Troubleshoot Finesse Error "SSLPeerUnverifiedException" for Gadgets Hosted on CA-Signed Servers

Contents

Introduction

This document describes the steps to troubleshoot the scenario where a Certificate Authority (CA)-signed certificate chain is uploaded to Finesse for an external web server that hosts a gadget but the gadget fails to load when you log in to Finesse and you see the error "SSLPeerUnverifiedException".

Contributed by Gino Schweinsberger, Cisco TAC Engineer.

Prerequisites

Requirements

Cisco recommends that you have knowledge of these topics:

- SSL Certificates
- Finesse administration
- Windows Server administration
- Packet capture analysis with Wireshark

Components Used

The information in this document is based on these software versions:

- Unified Contact Center Express (UCCX) 11.X
- Finesse 11.X

The information in this document was created from the devices in a specific lab environment. All of

the devices used in this document started with a cleared (default) configuration. If your network is live, ensure that you understand the potential impact of any command.

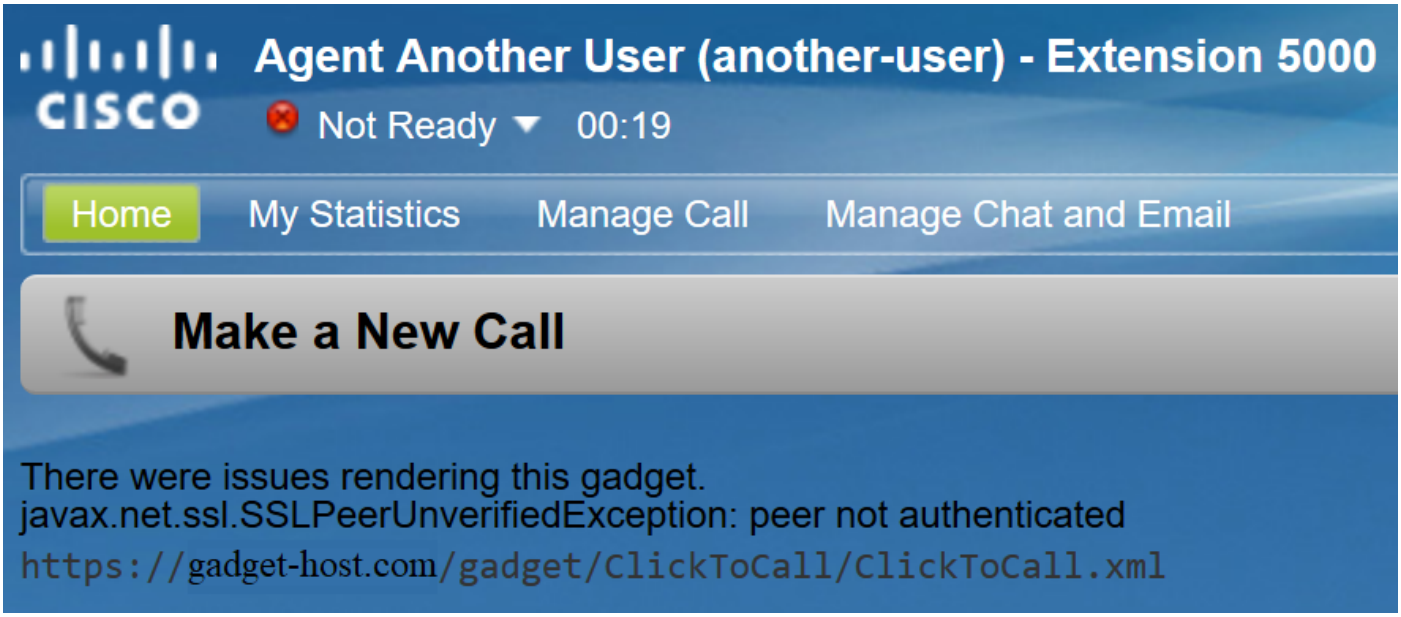
Background Information

These are the conditions for the error to occur:

- Assume certificate trust chain is uploaded to Finesse
- Ensure that the correct servers/services were restarted
- Assume the gadget has been added to the Finesse layout with an HTTPS URL and that the URL is reachable

This is the error observed when agent logs in to Finesse:

"There were issues rendering this gadget. javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated"



Problems

Scenario 1: The Hosting server negotiates unsecure TLS

When Finesse Server makes a connection request to the Hosting server, Finesse Tomcat advertises a list of encryption ciphers which it supports.

Some ciphers are not supported due to security vulnerabilities,

If the Hosting server selects either of these ciphers, the connection is refused:

- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA

These ciphers are known to use weak ephemeral Diffie-Hellman keys when it negotiates the connection, and the Logjam vulnerability makes these a bad choice for TLS connections.

Follow the TLS handshake process in a packet capture to see which cipher is negotiated.

1. Finesse presents its list of supported ciphers in the **Client Hello** step:

```
▾ TLSv1 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 67
  ▾ Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 63
      Version: TLS 1.0 (0x0301)
    › Random: 5cacb293b5efdb4cf1bb34464d7de9f5060b00a9beb81d29…
      Session ID Length: 0
      Cipher Suites Length: 24
    ▾ Cipher Suites (12 suites)
        Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
        Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
        Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
        Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
        Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
        Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
        Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
        Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)
        Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
        Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
      Compression Methods Length: 1
    › Compression Methods (1 method)
```

2. For this connection **TLS_DHE_RSA_WITH_AES_256_CBC_SHA** was selected by the Hosting server during the **Server Hello** step because that is higher on its list of preferred ciphers.

```
TLSv1 Record Layer: Handshake Protocol: Multiple Handshake Messages
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 2557
  Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 77
      Version: TLS 1.0 (0x0301)
    > Random: 5cacb292c4d7183627f620a066f9b6ce6460dcb849b59cae…
      Session ID Length: 32
      Session ID: 4c290000ce66098cc994a33e193b0da1244cb9f083f69c26…
      Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
      Compression Method: null (0)
      Extensions Length: 5
    > Extension: renegotiation_info (len=1)
  > Handshake Protocol: Certificate
  Handshake Protocol: Server Key Exchange
      Handshake Type: Server Key Exchange (12)
      Length: 1032
    > Diffie-Hellman Server Params
  Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

3. Finesse sends a Fatal alert and ends the connection:

```
TLSv1 Record Layer: Alert (Level: Fatal, Description: Internal Error)
    Content Type: Alert (21)
    Version: TLS 1.0 (0x0301)
    Length: 2
  > Alert Message
```

Solution

In order to prevent the use of these ciphers, the Hosting server must be configured to give these a low priority, or they must be removed from the list of available ciphers completely. This can be done on a Windows Server with the Windows Group Policy editor (gpedit.msc).

Note: For more details on the effects of Logjam in Finesse and the use of gpedit, check:

Scenario 2: The certificate has an unsupported signing algorithm

Windows Server certificate authorities can use newer signature standards to sign certificates. Even it offers greater security than SHA, adoption of these standards outside of Microsoft products

is low and administrators are likely to run into interoperability issues.

Finesse Tomcat relies on the SunMSCAPI security provider from Java to enable support for the various signature algorithms and cryptographic functions used by Microsoft. All current versions of Java (1.7, 1.8, and 1.9) support only these signature algorithms:

- MD5withRSA
- MD2withRSA
- NONEwithRSA
- SHA1withRSA
- SHA256withRSA
- SHA384withRSA
- SHA512withRSA

It is a good idea to check the version of Java that runs on the Finesse server to confirm which algorithms are supported in that version. The version can be checked from root access with this command: **java -version**

```
Using username "root".
Last login: Tue Apr 16 13:11:00 2019 from
[root@uccxl2pub ~]# java -version
java version "1.7.0_181"
OpenJDK Runtime Environment (rhel-2.6.14.8.el6_9-i386 u181-b00)
OpenJDK Server VM (build 24.181-b00, mixed mode)
[root@uccxl2pub ~]#
```

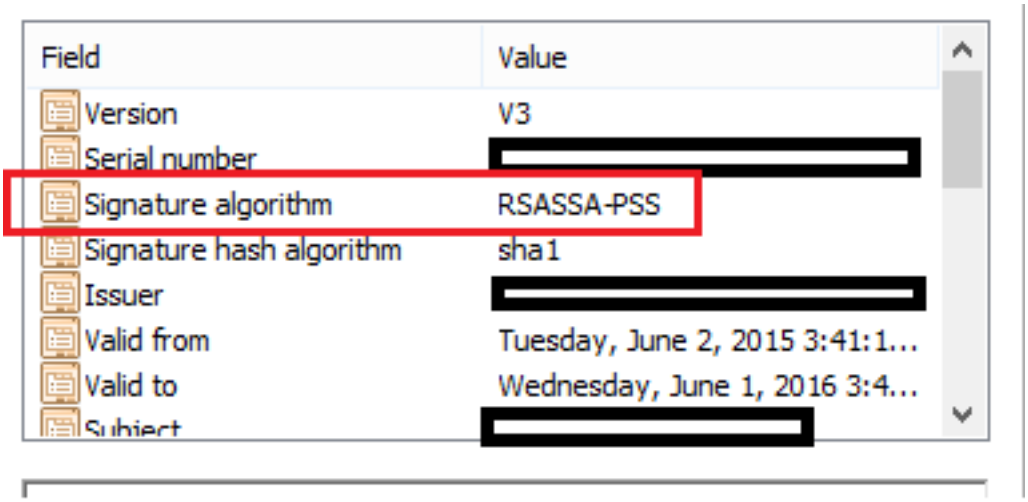
> **Note**: For more details on the Java SunMSCAPI provider refer to https://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SunMSCAPI

If a certificate is provided with a signature other than those listed above, Finesse is not able to use the certificate to create a TLS connection to the Hosting server. This includes certificates that are signed with a supported signature type but were issued by certificate authorities that have their own intermediate and root certificates signed with something else.

If you look at a packet capture, Finesse closes the connection with a "Fatal alert: Certificate Unknown" error, as shown in the image.

```
Secure Sockets Layer
⊟ TLSv1.2 Record Layer: Alert (Level: Fatal, Description: Certificate Unknown)
    Content Type: Alert (21)
    Version: TLS 1.2 (0x0303)
    Length: 2
  ⊟ Alert Message
      Level: Fatal (2)
      Description: Certificate Unknown (46)
```

At this point is is necessary to check the certificates presented by the Hosting server and look for unsupported signature algorithms. It is common to see **RSASSA-PSS** as the problematic signature algorithm:



If any certificate in the chain is signed with RSASSA-PSS, the connection fails. In this case the packet capture shows that the Root CA uses RSASSA-PSS for its own certificate:

```
Certificates (3906 bytes)
    Certificate Length: 1728
  Certificate: 308206bc308205a4a003020102021374000000243b805da9... (id-at-commonName=[redacted]
    signedCertificate
    algorithmIdentifier (sha256WithRSAEncryption)
      Padding: 0
      encrypted: e6230df257be9d34c0f57bc2f88c081c4186aaad092c8155...
    Certificate Length: 1114
  Certificate: 308204563082033ea003020102021316000000a93cd17d6... (id-at-commonName=[redacted]Issuing Authority[redacted]
    signedCertificate
    algorithmIdentifier (sha256WithRSAEncryption)
      Padding: 0
      encrypted: 889be6a1125c758cd0009b392d3b90a69b64546dcee09c84...
    Certificate Length: 1055
  Certificate: 3082041b308202cfa00302010202107b70dbb7c2760da74f... (id-at-commonName=[redacted]Root CA [redacted]
    signedCertificate
    algorithmIdentifier (id-RSASSA-PSS)
      Algorithm Id: 1.2.840.113549.1.1.10 (id-RSASSA-PSS)
    RSASSA-PSS-params
      Padding: 0
      encrypted: d8e9151adc76b4e55f9277fce916613ce26199e3b50dcb54...
```

Solution

In order to resolve this issue, a new certificate must be issued from a CA provider that only uses one of the supported SunMSCAPI signature types listed throughout the entire certificate chain as explained before.

> **Note**: For more details on the RSASSA-PSS signature algorithm, see https://pkisolutions.com/pkcs1v2-1rsassa-pss/

> **Note**: This issue is tracked in the defect CSCve79330