ICE Security Guide

Product guide for prerelease

Document version 1841, produced on Friday, September 06, 2024.

main 90adc8bf40040649230176bbdd465f6261a2d8e0

Contents

1 Introduction

This document covers security in Instant Connect Enterprise (ICE) and the Engage engine.

2 Technology, Regulatory Items, and Disclaimer

This section identifies the security technology used by Instant Connect Enterprise and the Engage engine. It also covers the regulatory restrictions and disclaimers relevant to ICE.

2.1 Technology

All components use the WolfSSL library, resulting in a lightweight, high-performance implementation.

The WolfSSL library is statically linked with ICE software. It is not dependent on the security features offered by any particular OS or hardware platform. The WolfSSL library uses the same crypto library across all platforms, providing ICE with a single, consistent, interoperable platform for security. And, because the WolfSSL library is independent of the operating system, regulatory considerations requiring inspection do not need to evaluate any particular individual platform.

> [!NOTE]
>
> ICE code is **FIPS-compliant**. ICE does not build WolfSSL with the FIPS140-2 “canister” that provides a **FIPS-validated** crypto layer.

2.2 Regulatory

Export and import of encryption technologies is a sensitive topic and varies in complexity across a range of use-cases, source of export, and target of import. Instant Connect does not sell or license its technologies to end-customers. Instant Connect licenses to OEM partners that incorporate Instant Connect technologies into their products, which are then sold/licensed to end-users. As a result, Instant Connect does not provide details of compliance with export or import regulations. It is the responsibility of OEM partners to seek the necessary approvals or exclusions on a case-by-case or product-by-product basis.

Instant Connect is open to inspection of its source code by authorized organizations, agencies, and individuals for compliance purposes. Requests are taken on a case-by-case basis. Please contact su pport@rallytac.com for assistance in this regard.

2.3 Disclaimer

Please remember that export/import of and/or use of strong cryptography software, providing cryptography hooks, or even just communicating technical details about cryptography software, is illegal in some parts of the world. When importing Instant Connect software to your country, re-distributing it from there, or even just emailing technical suggestions or software sources to Instant Connect or others: **You are strongly advised to pay close attention to any laws or regulations which apply to you**.

Rally Tactical Systems, Inc. is not liable for any violations made by users of its software.

3 Encryption

This section describes how the Engage engine employs different encryption algorithms.

3.1 Symmetric Encryption

All symmetric encryption in Engage is performed using the AES (Advanced Encryption Standard) algorithm with 256-bit keys operating in CBC (Cipher Block Chaining) mode.

3.2 Asymmetric Encryption

Asymmetric encryption is used in Engage during the TLS encryption setup phase between a client application and a Rallypoint. The algorithm used in this phase uses a cipher that the Rallypoint and client agree upon based on the X.509 certificates they present to each during the handshake.

3.3 Symmetric Key Derivation

Encryption keys are never stored or transmitted by Engage. Keys are algorithmically derived using the NIST-approved PBKDF2 algorithm. Engage takes the incoming passphrase/password/PIN (we refer to this as "**Baseline Key Material**" or simply "**BKM**") provided by the user application, then combines it with a 128-bit salt, and then performs 15,000 iterations of the algorithm (NIST recommends 10,000 iterations) to arrive at the derived, 256-bit, key.

3.3.1 Baseline Key Material is binary!

Engage is not limited to only printable characters for baseline key material. Rather, the software supports the full 256-bit range of binary data for BKM. For example, many encryption systems utilize passwords, passphrases, or PIN codes. This generally means that they need to be human-readable in some form and, therefore, printable. As Engage is not interacting directly with humans in this regard, the application using the Engage Engine can use any method appropriate to its use-case to obtain BKM from a user or from non-human entities such as provisioning systems.

3.4 Baseline Key Material Transport

Transportation of BKM is the domain of the application using the Engage Engine. For example, in implementations where human-processable data (e.g., passwords, passphrases, numbers) are involved, that information can be conveyed through out-of-band means (e.g., one-on-one interactions between team members, secured emails). Some implementations choose to use more exotic transport methods, e.g., encrypted QR codes, where the QR code itself is protected by a password known only to those using the QR code.

3.5 Traffic Encryption

Channels (or “groups”, as we refer to them) are encrypted using AES256-CBC, as described above. When the traffic is conveyed over UDP, the entire UDP payload is encrypted. This means that an attacker has no reliable way to know if the traffic is a known format (such as RTP) or a custom format implemented by the Engage Engine or the application using the Engine (in the case of “**raw**” group types). Also, even if the attacker assumes correctly that the payload is a standards-based format (such as RTP), since even the RTP headers are encrypted and the entire payload was preceded by an initialization vector produced uniquely for each packet; the task of cryptanalysis is exponentially more computationally expensive.

When traffic is conveyed over TCP, which is the case for client connections to Rallypoints and peer connections between Rallypoints operating in a mesh, that traffic is secured with TLS 1.3. This TLS-provided encryption is in addition to whatever encryption is already present for a group. Therefore, if the traffic for a group is encrypted, Engage will always encrypt it and, when that traffic then flows through TLS, the traffic is encrypted once again.

Engage generally views all traffic as packets conveyed over UDP. Even if the packets are conveyed over TCP, they are still treated by Engage as UDP (atomically). Essentially, Engage views TLS connections as secured tunnels (not unlike VPN connections) over which regular UDP is conveyed as a TCP stream.

4 X.509 Certificates

Engage uses X.509 certificates extensively for purposes of encryption, authentication, verification, and data creation. At minimum, an Engage Engine has a default certificate that it uses for all these purposes, but it is possible to use individual certificates for specialized purposes, e.g., connections to different Rallypoints (both within and external to an enterprise) can use different certificates.

4.1 Certificate Storage

When configuration (in the form of JSON) is provided by an application to the Engage Engine, the various certificates to be used may be passed directly in the certificate as PEM-encoded text. However, this certificate data often includes the private key with the certificate. Applications, or the platforms they execute on, may not always meet government or enterprise stringent data protection guidelines. To deal with this, Engage offers a set of certificate management APIs that secure certificates and private keys in an encrypted storage container known as a **certificate store**. This certificate store is typically physically implemented as an operating system file, which may be stored securely by the underlying operating system. That OS-provided secure storage notwithstanding, Engage encrypts this file following a sophisticated security algorithm where the encryption key (AES256-CBC) for the file is comprised of three components:

- An application-provided (binary) password/passphrase/PIN
- A 512-bit algorithmically-generated salt contained within the Engage Engine
- A 128-bit initialization vector embedded into the certificate store itself

Also, the initialization vector changes every time the file is modified by Engage, effectively creating a dynamic keying material baseline upon every store modification operation.

To aid in protecting against unauthorized certificate store modification, Engage embeds a SHA-256 digest in the certificate store and also validates each element in the store every time that store is opened. Therefore, for an adversary to obtain access to the content of the store - or to modify it in any way, they would have to know the application-defined passphrase, the algorithmically-generated Engage-provided salt, the IV inside the certificate store, and (finally) understand the method in which the encryption key is generated.

Each certificate (and its private key if applicable), is de-referenced with an application-provided identifier which is then used within the JSON configuration to “point” to the certificate. For example, JSON containing the actual PEM certificate could be represented as:

```
"security":{
```

```
  "certificate":"-----BEGIN CERTIFICATE-----\
     nMIICLDCCAdKgAwIBAgIBADAKBggqhkjOPQQDAjB9MQswCQYDVQQGEwJCRTEPMA0G\n
     .... ",
  "certificateKey":"-----BEGIN EC PARAMETERS-----\nBgUrgQQAIw==\n-----END
     EC PARAMETERS-----\n-----BEGIN EC PRIVATE KEY-----\nMIHbAgEBBEGXac
     ... "
}
```

Or, a certificate store can be used that contains a certificate (and private key) identified with an application-defined value, such as "**PrimaryX509**". In this case, the JSON would look as follows:

```
"security":{
  "certificate":"@certstore://PrimaryX509",
  "certificateKey":"@certstore://PrimaryX509"
}
```

In this case, Engage will extract the certificate and key for "**PrimaryX509**" from the active certificate store and insert it directly into its internal configuration without the certificate or key ever being exposed - even to the application itself.

4.1.1 Defaulting Using Certificate Store Tags

There is another way in which Engage can be instructed to use certificates from the certificate store without specifying anything in either the Engine's policy or the group configuration, i.e., a way to forgo the security JSON element altogether (as well as certificate-related details in Rallypoint configurations). This is done through **tagging** of elements in the certificate store.

Tagging is simply the process of attaching a well-known identifier (or set of identifiers) to a certificate element in the certificate store and then forcing Engage to retrieve elements from the active store using these well-known identifiers. If you leave a security element empty, Engage will look in the active certificate store for the first element it finds with the appropriate tag and use it's contents.

For example: Let's say we have a security element which is blank (or simply not in the configuration at all) as below:

```
"security":{
   "certificate":"",
   "certificateKey":""
}
```

When Engage encounters this situation, it will look in the active certificate store for the first element that has the tag `enginedefault`. If it finds that element, it uses it, but the onus is on the certificate store to have the elements correctly tagged. If they're not present, the above will result in Engage not finding any security-related information.

There are two tags that the Engage Engine will look for:

- `-enginedefault`: A certificate, including private key, used for general X.509 operations.
- `-cadefault`: A CA certificate.

On a Rallypoint, the same logic applies, but with a slightly different name:

- `-rpdefault`: A certificate, including private key, used for general X.509 operations.
- `-cadefault`: A CA certificate.

With a security configuration as shown in the example above, the certificate store needs it's elements tagged and Engage instructed to use that store. For example, if we use `ecstool` (described below) to configure a certificate store with the correct tagging, we'd have something like this:

```
./ecstool all-rts-certs.certstore list
--------------------------------------------------------------------------------

Engage Certstore Tool version 1.191.9028 [RELEASE] for darwin_x64
Copyright (c) 2019 Rally Tactical Systems, Inc.
--------------------------------------------------------------------------------

Build time: Nov 25 2020 @ 14:54:18

id.................: {415ffe01-efe5-4592-b984-c839b8fb074e}
fileName...........: all-rts-certs.certstore
version............: 1
3 CERTIFICATES
    rtsCA  (CERTIFICATE ONLY) [-cadefault]
    rtsFactoryDefaultRpSrv  (CERTIFICATE  + PRIVATE KEY) [-rpdefault]
    rtsFactoryDefaultEngage  (CERTIFICATE  + PRIVATE KEY) [-enginedefault]
```

ICE can use this certificate store for Engage Engines, as well as Rallypoints, because, in this case, the store contains all the certificates.

- Notice how `rtsCA` is tagged as `-cadefault`, telling Engage and Rallypoints that `rtsCA` is the default CA certificate to use.
- `rtsFactoryDefaultRpSrv` is tagged as `-rpdefault`, which tells Rallypoints using the certificate store that `rtsFactoryDefaultRpSrv` is the default certificate to use.
- Finally, `rtsFactoryDefaultEngage` is tagged as `-enginedefault`, which tells Engage Engines to use `rtsFactoryDefaultEngage` as the default certificate.

This capability is purposefully simple in nature and designed for situations where using defaults is sufficient for normal operation. For more sophisticated environments, such as using multiple and/or different certificates, CAs, and so forth; defaulting security is not sufficient and you will need to specify security-related information.

4.1.2 Certificate Store Loading & Distribution

Depending on the use-case of the application using the Engage Engine, you can choose to implement importing of certificates from within the application, or to use the ecstool command-line utility, which uses the Engage Engine to manage certificate store files. Those files are then distributed to end-user devices, servers, etc using the customer's preferred distribution mechanism, e.g., an MDM, ActiveDirectory, LDAP, dedicated servers, clouds, file copying, shared drive access. Even public means of distribution, e.g., email, social media, public drives, may be used, because the certificate store has self-contained security accessible only to Engage Engines where the above-mentioned constraints are satisfied.

5 Data Signing and Message Authentication

Signing and verification of messages and other secured data elements in Engage is driven by certificates, as well. Engage uses message-signing for an additional level of authentication between clients and Rallypoints, as well as signing of timeline events for anti-tampering purposes such as chain-of-evidence preservation. In both cases, Engages uses the Elliptic Curve Digital Signature Algorithm (EDCSA) to both sign and verify signed data.

In the case of TLS connections, Engage implements additional guards against attacks by using a message-signing strategy that incorporates keying material negotiated by the session's TLS handshake, and which is unique to that session. This is done in compliance with RFC 5705 whereby a portion of the session's unique key material is incorporated into session management messages and then signed with ECDSA.

For timeline signing, each timeline file (stored as a JSON or RIFF file depending on content) incorporates the public portion of the signing certificate in PEM format along with the ECDSA signature for the content based on the certificate public/private key pair.

6 Deploying Self-Signed Certificates

This section provides instructions for using self-signed certificates for ICE Desktop and Mobile users.

6.1 Self-sign certificates for ICE Desktop login

The ICE Desktop client supports the use of self-signed certificates by applying to the security context a root CA certificate file that was installed to the Windows certificate store. The root CA certificate must be in the .crt format.

> **Note:** Other common certificate formats (e.g., .cer) are not supported. The entire certificate trust chain must be present in the root CA file, so, depending on how the server identity certificate was setup, one or more intermediate CA certificates may also be required, in addition to the root CA certificate.

1. Download the root CA certificate file.
2. Right-click on the file and select 'Install Certificates'.
3. Select 'Open'.
4. Select 'Install for current users'.
5. Place the file in the 'Trusted Root Certification Authorities' store.
6. Navigate to the certificate store: `Certificates` > `Trusted Root Certification Authorities` > `Certificates`.
7. Verify the certificate file is there.
8. Launch ICE Desktop. Log out, if necessary, then log in.

When opened, the desktop client now queries the installed root CA certificates (system and user) and applies them to the security context. If the root CA certificate and all intermediates are present, the client will successfully connect to the ICE Server.

7 Self-sign certificates for ICE Mobile login

The ICE Mobile app (iOS, Android) supports the use of self-sign certificates by applying to the security context a root CA certificate file that was installed to the mobile device's OS or saved to mobile app's document directory. The root CA certificate must be in a supported format.

- iOS: .pem
- Android: .crt, .pem

> **Note:** Other common certificate formats (e.g., .cer) are not supported. The entire certificate trust chain must be present in the root CA file, so, depending on how the server identity certificate was setup, one or more intermediate CA certificates may also be required, in addition to the root CA certificate.

7.1 Convert .crt file to .pem file

Run the following command to convert a .crt file to a .pem file:

```
openssl x509 -in root.crt -out root.pem
```

7.2 Install the root CA certificate to the OS

The objective of this process is to install the root CA certificate to the device's OS. The examples used below are for reference and based on an Apple iPhone running iOS 14 and a Samsung phone running Android 10, respectively. Your device menus and settings may differ significantly, but the objective remains the same.

For iOS:

1. Save the root CA certificate file to the native Files app (e.g., from an email attachment): `Files > On my iPhone`.
2. From the 'Files' app, select the file so a 'Profile downloaded' message displays.
3. To install the certificate, open the native 'Settings' app and select 'Profile downloaded'.
4. To enable the certificate, navigate to `Settings > General > About > Certificate Trust Settings`.
5. Launch ICE Mobile. Log out, if necessary, then log in.

For Android:

1. Save the root CA certificate file to the device's storage.
2. Go to `Settings > Biometrics and security > Other security settings > Install from device storage`.
3. Select the file to install it.
4. Go to `Settings > Biometrics and security > Other security settings > View security certificates` to confirm the certificate is there.
5. Launch ICE Mobile. Log out, if necessary, then log in.

When opened, the app now queries the device's installed root CA certificates (system and user) and applies them to the security context. If the root CA certificate and all intermediates are present, the app will successfully connect to the ICE Server.

7.3 Load the root CA certificate to the ICE Mobile app's document directory

The objective of this process is to save the root CA certificate to the ICE Mobile app's document directory. The app references that file to support self-sign certification. The app will not find the root CA certificate unless the .pem file is placed in the correct document directory. The examples used below are for reference and based on an Apple iPhone running iOS 14 and a Samsung phone running Android 10, respectively. Your device menus and settings may differ significantly, but the objective remains the same.

1. Save the root CA certificate file to the ICE Mobile app's document directory. You can confirm the directory path is correct by looking for the presence of a README.txt document.

   ***For iOS*:** The directory is at: `Files` > `On my iPhone` > `ICE Mobile`.

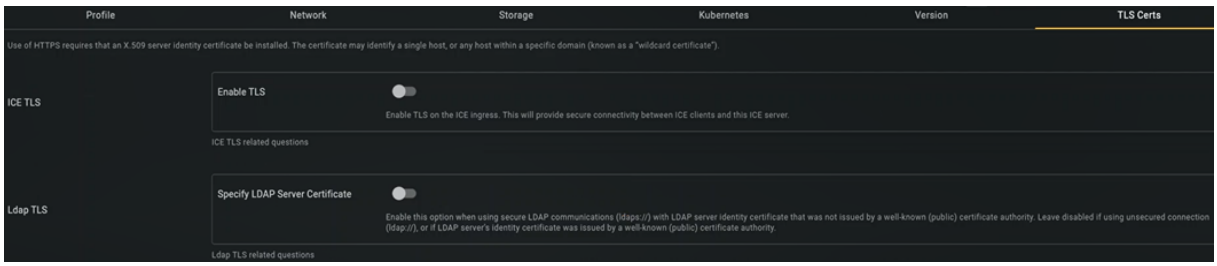
   1. Save the file to the native 'Files' app, e.g., from an email attachment.
   2. Open the 'Files' app.
   3. Long press on the root CA certificate file.
   4. From the resulting menu, select 'ICE Mobile', then select 'Copy'.

   ***For Android*:** The directory is typically at: `Internal storage` > `Android` > `data` > `com.dillonkane.ice.flutter` > `files`.

2. Open the ICE Mobile app. Log out, if necessary, then log in.

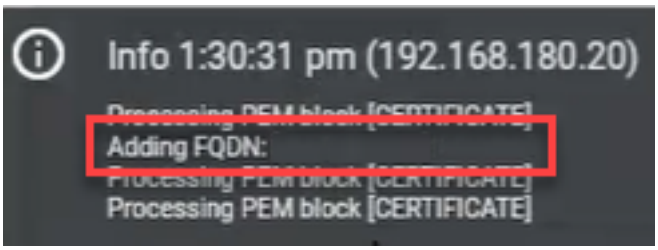
When opened, the app now checks that file directory and applies the root CA certificates located there to the security context. If the root CA certificate and all intermediates are present, the app will successfully connect to the ICE Server.

7.4 TLS Certs



> **Note:** Certificates must be valid for more than 60 days, at least. Open 'Status' > 'Client TLS Certificates' to check the validity of loaded certificates.
>
> **Note:** If a certificate contains an IP address prefaced by `http` or `https`, then do *NOT* list that IP address in the 'Cluster Ingress Hostname' field on the 'Server' screen.

- Enable TLS = If enabled, the following fields appear:
  - Site Certificate Private Key = Enter the private (host) key. X.509 certificate in PEM format.
  - Site Certificate Chain = Enter the public key. X.509 certificate in PEM format. The certificate chain is as follows: server certificate > intermediate certificate(s) (if any) > certificate authority (CA). When using the 'File upload' option, the certificate chain must be uploaded as a single file.

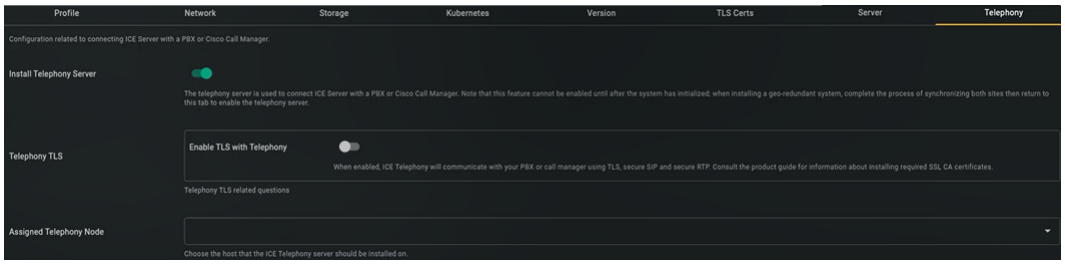
> **Note:** After entering both the private and public keys, an 'Adding FQDN: [XXX]' notification displays. Open the 'Notifications' screen and review the 'Adding FQDN' line of that notification to verify the domain is correct.
>
> 
>
>
> **Note:** The ICE Desktop web client (enabled on the 'Server' screen) requires certificates be entered here, otherwise, navigating to the web client address results in the following notification: 'Instant Connect is not available in this browser context. Contact your system administrator for more information.

- Specify LDAP Server Certificate = If enabled, the following field appears:
  - LDAP Certificate = Enter the LDAP server's identity certificate in PEM format.
- Apply = **Check the 'Status' dropdown and wait for 'Node Status' to turn green *before* proceeding to the next screen.**

  For air gap: Also wait for 'Air-gapped Extraction Status' to turn green *before* proceeding to the next screen.

  > **Note:** If you advance to the next screen *before* 'Node Status' turns green, an error message may display. If this occurs, wait for 'Node Status' to turn green, and the error will resolve itself.

7.5 Telephony Certs



- Install Telephony Server = Do *NOT* enable unless this feature is included in your ICE product license. If enabled, an 'Assigned node XXX for telephony' notification displays, also the following fields appear:

  - Enable TLS with Telephony = If enabled, the following fields appear:
    * Telephony Certificate Private Key = Required.
    * Telephony Certificate Chain = Required.

- Assigned Telephony Node = Prepopulated with the ICE host name, which is pulled from the value entered in the 'Node Name' field on the 'Network' screen.
- SIP Ports Specification = Only displays if the 'Advanced Questions' tool is enabled.
  - SIP TCP Port
  - SIP UDP Port
  - SIP TLS Port
  - SIP TCP6 Port
  - SIP UDP6 Port
  - SIP TLS6 Port
- Apply = Select to apply these values and proceed to the next screen.

7.6 Geo-Redundancy Certs

7.6.1 TLS Certs (DC2)

- Any certificates entered for DC1 still display here. If DC2 has different certificates, then enter those certificates instead.
- Verify the configurations on the screen are correct, then **check the 'Status' dropdown and wait for 'Node Status' to turn green *before* proceeding to the next screen.**

> **Note:** If you advance to the next screen *before* 'Node Status' turns green, an error message may display. If this occurs, wait for 'Node Status' to turn green, and the error will resolve itself.

7.7 Vector configuration for ICE Server

1. Add a section in `/etc/vector/vector.toml`:

```
#
# In this example, we are collecting all kubernetes logs
# from ICE OS VM 192.168.0.198
#
# "sources.K8S_CLUSTER" : User-friendly name of ICE Server
#
# "sinks.DATA_SOURCE_NAME"   : User-friendly name of data source
   output
```

```
# "inputs"                    : It should point to "sources.K8S_CLUSTER
    "
# "path"                      : Location of sink's output file

[sources.ice_192_168_0_198_vector]
type = "vector"
address = "0.0.0.0:9000"

[sinks.ice_192_168_0_198_vector_out]
type = "file"
inputs = ["ice_192_168_0_198_vector"]
path = "/var/log/vector/k8s/ice_192_168_0_198.log"
  encoding.codec = "raw_message"
```

2. For TLS:

    1. Add the following in the `source` section:

    ```
    tls.enabled = true
    tls.ca_file = "/etc/vector/tls/tls.vector.ca_file"
    tls.crt_file = "/etc/vector/tls/tls.vector.crt_file"
    tls.key_file = "/etc/vector/tls/tls.vector.key_file"
    ```

    2. `tls.ca_file` should point to a .PEM file containing the root certificate and the intermediate certificate, if applicable.
    3. `tls.crt_file` should point to a .PEM file containing the Vector server certificate.
    4. `tls.key_file` should point to a private key file corresponding to `tls.crt_file`.

3. Start/restart Vector:

```
sudo systemctl restart vector
```

4. You should see ICE Server logs streaming to: `/var/log/vector/k8s/ice_192_168_0_198.log`