- To validate a backend certificate, include truststore with target certs.
- Use a keystore and key alias to present a client certificate to the backend to prove the identity of the proxy.

```
<HTTPTargetConnection>
  <URL>https://internal-test.apigee.org/store</URL>
  <SSLInfo>
    <Enabled>true</Enabled>
    <TrustStore>ref://backend-truststore</TrustStore>
  </SSLInfo>
</HTTPTargetConnection>
```

```
<HTTPTargetConnection>
  <URL>https://internal-test.apigee.org/store</URL>
  <SSLInfo>
    <Enabled>true</Enabled>
    <TrustStore>ref://backend-truststore</TrustStore>
    <ClientAuthEnabled>true</ClientAuthEnabled>
    <KeyStore>ref://proxy-keystore</KeyStore>
    <KeyAlias>edgeKey</KeyAlias>
  </SSLInfo>
</HTTPTargetConnection>
```

It is typically recommended to have the server validate the proxy's identity using two-way TLS.

The truststore is configured the same way as for one-way TLS. In addition, the target's ClientAuthEnabled element should be set to true, and a keystore and key alias should be configured.

The keystore contains the certificate and private key. You should configure the keystore as a reference.

The KeyAlias is the name of the certificate and key within the keystore. Multiple certificate and key pairs can be stored in the same keystore by using different key alias names.

Key aliases cannot be specified by reference, so a new keystore should use the same key alias name when used for certificate rotation.

AccessControl policy

```
<AccessControl continueOnError="false" enabled="true"
name="AC-WhitelistIPs">
  <IPRules noRuleMatchAction="DENY">
    <MatchRule action="ALLOW">
      <SourceAddress mask="20">172.217.32.0</SourceAddress>
      <SourceAddress mask="16">173.194.0.0</SourceAddress>
    </MatchRule>
    <MatchRule action="ALLOW">
      <!-- use app custom attributes for CIDR -->
      <SourceAddress mask="verifyapikey.VK-VerifyKey.app_cidr_mask">
{verifyapikey.VK-VerifyKey.app_cidr_addr}</SourceAddress>
    </MatchRule>
  </IPRules>
</AccessControl>
```

- IP allow/deny lists for ranges of IPs
- Specify one or more rules to be evaluated in order.
- noRuleMatchAction specifies default functionality.

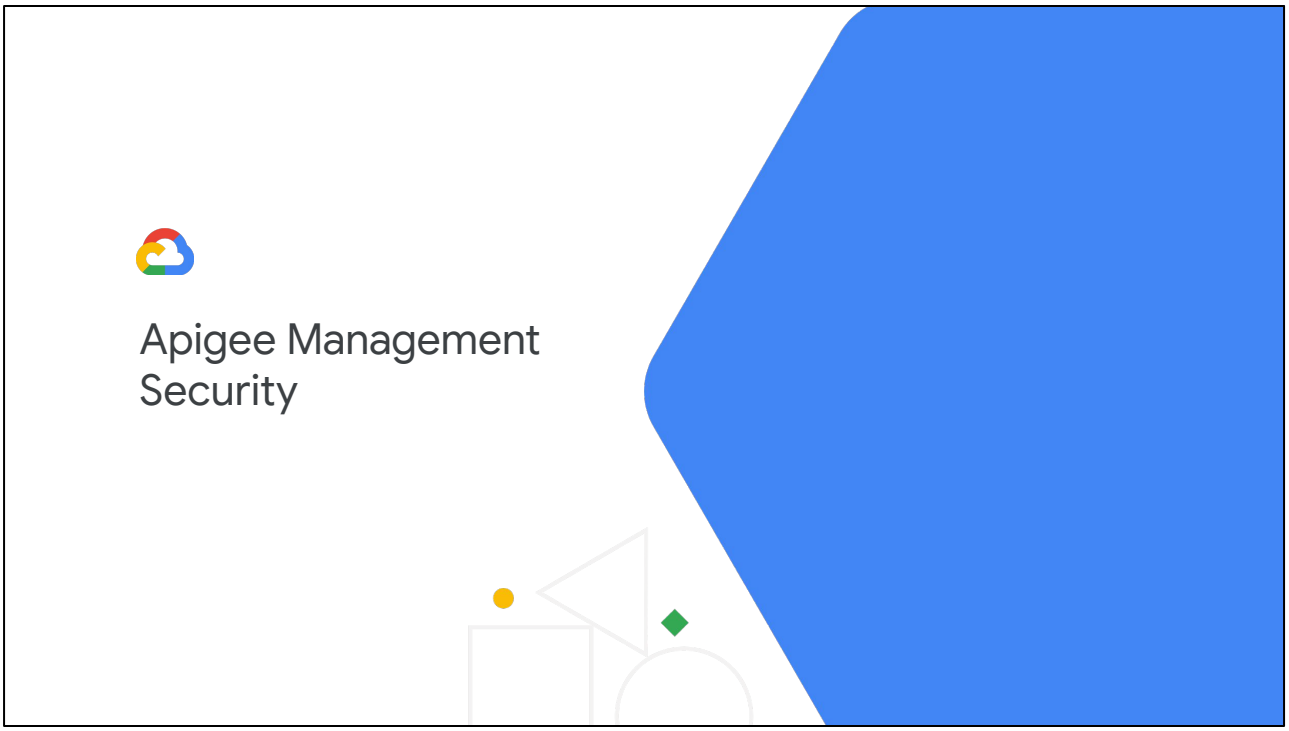
An additional method to restrict incoming connections to your API proxies is to use the AccessControl policy.

Allow and deny rules for IP addresses and IP address ranges can be specified, and they are evaluated in order.

The action of the first matching rule determines whether the traffic is allowed or denied.

Address and address ranges in rules can be hardcoded in the policy or specified using variables.

The noRuleMatchAction element specifies whether the default behavior is to allow or deny traffic, if none of the rules match.



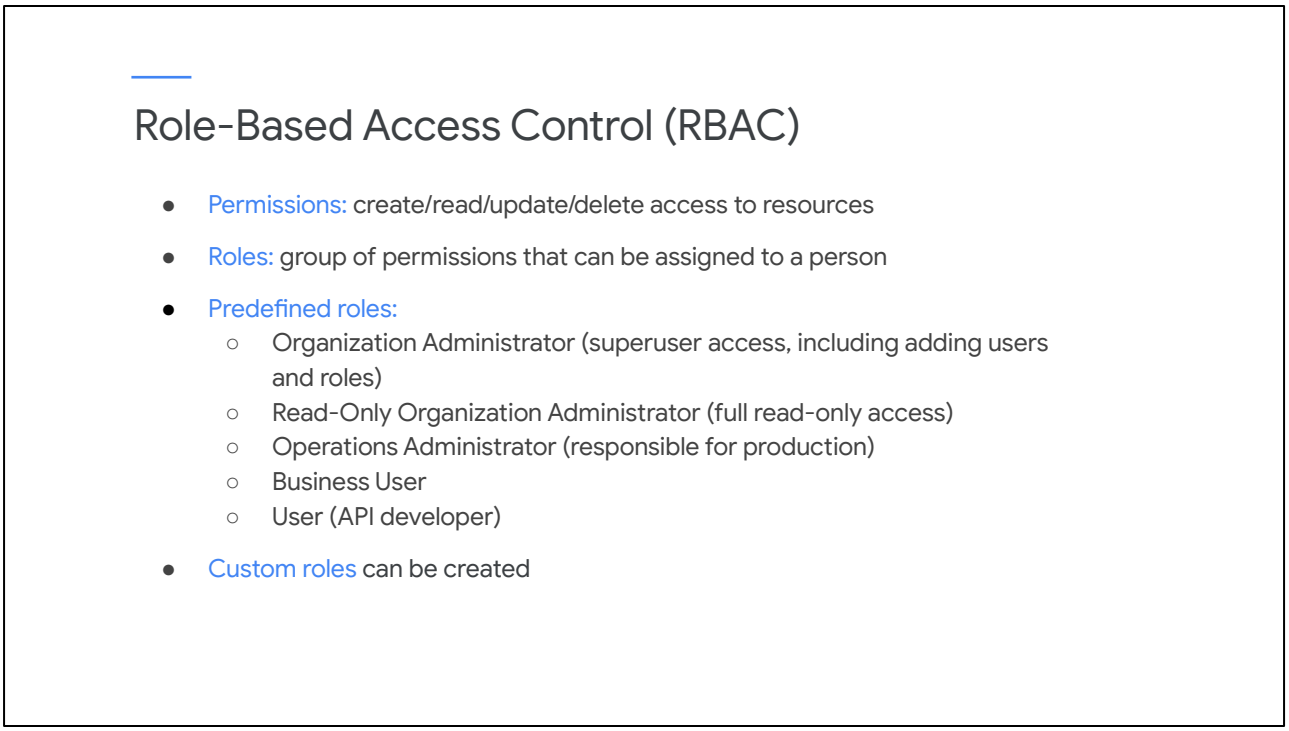
You've learned quite a bit about securing APIs from external threats.

Now we'll discuss internal security.

Role-based access control controls the level of access for users of Apigee.

Data masking and private variables can be used to prevent sensitive data from being visible when live traffic is being traced.

And Encrypted Key Value Maps can be used to store and use configuration data like credentials without allowing users of the Apigee platform to see the values.



Role-based Access control, or RBAC, provides controlled levels of access to users of the Apigee management platform.

Tasks that can be performed in the management UI are the equivalent of one or more management API calls.

Permissions are based on management API calls to resources: if the user can make an API call, she can perform the same action within the UI. Similarly, a user who does not have permission to make a management API call cannot perform the same action within the UI.

Roles are used to group permissions.

Roles can be assigned to users of the platform, and a user can have more than one role. When more than one role exists for a user, the user is able to perform any operation that either role is allowed to perform.

Apigee comes with several predefined roles.

Here are a few examples:

- An organization administrator, or orgadmin, is a superuser. A person with the orgadmin role has full control of the organization and can also add users and custom roles to the organization.
- A read-only orgadmin has full read access to the organization.

- An operations administrator manages API deployments and troubleshoots proxies, and can deploy, test, and trace APIs, but not create or edit them.
- A business user manages business entities like API products, developers, developer apps, and custom reports, but not API proxies or deployments.
- A user is an API developer, with the ability to edit API proxies.

You may sometimes find that the built-in roles might not work for your use case. Apigee also provides the ability to create custom roles. You can create custom roles to customize permissions for your use cases.

Data masking

- Trace tool allows users to see data being accessed during API calls, including sensitive data.
- Data masking can block fields from being viewed.
- Data masking is configured using the management API.

Use of the Apigee trace tool is vital for troubleshooting issues with your API. However, the user tracing live API traffic can, by default, see all of the fields that are being accessed during the API calls. This data can include sensitive user data, including names, passwords, or credit card numbers. A user can also see credentials that are being used to communicate with backend services. When you download a trace log, those values will also be in the trace file.

Data masking can block configured data from being visible in live trace or trace files. You specify certain patterns of field or variable names, and matching data will be masked using a series of asterisks.

Data masking is a feature that cannot be controlled using the Apigee management UI. Data masks can only be set up by using the management API. Data masks can be configured at the organization level, and these masks will be active for all APIs in the organization. They can also be configured for a specific API proxy, adding additional fields to mask that are specific to the API.

Because data masks are not visible in the UI, API teams sometimes forget about them. Protecting data is an important part of your API security, so you should create data masks for your organization and proxies and ideally store the data masks in source control with your API proxies.

Data masking configuration

All proxies in the org:
POST /v1/o/{org}/maskconfigs

Single proxy:

POST /v1/o/{org}/apis/{proxy}/maskconfigs

```
<MaskDataConfiguration name="default">
  <JSONPathsRequest>
    <JSONPathRequest>$.logonPassword</JSONPathRequest>
  </JSONPathsRequest>
  <Variables>
    <Variable>request.formparam.creditcard</Variable>
  </Variables>
</MaskDataConfiguration>
```

Request Content

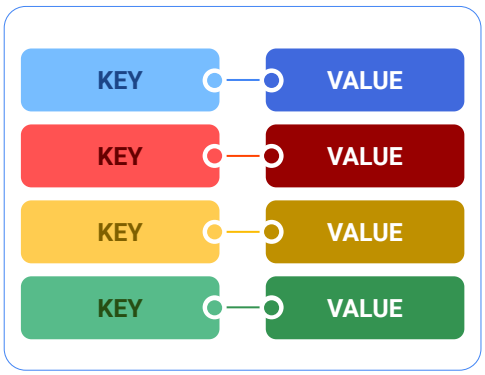
Body	{"logonPassword":"**********","lastName":"Smith","fi

Data masks are created by posting a request to the mask configs resource.

The payload contains a list of variables and JSONPath or XPath expressions for API requests or responses. In this case, both the logon password field and the form parameter named credit card of an incoming JSON request payload would be masked.

When tracing, you'll see the masked value replaced with asterisks. A downloaded trace file will also have the data masked.

Key Value Maps (KVMs)



- Storage for non-expiring data retrieved at runtime
- Keys and values both must be strings
- Typically environment-scoped
  - Ideal for environment-specific configuration
- Read/update/delete KVM entries via management API, management UI, or policy

Key value maps, or KVMs, are used to store non-expiring configuration data for use by proxies at runtime. You can think of a KVM as a replacement for a property file.

Both the key and the value for a key value map entry must be a string. You can convert the value to another data type, like a number, inside the proxy.

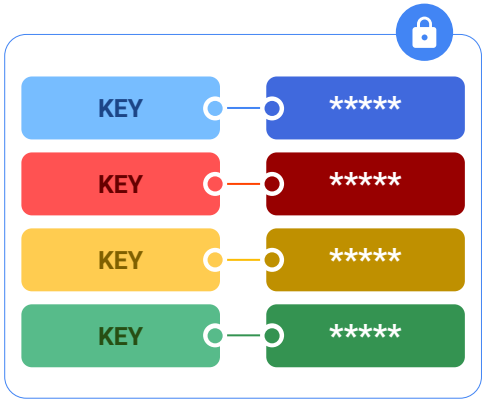
KVMs can be scoped to an organization, an environment, or a specific proxy, but environment-scoped KVMs are the most common. The management UI only allows you to manage environment-scoped KVMs.

KVMs are ideal for environment-specific configuration.

You can read and modify KVM entries using the management API, the management UI, or the KeyValueMapOperations policy.

Encrypted KVMs

- For sensitive data, like backend credentials
- Mark as encrypted at time of creation
  - Values encrypted with Apigee-generated AES-128 cypher key
- Values cannot be retrieved by using management UI or API.
- Can only retrieve values by using private variables in a proxy.



By default, key value maps are not encrypted.

Sensitive configuration data, like backend credentials, should not be stored in an unencrypted KVM.

You choose whether a KVM is encrypted or not when the KVM is created.

If the KVM is encrypted, all values in the key value map will be encrypted with an Apigee-generated AES 128 cypher key.

You cannot see the values of the KVM entries whenever you use the management API to retrieve KVM data or view the KVM in the management UI.

The only way to retrieve the unencrypted values from an encrypted KVM is to use the KeyValueMapOperations policy to retrieve the values into private variables.

Private variables

- *private.{varname}* is a private variable.
- Used like any other variable, but values do not show up in trace when created or accessed.
- When a private variable is assigned to another variable, the value is visible in the new variable.

A private variable is created by using a prefix of "private." in the variable name. So, private.password is a private variable.

A private variable can be used like any other variable, but, when an API is traced, the data value of a private variable is automatically masked.

Note that when you assign a private variable to another variable, the value will be visible when the non-private variable is accessed.

KeyValueMapOperations policy

- Inserts, updates and retrieves data from KVMs.

```
<KeyValueMapOperations continueOnError="false"
 enabled="true" name="KVM-GetItems"
 mapIdentifier="backend-creds">
  <Scope>apiproxy</Scope>
  <ExpiryTimeInSecs>300</ExpiryTimeInSecs>
  <Get assignTo="private.backendPassword">
    <Key>
      <Parameter ref="backendServiceName"/>
      <Parameter>password</Parameter>
    </Key>
  </Get>
</KeyValueMapOperations>
```

The KeyValueMapOperations policy can be used to access and update KVM data from within a proxy.

This policy can insert, update, or retrieve data from KVMs.

KeyValueMapOperations policy

```
<KeyValueMapOperations continueOnError="false"
 enabled="true" name="KVM-GetItems"
 mapIdentifier="backend-creds">
  <Scope>apiproxy</Scope>
  <ExpiryTimeInSecs>300</ExpiryTimeInSecs>
  <Get assignTo="private.backendPassword">
    <Key>
      <Parameter ref="backendServiceName"/>
      <Parameter>password</Parameter>
    </Key>
  </Get>
</KeyValueMapOperations>
```

- Inserts, updates and retrieves data from KVMs.
- *mapIdentifier* specifies the KVM to access.

The KVM's name is specified using the mapIdentifier field.

KeyValueMapOperations policy

```
<KeyValueMapOperations continueOnError="false"
 enabled="true" name="KVM-GetItems"
 mapIdentifier="backend-creds">
  <Scope>apiproxy</Scope>
  <ExpiryTimeInSecs>300</ExpiryTimeInSecs>
  <Get assignTo="private.backendPassword">
    <Key>
      <Parameter ref="backendServiceName"/>
      <Parameter>password</Parameter>
    </Key>
  </Get>
</KeyValueMapOperations>
```

- Inserts, updates and retrieves data from KVMs.
- *mapIdentifier* specifies the KVM to access.
- *Scope* defaults to environment.

The scope element specifies where that KVM is found.

In the example, the backend creds KVM is API proxy scoped, so the KVM is specific to the proxy containing this policy.

The map identifier field must always be set, but the scope will default to environment if not set.

KeyValueMapOperations policy

```
<KeyValueMapOperations continueOnError="false"
 enabled="true" name="KVM-GetItems"
 mapIdentifier="backend-creds">
  <Scope>apiproxy</Scope>
  <ExpiryTimeInSecs>300</ExpiryTimeInSecs>
  <Get assignTo="private.backendPassword">
    <Key>
      <Parameter ref="backendServiceName"/>
      <Parameter>password</Parameter>
    </Key>
  </Get>
</KeyValueMapOperations>
```

- Inserts, updates and retrieves data from KVMs.
- *mapIdentifier* specifies the KVM to access.
- *Scope* defaults to environment.
- *ExpiryTimeInSecs* indicates number of seconds to store entries (when adding) or cache entries (when retrieving).

The expiration time in seconds, when retrieving an entry, indicates how long the proxy can cache the value for that entry.

300 seconds is the default value.

Reading KVM data on every call can have a significant effect on proxy performance, so it is recommended to cache data for as long as possible.

KeyValueMapOperations policy

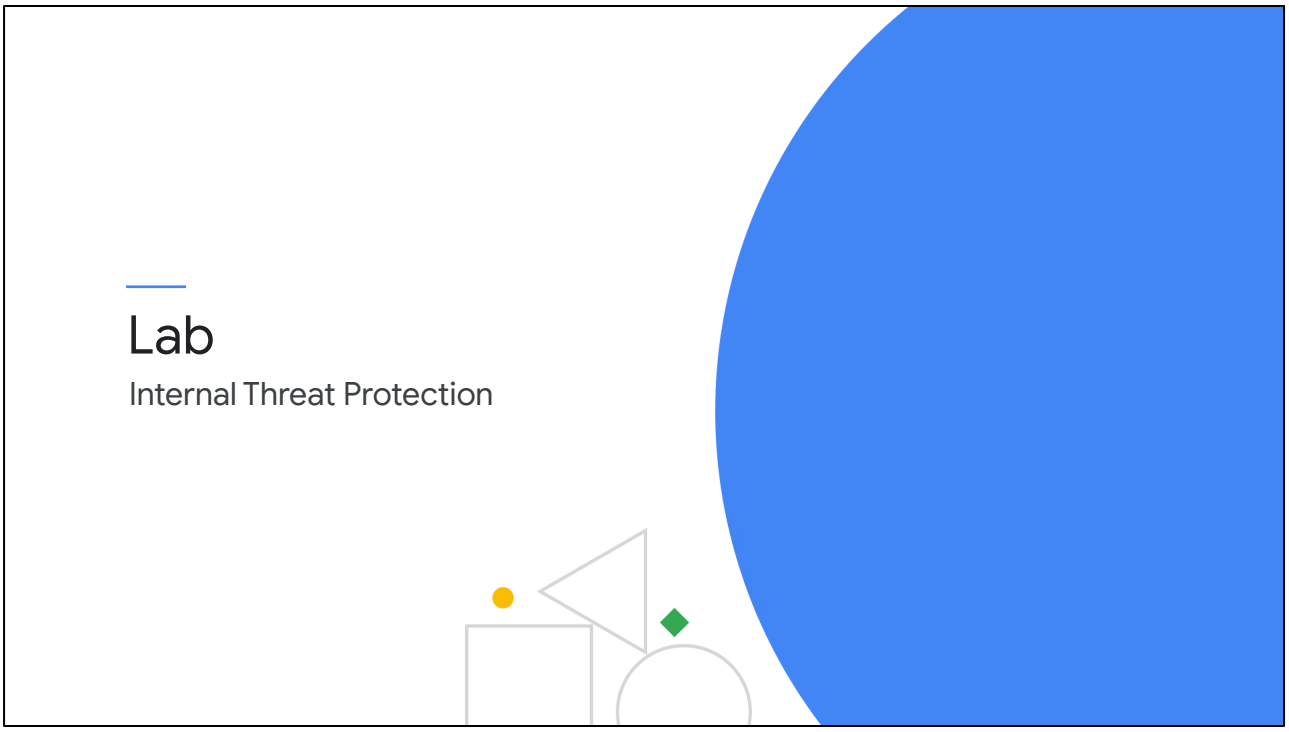
```
<KeyValueMapOperations continueOnError="false"
 enabled="true" name="KVM-GetItems"
 mapIdentifier="backend-creds">
  <Scope>apiproxy</Scope>
  <ExpiryTimeInSecs>300</ExpiryTimeInSecs>
  <Get assignTo="private.backendPassword">
    <Key>
      <Parameter ref="backendServiceName"/>
      <Parameter>password</Parameter>
    </Key>
  </Get>
</KeyValueMapOperations>
```

- Inserts, updates and retrieves data from KVMs.
- *mapIdentifier* specifies the KVM to access.
- *Scope* defaults to environment.
- *ExpiryTimeInSecs* indicates number of seconds to store entries (when adding) or cache entries (when retrieving).
- *Get* retrieves entries; *Put* creates or updates.
  - One or more *Parameter* fields as your key
  - Encrypted KVMs use private variables

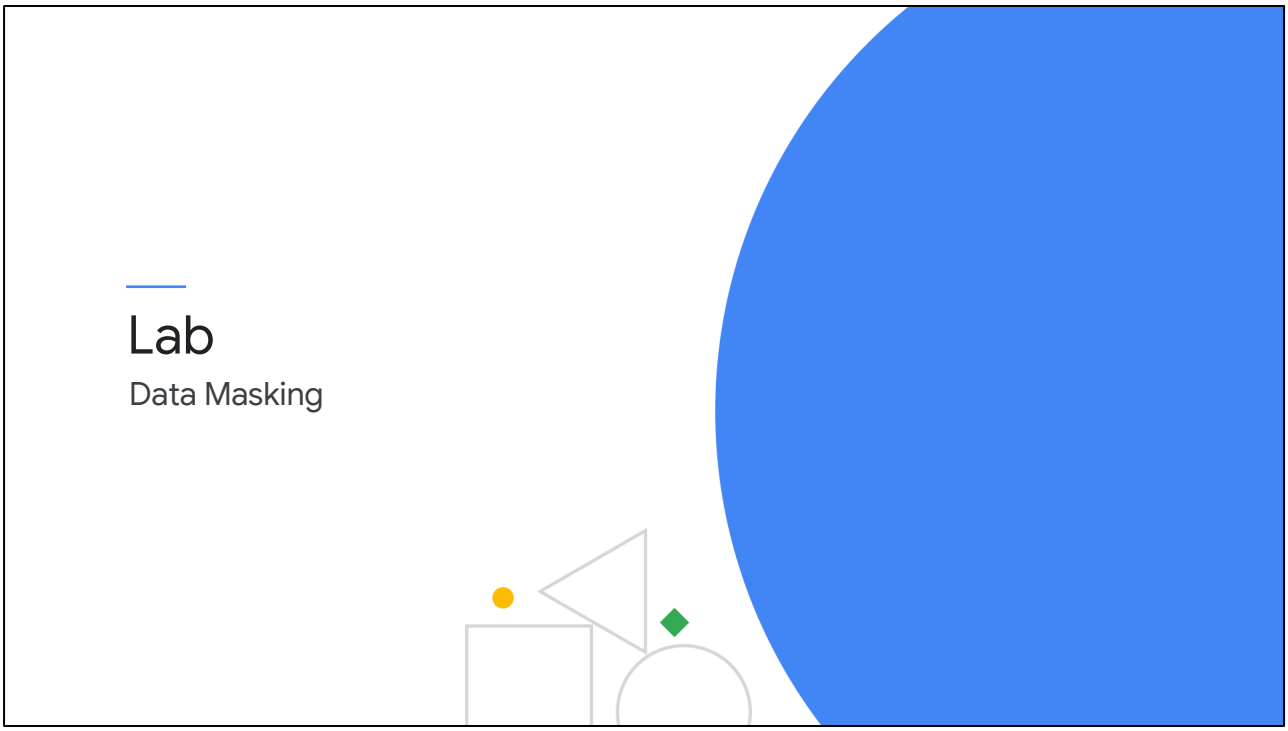
Entries are created or updated using a Put element, and retrieved by using a Get element.

One or more parameter fields are specified as your key, and each of these fields can be hardcoded or read from a variable. In the example, the backend service name is a variable, but password is a hardcoded string.

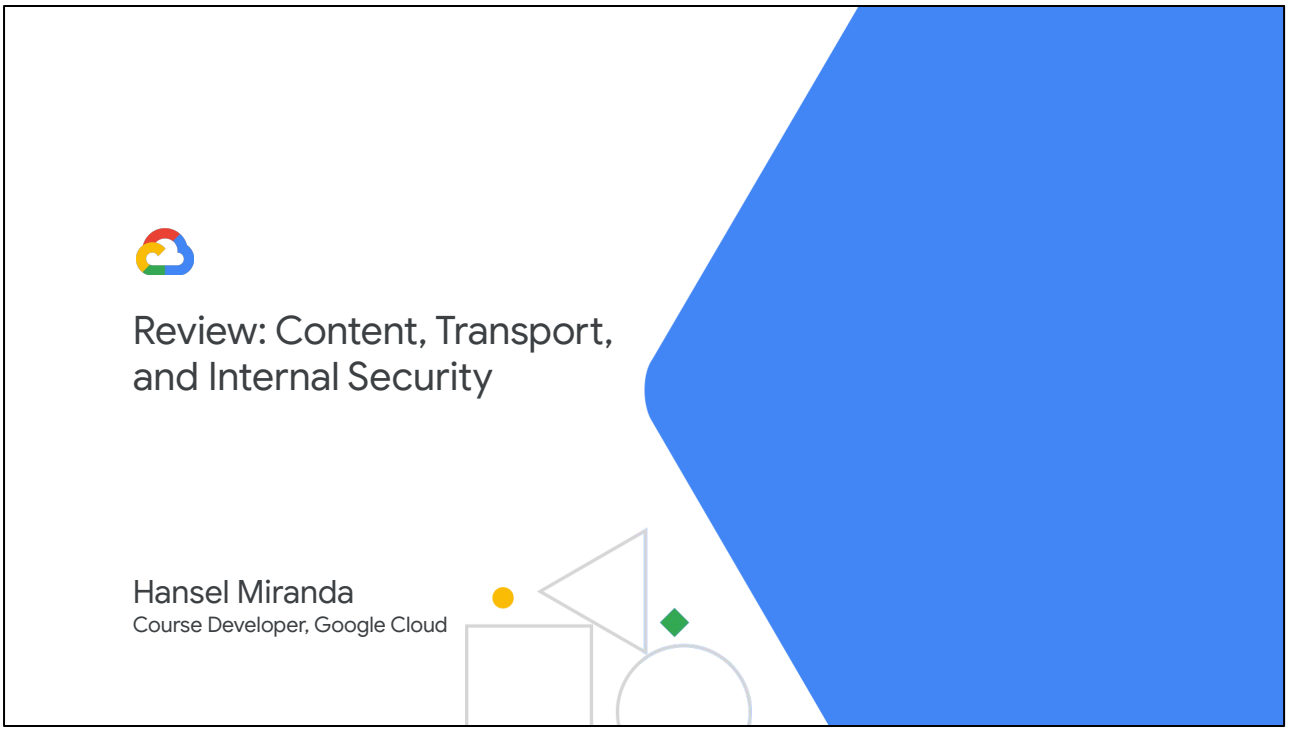
To retrieve data from an encrypted KVM, you must use assign the KVM entry value to a private variable, or the policy will throw a runtime error.



In this lab you protect backend credentials from being exposed in the Apigee management UI. You create an encrypted key/value map and add the credentials to it. You will use those credentials in your proxy to create a Basic Authentication header for calls to the backend.



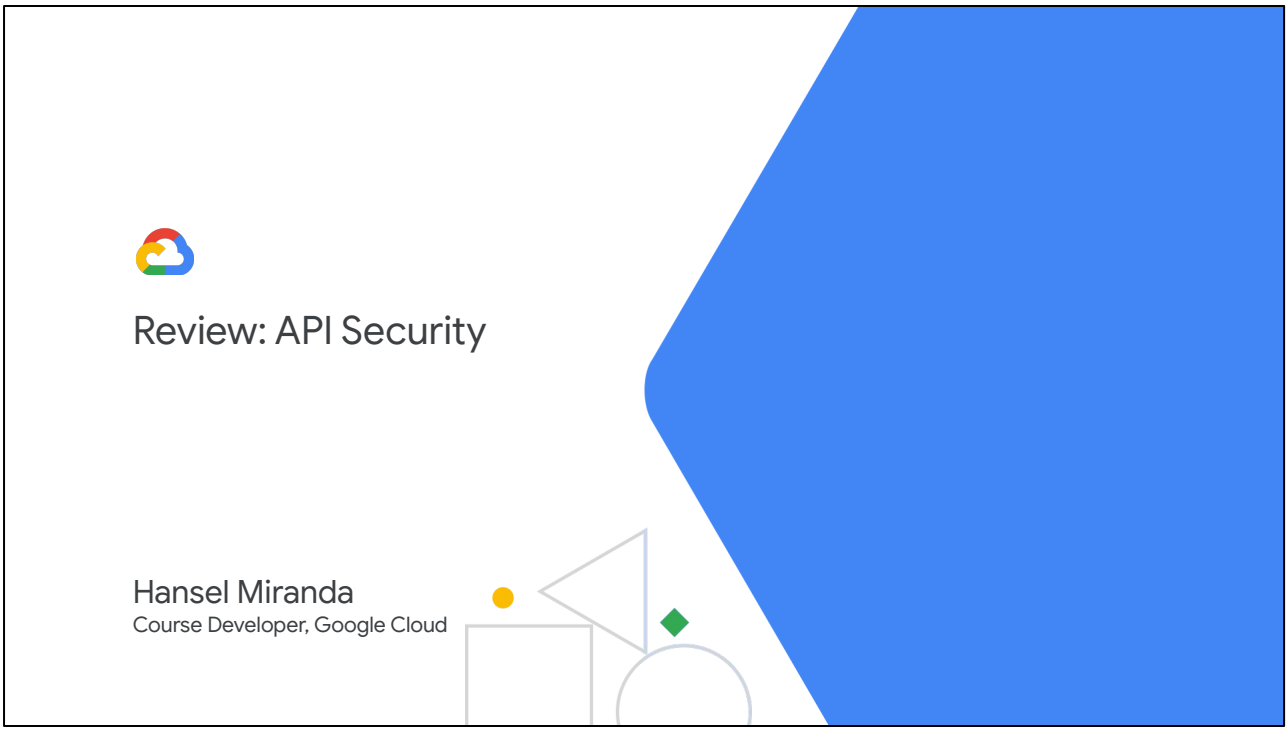
In this lab you learn how to protect against internal access of sensitive data by using private variables and data masking.



In this module, you learned how to protect your API proxies against content-based attacks, and how to keep users from seeing sensitive data in the Apigee Management UI.

You learned about transport security, and how to use one-way and two-way TLS for communicating between client apps and Apigee and backend services.

You also completed labs using threat protection policies, encrypted key value maps, and data masking.



Thank you for taking the API Security course.

During this course you learned about the different types of security concerns your APIs must protect against.

We learned about many aspects of OAuth, including the OAuth grant type flows and when we would use each grant type.

You learned about JWT tokens and federated security.

We discussed content-based attacks, and learned about the threat protection policies that can be used to protect against these attacks.

And you learned about Apigee features that can limit internal access to features and mask data for the users of the Apigee Management UI.

Next, we recommend you continue with the next course in this series, API Development and Operations on Google Cloud's Apigee API Platform.

In the API Development and Operations course, you'll learn about API mediation, traffic management, caching, and fault handling, and add these to your retail API proxy.

You will also create a developer portal, and publish your API product for use by app

developers.

We will learn about message logging and analytics, extensions, CI/CD, and deployment options for Apigee.

We hope to see you in the next course!