Nubeva SKI Documentation

Product

May 07, 2021

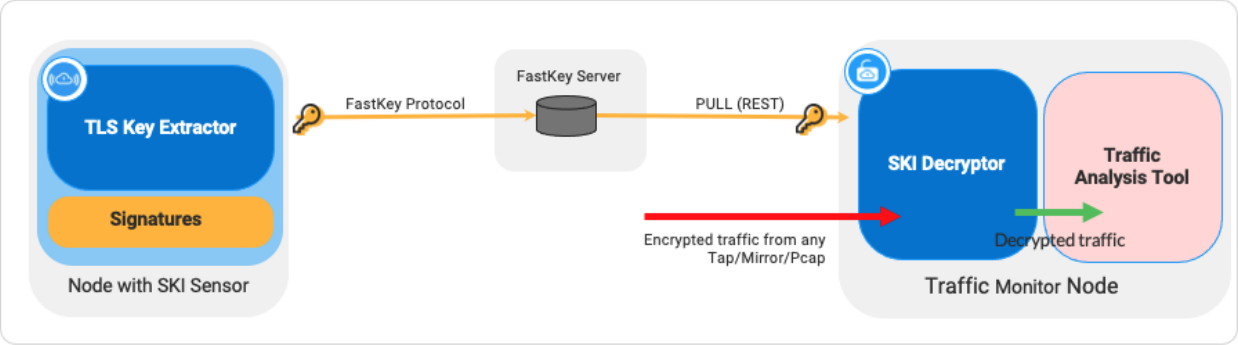
Evaluation and Testing Docs

CHAPTER 1

SKI Overview

The increasing use of encryption both across the public Internet and within private enterprise networks is well documented. TLS 1.2 PFS and TLS 1.3 make it much harder for enterprises to monitor and inspect traffic for malware, data breaches, and malicious activity, as well as troubleshoot availability or performance issues on the network. Passive decryption is not possible. Inline decryption increases latency significantly increasing costs of required SLAs, and cannot inspect sessions that use pinned certificates. These concerns are shared by manufacturers of out-of-band and in-band network security and monitoring solutions and their customers. Disabling PFS in TLS 1.2 is no longer an unacceptable compromise. Therefore there is a growing need to enhance TLS visibility for TLS 1.2 PFS and 1.3 with continued support for legacy protocols in cloud networks and private data centers.

Nubeva's Symmetric Key Intercept (SKI) solves these problems. Nubeva SKI uses sensors that discover ephemeral session secrets on active TLS client and TLS servers without certificates, PKI, man-in-the-middle, session termination, or session replay. Session secrets are detected in memory using signatures specific to the libraries or DLLs used by the application. SKI is read-only and does not require modifications to applications. SKI works for new TLS sessions as well as TLS session resumption. Discovered keys are sent to key depots or directly to Nubeva's decryptors or 3rd party decryptors that implement Nubeva's low-latency protocol.



The SKI architecture is comprised of `Signatures`, `SKI Sensors`, `FastSKI Protocol`, `SKI Decrypt Library` and `SKI Decryptor`.

1.1 Signatures

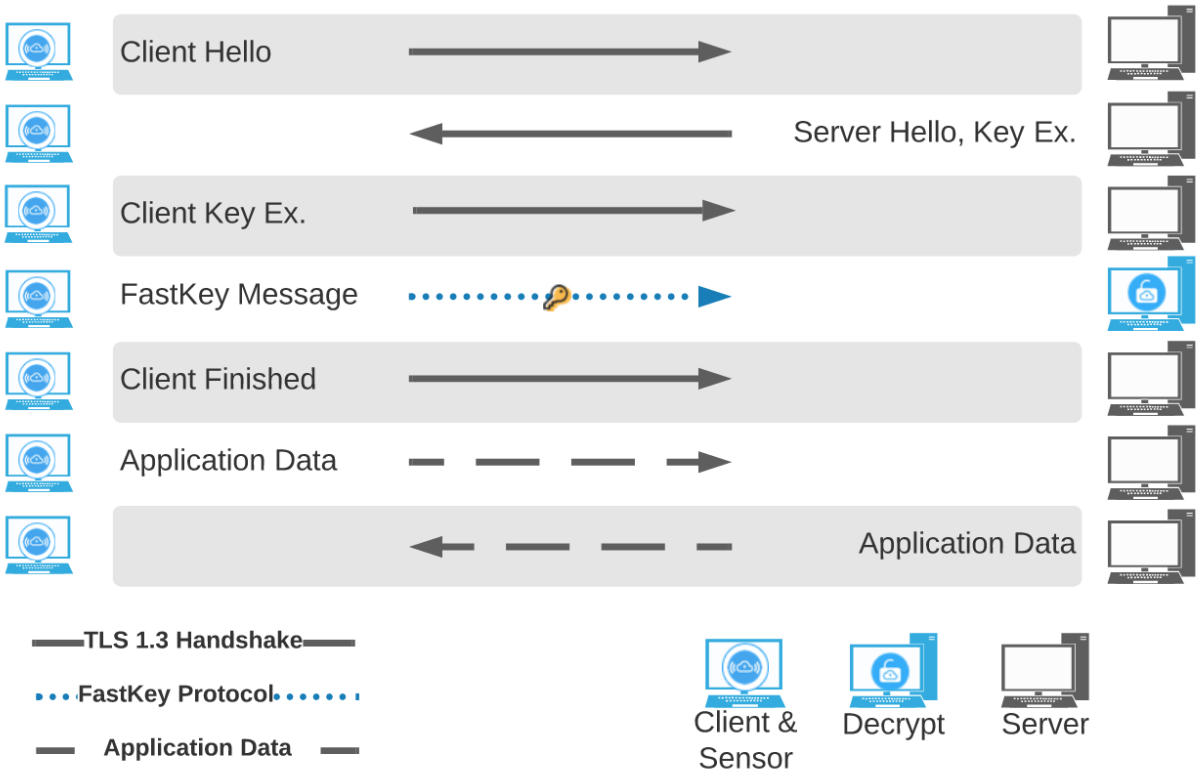
TLS libraries create session secrets during the TLS session handshake. Each TLS library creates session keys in a unique way in the process memory space. Nubeva's core IP finds the session keys using an algorithm designed specifically for each library or application and outputs a set of instructions called a `signature`. Such signatures are then used to gain access to the session keys at runtime in an extremely efficient manner. TLS signatures are compiled by Nubeva's back-end and distributed in binary form to licensed customers.

1.2 SKI Sensors

`SKI Sensors` are endpoint software that uses memory hooks and signatures to discover and extract session secrets from process memory. Signature based key extraction does not participate in TLS sessions, and does not require certificates. Therefore it is possible to extract keys from any session, including sessions that traditional methods have to bypass, such as sessions using pinned certificates or client certificates or sessions to 3rd party services where certificates are not available. SKI Sensors upload signature updates regularly from a licensed source.

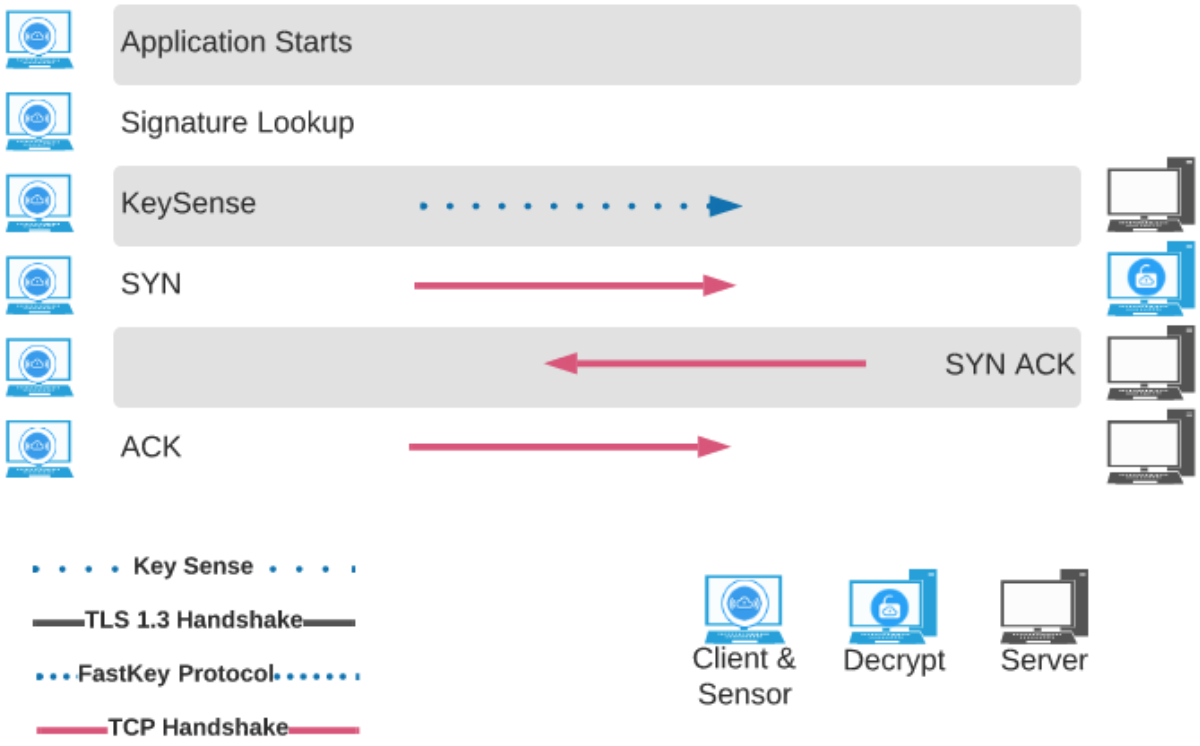
1.3 FastKey

SKI Sensors capture and send session secrets using Nubeva's `FastKey` *protocol*. Session keys are sent 200μs after the secrets are created, before handshakes are completed, and 500μs before the first encrypted application data is received for inspection. This assures that sessions keys are available for decryption before the first encrypted packet of a session arrives.



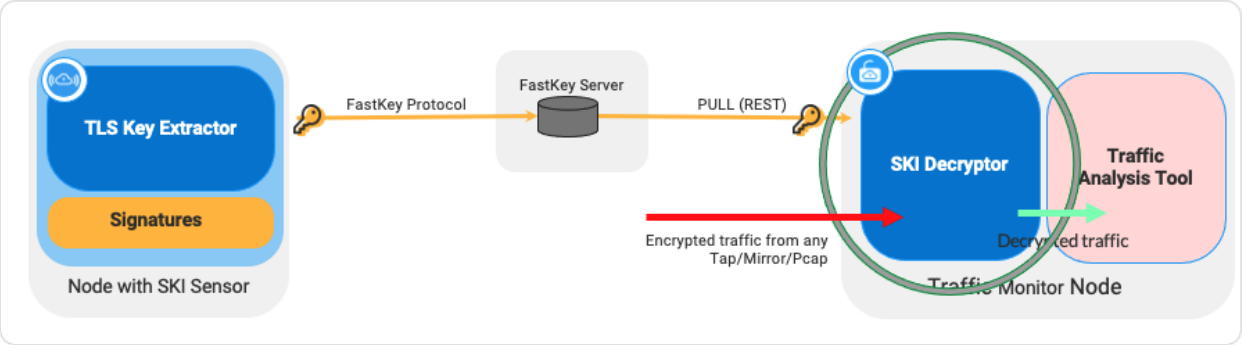
1.4 KeySense

In the rare event that an application uses a library does not have a signature, the SKI Sensor sends a `KeySense` indication to the decryption system telling the system not to expect a key for the current application. This indication is sent as soon as a TCP handshake is detected. KeySense operates before TLS handshakes begin, giving the inspection system time to select the decryption mechanism to use.



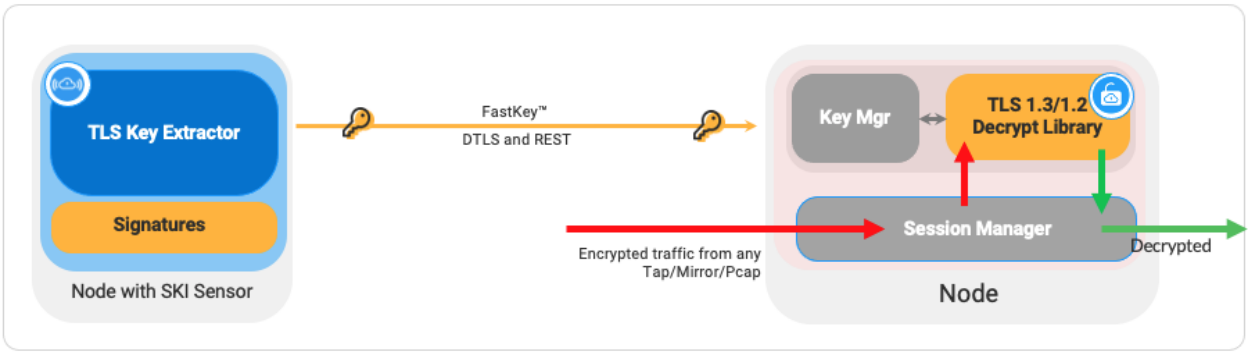
1.5 SKI Decryptors

Nubeva `SKI Decryptors` are a turnkey container solution that receives mirrored traffic and session keys from TLS sensors and exposes a standard network interface from which inspection software can read both decrypted and encrypted traffic. SKI Decryptors can be integrated with any deep packet inspection software that does not have decryption capabilities or cannot decrypt specific cyphers. Some of the supported open-source tools are Arkime (Moloch), Bro, ntop, Suricata, and Wireshark. Decryption capacity can be increased by scaling up and out using standard load balancing methods.



1.6 High-Performance Decryption Library

Nubeva also provides a high-performance `Decryption Library`, designed for applications that need high decryption throughput. The Decryption Library are integrated into the inspection software and operates independently of underlying TCP/IP stack customizations such as higher throughput optimizations using DPDK. The Decryption Library provides high decryption throughput upwards of 10 Gbps on a single core on standard hardware for TLS 1.3 AES ciphers. Multi-threading enables decryption at rates of 100Gbps and higher. The library is augmented by an optional `FastKey Buffer` library for receiving and storing keys in case the inspection tools do not receive and retrieve session keys directly.



1.7 Products

The table shows products available for evaluation and products which require a license.

Table 1: SKI Products Licensing Options

Product	Evaluation	Full License	Product Brief
Supported Signatures	Binary	Binary	
SKI Sensor	Binary	Binary and code	Product brief
SKI Decryptor	Binary	Binary and code	
SKI Decryption Library		Binary and reference code	Product brief
FastKey Buffer	Binary	Binary and reference code	
Key Buffer Example	Code	Code	

CHAPTER 2

System Requirements

Note: Nubeva SKI Sensors are supported on Linux - kernel 4.4 or above, Windows Server 2012, Server 2012 R2, Server 2016, Server 2019 and Windows 10, Kubernetes 1.13 and above and OpenShift 4.x.

2.1 Linux Systems

SKI Sensors extract key from any Linux version running a Linux Kernel version 4.4 or higher.

2.1.1 Supported Linux Kernel Versions

Kernel Version	Release Date
4.4	10 January 2016
4.5	13 March 2016
4.6	15 May 2016
4.7	24 July 2016
4.8	25 September 2016
4.9	11 December 2016
4.10	19 February 2017
4.11	30 April 2017
4.12	2 July 2017
4.13	3 September 2017
4.14	12 November 2017
4.15	28 January 2018
4.16	1 April 2018
4.17	3 June 2018
4.18	12 August 2018
4.19	22 October 2018
4.20	23 December 2018
5.1	5 May 2019
5.2	7 July 2019
5.3	15 September 2019
5.4	24 November 2019
5.5	26 January 2020
5.6	29 March 2020
5.7	31 May 2020
5.8	2 August 2020
5.9	11 October 2020
5.10	13 December 2020
5.11	14 February 2021

2.1.2 Supported Linux Operating Systems

The table lists commonly used Linux operating systems that use supported kernel versions.

Note: The table is not exhaustive. Key extraction is also supported on other Linux operating system that use one of the supported Linux kernel versions listed in the previous section.

OS	Version	Kernel
Ubuntu	16.04 LTS	4.4
	16.1	4.8
	17.04	4.1
	17.1	4.13
	18.04 LTS	4.15
	18.1	4.18
	19.04	5
	19.1	5.3

Continued on next page

Table 1 – continued from previous page

OS	Version	Kernel
	20.04 LTS	5.4
RHEL	8	4.18.0-80
	8.1	4.18.0-147
	8.2	4.18.0-193
	8.3	4.18.0-240
CentOS	8.0-1905	4.18.0-80
	8.1-1911	4.18.0-147
	8.2-2004	4-18-0-193
	8.3-2011	4.18-0-240
Debian	9	4.9
	9.1	4.9
	9.2	4.9
	9.3	4.9
	9.4	4.9
	9.5	4.9
	9.6	4.9
	9.7	4.9
	9.8	4.9
	9.9	4.9
	9.10	4.9
	9.11	4.9
	9.12	4.9
	9.13	4.9
	10	4.19
	10.1	4.19
	10.2	4.19
	10.3	4.19
	10.4	4.19
	10.5	4.19
	10.6	4.19
	10.7	4.19
	10.8	4.19
	10.9	4.19
Fedora	24	4.5
	25	4.8
	26	4.11
	27	4.13
	28	4.16
	29	4.18
	30	5
	31	5.3
	32	5.6
	33	5.8
AWS AMI	2016.03.1	4.4
	2016.03.2	4.4
	2016.03.3	4.4
	2016.09.1	4.4
	2017.03.1	4.9
	2017.09.1	4.9

Continued on next page

Table 1 – continued from previous page

OS	Version	Kernel
	2018.03	4.14

2.2 MS Windows Systems

2.2.1 Supported MS Windows versions

```
Windows Server 2012 September 2012, Foundation, Essentials, Standard, Datacenter
Windows Server 2012 R2 April 2014
Windows Server 2016 public release September 26 2016
Windows Server 2016 version 1709 October 17 2017
Windows Server 2016 version 1803 April 10 2018
Windows server 2019 version 1809 October 2 2018
Windows 10 version 1507 July 2015
Windows 10 version 1511 November 2015
Windows 10 version 1607 July 2016
Windows 10 version 1703 April 2017
Windows 10 version 1709 October 2017
Windows 10 version 1803 April 2018
Windows 10 version 1809 October 2018
Windows 10 version 1903 May 2019
Windows 10 version 1909 November 2019
Windows 10 version 2004 May 2020
Windows 10 version 20H2 October 2020
```

2.3 Container Orchestration Platforms

A container is a normal operating system process except that is isolated and has its own file system, networking, and process tree. *SKI Sensors* extract session keys from containers regardless of how these containers are run, whether using Docker on Linux, a container orchestration system like Kubernetes or OpenShift, or Docker Desktop.

2.3.1 Kubernetes

SKI Sensors can be deployed on all Kubernetes versions 1.13 or higher.

```
Version 1.13
Version 1.14
Version 1.15
Version 1.16
Version 1.17
Version 1.18
Version 1.19
Version 1.20
```

SKI Sensor containers may be run by Docker or CRI-O.

2.3.2 OpenShift

SKI Sensors can be deployed on OpenShift 4.x.

```
Openshift 4.0
Openshift 4.1
Openshift 4.2
Openshift 4.3
Openshift 4.4
Openshift 4.5
Openshift 4.6
Openshift 4.7
```

2.3.3 Docker Desktop

You can run SKI Sensors in Linux containers on Windows and MACOS if you have Docker Desktop installed. Docker Desktop is supported on MACOS version 10.10.13 (Yosemite) or higher (Catalina, Big Sur). Docker Desktop for Windows requires WSL (Windows Subsystem for Linux) which is a Linux compatibility layer for running Linux binary executables natively on Windows 10 and Windows Server 2019. WSL 2 was released in May 2019 and introduced a real Linux kernel. As such, Docker Desktop appears as a Linux system to SKI Sensors. A single sensor deployed on Docker Desktop on a MAC or Windows node will extract TLS session keys from all the containers running in Docker Desktop.

Since SKI sensors do not require CAs or certificates, running SKI sensors containers in docker desktop environments, enables MITM configurations to inspect TLS traffic. Such TLS inspection would very difficult to configure using MITM, because it would not be practical install and maintain CAs in each container.

CHAPTER 3

Supported Signatures

TLS libraries create session secrets during the TLS session handshake. Each TLS library creates session keys in a unique way in the process memory space. Nubeva's core IP finds the session keys using an algorithm designed specifically for each library or application and outputs a set of instructions called a `signature`. Such signatures are then used to gain access to the session keys at runtime in an extremely efficient manner. A *SKI Sensor* uses memory hooks and signatures to extract session secrets to TLS library memory. The ability to hook into memory depends on the Linux Kernel version and Windows OS version. Therefore SKI is applicable to any application that runs on a supported OS and kernel version, and is linked with a supported TLS library.

3.1 Application and Signature Categories

The following figure depicts categories of applications in relation to the TLS libraries they use.

- **Linux**
  - **Shared Libraries**
    - OpenSSL, NSS, WolfSSL
  - **Containers**
    - **Shared Libraries**
      - OpenSSL, NSS, WolfSSL
  - **Application Specific**
    - BoringSSL
    - Others
  - Java
- **Windwos**
  - **Shared Libraries**
    - Schanel

  - **Application Specific**
    - BoringSSL
    - Others
  - Java

The classification is based on the way TLS is included in the application executable. The TLS code could be a shared library, which is linked to the application code, or TLS code which is compiled with the rest of the application. Applications that are linked with TLS libraries share the library signature. Applications that use a private library or compile code, require an application specific signature.

3.2 Linux

On Linux, the vast majority of applications use standard TLS *OpenSSL Libraries* and *NSS Libraries*, and to a lesser extent *WolfSSL Libraries*. Nubeva provides signatures for every version of OpenSSL starting with version 0.9.7, every version of NSS Libraries starting with version NSS_3_15_1_RTM and WolfSSL versions starting from v4.3.0. This means that session keys can extracted from the vast majority of Linux applications. For containers, the signatures also include a unique method to find the path into the directories of the containers as those have a different namespace than the host Linux. Therefore each of the *Supported Linux TLS Libraries* listed below for OpenSSL, NSS and WolfSSL has two signatures. Application specific signatures are required for applications that use BoringSSL, since BoringSSL is provided by Google as a code repository on Github, and this code is compiled with the rest of the application code. *Java* applications use TLS functions included with each version of the JDK. Signatures for Java are supported for JDK version 8 through 17. TLS key extraction is supported for all Java applications using one of these JDK versions.

Please refer to *Linux Systems* for details on Linux Kernel version and Linux version support.

3.2.1 Supported Linux TLS Libraries

OpenSSL and NSS libraries are included with the Linux OS. Applications link to the shared libraries. Therefore key extraction is supported for the vast majority (over 99%) of applications that use OpenSSL or NSS.

OpenSSL Libraries

```
OpenSSL-0.9.7
OpenSSL-0.9.7-beta4
OpenSSL-0.9.7-beta5
OpenSSL-0.9.7-beta5
OpenSSL-0.9.7-beta6
OpenSSL-0.9.7a
OpenSSL-0.9.7b
OpenSSL-0.9.7c
OpenSSL-0.9.7d
OpenSSL-0.9.7e
OpenSSL-0.9.7f
OpenSSL-0.9.7g
OpenSSL-0.9.7h
OpenSSL-0.9.7i
OpenSSL-0.9.7j
OpenSSL-0.9.7k
OpenSSL-0.9.7l
OpenSSL-0.9.7m
OpenSSL-0.9.8
```

(continues on next page)

(continued from previous page)

```
OpenSSL-0.9.8-beta1
OpenSSL-0.9.8-beta2
OpenSSL-0.9.8-beta3
OpenSSL-0.9.8-beta4
OpenSSL-0.9.8-beta5
OpenSSL-0.9.8-beta6
OpenSSL-0.9.8-post-auto-reformat
OpenSSL-0.9.8-post-reformat
OpenSSL-0.9.8-pre-auto-reformat
OpenSSL-0.9.8-pre-reformat
OpenSSL-0.9.8a
OpenSSL-0.9.8b
OpenSSL-0.9.8c
OpenSSL-0.9.8d
OpenSSL-0.9.8e
OpenSSL-0.9.8f
OpenSSL-0.9.8g
OpenSSL-0.9.8h
OpenSSL-0.9.8i
OpenSSL-0.9.8j
OpenSSL-0.9.8k
OpenSSL-0.9.8l
OpenSSL-0.9.8m
OpenSSL-0.9.8m-beta1
OpenSSL-0.9.8n
OpenSSL-0.9.8o
OpenSSL-0.9.8p
OpenSSL-0.9.8q
OpenSSL-0.9.8r
OpenSSL-0.9.8s
OpenSSL-0.9.8t
OpenSSL-0.9.8u
OpenSSL-0.9.8v
OpenSSL-0.9.8w
OpenSSL-0.9.8x
OpenSSL-0.9.8y
OpenSSL-0.9.8za
OpenSSL-0.9.8zb
OpenSSL-0.9.8zc
OpenSSL-0.9.8zd
OpenSSL-0.9.8ze
OpenSSL-0.9.8zf
OpenSSL-0.9.8zg
OpenSSL-0.9.8zh
OpenSSL-1.0.0
OpenSSL-1.0.0-beta1
OpenSSL-1.0.0-beta2
OpenSSL-1.0.0-beta3
OpenSSL-1.0.0-beta4
OpenSSL-1.0.0-beta5
OpenSSL-1.0.0-post-auto-reformat
OpenSSL-1.0.0-post-reformat
OpenSSL-1.0.0-pre-auto-reformat
OpenSSL-1.0.0-pre-reformat
OpenSSL-1.0.0a
OpenSSL-1.0.0b
OpenSSL-1.0.0c
```

(continues on next page)

(continued from previous page)

```
OpenSSL-1.0.0d
OpenSSL-1.0.0e
OpenSSL-1.0.0f
OpenSSL-1.0.0g
OpenSSL-1.0.0h
OpenSSL-1.0.0i
OpenSSL-1.0.0j
OpenSSL-1.0.0k
OpenSSL-1.0.0l
OpenSSL-1.0.0m
OpenSSL-1.0.0n
OpenSSL-1.0.0o
OpenSSL-1.0.0p
OpenSSL-1.0.0q
OpenSSL-1.0.0r
OpenSSL-1.0.0s
OpenSSL-1.0.0t
OpenSSL-1.0.1
OpenSSL-1.0.1-beta1
OpenSSL-1.0.1-beta2
OpenSSL-1.0.1-beta3
OpenSSL-1.0.1-post-auto-reformat
OpenSSL-1.0.1-post-reformat
OpenSSL-1.0.1-pre-auto-reformat
OpenSSL-1.0.1-pre-reformat
OpenSSL-1.0.1a
OpenSSL-1.0.1b
OpenSSL-1.0.1c
OpenSSL-1.0.1d
OpenSSL-1.0.1e
OpenSSL-1.0.1f
OpenSSL-1.0.1g
OpenSSL-1.0.1h
OpenSSL-1.0.1i
OpenSSL-1.0.1j
OpenSSL-1.0.1k
OpenSSL-1.0.1l
OpenSSL-1.0.1m
OpenSSL-1.0.1n
OpenSSL-1.0.1o
OpenSSL-1.0.1p
OpenSSL-1.0.1q
OpenSSL-1.0.1r
OpenSSL-1.0.1s
OpenSSL-1.0.1t
OpenSSL-1.0.1u
OpenSSL-1.0.2
OpenSSL-1.0.2-beta1
OpenSSL-1.0.2-beta1-fips
OpenSSL-1.0.2-beta2
OpenSSL-1.0.2-beta2-fips
OpenSSL-1.0.2-beta3
OpenSSL-1.0.2-beta3-fips
OpenSSL-1.0.2-fips
OpenSSL-1.0.2-post-auto-reformat
OpenSSL-1.0.2-post-auto-reformat-fips
OpenSSL-1.0.2-post-reformat
```

(continues on next page)

(continued from previous page)

```
OpenSSL-1.0.2-post-reformat-fips
OpenSSL-1.0.2-pre-auto-reformat
OpenSSL-1.0.2-pre-auto-reformat-fips
OpenSSL-1.0.2-pre-reformat
OpenSSL-1.0.2-pre-reformat-fips
OpenSSL-1.0.2a
OpenSSL-1.0.2a-fips
OpenSSL-1.0.2b
OpenSSL-1.0.2b-fips
OpenSSL-1.0.2c
OpenSSL-1.0.2c-fips
OpenSSL-1.0.2d
OpenSSL-1.0.2d-fips
OpenSSL-1.0.2e
OpenSSL-1.0.2e-fips
OpenSSL-1.0.2f
OpenSSL-1.0.2f-fips
OpenSSL-1.0.2g
OpenSSL-1.0.2g-fips
OpenSSL-1.0.2h
OpenSSL-1.0.2h-fips
OpenSSL-1.0.2i
OpenSSL-1.0.2i-fips
OpenSSL-1.0.2j
OpenSSL-1.0.2j-fips
OpenSSL-1.0.2k
OpenSSL-1.0.2k-fips
OpenSSL-1.0.2l
OpenSSL-1.0.2l-fips
OpenSSL-1.0.2m
OpenSSL-1.0.2m-fips
OpenSSL-1.0.2n
OpenSSL-1.0.2n-fips
OpenSSL-1.0.2o
OpenSSL-1.0.2o-fips
OpenSSL-1.0.2p
OpenSSL-1.0.2p-fips
OpenSSL-1.0.2q
OpenSSL-1.0.2q-fips
OpenSSL-1.0.2r
OpenSSL-1.0.2r-fips
OpenSSL-1.0.2s
OpenSSL-1.0.2s-fips
OpenSSL-1.0.2t
OpenSSL-1.0.2t-fips
OpenSSL-1.0.2u
OpenSSL-1.0.2u-fips
OpenSSL-1.1.0
OpenSSL-1.1.0-pre1
OpenSSL-1.1.0-pre2
OpenSSL-1.1.0-pre3
OpenSSL-1.1.0-pre4
OpenSSL-1.1.0-pre5
OpenSSL-1.1.0-pre6
OpenSSL-1.1.0a
OpenSSL-1.1.0b
OpenSSL-1.1.0c
```

(continues on next page)

(continued from previous page)

```
OpenSSL-1.1.0d
OpenSSL-1.1.0e
OpenSSL-1.1.0f
OpenSSL-1.1.0g
OpenSSL-1.1.0h
OpenSSL-1.1.0i
OpenSSL-1.1.0j
OpenSSL-1.1.0k
OpenSSL-1.1.0l
OpenSSL-1.1.1
OpenSSL-1.1.1-pre1
OpenSSL-1.1.1-pre2
OpenSSL-1.1.1-pre3
OpenSSL-1.1.1-pre4
OpenSSL-1.1.1-pre5
OpenSSL-1.1.1-pre6
OpenSSL-1.1.1-pre7
OpenSSL-1.1.1-pre8
OpenSSL-1.1.1-pre9
OpenSSL-1.1.1a
OpenSSL-1.1.1b
OpenSSL-1.1.1c
OpenSSL-1.1.1d
OpenSSL-1.1.1e
OpenSSL-1.1.1f
OpenSSL-1.1.1g
OpenSSL-1.1.1h
OpenSSL-1.1.1i
OpenSSL-1.1.1j
OpenSSL-1.1.1k
OpenSSL-FIPS.1.0
OpenSSL-fips-1.2.0
OpenSSL-fips-1.2.1
OpenSSL-fips-1.2.2
OpenSSL-fips-1.2.3
OpenSSL-fips-2.0
OpenSSL-fips-2.0-pl1
OpenSSL-fips-2.0-rc1
OpenSSL-fips-2.0-rc2
OpenSSL-fips-2.0-rc3
OpenSSL-fips-2.0-rc4
OpenSSL-fips-2.0-rc5
OpenSSL-fips-2.0-rc6
OpenSSL-fips-2.0-rc7
OpenSSL-fips-2.0-rc8
OpenSSL-fips-2.0-rc9
OpenSSL-fips-2.0.1
OpenSSL-fips-2.0.2
OpenSSL-fips-2.0.3
OpenSSL-fips-2.0.4
OpenSSL-fips-2.0.5
OpenSSL-fips-2.0.6
OpenSSL-fips-2.0.7
OpenSSL-fips-2.0.8
OpenSSL-fips-2.0.9
OpenSSL-fips-2.0.10
OpenSSL-fips-2.0.11
```

(continues on next page)

(continued from previous page)

```
OpenSSL-fips-2.0.12
OpenSSL-fips-2.0.13
OpenSSL-fips-2.0.14
OpenSSL-fips-2.0.15
OpenSSL-fips-2.0.16
```

Sample Linux Applications that use OpenSSL

The samples below show application versions that use OpenSSL. All versions are in the supported list above.

```
Python 3.9.2, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.9.1, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.9.0, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.8, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.7, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.6, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.5, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.4, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.3, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.2, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.1, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.8.0, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.10, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.9, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.8, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.7, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.6, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.5, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.4, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.3, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.2, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.1, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.7.0, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.13, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.12, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.11, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.10, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.9, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.8, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.7, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.6, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.5, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.4, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.3, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.2, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.1, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.6.0, OpenSSL 1.0.2, 1.1.0, and 1.1.1
Python 3.5.7 OpenSSL 0.9.8, 1.0.2
Python 3.5.6 OpenSSL 0.9.8, 1.0.2
Python 3.5.5 OpenSSL 0.9.8, 1.0.2
Python 3.5.4 OpenSSL 0.9.8, 1.0.2
Python 3.5.3 OpenSSL 0.9.8, 1.0.2
Python 3.5.2 OpenSSL 0.9.8, 1.0.2
Python 3.5.1 OpenSSL 0.9.8, 1.0.2
Python 2.7.8 OpenSSL 0.9.8, 1.0.1
```

(continues on next page)

(continued from previous page)

```
Python 2.7.7 OpenSSL 0.9.8, 1.0.1
Python 2.7.6 OpenSSL 0.9.8, 1.0.1
Python 2.7.5 OpenSSL 0.9.8, 1.0.1
Python 2.7.4 OpenSSL 0.9.8, 1.0.1
Python 2.7.3 OpenSSL 0.9.8, 1.0.1
Python 2.7.2 OpenSSL 0.9.8, 1.0.1
Python 2.7.1 OpenSSL 0.9.8, 1.0.1
Python 2.7 OpenSSL 0.9.8, 1.0.1

AWS CLI depends on Python versions:

AWS CLI v2 requires Python 3.6+
AWS CLI v1 1.19.0 - current requires Python 3.6*
AWS CLI v1 1.17 - 1.18.x requires Python 2.7+ or Python 3.6+

curl 7.76.0 OpenSSL 1.1.0, 1.1.1
curl 7.75.0 OpenSSL 1.1.0, 1.1.1
curl 7.74.0 OpenSSL 1.1.0, 1.1.1
curl 7.73.0 OpenSSL 1.1.0, 1.1.1
curl 7.72.0 OpenSSL 1.1.0, 1.1.1
curl 7.71.1 OpenSSL 1.1.0, 1.1.1
curl 7.71.0 OpenSSL 1.1.0, 1.1.1
curl 7.69.1 OpenSSL 1.1.0, 1.1.1
curl 7.69.0 OpenSSL 1.1.0, 1.1.1
curl 7.68.1 OpenSSL 1.1.0, 1.1.1
curl 7.68.0 OpenSSL 1.1.0, 1.1.1
```

NSS Libraries

Released versions starting with v3.15 (released 2013-05-28 23:37:46)

```
    NSS_3_15_1_RTM
    NSS_3_15_2_RTM
    NSS_3_15_3_1_RTM
    NSS_3_15_3_RTM
    NSS_3_15_4_RTM
    NSS_3_15_5_RTM
    NSS_3_15_RTM
    NSS_3_16_1_RTM
    NSS_3_16_2_1_RTM
    NSS_3_16_2_2_RTM
    NSS_3_16_2_3_RTM
    NSS_3_16_2_RTM
    NSS_3_16_3_RTM
    NSS_3_16_4_RTM
    NSS_3_16_5_RTM
    NSS_3_16_6_RTM
    NSS_3_16_RTM
    NSS_3_17_1_RTM
    NSS_3_17_2_RTM
    NSS_3_17_3_RTM
    NSS_3_17_4_RTM
    NSS_3_17_RTM
    NSS_3_18_1_RTM
    NSS_3_18_RTM
```

(continues on next page)

(continued from previous page)

```
    NSS_3_19_1_RTM
    NSS_3_19_1_WITH_CKBI_1_98_RTM
    NSS_3_19_2_1_RTM
    NSS_3_19_2_2_RTM
    NSS_3_19_2_3_RTM
    NSS_3_19_2_4_RTM
    NSS_3_19_2_RTM
    NSS_3_19_3_RTM
    NSS_3_19_4_RTM
    NSS_3_19_RTM
    NSS_3_20_1_RTM
    NSS_3_20_2_RTM
    NSS_3_20_RTM
    NSS_3_21_1_RTM
    NSS_3_21_2_RTM
    NSS_3_21_3_RTM
    NSS_3_21_4_RTM
    NSS_3_21_RTM
    NSS_3_22_1_RTM
    NSS_3_22_2_RTM
    NSS_3_22_3_RTM
    NSS_3_22_RTM
    NSS_3_23_RTM
    NSS_3_24_RTM
    NSS_3_25_1_RTM
    NSS_3_25_RTM
    NSS_3_26_1_RTM
    NSS_3_26_2_RTM
    NSS_3_26_RTM
    NSS_3_27_1_RTM
    NSS_3_27_2_RTM
    NSS_3_27_RTM
    NSS_3_28_1_RTM
    NSS_3_28_2_RTM
    NSS_3_28_3_RTM
    NSS_3_28_4_RTM
    NSS_3_28_5_RTM
    NSS_3_28_6_RTM
    NSS_3_28_RTM
    NSS_3_29_1_RTM
    NSS_3_29_2_RTM
    NSS_3_29_3_RTM
    NSS_3_29_5_RTM
    NSS_3_29_RTM
    NSS_3_30_1_RTM
    NSS_3_30_2_RTM
    NSS_3_30_RTM
    NSS_3_31_1_RTM
    NSS_3_31_RTM
    NSS_3_32_1_RTM
    NSS_3_32_RTM
    NSS_3_33_RTM
    NSS_3_34_1_RTM
    NSS_3_34_RTM
    NSS_3_35_RTM
    NSS_3_36_1_RTM
    NSS_3_36_2_RTM
```

(continues on next page)

(continued from previous page)

```
    NSS_3_36_4_RTM
    NSS_3_36_5_RTM
    NSS_3_36_6_RTM
    NSS_3_36_7_RTM
    NSS_3_36_RTM
    NSS_3_37_1_RTM
    NSS_3_37_3_RTM
    NSS_3_37_RTM
    NSS_3_38_RTM
    NSS_3_39_RTM
    NSS_3_40_1_RTM
    NSS_3_40_RTM
    NSS_3_41_1_RTM
    NSS_3_41_RTM
    NSS_3_42_1_RTM
    NSS_3_42_RTM
    NSS_3_43_RTM
    NSS_3_44_1_RTM
    NSS_3_44_2_RTM
    NSS_3_44_3_RTM
    NSS_3_44_4_RTM
    NSS_3_44_RTM
    NSS_3_45_RTM
    NSS_3_46_1_RTM
    NSS_3_46_RTM
    NSS_3_47_1_RTM
    NSS_3_47_RTM
    NSS_3_48_1_RTM
    NSS_3_48_RTM
    NSS_3_49_1_RTM
    NSS_3_49_2_RTM
    NSS_3_49_RTM
    NSS_3_50_RTM
    NSS_3_51_1_RTM
    NSS_3_51_RTM
    NSS_3_52_1_RTM
    NSS_3_52_RTM
    NSS_3_53_1_RTM
    NSS_3_53_RTM
    NSS_3_54_RTM
    NSS_3_55_RTM
    NSS_3_56_RTM
    NSS_3_57_RTM
    NSS_3_58_RTM
    NSS_3_59_1_RTM
    NSS_3_59_RTM
    NSS_3_60_1_RTM
    NSS_3_60_RTM
    NSS_3_61_RTM
    NSS_3_62_RTM
    NSS 3_63_RTM
    NSS_3_63_1_RTM
NSS_3_64_RTM
```

WolfSSL Libraries

```
Version v4.3.0
Version v4.4.0
Version v4.5.0
Version v4.6.0
Version v4.7.0
```

BoringSSL

Google's BoringSSL code is available on Github. Applications using BoringSSL require application specific signatures.

3.2.2 Application Signatures

Envoy

```
1.9.0
1.9.1
1.10.0
1.11.0
1.11.1
1.11.2
1.12.0
1.12.1
1.12.2
1.12.3
1.12.4
1.12.6
1.12.7
1.13.0
1.13.1
1.13.2
1.13.3
1.13.4
1.13.5
1.13.6
1.13.6
1.13.8
1.14.0
1.14.1
1.14.2
1.14.3
1.14.4
1.14.5
1.14.6
1.14.7
1.15.0
1.15.1
1.15.2
1.15.3
1.15.4
1.16.0
1.16.1
```

(continues on next page)

(continued from previous page)

```
1.16.2
1.16.3
1.17.0
1.17.1
1.17.2
1.18.0
1.18.1
1.18.2
1_18_dev
1_19_dev
```

3.3 MS Windows

On Windows, the majority of applications use the standard TLS *Schannel* library. Nubeva provides signatures for every version of Schannel starting with version 6.3.9600.19941. Application specific signatures are required for applications that use BoringSSL, since BoringSSL is provided by Google as a code repository on Github, and this code is compiled with the rest of the application code. Two such applications are Google Chrome and MS Edge Chromium. Their supported versions are listed below. Until Recently DropBox used a specific version of TLS that required an application signatures. The supported versions of DropBox are also provided. DropBox has recently switched to using Schannel, and therefore future versions of DropBox will no longer require dedicated signatures.

Please refer to *MS Windows Systems* for a list of supported operating system versions.

3.3.1 Supported TLS DLLs

Schannel

Schannel.dll versions supported:

```
10.0.14393.0
10.0.14393.1613
10.0.14393.3269
10.0.14393.3750
10.0.14393.3808
10.0.14393.3930
10.0.14393.4225
10.0.17763.1
10.0.17763.1217
10.0.17763.1282
10.0.17763.1339
10.0.17763.1457
10.0.17763.1728
10.0.17763.802
10.0.18362.1082
10.0.18362.418
10.0.18362.900
10.0.18362.959
10.0.18362.997
10.0.19041.1
10.0.19041.329
10.0.19041.388
10.0.19041.508
```

(continues on next page)

(continued from previous page)

```
10.0.19041.546
10.0.19041.789
10.0.20295.1
6.1.7601.17514
6.2.17763.802
6.2.9200.22562
6.3.9600.17031
6.3.9600.17415
6.3.9600.19473
6.3.9600.19941
```

Applications that use SChannel on these Microsoft platforms include but are not limited to:

Access, Cortana, Excel, Groove Music, InfoPath, Internet Explorer, Microsoft Store, Microsoft News, Microsoft Sway, Microsoft Teams, Microsoft Word, Movies and TV, MS Dynamics 365 CRM, MSN Money, MSN Sports, MSN Weather, OneDrive, OneNote, Outlook, Power BI Desktop, PowerPoint, Publisher, Skype.

BoringSSL

Google's BoringSSL code is available on Github. Applications using BoringSSL require application specific signatures. In the Windows Applications examples below Google Chrome and MS Edge Chromium use BoringSSL.

3.3.2 Windows Applications

Application specific signatures are supported for:

Application	Version
Dropbox	91.4.548, 92.4.382, 94.4.384, 93.4.237, 93.4.273, 94.4.384, 95.4.441, 96.4.172, 97.4.467, 98.4.158, 99.4.501, 100.4.409, 101.4.434, 102.4.431, 103.4.383, 104.4.175, 105.4.651, 106.4.368, 107.4.443, 108.4.453, 109.4.517 110.4.458, 111.4.472. 112.4.321, 113.4.507, 114.4.426, 115.4.601, 116.4.368, 117.4.378, 119.4.1772
Google Chrome	80.0.3987.132, 80.0.3987.149, 80.0.3987.163, 81.0.4044.113, 83.0.4103.97, 81.0.4044.113, 81.0.4044.122, 81.0.4044.129, 81.0.4044.138, 81.0.4044.92, 83.0.4103.61, 83.0.4103.97, 83.0.4103.106, 83.0.4103.116, 84.0.4147.89, 84.0.4174.105, 84.0.4147.125, 84.0.4147.135, 85.0.4183.83, 85.0.4183.102, 85.0.4183.102, 85.0.4183.121, 86.0.4240.111, 86.0.4240.75, 86.0.4240.183, 86.0.4240.193, 86.0.4240.198, 87.0.4280.66, 87.0.4280.88, 87.0.4280.141, 88.0.4324.104, 88.0.4324.146, 88.0.4324.150, 88.0.4324.182, 88.0.4324.190, 89.0.4389.72, 89.0.4389.82, 89.0.4389.90, 89.0.4389.114, 89.0.4389.128, 90.0.4430.72, 90.0.4430.85, 90.0.4430.93
MS Edge Chromium	80.0.361.66, 80.0.361.69, 80.0.361.109, 83.0.478.44, 83.0.478.45, 83.0.478.54, 83.0.478.56, 83.0.478.58, 83.0.478.61, 84.0.522.44, 84.0.522.49, 84.0.522.52, 84.0.522.59, 84.0.522.63, 85.0.564.41, 85.0.564.44, 85.0.564.51, 85.0.564.63, 86.0.4240.75, 86.0.622.38, 86.0.622.43, 86.0.622.51, 86.0.622.56, 86.0.622.61, 86.0.622.69, 87.0.664.41, 87.0.664.47, 87.0.664.52, 87.0.664.55, 87.0.664.57, 87.0.664.60, 87.0.664.66, 87.0.664.75, 87.0.664.75, 88.0.705.50, 88.0.705.53, 88.0.705.56, 88.0.705.62, 88.0.705.63, 88.0.705.68, 88.0.705.74, 88.0.705.81, 89.0.774.45, 89.0.774.48, 89.0.774.50, 89.0.774.54, 89.0.774.57, 89.0.774.63, 89.0.774.68, 89.0.774.75, 89.0.774.76, 89.0.774.77, 90.0.818.41, 90.0.818.42, 90.0.818.46, 90.0.818.49, 90.0.818.51
MS Edge (old)	44.18362.449.0
Zscaler App	2.1.0.210

3.4 Java

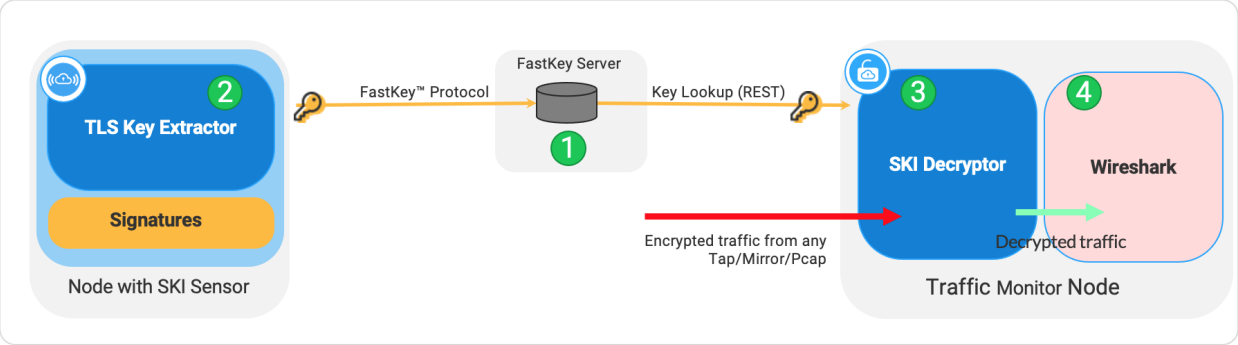
Key extraction works on Linux and Windows for all Java applications using supported JDK versions on the supported operating system versions listed above.

```
Java 8
Java 9
Java 10
Java 11
Java 12
Java 13
Java 14
Java 15
Java 16
Java 17
```

CHAPTER 4

Getting Started

SKI is a complete TLS visibility solution. In this section we show a simple TLS visibility example using a `SKI FastKey Buffer`, `SKI Sensor`, `SKI Decryptor` and WireShark. The SKI Sensor discovers session secrets from processes created by a traffic generator and sends the them to the FastKey Buffer. A `SKI Decryptor` receives mirrored encrypted traffic and retrieves session keys from the FastKey Buffer. The Decryptor uses these keys to decrypt the traffic, and outputs decrypted traffic on an interface monitored by WireShark. You can inspect the decrypted traffic using WireShark user interface.

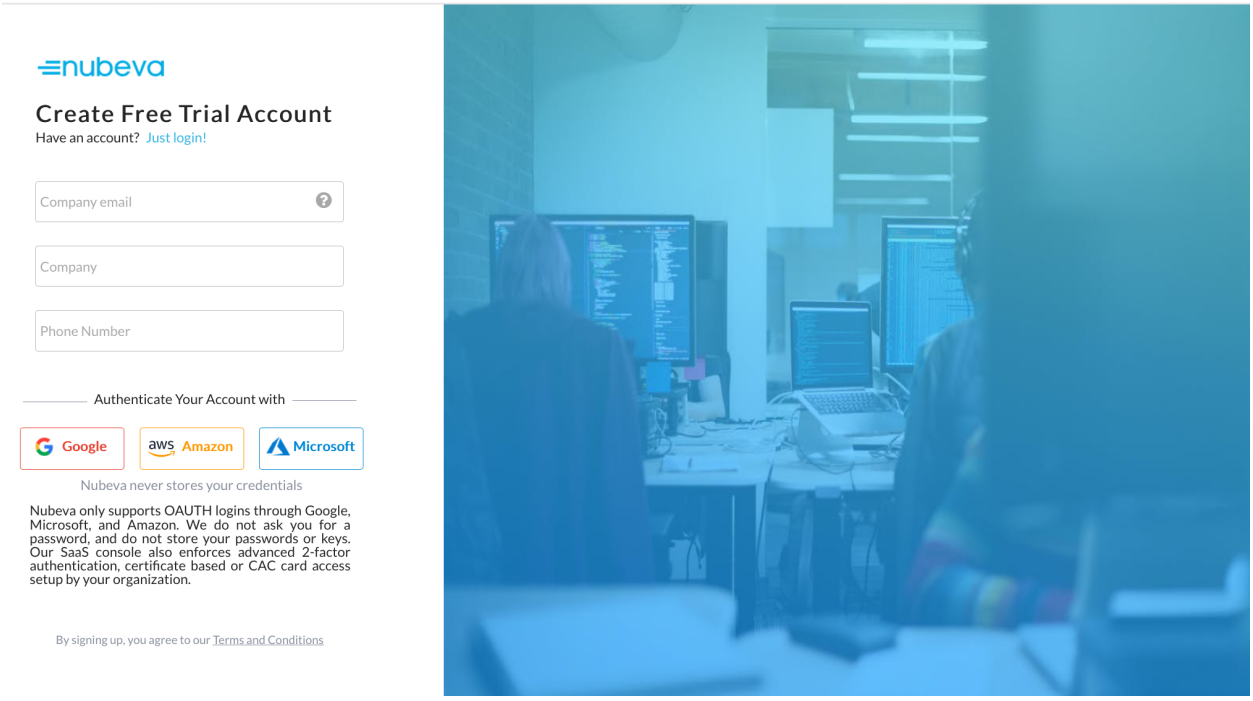


Tip: SKI can be used in a wide variety of use cases. This example is meant to illustrate SKI's basic features. Please review the full document to understand the broader applicability and scalability of SKI.

4.1 Create an Evaluation Account

To get started with Nubeva TLS Visibility Solution, start by creating an account on Nubeva's Account Console. Enter you email and company name and login with one the three OAuth providers.

Note: Nubeva only supports OAUTH logins through Google, Microsoft, and Amazon. We do not ask you for a



password, and do not store your passwords or keys.

Once your account is created you will see your account details and your account token.

Once you have a token you can launch and test SKI components.

4.2 Step 1: Run a FastKey Buffer

Please see *FastKey Buffer* for details about the FastKey(TM) key buffer.

4.3 Step 2: Run a SKI Sensor

Please see *SKI Sensors* for instructions to launch a SKI Sensor. A sample traffic generation script is provided in the section. You may generate traffic using the sample script or by any other means.

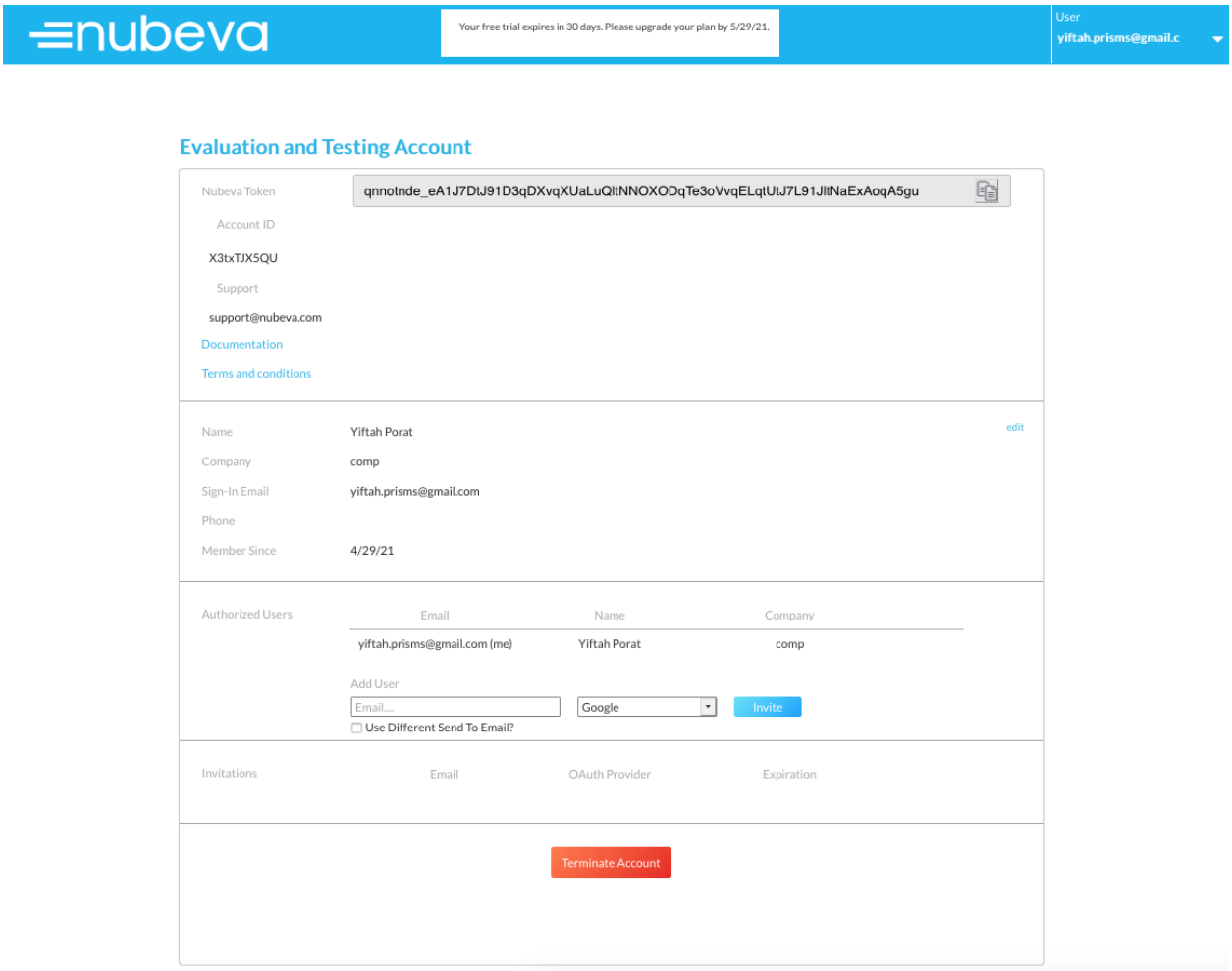
4.3.1 Test Key Discovery

You can see the keys by running:

```
curl https://<key server domain>:4433/dumpkeys
```

4.4 Step 3: Run a SKI Decryptor

Please see *SKI Decryptors* for instructions to launch a SKI Decryptor.


nubeva
Your free trial expires in 30 days. Please upgrade your plan by 5/29/21.
User
yiftah.prisms@gmail.c
Evaluation and Testing Account
Nubeva Token
qnnotnde_eA1J7DtJ91D3qDXvqXUaLuQltNNOXODqTe3oVvqELqtUtJ7L91JltNaExAoqA5gu
Account ID
X3txTJX5QU
Support
support@nubeva.com
Documentation
Terms and conditions
Name
Yiftah Porat
edit
Company
comp
Sign-In Email
yiftah.prisms@gmail.com
Phone
Member Since
4/29/21
Authorized Users
Email
Name
Company
yiftah.prisms@gmail.com (me)
Yiftah Porat
comp
Add User
Email....
Google
Invite
Use Different Send To Email?
Invitations
Email
OAuth Provider
Expiration
Terminate Account

Note: You should mirror traffic from the sensor instance to the decryptor. You can use AWS VPC Traffic Mirrors or your own tapping or spanning tools.

4.5 Step 4: Run Wireshark

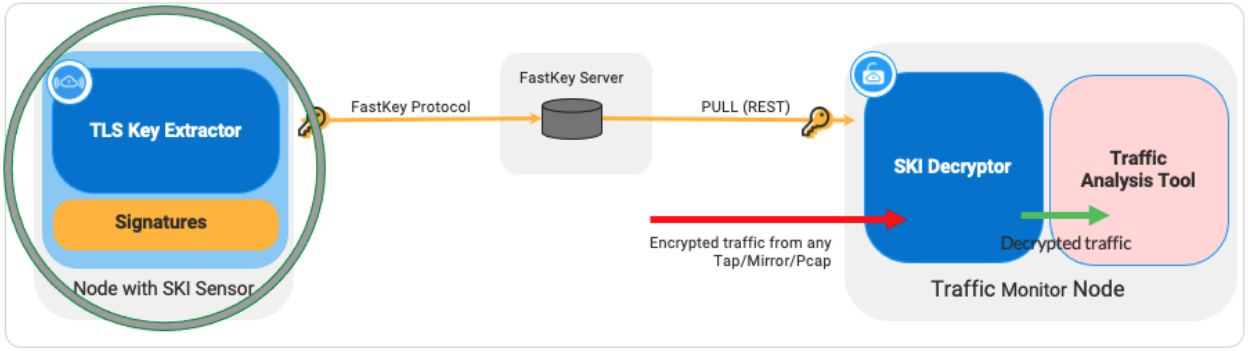
You can launch a containerized version of *Wireshark* to see encrypted and decrypted traffic.

Tip: If you are running on AWS you can deploy the above configuration using the following cloud formation template: https://nubevalabs.s3.amazonaws.com/nudemo/nubeva-demo.template.yaml

CHAPTER 5

SKI Sensor

A SKI Sensor should be deployed on a node where you wish to discover session secrets. SKI Sensors can be deployed as containers on Linux, Kubernetes and OpenShift using Docker or CRI-O as well as a MS-Windows service. SKI Sensors send keys to a Key Server using Nubeva's `FastKey` protocol.



System Requirements

A single SKI sensor is required on a physical (bare metal) or virtual node. For example a virtual node could be an EC2 instance on AWS, an Azure virtual machine or a GCP compute engine instance. A node could also be part of an EKS, AKS, or GKS Kubernetes or OpenShift cluster. The SKI Sensor extracts session keys from all the processes and containers running on the node even if a container is up for less than a second, which is possible in cloud native environments. Once launched, a SKI Sensor periodically checks and downloads code and signature updates and does not require maintenance.

When a SKI Sensor detects a new process or container it checks which TLS library the process uses, and selects that signature to extract TLS session keys. If the process does not use a known library, the sensor looks for an application specific signature. If such a signature is found, the signature is used. In the rare case that both lookups fail, the sensor sends a *KeySense* notification.

5.1 Sensor Command Line Parameters

SKI Sensors read their basic setup from command line parameters listed below. Highlighted `parameters` indicate parameters used in the command lines provided in this guide.

- `accept_eula`: adding this flag indicates you accept the Nubeva's EULA
- `baseurl`: (string) For simple integration with orchestration systems, SKI sensors check for configuration changes by reading from local files. On Linux this is done by setting the `baseurl` to `file:///host/` followed by the path to the directory where the configuration files are stored, for example `file:///host/home/ubuntu/` (including the trailing slash). On Windows `baseurl` should be set to `file://c:` followed by the path to the directory that contains the configuration files, for example `file://c:/nubeva` (a trailing slash should not be included). The files are described in the next section.
- `debug`: value [all | none | main | nutap | nuagentclient | argparser | docker | ssl | keys | keyability] default:none
- `disable`: value [all | none | panicwrapper | ssl | tagupdates | containerssl | `metrics` | keyability] default:none
- `noautoupdate`: When set the panic-wrapper will not automatically update the sensor binary.
- `nocloudwatch`: Stops logging to CloudWatch if this flag is set. Don't even start the thing.
- `nutoken`: (string) Nubeva token for authenticating to the API
- `ssl-baseurl` (string) the base URL for key signature updates.

Note: Unlicensed users should set this value to point to Nubeva's saas backend: `https://i.nuos.io/api/1.1/wf/`.

- `sslcredobj` (string) defines a key store authentication parameters. This string is a base64 encoded json object that can be generated using the following command:

Note:

```
# Encode
echo '{"type":"dtls","domain":"key.nubedge.com:4433","region":"test","ak":"user","sk":
↪"password"}' | base64
```

```
# Decode
echo
↪'eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlc
↪' | base64 --decode
```

Type "dtls" value means a Key Server, 'domain' is the key server's destination host name, 'region' is 'test', user, and password fields can be left the same. They are not used but they are required. You should change the domain.

To export session keys to a local file in nsskeylog format, set the `type` to `lcl` and `region` to specify a file:

```
#Encode
echo '{"type":"lcl","region":"/host/tmp/","domain":"keys.log"}' | base64

#Decode
echo 'eyJ0eXBlIjoibGNsIiwicmVnaW9uIjoiL2hvc3QvdG1wLyIsImRvbWFpbiI6ImtleXMubG9nIn0K' |␣
↪base64 --decode

#Windows local files encode
echo '{"type":"lcl","region":"c:\\\\nubeva\\\\","domain":"keys.log"}' | base64
```

(continues on next page)

(continued from previous page)

```
#Decode
echo
↪'eyJ0eXBlIjoibGNsIiwicmVnaW9uIjoiYzpcXFxcbnViZXZhXFxcXCIsImRvbWFpbiI6ImtleXMubG9nIn0K
↪' | base64 --decode
```

- **version**: show current code/binary version

5.2 Sensor Configuration Files

The simplest way configure SKI Sensors is by using three configuration files: `sensor_login`, `sensor_create` and `sensor_get`. The contents of the files are listed below.

- **sensor_login**:

```
{
        "status": "success",
        "response": {
                "user_id": "default",
                "token": "default",
                "expires": 31536000,
        }
}
```

- **sensor_create**:

```
{
        "status": "success",
        "response": {
                "sensorid": "i-0216d54994df6bcc1x631140662218946579075626",
                "account_id": "default",
                "plan_id": "default"
    }
}
```

Note: Sensor ids are not random. You should not modify the value of the `sensorid` unless you have a valid alternative.

- **sensor_get**:

```
{
        "status": "success",
        "response": {
        "SslCredObj":
↪"eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlc
↪",
        "SrcGroups": [
                {
                        "_type": "custom.srcgroup",
                        "sensorlist_list_custom_sensor": [
                        "i-0216d54994df6bcc1x631140662218946579075626"
                        ],
```

(continues on next page)

(continued from previous page)

```
                "ssl": true,
                "_id": "1234567890AAAA"
                }
        ]
    }
}
```

5.3 Deploying SKI Sensors

You should modify `baseurl` file path the and the value of `sslcredobj` to the domain of your key server. The key server's DNS name must resolve to its IP address (usually with /etc/hosts).

5.3.1 Linux Container

The following command launches a SKI Sensor container.

```
docker run -v /:/host -v /var/run/docker.sock:/var/run/docker.sock  \
--cap-add NET_ADMIN --cap-add SYS_ADMIN \
--cap-add SYS_RESOURCE --cap-add SYS_PTRACE \
--name nubeva-agent -d --restart=always  \
--net=host --pid host nubeva/nuagent \
--accept-eula --contained on \
-nutoken {YOUR_NUTOKEN} \
-noautoupdate --nocloudwatch  --debug=none \
--disable metrics --disable tagupdates \
--sslcredobj␣
↪eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlci
↪\
--baseurl file:///host/<path>/ --ssl-baseurl https://i.nuos.io/api/1.1/wf/
```

You may use a *Traffic Generator* to create encrypted traffic from which the SKI Sensor extracts keys.

5.3.2 Windows Sensor

To run a Windows sensor that sends keys to `key.nubedge.com` run the following PowerShell command:

```
$DownloadDir = $env:TEMP;   $BaseUrl="file:///c:/<path>"; $SSLUrl="https://i.nuos.io/
↪api/1.1/wf/";
$InstallerArg="--noautoupdate --nocloudwatch  -debug none -disable metrics -disable␣
↪tagupdates --accept-eula -baseurl ${BaseUrl} -ssl-baseurl ${SSLUrl} -sslcredobj␣
↪eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlci
↪";
$NubevaTok="{YOUR_NUTOKEN}";
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
Invoke-WebRequest -Uri "https://i.nuos.io/NubevaSensor.latest.setup.exe" -OutFile "
↪$DownloadDir\installer.exe"; & "$DownloadDir\installer.exe" NUTOKEN_USERINPUT=
↪$NubevaTok  API_URL_ARG=${InstallerArg} /q;
```

5.3.3 Kubernetes DaemonSet

SKI Sensors can be deployed as a DaemonSet in K8s. To launch a SKI Sensor DaemonSet modify the following command with a appropriate `baseurl` and `sslcredobj` values.

```
kubectl apply -f https://nubevalabs.s3.amazonaws.com/nuAgentDaemonSet.yaml
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nuagent
spec:
  selector:
    matchLabels:
      name: nuagent
  template:
    metadata:
      labels:
        name: nuagent
    spec:
      hostNetwork: true
      containers:
      - name: nuagent
        image: nubeva/nuagent
        imagePullPolicy: Always
        args: ["--accept-eula", "--nutoken", "avnujtoj_
↪D1don13jxtUQoGovougnelugODOAdlOdxOu7jVx9jDq5Vx7goL1QqVgl9AxulunG", "--sslcredobj",
↪"eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlc
↪","-noautoupdate", "--nocloudwatch", "--disable", "metrics", "--disable",
↪"tagupdates", "--baseurl", "file:///host/<path>", "--ssl-baseurl", "https://i.nuos.
↪io/api/1.1/wf/"]
        securityContext:
          capabilities:
            add: ["NET_ADMIN", "SYS_ADMIN", "SYS_PTRACE", "SYS_RESOURCE"]
        volumeMounts:
          - name: dockersocket
            mountPath: /var/run/docker.sock
          - name: vhost
            mountPath: /host
      volumes:
        - hostPath:
            path: /var/run/docker.sock
          name: dockersocket
        - hostPath:
            path: /
          name: vhost
```

TLS Traffic Generator

```
kubectl apply -f <path of file containing the code below>
```

You can create a batch job that will spawn a TLS traffic generator every minute using the following yaml file:

```
apiVersion: batch/v1beta1
kind: CronJob
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: tlstraffic
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec: # JOB
      template:
        spec:
          containers:
          - name: tlsgenerator
            image: nubevalab/tlsgenerator
          restartPolicy: Never
      backoffLimit: 2
```

5.3.4 Native Linux Sensor

SKI Sensors can be deployed as native Linux services by running the following command:

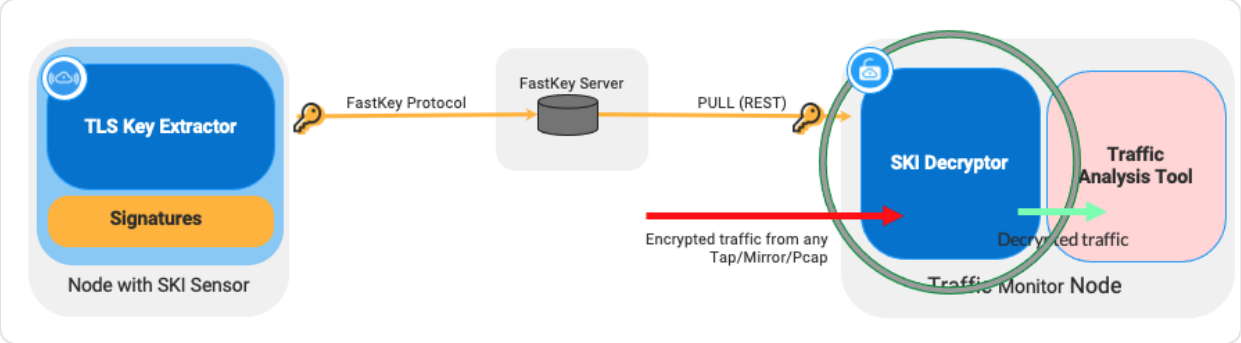
```
curl -s https://nubevalabs.s3.amazonaws.com/install_linux.sh | sudo bash -s -- \
--accept-eula --versionurl https://i.nuos.io/api/1.1/wf/ \
--nutoken avnujtoj_D1don13jxtUQoGovougnelugODOAdlOdxOu7jVx9jDq5Vx7goL1QqVgl9AxulunG \
--baseurl file:///host/<path>/ --ssl-baseurl https://i.nuos.io/api/1.1/wf/ \
--sslcredobj␣
↪eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlci
↪\
--debug none --disable metrics --noautoupdate --nocloudwatch
```

Note: The parameter `versionurl` instructs the installation script from where to download the sensor executable. This parameter is not passed to the sensor.

CHAPTER 6

SKI Decryptor

SKI Decryptors decrypt TLS records with session secrets extracted by SKI Sensors. SKI Decryptors pull keys from key depots: AWS DynamoDB, MongoDB or Nubeva's Key Server. Keys are looked up in a key depot based on client-random values extracted from client-hello handshake messages. This approach assures that decryptors decrypt TLS sessions and TLS resumed sessions in the same manner. Packets are decrypted and delivered out of the virtual interface to be inspected, monitored, or forwarded. SKI Decryptors decrypt TLS records with session secrets extracted by SKI Sensors. SKI Decryptors pull keys from key depots: AWS DynamoDB, MongoDB or Nubeva's Key Server. Keys are looked up in a key depot based on client-random values extracted from client-hello handshake messages. This approach assures that decryptors decrypt TLS sessions and TLS resumed sessions in the same manner. Packets are decrypted and delivered out of the virtual interface to be inspected, monitored, or forwarded.



Note:

- Delivered as Linux containers, supported on Linux - kernel 4.4 or above, Kubernetes 1.13 and above and OpenShift 4.x.
- TLS 1.3, TLS 1.2 PFS and legacy ciphers.
- 2Gbps throughput.
- Decrypted packets can be processed by commercial and open source tools such as Arkime (Moloch), Bro, ntop, Suricata and Wireshark.

Decryptors read VXLAN traffic from the NIC and use session secrets received from SKI Sensors to decrypt traffic. Output is sent to an interface called `nurx0`. Encrypted packets are forwarded to port 443. Decrypted packers are forwarded to port 80.

Tip: If you are running a customized network stack (e.g. DPDK), require 10Gbps or higher decryption throughput, or have other specialized decryption configuration requirements, please review the *SKI Decryption Library* section.

6.1 Decryptor Configuration Files

SKI Decryptors require three configuration files: `rx_login`, `rx_create` and `rx_get`. The contents of the files are listed below.

- **rx_login**: login request. The Key Server returns the following JSON object:

```
{
        "status": "success",
        "response": {
                "user_id": "default",
                "token": "default",
                "expires": 31536000,
        }
}
```

- **rx_create**: The SKI Decryptor sends its meta data. The Key Server returns a the following JSON :

```
{
        "status": "success",
        "response": {
                "rxid": "01234567890",
                "account_id": "default",
                "plan_id": "default"
    }
}
```

- **rx_get**: decryptor configuration request. The Key Server returns a minimal configuration in JSON format:

```
{
        "response": {
                "Mtu": 65535,
                "SslCredObj":
↪"eyJ0eXBlIjoiZHRscyIsImRvbWFpbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwicmVnaW9uIjoidGVzdCIsImFrIjoidXNlc
↪"
        },
        "status": "success"
}
```

6.2 Decryptor Command Line Parameters

Highlighted `parameters` indicate parameters used in the command lines provided in this guide.

- `accept_eula`: adding this flag indicates you accept the Nubeva's EULA

- `baseurl`: (string) Base URL for requests, include final slash 'file:///host/<local path>/'
- `both-traffic`: send both encrypted and decrypted traffic to interface (default true)
- `debug`: value [all | none | main | nutap | nuagentclient | argparser | metrics] default:none
- **dint**: interface to send decrypted traffic to (default is "nurx0")
- `disable`: value [all | none | client | cwlogs | panicwrapper | `metrics`] default:none
- `noautoupdate`: Automatically update binary
- `nutoken`: (string) Nubeva token for authenticating to the API
- `sslcredobj` (string) defines a key store authentication parameters:

Note: Make sure you use an sslcredobj value that contain a `type` field value set to `kdb`.

```
--sslcredobj␣
↪eyJ0eXBlIjoia2RiIiwiZG9tYWluIjoidGVzdCIsInJlZ2lvbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwiYWsiOiJ1c2VyIiw
```

This string is a base64 encoded json object that can be generated using the following command:

```
echo '{"type":"kdb","domain":"key.nubedge.com:4433","region":"test","ak":"user","sk":
↪"password"}' | base64
```

Type `kdb` instructs the SKI Decryptor to use a REST API to retrieve session secrets, 'domain' is the destination host name, `region` should be set to "test", `user`, and `password` fields can be left the same. They are not used but they are required. You can change the domain to anything however you MUST have a valid cert. This docker container contains a valid cert for key.nubedge.com.

6.3 Deploying SKI Decryptors

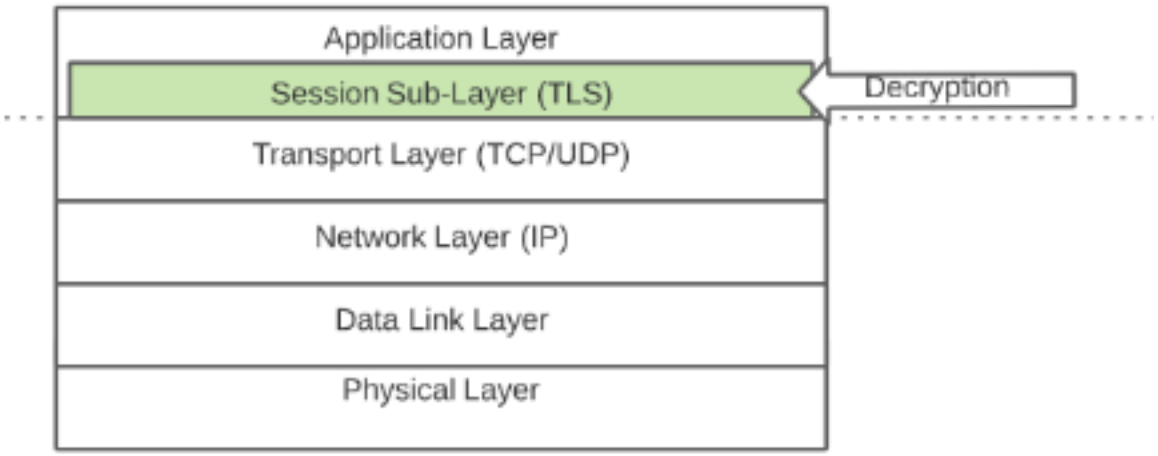
To launch a SKI Decryptor run the following command:

```
docker run -v /:/host -v /var/run/docker.sock:/var/run/docker.sock \
--cap-add NET_ADMIN \
--name nubeva-rx -d --restart=on-failure \
--net=host nubeva/nurx --accept-eula \
--disable metrics -noautoupdate \
--nutoken {YOUR_NUTOKEN} \
--sslcredobj␣
↪eyJ0eXBlIjoia2RiIiwiZG9tYWluIjoidGVzdCIsInJlZ2lvbiI6ImtleS5udWJlZGdlLmNvbTo0NDMzIiwiYWsiOiJ1c2VyIiw
↪\
--baseurl file:///host/<path>/
```

CHAPTER 7

SKI Decryption Library

The decryption library is a static C library designed to provide high decryption throughput of TLS records. The underlying packet processing system need not be aware of TLS records, or perform any TLS parsing operations. The underlying packet processing system should perform standard TCP layer termination and re-assembly, and eliminate packet deduplication that could occur if traffic is tapped at multiple points. All packets belonging to a particular TCP session must arrive at the same thread processing the data. Each decryption thread operates as a completely independent system.

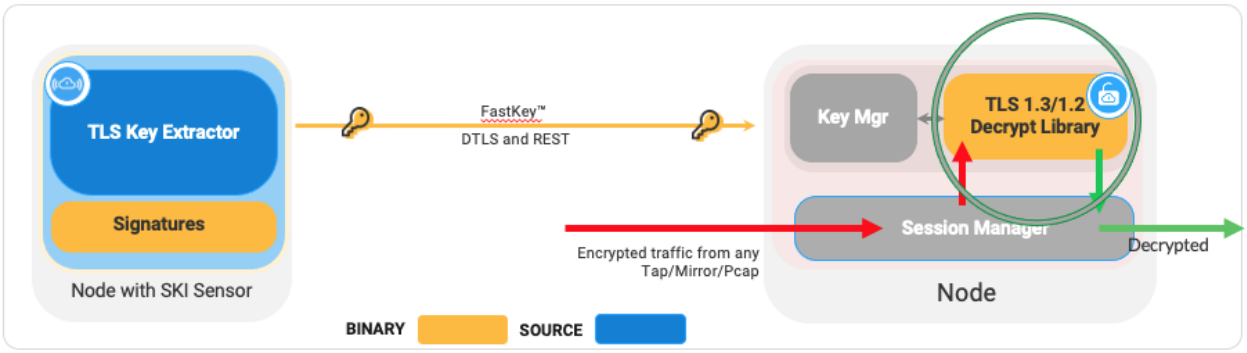


The decryption library supports TLS 1.2, TLS 1.2 PFS, and TLS 1.3 ciphers. The library provides high decryption throughput at 80% of an OpenSSL speed test. The table below shows decryption throughput in Gbps on an AWS m4.xlarge AWS Linux 2, with an Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, with AES-NI support enabled.

Cipher	Block Size	Decryption Gbps
AES-128-GCM Mac=AEAD	1024	13.95
AES-256-GCM Mac=AEAD	1024	11.12
CHACHA20-POLY1305	1024	5.86
AES-128-CBC-HMAC=SHA1	1024	4.01
AES-128-CBC-HMAC=SHA256	1024	2.04
AES-256-CBC-HMAC-SHA1	1024	3.80
AES-256-CBC-HMAC-SHA256	1024	1.97

Note: The numbers above are for a single thread. Higher throughput can be achieved by running multiple decryption threads.

7.1 Using the Library



7.1.1 Data Types

- **Thread Info** (`nuthread`) Some per thread info is required in order to keep configuration information and re-usable buffers. This per thread information is represented by the `nuthread` structure type. It is expected that this information be consistent for every function call in the same thread.
- **Session Info** (`nusession`) Session state information is required so that the same thread can handle multiple independent sessions. The session info pointer is expected to be saved in the same lookup table as the TCP session information and passed into the decryptor so as to have the decryptor be up to date with current TLS record processing for this session. The contents of this pointer is opaque to the user of the library.
- **Packet Info** (`nupacket`) Each record that is passed into the library also requires information to determine directionality (was it sent from the client to the server or the server to the client). This packet information is represented by the `nupacket` structure.

7.1.2 Library Functions

The decryption library interface has the following functions:

nu_record_processor_init

This is the initialization function.

- **Parameters: nu_processor_options:** a structure containing two function pointers: a function to lookup master keys based on a `client-random` value, and a function to free key memory. **nu_processor_init_response:** An output parameter which indicates a successful initialization, or that invalid data was passed to the function. The latter is likely due to null pointers.
- **Returns: nutldh:** Nubeva thread local data handle.

nu_new_session

Called at the beginning of a new session in order to initialize the session state information.

- **Inputs: nutldh:** The nubeva thread local data handle that is returned from the call to `nu_record processor_init()`.
- **Returns: nush:** a new nubeva session handle.

nu_record_processor

This is the actual record processor. It will keep state information and decrypt the data and return the decrypted data and status.

- **Parameters: nush:** The nubeva session handle returned from the call to `nu_new_session()`.

  nupacket: The input packet information.

  indata: A buffer containing the encrypted data (TCP payload).

  outdata: a buffer to contain the decrypted data. The size of this buffer should be the maximal size of a TLS record 16384 + 1024 (17408) + `len` (the size of the input data buffer).

  len: On input `len` indicates the length of the input data. On output `len` indicates the length of the decrypted data in the output buffer.
- **Returns:** record processing status enumeration.

nu_record_processor_cleanup

This is the cleanup function. It releases memory and closes all resources used.

- **Parameters: nutldh:** the nubeva thread local data handle.

nu_session_cleanup

Called when freeing a TLS session to release all session records.

- **Parameters: nush:** the nubeva session handle.

nu_lost_client_frame

Should be called to notify the library of the first loss in a consecutive set of lost packets from the client to the server.

- **Parameters: nush:** the nubeva session handle.

nu_lost_server_frame

Should be called to notify the library of the first loss in a consecutive set of lost packets from the server to the client.

- **Parameters:** **nush:** the nubeva session handle.

nu_license_check

- **Returns:** A string containing the library license type and expiration date.

7.1.3 Initialization

Call `nu_record_processor_init()` with configuration parameters. Check for errors. After the initialization, you will have a `nuthread` structure instance that you should pass into every process request.

7.1.4 Processing TLS Records

For each new TCP Session: Call `nu_new_session()` to get a session state pointer. Keep track of this pointer within your TCP state information structure.

For each TCP packet generate a `nupacket` info structure with information about this particular packet. Pass the `nusession` and `nupacket` structs, together with an encrypted packet, a buffer for decrypted TLS data, and a pointer to length information to `nu_record_processor()`. The return code indicates what type of data has been decrypted or if any errors occurred.

The following section explains how `nu_record_processor()` parses and decrypts all possible TLS encapsulations in TCP payloads:

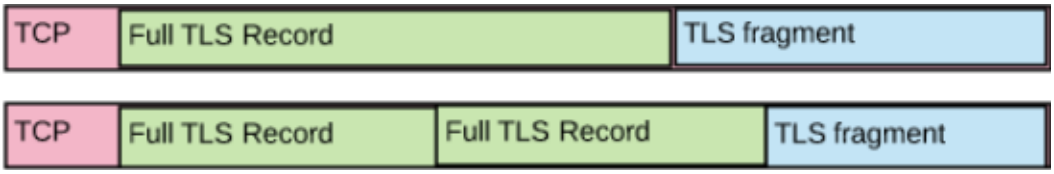
- If the TCP payload contains a single complete TLS record (shown below):

  `nu_record_processor()` outputs the decrypted data in the `outdata` buffer, and `len` is set to the length of the decrypted data. The return value indicates the type of TLS record that was decrypted: RECORD_HANDSHAKE_DECRYPTED, RECORD_DECRYPTED, or RECORD_ALERT_DECRYPTED.

- If the TCP payload contains multiple TLS records (shown below):

  `nu_record_processor()` outputs the decrypted data from all the TLS records in the `outdata` buffer, and `len` is set to the length of the decrypted data. The return value is RECORD_HANDSHAKE_DECRYPTED, RECORD_DECRYPTED, or RECORD_ALERT_DECRYPTED.

- If the TCP payload contains one or more TLS records followed by the first fragment of the next TLS record (shown below):



`nu_record_processor()` outputs the decrypted data from all the TLS records in the `outdata` buffer, and `len` is set to the length of the decrypted data. The return value is RECORD_NO_ERROR, indicating that more packet payload inputs are required to complete the next TLS record.

- If the TCP payload contains a fragment of a TLS record which is not the last fragment (shown below):

`nu_record_processor()` sets the value of `len` to 0 and returns RECORD_NO_ERROR.

- If the TCP payload contains the last fragment of a TLS record and one or more TLS records (shown below):

`nu_record_processor()` outputs the decrypted data from all the TLS records in the `outdata` buffer, and `len` is set to the length of the decrypted data. The return value is RECORD_HANDSHAKE_DECRYPTED, RECORD_DECRYPTED, or RECORD_ALERT_DECRYPTED.

Note: A return value other than `RECORD_HANDSHAKE_DECRYPTED`, `RECORD_DECRYPTED`, `RECORD_ALERT_DECRYPTED` or `RECORD_NO_ERROR` is an indication that the library will not continue decrypting the TLS session. The caller can drop the session state returned from `nu_new_session()`.

7.1.5 Recovering from Lost Packets

The library attempts to recover lost packets for AES_128_GCM and AES_256_GCM. You need to notify the library Directionality matters. Your TCP processing logic should notify the library of the first loss in a consecutive set of lost packets. Since only complete TLS records can be decrypted, the library will wait for the next TLS record header after a loss is reported when attempting to recover from lost packets. Call `nu_lost_client_frame()` if frames from the client to the server were lost. Call `nu_lost_server_frame()` of frames from the server to the client were lost.

7.1.6 Cleanup

Before cleaning up a TCP session, you should ensure that `nu_session_cleanup()` is called in order to release session memory. Before stopping a thread, you should ensure that `nu_record_processor_cleanup()` is called in order to release memory.

7.2 Sample Code and Test Files

Users who have licensed the decryption library have access to a repository containing:

- The decryption library binary file.
- A sample test program that reads in PCAP files, passes TCP payloads to the library, which returns decrypted TLS records.
- Documented header files containing the function declarations and data structure definitions required for library initialization, session initialization, processing TLS records, notifying of lost packets, cleanup and license display.
- A Makefile to compile the program.
- A directory of test cases.
- A Readme file with instruction how to run performance and decryption validation tests.
- A file containing frequently asked questions.

The sample program shows how to pass TCP payloads to the decryption library. TCP records without a payload (SYN, ACK, SYN/ACK, RST, FIN) need not be passed to the library. Packets should be passed in order (see specifics in the FAQ section).

Note: A TCP packet could be quite large in environments where NICs coalesce packets to reduce the number of packets forwarded up the stack. This could be the case when the decryption library is used in out-of-band cases where the MTU can be large.

7.3 Frequently Asked Questions

Note: A more detailed set of questions is available in the repositories provided to organizations that have licensed Nubeva solutions.

Q: Is a complete TCP session required for decryption? The decryption library does not need to receive TCP packets that do not contain a payload such as SYN, SYN/ACK, plain ACK, RST, FIN. All other packets should be passed to the decryption library.

Q: Is it possible to decrypt application records without decrypting the handshake? Handshake information must be available. The encryption/decryption/MAC starts at the end of the TLS 1.2 handshake and in TLS 1.3, the handshake is mostly encrypted. The handshake information is required in order to decrypt the application data that follows the handshake.

Q: Does `nu_record_processor()` need TCP packets in the exact order? Packets containing TLS handshakes must be passed in the right order. For TLS 1.2 the handshake sequence is:

- Client → server: client hello
- Server → client: server hello, server certificate, server key exchange, server hello done
- Client → server: client key exchange, change cipher spec, client handshake finished
- Server → client: server change cipher spec, server handshake finished.

For TLS 1.3 the sequence is:

- Client → server: client hello
- Server → client: server hello, change cipher spec, certificate, handshake finished
- Client → server: change cipher spec, handshake finished

The caller does not need to parse these records. Once the handshake is complete the order needs to be maintained in each direction.

Q: What should the size of the decryption buffer be? The decrypted buffer length must be at least 17408 (max TLS record length) + size of the input length.

Q: How does the decryption library match a TLS session with a master key? There is a callback function pointer that is set by the caller when the library is initialized. The library calls this function with a client-random value contained in client-hello messages.

Q: Do packets need to be buffered until keys are available? You should not have to buffer packets when using Nubeva Sensors to receive keys. Nubeva Sensors send keys within 200usec of key creation. Therefore keys get to the destination well before anything needs to be decrypted.

Q: What happens if the key lookup function fails to find a key? The nu_record_processor will return as error status indicating that a key was not found.

Q: Does the library recover from lost packets? The library currently attempts to recover lost packets for AES_128_GCM and AES_256_GCM provided that not more than 128 packets were lost. There is a function to notify the library of lost packets from a client to the server and a function to notify the library of lost packets from the server to the client.

Q: Where can I find detailed information about TLS records? These are two good references: TLS 1.2 and TLS 1.3.

CHAPTER 8

Supported Ciphers

```
"TLSv1.3"
TLS_AES_256_GCM_SHA384          Kx=any          Au=Any  Enc=AESGCM(256) Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256    Kx=any          Au=Any  Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
TLS_AES_128_GCM_SHA256          Kx=Any          Au=Any  Enc=AESGCM(128) Mac=AEAD

"TLSv1.2"
ECDHE-ECDSA-AES256-GCM-SHA384   Kx=ECDH         Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384     Kx=ECDH         Au=RSA  Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-GCM-SHA384       Kx=DH           Au=RSA  Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-CHACHA20-POLY1305   Kx=ECDH         Au=ECDSA Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
ECDHE-RSA-CHACHA20-POLY1305     Kx=ECDH         Au=RSA  Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
DHE-RSA-CHACHA20-POLY1305       Kx=DH           Au=RSA  Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
ECDHE-ECDSA-AES128-GCM-SHA256   Kx=ECDH         Au=ECDSA Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES128-GCM-SHA256     Kx=ECDH         Au=RSA  Enc=AESGCM(128) Mac=AEAD
DHE-RSA-AES128-GCM-SHA256       Kx=DH           Au=RSA  Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES256-SHA384       Kx=ECDH         Au=ECDSA Enc=AES(256)  Mac=SHA384
ECDHE-RSA-AES256-SHA384         Kx=ECDH         Au=RSA  Enc=AES(256)  Mac=SHA384
DHE-RSA-AES256-SHA256           Kx=DH           Au=RSA  Enc=AES(256)  Mac=SHA256
ECDHE-ECDSA-AES128-SHA256       Kx=ECDH         Au=ECDSA Enc=AES(128)  Mac=SHA256
ECDHE-RSA-AES128-SHA256         Kx=ECDH         Au=RSA  Enc=AES(128)  Mac=SHA256
DHE-RSA-AES128-SHA256           Kx=DH           Au=RSA  Enc=AES(128)  Mac=SHA256
AES256-GCM-SHA384               Kx=RSA          Au=RSA  Enc=AESGCM(256) Mac=AEAD
AES128-GCM-SHA256               Kx=RSA          Au=RSA  Enc=AESGCM(128) Mac=AEAD
AES256-SHA256                   Kx=RSA          Au=RSA  Enc=AES(256)  Mac=SHA256
AES128-SHA256                   Kx=RSA          Au=RSA  Enc=AES(128)  Mac=SHA256

ECDHE-PSK-CHACHA20-POLY1305     Kx=ECDHEPSK     Au=PSK  Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
RSA-PSK-AES256-GCM-SHA384       Kx=RSAPSK       Au=RSA  Enc=AESGCM(256) Mac=AEAD
DHE-PSK-AES256-GCM-SHA384       Kx=DHEPSK       Au=PSK  Enc=AESGCM(256) Mac=AEAD
```

(continues on next page)

(continued from previous page)

```
RSA-PSK-CHACHA20-POLY1305       Kx=RSAPSK       Au=RSA  Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
DHE-PSK-CHACHA20-POLY1305       Kx=DHEPSK       Au=PSK  Enc=CHACHA20/POLY1305(256)␣
↪Mac=AEAD
RSA-PSK-AES128-GCM-SHA256       Kx=RSAPSK       Au=RSA  Enc=AESGCM(128) Mac=AEAD
DHE-PSK-AES128-GCM-SHA256       Kx=DHEPSK       Au=PSK  Enc=AESGCM(128) Mac=AEAD
```

CHAPTER 9

FastKey Protocol

FastKey combines a binary protocol over DTLS and a REST API to send keys to key targets. The rest API uses a JSON object. The object is a key:value pair, where the key is a 'client random' and the value is a set of key fields and a meta-data structure. Meta-data data is not required for decryption.

The structure is the same for TLS 1.2 and TLS 1.3 keys. Fields are populated based on the TLS protocol type. The names of the fields match the specification. The JSON object uses the following abbreviations:

- **MK:** master key - this field is populated when the type is "TLS 1.2"
- **CETS:** client early traffic secret - TLS 1.3
- **CHTS:** client handshake traffic secret - TLS 1.3
- **SHTS:** server handshake traffic secret - TLS 1.3
- **CTS0:** client traffic secret 0 - TLS 1.3
- **STS0:** server traffic secret 0 - TLS 1.3
- **XS:** exported secret - TLS 1.3

The figure below depicts a TLS 1.2 key object


```
{
  "CETS": "",
  "CHTS": "",
  "CR": "07ad5a5d9745c73e599e2147c7bb471249ed427a209dae6b77a807e898b9e5ec",
  "CTS0": "",
  "LastUsed": "2020-06-10T23:59:58",
  "MD": {
        "command": "curl",
        "hostname": "ip-172-31-5-150.ec2.internal",
        "instance": "i-05f1e99c1cd9e3b28",
        "lib": "osl",
        "pid": 22909
  },
  "MK":
↪"8b48f8a3f80ee7bbf26166d20913ce5ff068924d23b79199adc1babc4385c084c9cccc143ffa19963db60e010157fe33
↪",
```

(continues on next page)

(continued from previous page)


```
  "SHTS": "",
  "STS0": "",
  "Type": "1.2",
  "XS": ""
}
```


The following figure depicts a TLS 1.3 key object


```
{
 "CETS": "",
 "CHTS":
↪"34110bb15d8fad38f0e2cda16cebe43e5f30cf60066ab04bf6cabf9553b175c6e2d1a3005b1d1c5527d9d1ad9a750679
↪",
 "CR": "01fc0baa6eca082096d69f047e232ed762ba317b1e7392178ca8c2579c73c464",
 "CTS0":
↪"1bfa4c5d2351a746fb4472f5ca0276c81864ac613f994b06570431c1751f3aa88a4a67a66375a403dc9792e913b830b8
↪",
 "LastUsed": "2020-06-11T00:01:43",
 "MD": {"command": "curl",
     "hostname": "ip-172-31-5-150.ec2.internal",
     "instance": "i-05f1e99c1cd9e3b28",
     "lib": "osl",
     "pid": 23311},
 "MK": "",
 "SHTS":
↪"a86ae64b0fe1bd9e065125768251fd279089d0e544e2200f2d3df87edc2692d45befce649fac006d0fab153f2365a41a
↪",
 "STS0":
↪"4c84bdae94562490dc2371cf32a48489fe03e89f7c464efae93afdc1df522d774b875ffabf95ce8e35cdf9b89773855a
↪",
 "Type": "1.3",
 "XS":
↪"7dce5eec25e6e48a4f0a6bad9ece783fe7ef208d2508a1112b9a074d000e8cc9425ff2d59ac99b33715c2bdf98547510
↪"
}
```


Two communication channels are used when sensors send keys to a DTLS key target. A control channel uses a REST API to send keys. A low latency protocol uses DTLS to send a binary representation of key records. The structure of the binary format is the following:

- **Byte 0:** Version: a running counter of Nubeva's low latency protocol version. The current version is 0x2. This is the same number returned with the bearer token.
- **Byte 1: Type: indicates what function is requested:**
  - 0xC8 (200) = putkey TLS 1.2
  - 0xCB (203) = putkey TLS 1.3 32B key
  - 0xCC (204) = putkey TLS 1.3 48B key
- **Bytes 2-3:** Protocol Version: 0303 = TLS 1.2, 0304 = TLS 1.3
- **Bytes 4-5:** Length: indicates the length of the keys + the length of the bearer token (64 + 32 + 48 + 384 = 528).
- **Bytes 6-69:** Bearer-token (64 bytes)
- **Bytes 70-101:** Client random (32 bytes)
- **Bytes 102-149:** Master key (48 bytes). 0's if 'protocol version' is 0304

- **Bytes 150-213:** CETS (client early traffic secret) 0's if 'protocol version' is 0303
- **Bytes 214-277:** CHTS (client handshake traffic secret) 0's if 'protocol version' is 0303
- **Bytes 278-341:** SHTS (server handshake traffic secret) 0's if 'protocol version' is 0303
- **Bytes 342-405:** CTSZ (client traffic secret 0) 0's if 'protocol version' is 0303
- **Bytes 406-469:** STSZ (server traffic secret 0) 0's if 'protocol version' is 0303
- **Bytes 470-533:** XS (exported secret) 0's if 'protocol version' is 0303

Note: If the Byte 0 (version) is 0x1 then the Byte 1 (function) is 0xC0 (putkey). Key length indication was added in version 2.

Sample code that shows how to receive REST API and UDP messages for storing keys is available at https://nubevalabs.s3.amazonaws.com/dtlskeydb/dtlskeydb.py. Instructions to run the script are provided in the `DTLS Key DB` section at the bottom of the previous page.

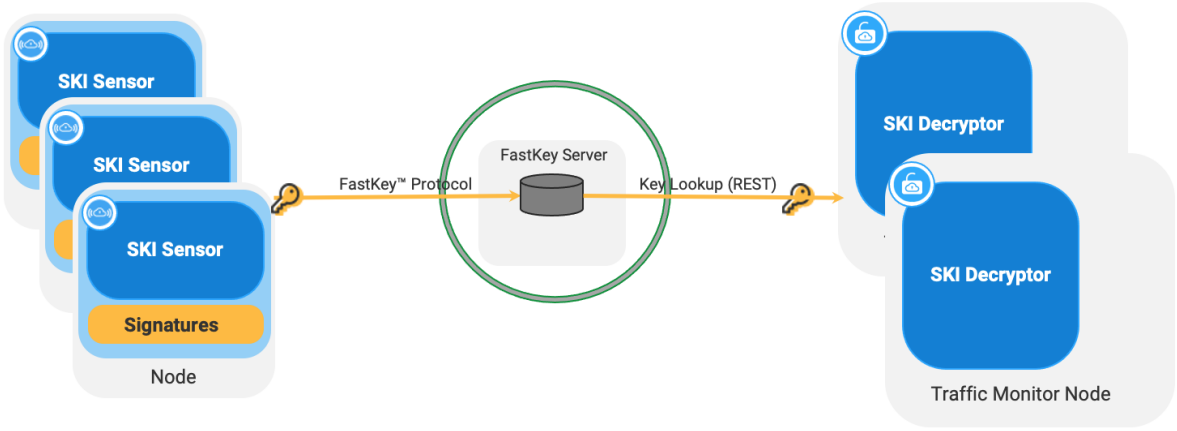
Note: Since the client random value is used as the unique key for looking up session master keys, it is possible to store the same master key more than once. This assures that TLS session renewals are supported as well.

CHAPTER 10

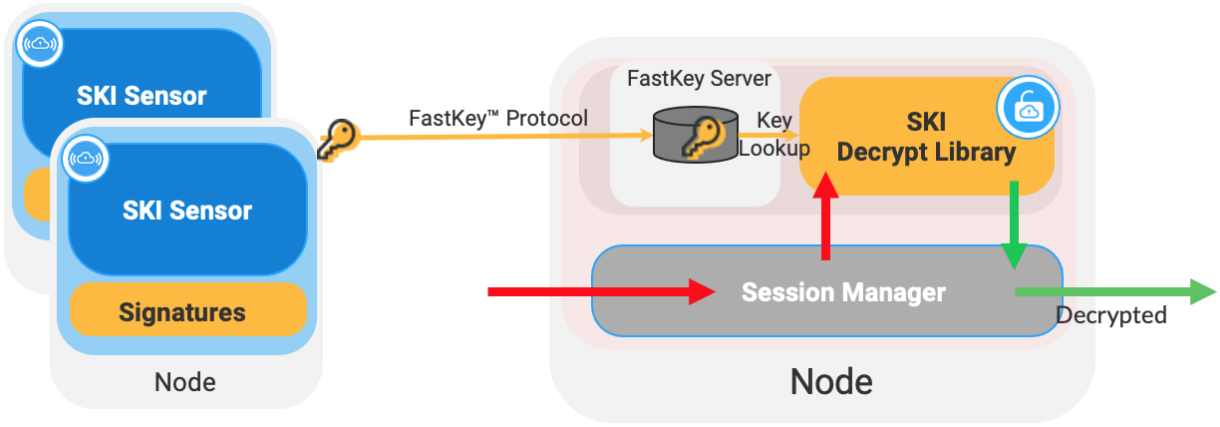
FastKey Buffer

The `Fastkey Buffer` maintains a mapping between session secrets and client random values. The FastKey Server receives session secrets from *SKI Sensors* using the *FastKey Protocol*, and buffers the keys in memory for a configurable amount of time. When session keys expire, their memory is set to 0 and then freed. The Key Server exposes an REST API call to lookup keys based on the value of the `Client Random` which is when a new session is created or a session is restarted.

FastKey Servers can be used in multiple use cases. The figure below depicts a FastKey Server receiving keys from multiple SKI Sensors, and returning keys to multiple *SKI Decryptors*.



The following figure shows a FastKey Server receiving keys from multiple sensors and returning keys to a *SKI Decryption Library*.



10.1 Deploying a FastKey Buffer

To launch a FastKey Buffer run the following command:

```
docker run --name nubeva-ks -d -p4433:4433 -p4433:4433/udp  \
-v $(pwd)/../certs:/certs nubeva/fastkey \
--cert /certs/yourcert.pem --key /certs/yourkey.pem
```

Note: You should adjust the mounted directory `-v $(pwd)../certs` to match your certs directory, and `yourcert.pem` and `yourkey.pem` files to match your cetificiate and key file names.

The instance running the FastKey Server must resolve your server DNS to `127.0.0.1` (usually with /etc/hosts).

10.2 Command Line Parameters

- **bind** <addr> : Address to bind to. Default 0.0.0.0
- **port** <port> : Port to use (both TCP & UDP). Default 4433
- `cert` <file> : Server PEM-encoded X.509 Certificate chain file
- `key` <file> : Server PEM-encoded Private Key file
- **client-timeout** <n> : Timeout in seconds before client disconnects, default 60
- **key-timeout** <n> : Timeout in seconds before a key is forgotten, default 300
- **max-key-count** <n> : Maximum number of keys to keep. Default 32768
- **ciphers** <str> : Accepted SSL Ciphers to use
- **enable-protocol** <p> : Enable one of the following protocols: SSLv3, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3 or ALL
- **disable-protocol** <p> : Disable one of the following protocols: SSLv3, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3 or ALL

Note: You do not need to launch a FastKey server if you have your own key management solution that implements the FastKey protocol.

10.3 API Calls

If you use Nubeva's demo certificate the the FastKey Server must resolve to `key.nubedge.com`. The following API calls are supported:

- **stats**: returns server statistics

```
# Call stats
curl https://[server DNS or IP]:4433/stats

# Returns
{
"cur_size":175,
"get_req_count":0,
"put_req_count":602,
"put_from_dtls":301,
"unneeded_put_from_dtls":1,
"put_from_tcp":301,
"unneeded_put_from_tcp":299,
"get_success_count":0,
"get_fail_count":0,
"timedout_keys":127,
"ssl_error":0,
"ssl_error_other":1,
"dtls_zero_read":0,
"dtls_invalid_version":0,
"dtls_invalid_len":0,
"dtls_keepalive":2,
"dtls_unknown_type":0,
"dtls_unknown_size":0,
"dtls_client_free":0,
"dtls_client_prepare":2,
"key_timeout_checks":3,
"client_timeout_checks":18,
"client_timeouts":0,
"dtls_packets":306,
"dtls_packet_bytes":172548
}
```

- **dumpkeys** dumps the contents of the key buffer

```
# Call dumpkeys

curl https://[server DNS or IP]:4433/stats

# Returns
f98cde0d83171d5cb6e53a1d3894b49d3757687a0b7e0aacffdc236d26229c53 (1s): {
"Type":"1.3",
"CR":"f98cde0d83171d5cb6e53a1d3894b49d3757687a0b7e0aacffdc236d26229c53",
"CHTS":
↪"ffa69c63527f2a8fe1a9dc974bf0c9c841283ebe28278d27d4a32f888bf91eff2b7d66b490954843b458c386ccac90e6
↪",
```

(continues on next page)

(continued from previous page)

```
"SHTS":
↪"e1fbea741bebc77786bee83b571a463476bd0a0daf0af0f39ca122b0032631a50fe3987ef6b619aba21216c5ad3b2349
↪",
"CTS0":
↪"5f1f76bed96460469b52d31c8c2539030d3f35b41a67f476cc277473787efd35b4598ef515ca777157a18786e59a5d85
↪",
"STS0":
↪"873180754f348e964060fc8b76292118cf4b535880bd9be94fa105b1f1c133f6b0366a2fcd88b666f545e00898af7698
↪",
"XS":
↪"8aab03b0bb908fb829ced89bb233255203995fb03151ede766dd737b298186fa1ed83016e13684473bb6bc5120452be5
↪"
}
faf7ab15ba9c1872dc92ec7d17607e7f13844afb9beed83b28615cd7647f4ccd (1s): {
"Type":"1.3",
"CR":"faf7ab15ba9c1872dc92ec7d17607e7f13844afb9beed83b28615cd7647f4ccd",
"CHTS":
↪"feb0418af64c0c8fc1059389a8e50e10e124092196c2b2781a012274ab9eea5f4625f1037f329bc983c5b30cba1eb6f9
↪",
"SHTS":
↪"d2611d319b2e43523098d9b87339ebca574d5b5d397dab3a74ef3f7f6e21b12472903e88ea364aea75a72b39b276271f
↪",
"CTS0":
↪"41beb5c8ea2c0fa073722e3addd4b39aaaf9599f309f6192df710e03417d198081a54da135d5d2a2c3f454a74619ba8c
↪",
"STS0":
↪"9e75b7418d3877548556e64b2079bba42c11ad1563add1002c551b4d63fb50a6cc3ab7c4f0429904d65fd84071587a7b
↪",
"XS":
↪"50cc7fa146b61476a7a493162b76e95c64d8256cf54a342f34eab983362a8b435870b4bd5fbef33b6bf7edf995e77278
↪"
}
```

- **flush_keys** clears the key buffer

```
# Call flush_keys
curl https://[server DNS or IP]:4433/flush_keys

# No return value
```

CHAPTER 11

Key Buffer Example

Nubeva provides a Python script running a Flask application server that implements a DTLS key target and buffer for receiving and storing keys in memory.

Note: The Key Buffer script is provided for illustrating the use of the APIs. You may modify this code any way you like, or use it as a reference to build your own services.

11.1 Run the Server

1. Add DNS resolution of `key.nubedge.com` to 127.0.0.1 to /etc/hosts on the Key Server instance.
2. Run the API Server container:

```
docker run -p 4443:4443/TCP -p 4443:4443/UDP --name nubeva-ks -dit  nubevalab/nubeva-
↪ks
```

1. Add DNS resolution of `key.nubedge.com` to point to the IP address of the Key Server to instances running SKI sensors and SKI Decryptors.
2. If you would like to configure SKI Sensors or SKI Decryptors from the Key Server, set the `baseurl` to `https://key.nubedge.com:4443/`
3. If you would like SKI Sensors to send session keys to the Key Server, set the `sslcredobj` to point to the API server as described in *Sensor Command Line Parameters*. Do the same for SKI Decryptors as described in *Decryptor Command Line Parameters*.

11.2 Server Script

You can inspect the contents of the Key Server container by running:

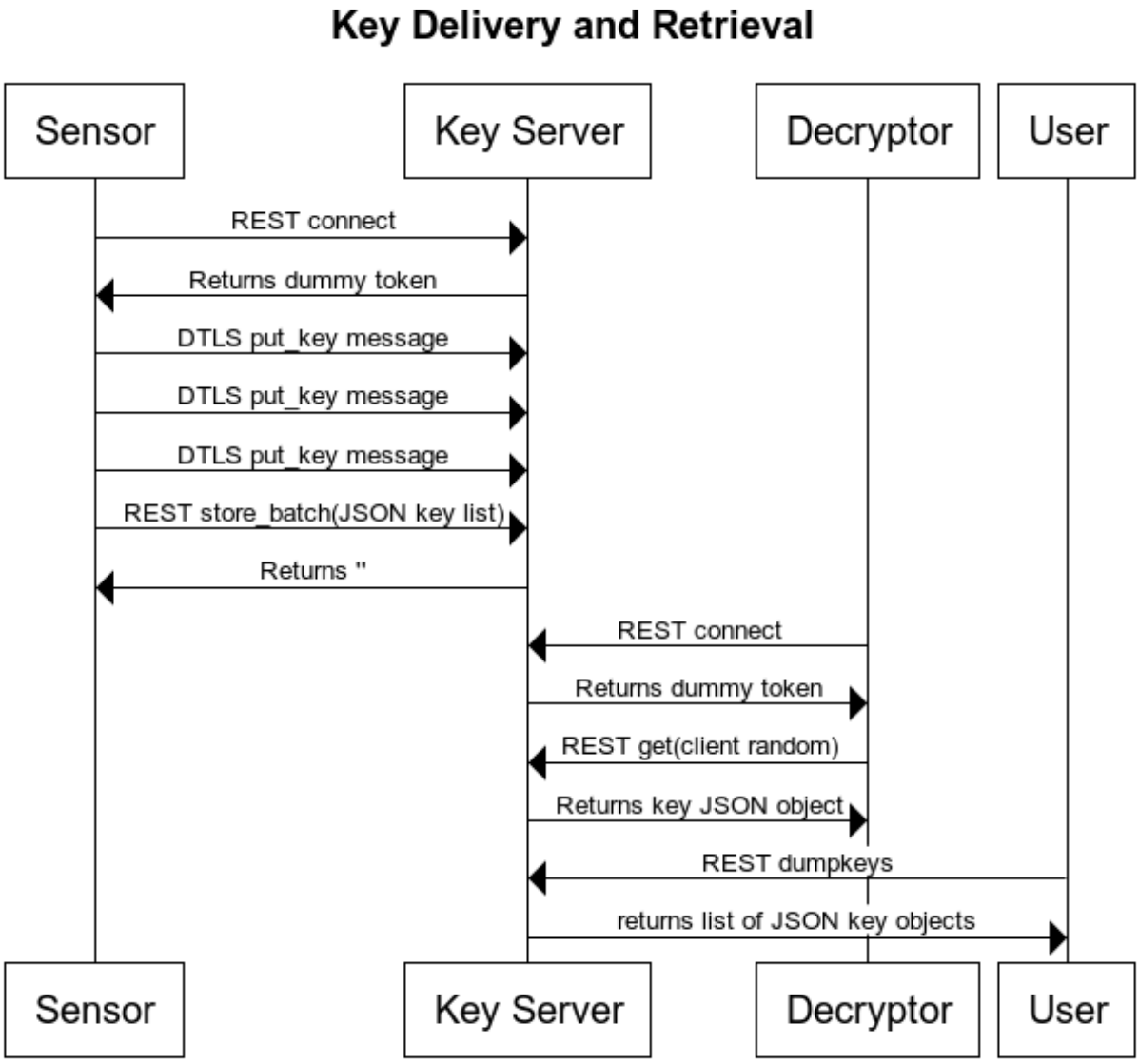
```
docker exec -it nubeva-ks bash
```

11.2.1 Input Parameters

The container runs the script using default parameter values:

- **ca_path:** Path to CA cert. Linux Default: `/opt/nubevaTools/ca`. Windows Default: `C:\\Users\\`
- **certs_path:** Path to certs. Linux Default: `/opt/nubevaTools/certs`. Windows Default: `C:\\Users\\`
- **certs_name:** Name for .ca, .key, .pem files. The default is `nubedge`: `nubedge.ca`, `nubedge.key`, `nubedge.pem`
- **nssfile_path:** Path to nsslogfile including filename e.g. `/host/home/ec2-user/keys.log`

11.2.2 FastKey Protocol API



The scripts runs a thread to process process *FastKey Protocol* DTLS messages. The DTLS handler function is `dtls`, which calls `readdtls` to process each message.

dtls()

Sets up the DTLS channel and listens for DTLS messages.

- **Parameters:** **nssfile_path:** passed from main()

  ca_path: passed main()

  certs_path: passed from main()

  certs_name: passed from main()

readdtls()

Processes a DTLS message, stores the key object in memory and if specified, writes keys to the key log file.

- **Parameters:** **nssfile_path:** passed in from dtls()

  conn: DTLS connection object passed dtls()

storebatch()

REST backup call for passing keys. Processes a list of JSON key objects. Stores each key in memory and if specified, writes keys to the key log file.

get()

REST key lookup based on a client-random value.

dumpkeys()

REST key dump helper function to test that keys are received and stored.

CHAPTER 12

Tools

12.1 Traffic Generator

You can add a traffic generator container to the source, however this is not required. This is Nubeva's standard traffic generator container. A container runs for approximately 60-120 seconds. You may use a cron job to run a container every minute. The actual docker command to run this generator once is:

```
docker run -dti nubevalab/tlsgenerator
```

You can check that Fast Key DB is receiving keys accessing the URL below from a browser:

```
https://key.nubedge.com:4433/dumpkeys
```

You may also run the generator as a script using the following code:

```
#!/bin/bash

# Sample sources of TLS traffic

while true; do
aws iam get-user  --output json
sleep 5
aws ec2 describe-vpcs --output json --region us-east-1
sleep 5
#Grab EICAR first as binary then as text
curl --output /dev/null https://secure.eicar.org/eicar.com
sleep 5
#TLS version of TestmyIDS.com
curl --output /dev/null https://nubevalabs.s3.amazonaws.com/testmyids.txt
sleep 5
#Download Google Homepage via TLS
curl --tlsv1.3 --output /dev/null https://www.google.com
sleep 5
#Download ESPN Homepage via TLS
```

(continues on next page)

(continued from previous page)

```
curl --output /dev/null https://www.bbc.com
sleep 5
done
```

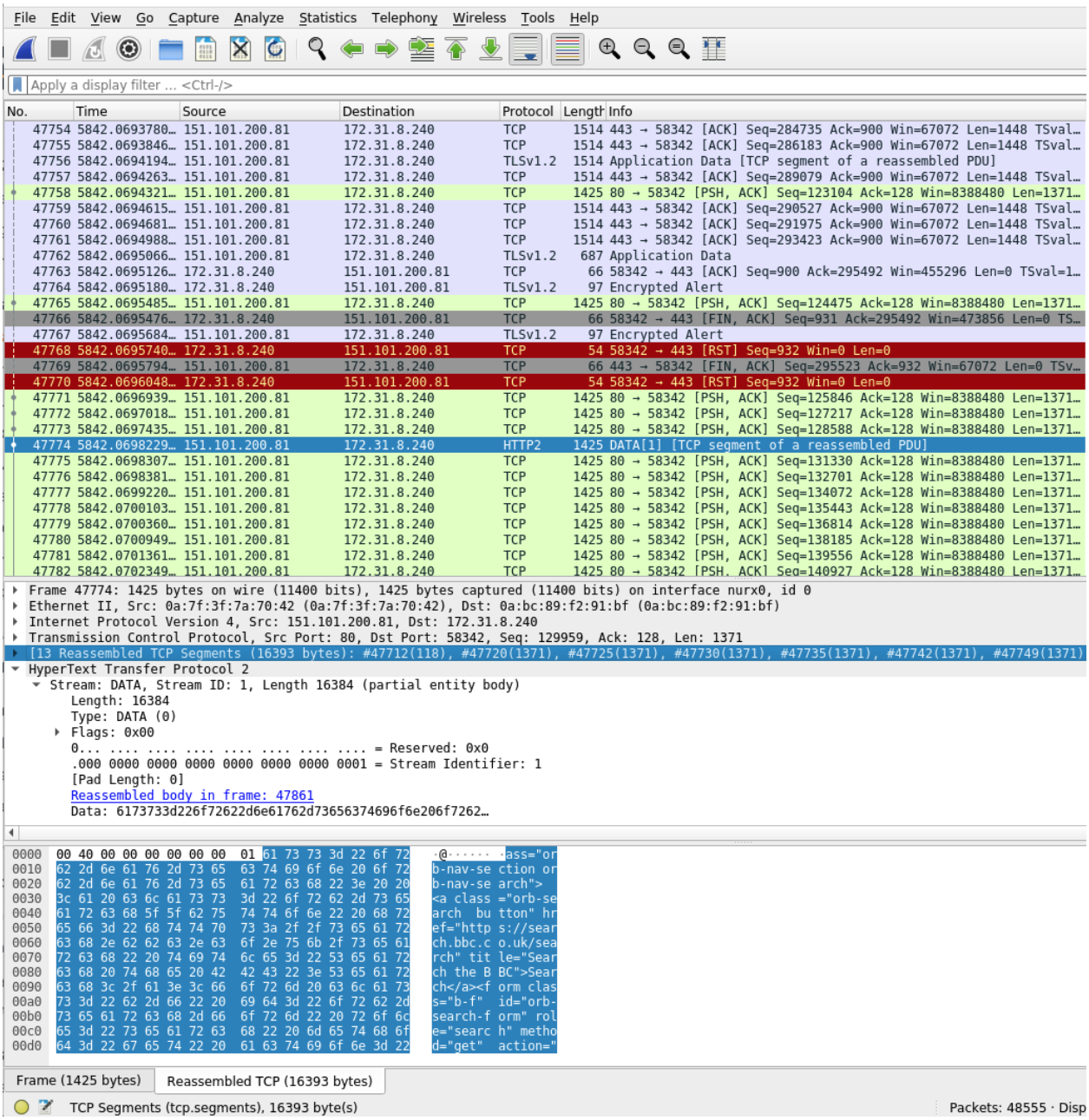
12.2 Wireshark

A containerized version of Wireshark can be deployed using the following command:

```
docker run -v /tmp:/keys -p 14500:14500 --restart unless-stopped -dti --cap-add NET_
↪ADMIN --net=host --name wireshark  ffeldhaus/wireshark
```

The default WireShark credentials are `wireshark`, `wireshark`.

Set Wireshark to monitor the `nurx0` interface. You will be able to see encrypted and decrypted traffic:


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
No. Time Source Destination Protocol Length Info
47754 5842.0693780… 151.101.200.81 172.31.8.240 TCP 1514 443 → 58342 [ACK] Seq=284735 Ack=900 Win=67072 Len=1448 TSval…
47755 5842.0693846… 151.101.200.81 172.31.8.240 TCP 1514 443 → 58342 [ACK] Seq=286183 Ack=900 Win=67072 Len=1448 TSval…
47756 5842.0694194… 151.101.200.81 172.31.8.240 TLSv1.2 1514 Application Data [TCP segment of a reassembled PDU]
47757 5842.0694263… 151.101.200.81 172.31.8.240 TCP 1514 443 → 58342 [ACK] Seq=289079 Ack=900 Win=67072 Len=1448 TSval…
47758 5842.0694321… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=123104 Ack=128 Win=8388480 Len=1371…
47759 5842.0694615… 151.101.200.81 172.31.8.240 TCP 1514 443 → 58342 [ACK] Seq=290527 Ack=900 Win=67072 Len=1448 TSval…
47760 5842.0694681… 151.101.200.81 172.31.8.240 TCP 1514 443 → 58342 [ACK] Seq=291975 Ack=900 Win=67072 Len=1448 TSval…
47761 5842.0694988… 151.101.200.81 172.31.8.240 TCP 1514 443 → 58342 [ACK] Seq=293423 Ack=900 Win=67072 Len=1448 TSval…
47762 5842.0695066… 151.101.200.81 172.31.8.240 TLSv1.2 687 Application Data
47763 5842.0695126… 172.31.8.240 151.101.200.81 TCP 66 58342 → 443 [ACK] Seq=900 Ack=295492 Win=455296 Len=0 TSval=1…
47764 5842.0695180… 172.31.8.240 151.101.200.81 TLSv1.2 97 Encrypted Alert
47765 5842.0695485… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=124475 Ack=128 Win=8388480 Len=1371…
47766 5842.0695476… 172.31.8.240 151.101.200.81 TCP 66 58342 → 443 [FIN, ACK] Seq=931 Ack=295492 Win=473856 Len=0 TS…
47767 5842.0695684… 151.101.200.81 172.31.8.240 TLSv1.2 97 Encrypted Alert
47768 5842.0695740… 172.31.8.240 151.101.200.81 TCP 54 58342 → 443 [RST] Seq=932 Win=0 Len=0
47769 5842.0695794… 151.101.200.81 172.31.8.240 TCP 66 443 → 58342 [FIN, ACK] Seq=295523 Ack=932 Win=67072 Len=0 TSv…
47770 5842.0696048… 172.31.8.240 151.101.200.81 TCP 54 58342 → 443 [RST] Seq=932 Win=0 Len=0
47771 5842.0696939… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=125846 Ack=128 Win=8388480 Len=1371…
47772 5842.0697018… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=127217 Ack=128 Win=8388480 Len=1371…
47773 5842.0697435… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=128588 Ack=128 Win=8388480 Len=1371…
47774 5842.0698229… 151.101.200.81 172.31.8.240 HTTP2 1425 DATA[1] [TCP segment of a reassembled PDU]
47775 5842.0698307… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=131330 Ack=128 Win=8388480 Len=1371…
47776 5842.0698381… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=132701 Ack=128 Win=8388480 Len=1371…
47777 5842.0699220… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=134072 Ack=128 Win=8388480 Len=1371…
47778 5842.0700103… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=135443 Ack=128 Win=8388480 Len=1371…
47779 5842.0700360… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=136814 Ack=128 Win=8388480 Len=1371…
47780 5842.0700949… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=138185 Ack=128 Win=8388480 Len=1371…
47781 5842.0701361… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=139556 Ack=128 Win=8388480 Len=1371…
47782 5842.0702349… 151.101.200.81 172.31.8.240 TCP 1425 80 → 58342 [PSH, ACK] Seq=140927 Ack=128 Win=8388480 Len=1371…
Frame 47774: 1425 bytes on wire (11400 bits), 1425 bytes captured (11400 bits) on interface nurx0, id 0
Ethernet II, Src: 0a:7f:3f:7a:70:42 (0a:7f:3f:7a:70:42), Dst: 0a:bc:89:f2:91:bf (0a:bc:89:f2:91:bf)
Internet Protocol Version 4, Src: 151.101.200.81, Dst: 172.31.8.240
Transmission Control Protocol, Src Port: 80, Dst Port: 58342, Seq: 129959, Ack: 128, Len: 1371
[13 Reassembled TCP Segments (16393 bytes): #47712(118), #47720(1371), #47725(1371), #47730(1371), #47735(1371), #47742(1371), #47749(1371)
HyperText Transfer Protocol 2
Stream: DATA, Stream ID: 1, Length 16384 (partial entity body)
Length: 16384
Type: DATA (0)
Flags: 0x00
0... .... = Reserved: 0x0
.000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
[Pad Length: 0]
Reassembled body in frame: 47861
Data: 6173733d226f72622d6e61762d73656374696f6e206f7262…
0000 00 40 00 00 00 00 00 00 01 61 73 73 3d 22 6f 72 ·@····· ·ass="or
0010 62 2d 6e 61 76 2d 73 65 63 74 69 6f 6e 20 6f 72 b-nav-se ction or
0020 62 2d 6e 61 76 2d 73 65 61 72 63 68 22 3e 20 20 b-nav-se arch">
0030 3c 61 20 63 6c 61 73 73 3d 22 6f 72 62 2d 73 65 <a class ="orb-se
0040 61 72 63 68 5f 5f 62 75 74 74 6f 6e 22 20 68 72 arch__bu tton" hr
0050 65 66 3d 22 68 74 74 70 73 3a 2f 2f 73 65 61 72 ef="http s://sear
0060 63 68 2e 62 62 63 2e 63 6f 2e 75 6b 2f 73 65 61 ch.bbc.c o.uk/sea
0070 72 63 68 22 20 74 69 74 6c 65 3d 22 53 65 61 72 rch" tit le="Sear
0080 63 68 20 74 68 65 20 42 42 43 22 3e 53 65 61 72 ch the B BC">Sear
0090 63 68 3c 2f 61 3e 3c 66 6f 72 6d 20 63 6c 61 73 ch<f orm clas
00a0 73 3d 22 62 2d 66 22 20 69 64 3d 22 6f 72 62 2d s="b-f" id="orb-
00b0 73 65 61 72 63 68 2d 66 6f 72 6d 22 20 72 6f 6c search-f orm" rol
00c0 65 3d 22 73 65 61 72 63 68 22 20 6d 65 74 68 6f e="searc h" metho
00d0 64 3d 22 67 65 74 22 20 61 63 74 69 6f 6e 3d 22 d="get" action="
Frame (1425 bytes)
Reassembled TCP (16393 bytes)
TCP Segments (tcp.segments), 16393 byte(s)
Packets: 48555 · Disp

CHAPTER 13

Release Notes

13.1 Nubeva Sensor Release Notes

13.1.1 Jan-2021

Features

- Key Sense
- FastKey key length indication
- Configurable TCP and UDP port

13.1.2 Sep-2021

Features

- FastKey Protocol

13.2 Nubeva Decryptor Release Notes

13.2.1 Oct-2020

13.3 Decryption Library Release Notes

13.3.1 Jan-2021

Features

- FastKey protocol key length indicator for TLS 1.3

13.3.2 Oct-2020

Features

- TLS 1.2 PFS and TLS 1.3 decryption
- Over 10Gbps per thread decryption throughput
- TLS parsing and assembly
- Recovery from packet loss for AES-128-GCM and AES-256-GCM ciphers

13.4 FastKey Server Release Notes

13.4.1 Mar-2021

Features

- C Library
- Session key TTL
- Configurable key buffer size

13.4.2 Jan-2021

Features

- Python/Flask
- Sensor configuration REST API
- Decryptor configuration REST API
- Session secret read/write REST API
- FastKey protocol

CHAPTER 14

Frequently Asked Questions

Question: What measures & research has Nubeva undertaken to ensure that your products continue to operate given the following challenges:

- OS vendor changes
- Compliance issues
- Interaction with AV/Malware/EDR products
- TPM-type solutions
- Protocol updates, changes and additions

14.1 OS Vendor Changes

Today on Microsoft Windows, Nubeva uses 'hooking' for Symmetric Key Intercept. That means we act, with permission, in the user space application process and memory space. So whatever mechanism segmentation functions, OS or hardware, Nubeva acts from the user-space process perspective to get the memory segment that has the keys.

For almost twenty years, 'hooking' has been licensed by hundreds of ISVs, used by nearly every product team at Microsoft, and is a generally accepted method.

Microsoft Windows is moving towards tracing, having joined Open DTrace. Windows itself had kernel tracing mechanisms such as ETW already. But is now choosing to invest further into more advanced tracing methods such as DTrace. Once tracing mechanisms are widely available in all Windows distributions, Nubeva looks forward to using tracing instead of hooking.

Nubeva already uses tracing inside the Linux operating system to access process memory. Just like in Windows hooking, Nubeva accesses the memory just as the process does. The biggest difference with tracing is that the tracing works directly in the operating system and is first validated as safe, unobtrusive, and not overly complex by the OS kernel. This method is safer and more performant than hooking. That is why it's expected that all operating systems are likely to evolve to support similar systems.

14.2 Government/Compliance Issues

Nubeva has been classified as EAR99. Software classified as EAR99 does not require any additional licenses for export. Nubeva Sensors do not participate in encryption processes in any way. Therefore there are no additional requirements or considerations required for export.

14.3 Interaction with AV/Malware/EDR Products

Nubeva sensors do not trigger an alert from products such as Windows defenders and other AV/Malware/EDR products in their default form. However, if the products change from their defaults and have all alerts turned on, then some will detect Nubeva's hooking. They all have an easy method to add the Nubeva process to the "allowed processes list". This requirement goes hand in glove with the concept that Nubeva is a solution the must be explicitly given permissions to operate by administrators.

14.4 TPM-type solutions

Trusted Platform Modules (TPM) are used to secure long-term asymmetric encryption and authentication secrets, such as certificates, private keys, and passwords. Microsoft Pluton further secures TPMs to protect the communication between the TPM and CPU so it cannot be intercepted. Neither TPM nor Pluton affects Nubeva. Nubeva looks for ephemeral keys from the user space process memory. The Nubeva process does not depend on any asymmetric keys nor the TPM architecture. By acting exactly as the user process would (DTrace on Linux, hooking in Windows), Nubeva can access the key from memory exactly as the user process does. The user process ultimately needs to access the symmetric key given that TLS is an application encryption protocol. Not network layer protocol like IPSEC. So it must have access to symmetric keys

14.5 Protocol Updates, Changes & Additions

Nubeva operations have a complete set of automated tests to detect new versions of software, protocols, and applications. As new versions are detected, Nubeva automatically creates new signatures and tests for viability. If key extraction is successful, then the new signatures are automatically pushed to the master repo as well as to all partners. If key extraction is not successful, then the Nubeva R&D team manually creates the new signatures and extends the testing process. The automated signature creation takes 60-90 minutes to complete and if manual intervention is required, another 2-4 hours is required on average for a new signature.

For applications and libraries that we have source code access to, for instance, open-source, it generally takes us a day or two to apply our intellectual property to extract the keys. Nubeva also tests the preview version of software to ensure we can support them at launch.