Clear Key Encryption with IBM WebSphere Liberty on Linux on Z

Hybrid Cloud Integration Test Team

March 21, 2023

Frank Mitarotonda (fpmitaro@us.ibm.com)

Who are we?

IBM Hybrid Cloud Integration Test (HCIT) is an effort within IBM Systems to build and maintain a customer like software environment running multiple application stacks on the LinuxONE server platform. The objective is to find defects across IBM and open-source software as well as LinuxONE and IBM Storage firmware, which can only be found when running a complex software configuration for a long stretch of time. This now includes software stacks running on top of RedHat OpenShift Container Platform (OCP) and some potential Independent Software Vendor products (ISVs). This document focuses on the implementation of clear key encryption in a Linux on Z environment running IBM Java 8.

Introduction

Encryption is a coveted feature that Linux on Z customers want to take advantage of and implement in their production environments. The IBM PCIE crypto adapter supports three different encryption modes, Accelerator, Co-processor, and EP11 mode. Accelerator mode makes use of clear key encryption, meaning the cryptographic adapter will perform RSA asymmetric encryption operations while the CPU handles all other symmetric encryption operations such as, AES and DES. In addition, the keys responsible for encryption are stored in plain text on the Linux OS in memory, hence the term "clear key", and do not reside on the cryptographic adapter. This report will specifically be focusing on clear encryption and go into detail about how to configure a Linux on System Z environment to take advantage of clear key cryptographic acceleration. The goal of this document is to highlight the performance metrics gathered from driving clear key encryption in my team's production environment and emphasis the configuration that produced the best performance. Ultimately, this information aims to aid in providing a recommended solution for any future clear key encryption users.

Prerequisites

- Running Linux on Z
- IBM Java 8 Installed
- IBM WebSphere Liberty Installed
- Cryptographic Accelerator card `(CEXxA)` attached to machine
- `z90crypt` or AP module is loaded
- `openCryptoki`
- `libica`

Environment

The majority of testing conducted in this report was performed in a KVM virtual guest environment with the following attributes, software, and package versions:

- z15 Processor
- RHEL 8.7
- Memory: 19G
- Kernel: `4.18.0-425.13.1.el8_7.s390x`
- Disk: 54G
- CEX6A Accelerator card attached
- Java Version 8.0.7.0
- IBM J9 VM
- WebSphere Application Server Liberty 21.0.0.10
- `opencryptoki-icatok-3.18.0-5.el8_7.s390x`
- `opencryptoki-libs-3.18.0-5.el8_7.s390x`
- `opencryptoki-3.18.0-5.el8_7.s390x`
- `libica-4.0.2-1.el8.s390x`

Clear Key Encryption

When configuring clear key encryption for your environment, it is important to be aware of the Java security providers you are using to perform cryptographic operations. Depending on the version of Java being used in your environment, the security provider list can differ. At the time of writing this report, for Java versions 8 and 11, cryptographic acceleration capabilities via clear key in a Linux on Z environment depend on the following security providers:

IBM Java 8:

- `IBMPKCS11Impl`
    - Exploits RSA acceleration on the attached cryptographic adapter (`CEX_A` card)
    - Must be added to the Java security provider list and takes a PKCS11 configuration file as a parameter *(Provider list order can be changed manually or programmatically)*
    - **Note**: It is recommended to modify the PKCS11 configuration file such that the `IBMPKCS11Impl` only uses the encryption mechanisms needed to support RSA (*and ECC pre z/15*) encryption to avoid deteriorating performance

- `IBMJCE`
    - On z14 hardware and below, exploits CPACF to perform symmetric encryption operations
        - (AES, DES, 3DES, etc).
        - CEX adapter can be used to accelerate ECC requests via `IBMPKCS11Impl`
    - Provided by default in the Java security provider list

- `IBMJCEplus`
    - As of z15 and higher, exploits CPACF to perform SHA/SH2 digests and symmetric encryption operations
        - (AES, DES, 3DES, etc)
    - As of z15 and higher, exploits CPACF to handle ECC operations faster
        - **Recommended** to no longer have the `IBMPKCS11Impl` handle ECC encryption mechanisms for performance purposes
    - Provided by default in the Java security provider

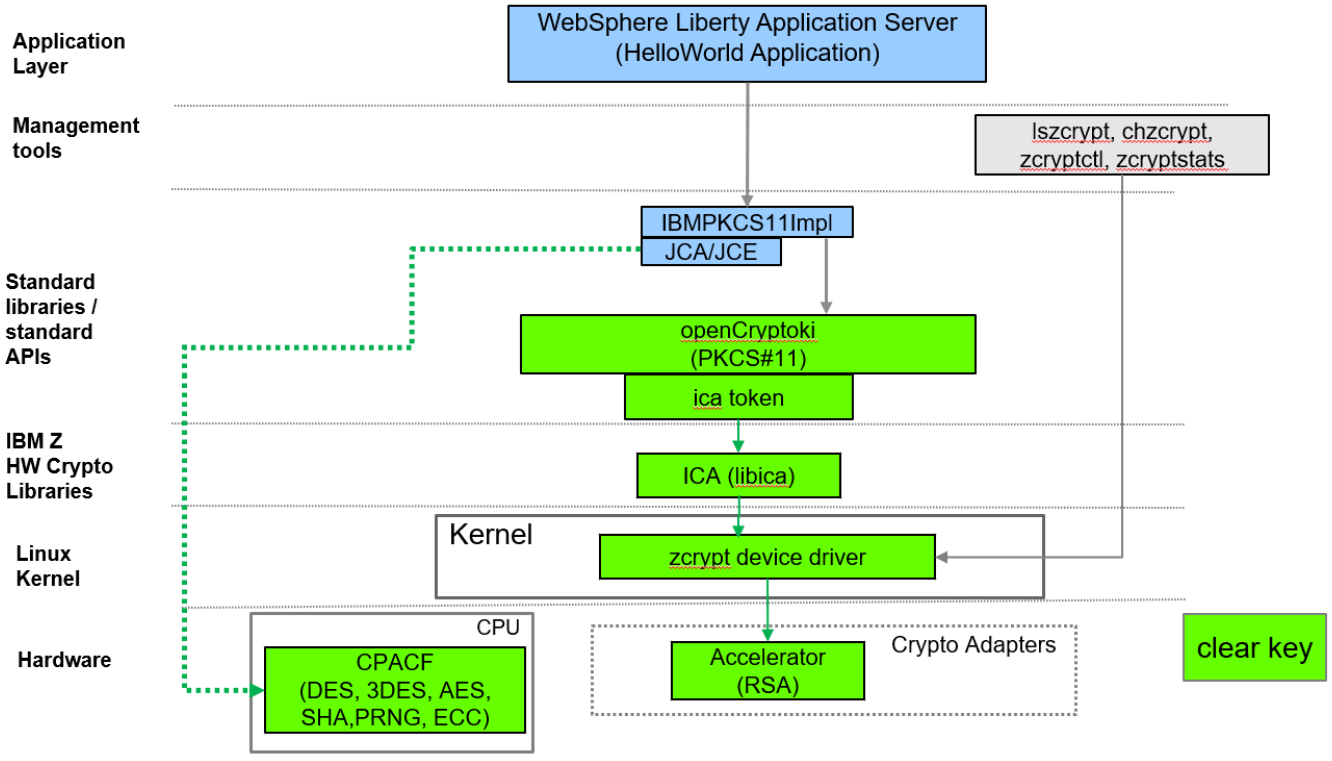
IBM Java 11:

- `SUNPKCS11` *(Very similar to the `IBMPKCS11Impl`)*
    - Responsible for exploiting RSA acceleration on the attached cryptographic adapter (performance is said to be better than the `IBMPKCS11Impl` java security provider)
    - Provided by default in the Java security provider list but still must be moved to the top of the Java security provider list *(Provider list order can be changed manually or programmatically)*
    - Takes a PKCS11 configuration file as a parameter and it is **recommended** to modify the configuration file such that the mechanisms needed to support RSA (*and ECC pre z/15*) are only used
- `SunJCE`
    - Configured to use `libcrypto` functionality to call CPACF directly and perform ECC and symmetric encryption operations as well
        - (ECC, AES, DES, 3DES, etc)
    - Provided by default in the Java security provider list

Note: The testing done in this report uses **IBM Java 8**, as a result the following documentation may not be fully applicable for those using IBM Java 11.

Linux on IBM Z Clear Key Flow

The below diagram represents a high-level flow of how clear encryption is performed relative to the testing done in this report:



Asymmetric (RSA) and symmetric (AES, DES, 3DES, etc) cryptographic requests driven to my "Hello World" application are passed down through the above encryption stack. The Java security providers in blue handle each request accordingly and the requests are either accelerated via my attached crypto adapter or on the CPU via CPACF.

System Configuration

To utilize clear key encryption, you will need to modify your system to exploit the cryptographic acceleration features provided on your attached adapter. Most of the steps used to configure clear key encryption came from this crypto performance study, conducted in 2013. I felt it would be redundant to replicate all the commands and output in this report that had already been previously done in the 2013 clear key encryption study. I would highly recommend viewing that test report for complete command examples and use this report as an accompanying guide for things that have changed in the stack over time.

Step 1:

Be sure the `z90crypt` or `AP` module is loaded and installed in your environment. Depending on your environment you may be required to load the module with the `modprobe` command. Upon issuing the `lszcrypt` command, if you receive the following error message:

```
error - cryptographic device driver zcrypt is not loaded!
```

The module has not been successfully loaded. In my testing with RHEL8.7, the kernel already had basic cryptographic device driver support and the `modprobe` command was not necessary. If the module has been loaded correctly, `lszcrypt` will show the online cryptographic adapters attached to your system and their request counts.

```
[lozcoc@helloWorld ~]$ lszcrypt
CARD.DOM TYPE  MODE        STATUS     REQUESTS
------------------------------------------------
01       CEX6A Accelerator online           20
01.0047  CEX6A Accelerator online           20
```

Step 2:

Start the `pkcsslotd` daemon to allow for the ICA token to be configured and enable the libICA interface to communicate with the PKCS11 API. If the `pkcsslotd` service is running, you should see the following response when checking its status with `systemctl`:

```
[lozcoc@helloWorld ~]$ sudo systemctl status pkcsslotd
● pkcsslotd.service - Daemon which manages cryptographic hardware tokens for the openCryptoki package
   Loaded: loaded (/usr/lib/systemd/system/pkcsslotd.service; enabled; vendor preset: disabled)
   Active: active (running) since Thu 2023-01-19 13:49:06 EST; 2h 33min ago
 Main PID: 153135 (pkcsslotd)
    Tasks: 1 (limit: 124607)
   Memory: 6.1M
   CGroup: /system.slice/pkcsslotd.service
           └─153135 /usr/sbin/pkcsslotd

Jan 19 13:49:06 helloWorld.fpet.pokprv.stglabs.ibm.com systemd[1]: Starting Daemon which manages cryptographic hardware tokens for the openCrypt
Jan 19 13:49:06 helloWorld.fpet.pokprv.stglabs.ibm.com systemd[1]: Started Daemon which manages cryptographic hardware tokens for the openCrypto
```

Step 3:

Initialize the ICA token to allow for the PKCS11 API to be utilized:

- Issue `sudo pkcsconf -t` to view what slot number the ICA token is using:
  (On my system the ICA token was in slot 1)

```
Token #1 Info:
            Label: IBM ICA PKCS #11
            Manufacturer: IBM
            Model: ICA
            Serial Number:
            Flags: 0x880045
(RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED)
            Sessions: 0/[effectively infinite]
            R/W Sessions: [information unavailable]/[effectively infinite]
            PIN Length: 4-8
            Public Memory: [information unavailable]/[information unavailable]
            Private Memory: [information unavailable]/[information unavailable]
            Hardware Version: 0.0
            Firmware Version: 0.0
            Time: 2023011916272800
```

- Initialize the token using the default SO pin of `87654321` and enter a new token label:

```
sudo pkcsconf -c 1 -I

Enter the SO PIN: ########
Enter a unique token label: icatok
```

- Set a new SO PIN:

```
sudo pkcsconf -c 1 -P

Enter the SO PIN: ########
Enter the new SO PIN: ########
Re-enter the new SO PIN: ########
```

- Initialize and set a user PIN:

```
sudo pkcsconf -c 1 -u

Enter the SO PIN: ########
Enter the new user PIN: ########
Re-enter the new user PIN: ########
```

- Set a new user PIN, after setting a user PIN for the first time above, the flag `USER_PIN_TO_BE_CHANGED` is set in your token. It is recommended to explicitly set a new user PIN afterwards:

```
sudo pkcsconf -c 1 -p

Enter the SO PIN: ########
Enter the new user PIN: ########
Re-enter the new user PIN: ########
```

- The ICA token's hex flags should now be set to `0x44D` to indicate the token is now initialized:

```
sudo pkcsconf -t
Token #1 Info:
    Label: icatok
    Manufacturer: IBM
    Model: ICA
    Serial Number:
    Flags: 0x44D
(RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED)
    Sessions: 0/[effectively infinite]
    R/W Sessions: [information unavailable]/[effectively infinite]
    PIN Length: 4-8
    Public Memory: [information unavailable]/[information unavailable]
    Private Memory: [information unavailable]/[information unavailable]
    Hardware Version: 0.0
    Firmware Version: 0.0
    Time: 2023011916401400
```

Note: Any non-root users running applications using the PKCS11 API must be added to the pkcs11 group

- `(usermod -G pkcs11 <non_root_user>)`

(The testing done in this report was done with a user running with root privileges)

Step 4:

Create a PKCS11 configuration file, this configuration file will later be passed to the IBM Java PKCS11 security provider. This file allows for IBM Java to communicate with the ICA token which in turn will talk down to the zcrypt device driver and perform encryption. The PKCS11 configuration file can be given any name, as it does not follow a naming convention and should look something like this:

```
name = arbitrary config name
description = brief explanation of config file
library = path/to/your/pkcs11/implementation
tokenlabel = name of the ICA token previously specified
disabledmechanisms = {
                        List of PKCS11 mechanisms you want to disable
(Note: These mechanisms are explicitly disabled for the PKCS11 provider and not
the entire Java environment. They can still be accelerated by other encryption
providers below the PKCS11 provider in the Java security stack)
                        }
```

I have noticed, depending on your Linux distribution the path of your PKCS11 implementation can vary. In my case on RHEL 8.7, the `PKCS11_API.so` file was located `/usr/lib64/opencryptoki`. It is also **important to note,** that in my PKCS11 configuration, I decided to disable numerous encryption mechanisms such that the PKCS11 Java security provider, `IBMPKCS11Impl`, refrains from performing symmetric cryptographic operations. I **HIGHLY** recommend taking this approach and configuring the PKCS11 provider to only use the mechanisms needed to support RSA. If you choose not to disable any encryption mechanisms, beware of the performance impact it can have on your application and the defect impacting RHEL8 and SLES15.

In my implementation of the PKCS11 configuration file, I created a file called, `cex.cfg`, and stored it in the `/etc/` directory on my system:

```
[lozcoc@helloWorld security]$ cat /etc/cex.cfg
name = crypto_hw
description = config for IBM Crypto Express HW (icatok)
library = /usr/lib64/opencryptoki/PKCS11_API.so
tokenlabel = icatok
disabledmechanisms = {
                      CKM_DES_KEY_GEN
                      CKM_DES_ECB
                      CKM_DES_CBC
                      CKM_DES_CBC_PAD
                      CKM_DES3_KEY_GEN
                      CKM_DES3_ECB
                      CKM_DES3_CBC
                      CKM_DES3_MAC
                      CKM_DES3_MAC_GENERAL
                      CKM_DES3_CBC_PAD
                      CKM_MD5
                      CKM_MD5_HMAC
                      CKM_MD5_HMAC_GENERAL
                      CKM_SHA_1
                      CKM_SHA_1_HMAC
                      CKM_SHA_1_HMAC_GENERAL
                      CKM_SHA256
                      CKM_SHA256_HMAC
                      CKM_SHA256_HMAC_GENERAL
                      CKM_SHA224
                      CKM_SHA224_HMAC
                      CKM_SHA224_HMAC_GENERAL
                      CKM_SHA384
                      CKM_SHA384_HMAC
                      CKM_SHA384_HMAC_GENERAL
                      CKM_SHA512
                      CKM_SHA512_HMAC
                      CKM_SHA512_HMAC_GENERAL
                      CKM_GENERIC_SECRET_KEY_GEN
                      CKM_EC_KEY_PAIR_GEN
                      CKM_ECDSA
                      CKM_ECDSA_SHA1
                      CKM_ECDH1_DERIVE
                      CKM_AES_KEY_GEN
                      CKM_AES_ECB
                      CKM_AES_CBC
                      CKM_AES_MAC
                      CKM_AES_MAC_GENERAL
                      CKM_AES_CBC_PAD
                      CKM_AES_GCM
                      }
```

As you can see, I disabled numerous encryption mechanisms, mostly symmetric, in an effort to force the PKCS11 security provider to only handle accelerating RSA requests. The `pkcsconf -m -c <token_slot_number> n` command can list all the encryption mechanisms supported for your token. However, finding the above list of mechanisms was not a trivial process because the `pkcsconf` command lists both the supported and unsupported mechanisms for the Java PKCS11 provider.

In my testing, I captured this list and wrote the results to a file with the following command:

```
pkcsconf -m -c 1 n > ~/crypto.text
```

Note: In the long list of mechanisms captured from the above command, it is not clear which are supported by the PKCS11 security provider. Unfortunately, if the PKCS11 security provider does not recognize an encryption mechanism it will fail to initialize **silently**. The disabled mechanism list configured above required a lot of trial and error, currently there is not a user-friendly way for determining the encryption mechanisms known by IBMPKCS11Impl. IBM Java documents a list of supported PKCS11 mechanisms and this is a good resource to reference. **Keep in mind** this is not a complete list and does not cover all the supported PKCS11 mechanisms.

Configuring the Java Security Provider

By default, IBM Java is not configured to use PKCS11 and perform RSA acceleration capabilities, a modification to the `java.security` file installed with your version of IBM Java is required. The location of this file can vary depending on how Java is installed on your system and what version of Java you are running with in your environment. On my test system, running IBM Java 8, the `java.security` file was located in my Java installation directory at `jre/lib/security`.

Note: Alternatively, if you do not wish to make changes to the `java.security` file included with your installation of Java. You can create a copy of the existing `java.security` file, make the necessary modifications, and then tell Java to use the new copy `java.security` file with the following JVM argument:

- `Djava.security.property=<java.security path and file name>`

This file is quite verbose, however, the only area you will need to modify is the section that lists the java security providers. Looking at the `java.security` file on my test machine, the default provider list displays the following:

```
# List of providers and their preference orders (see above):
#
security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
security.provider.2=com.ibm.crypto.plus.provider.IBMJCEPlus
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
security.provider.7=com.ibm.xml.crypto.IBMXMLCryptoProvider
security.provider.8=com.ibm.xml.enc.IBMXMLEncProvider
security.provider.9=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
security.provider.10=sun.security.provider.Sun
```

PKCS11

To use PKCS11 for clear key encryption, the IBMPKCS11Impl provider needs to be added to the provider list along with the location of the PKCS11 configuration file previously created.

Following this modification, my system's java.security file looks as such:

```
# List of providers and their preference orders (see above):
#
security.provider.1=com.ibm.crypto.pkcs11impl.provider.IBMPKCS11Impl /etc/cex.cfg
security.provider.2=com.ibm.jsse2.IBMJSSEProvider2
security.provider.3=com.ibm.crypto.plus.provider.IBMJCEPlus
security.provider.4=com.ibm.crypto.provider.IBMJCE
security.provider.5=com.ibm.security.jgss.IBMJGSSProvider
security.provider.6=com.ibm.security.cert.IBMCertPath
security.provider.7=com.ibm.security.sasl.IBMSASL
security.provider.8=com.ibm.xml.crypto.IBMXMLCryptoProvider
security.provider.9=com.ibm.xml.enc.IBMXMLEncProvider
security.provider.10=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
security.provider.11=sun.security.provider.Sun
```

Given the newly modified security provider list above, coupled with the disabled mechanisms in my PKCS11 configuration file. The IBMPKCS11Impl provider should handle the acceleration of RSA requests using my attached cryptographic adapter. All other requests such as, AES, DES, and 3DES, will fall to the IBMJCE and IBMJCEplus providers which will handle acceleration on the CPU via CPACF.

Once you have fully configured your system and IBM Java to take advantage of the PKCS11 provider, you can verify your configuration by ensuring the encryption statistics displayed from the icastats command are increasing.

The icastats and icainfo commands are utilities provided by the libica package for monitoring the acceleration features being used on your system. (**Note:** These commands display data of the encryption operations pertaining to libica, which is called indirectly by the IBMPKCS11Impl provider if configured with openCryptoki and the ica token)

For verification, I issued a few curl requests to my test application to drive RSA activity, I was able to see the crypto counters increase:

```
  function       |             hardware          |              software
---------------+-------------------------------+-------------------------------
                 |        ENC     CRYPT     DEC  |        ENC     CRYPT     DEC
---------------+-------------------------------+-------------------------------
          SHA-1 |                  0            |                  0
        SHA-224 |                  0            |                  0
        SHA-256 |                  5            |                  0
        SHA-384 |                  0            |                  0
        SHA-512 |                  0            |                  0
```

SHA-512/224	0		0	
SHA-512/256	0		0	
SHA3-224	0		0	
SHA3-256	0		0	
SHA3-384	0		0	
SHA3-512	0		0	
SHAKE-128	0		0	
SHAKE-256	0		0	
GHASH	0		0	
P_RNG	0		0	
DRBG-SHA-512	9		0	
ECDH	0		0	
ECDSA Sign	0		0	
ECDSA Verify	0		0	
EC Keygen	0		0	
Ed25519 Keygen	0		0	
Ed25519 Sign	0		0	
Ed25519 Verify	0		0	
Ed448 Keygen	0		0	
Ed448 Sign	0		0	
Ed448 Verify	0		0	
X25519 Keygen	0		0	
X25519 Derive	0		0	
X448 Keygen	0		0	
X448 Derive	0		0	
RSA-ME	**1**		**0**	
RSA-CRT	**1**		**0**	
DES ECB	0	0	0	0
DES CBC	0	0	0	0
DES OFB	0	0	0	0
DES CFB	0	0	0	0
DES CTR	0	0	0	0
DES CMAC	0	0	0	0
3DES ECB	0	0	0	0
3DES CBC	0	0	0	0
3DES OFB	0	0	0	0
3DES CFB	0	0	0	0
3DES CTR	0	0	0	0
3DES CMAC	0	0	0	0
AES ECB	0	0	0	0
AES CBC	0	0	0	0
AES OFB	0	0	0	0
AES CFB	0	0	0	0
AES CTR	0	0	0	0
AES CMAC	0	0	0	0
AES XTS	0	0	0	0
AES GCM	0	0	0	0

Note: Be sure your verification requests are responsible for impacting the crypto statistic counters. The counters may increase because of other processing on your system. For example, SSH session requests have been noted to increase the counters depending on how your environment is configured.

Testing

The testing documented below, encompasses the performance statistics consisting of application response times and throughput. These metrics have been gathered by driving simulated customer workloads in my team's production test environment. (**Note**: All the machines in the test environment used to gather the performance metrics documented below are shared across other teams and departments, this is not a dedicated environment specifically geared to run performance measurements. The data captured is for comparison purposes, not benchmark.)

These workloads are driven via the testing framework known as JMeter. The JMETER testcases execute inside containerized images and target a web application running on IBM Liberty. To accurately capture the performance of IBM Java and its impact on cryptographic acceleration, a simple "hello world" web application was utilized, as opposed to a complex web application that may incur performance debt as a result of natural web application processing.

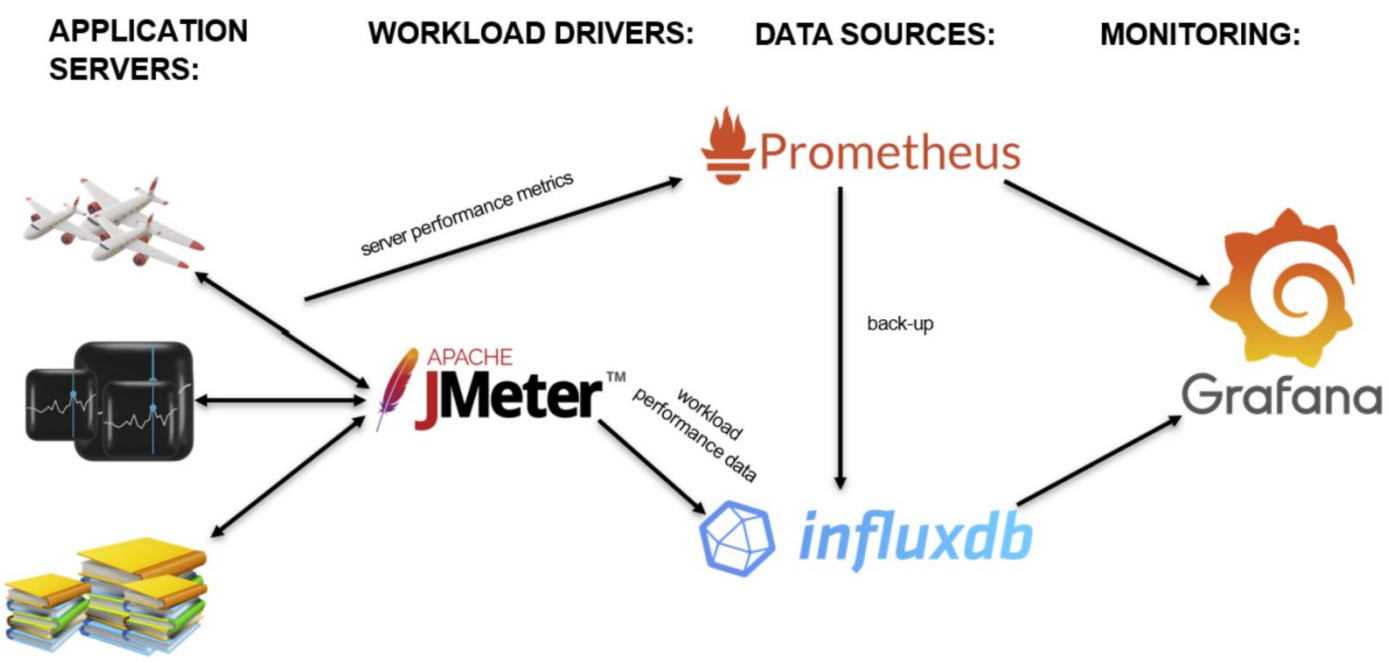
The testcases stress TLS session usage and drive the RSA acceleration capabilities of the crypto card while also performing symmetric encryption operations to be accelerated on the CPU. The results of these testcases are captured in the form of Grafana dashboards and can be viewed under each respective test below.

The tests were consistent across each variation, including a baseline test in which encryption was not performed. The parameters for each test including the following:

- Test Duration: 12 hours
- ThinkTime *(How much time to wait in between requests)*: 3 Milliseconds
- Request Type: HTTPS
- Users: 100
- Threads: 50
- JVM Options:

```
-Xdump:java+snap+heap:events=systhrow,filter=java/lang/OutOfMemoryError,range=1..3
-verbose:gc
-Xdump:directory=/home/lozcoc/oom_debug
-Xverbosegclog:/home/lozcoc/oom_debug/gc.log
-Xmx2g
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/home/lozcoc/oom_debug/my-heap-dump.hprof
```

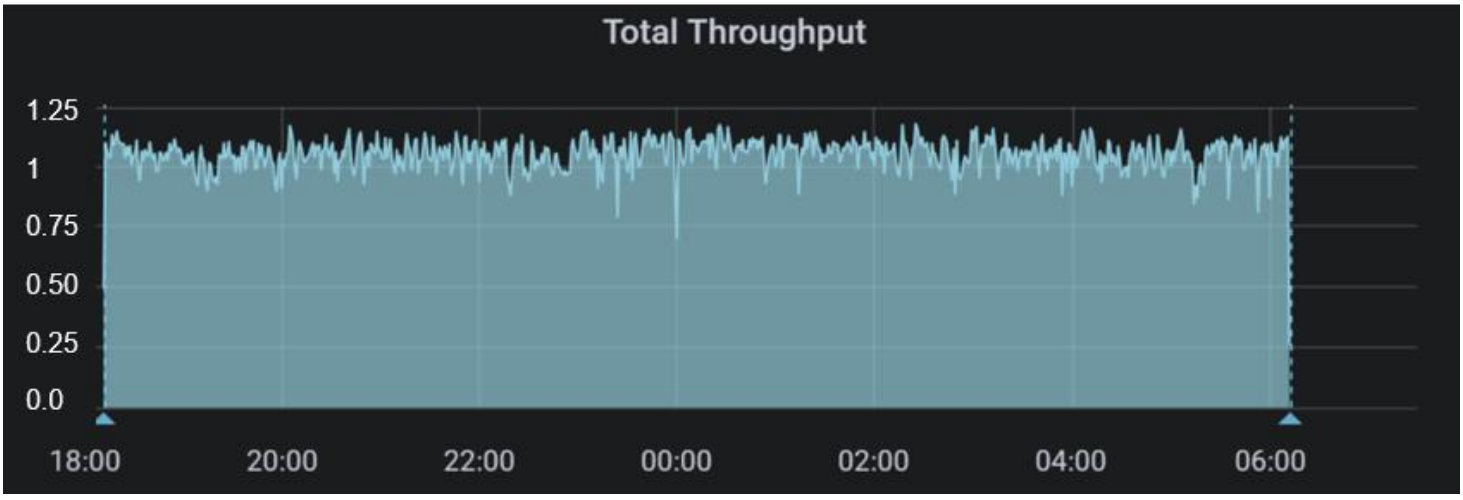
The complete testing solution my team used for driving these "production like" customer workloads, can be best depicted by the diagram below:



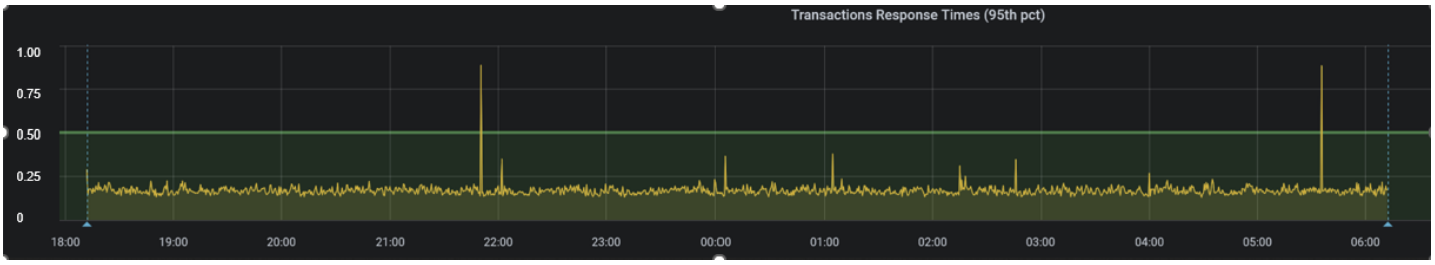
The explanation and configuration of the technologies used to create the test environment above are outside the scope of this document and will be left as an exercise for the reader.

No Encryption Test Results *(Baseline)*

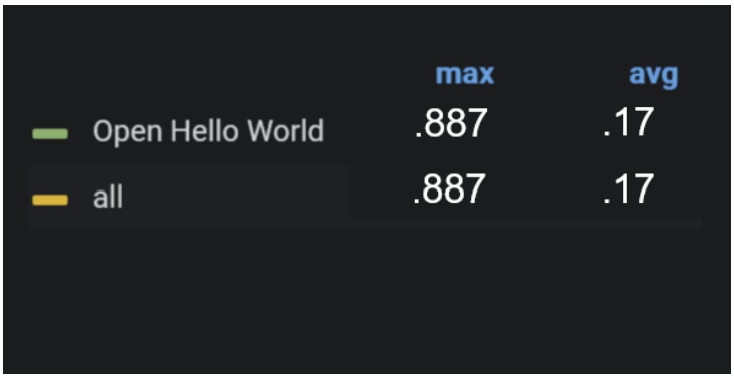
Without performing encryption at all and only issuing requests over a HTTP connection to my test application. The JMETER workload shows very promising performance, reaching a normalized average throughput performance value of 1 and normalized average response time of .17. We normalized both the average throughput and response time values to 1 and .17 respectively, the future metrics recorded in this test report are relative to these measurements. Although with encryption enabled in the upcoming testing a slight dip in performance is expected, we would still like these metrics to remain consistent.



Note: The response times in this graph have been normalized to a peak of 1

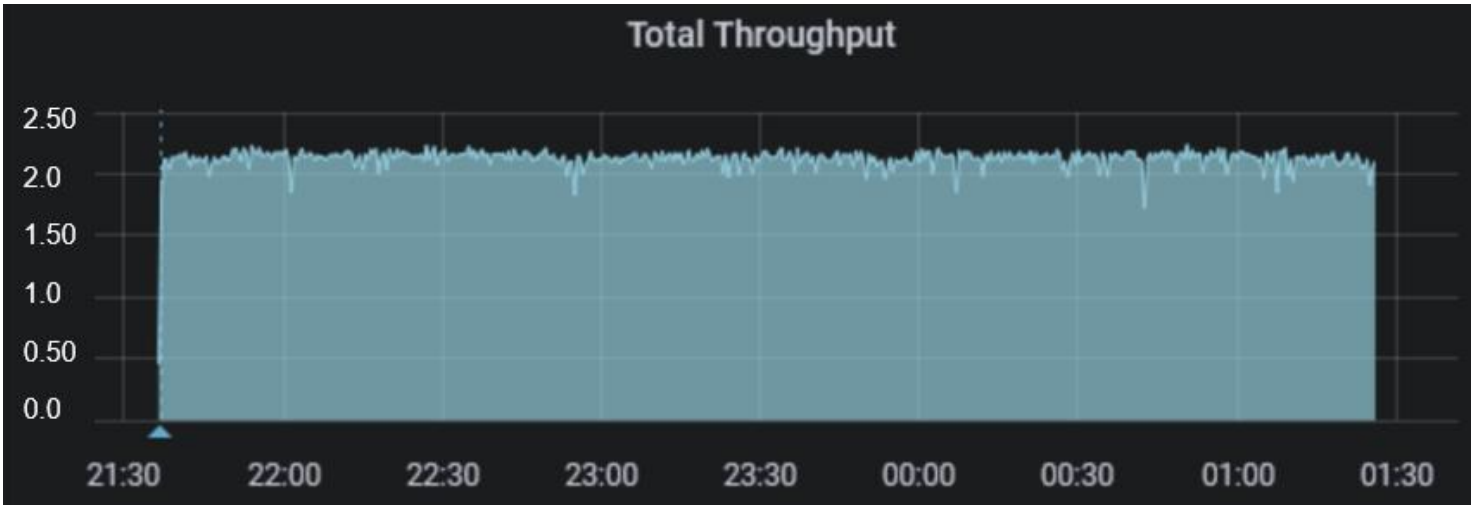


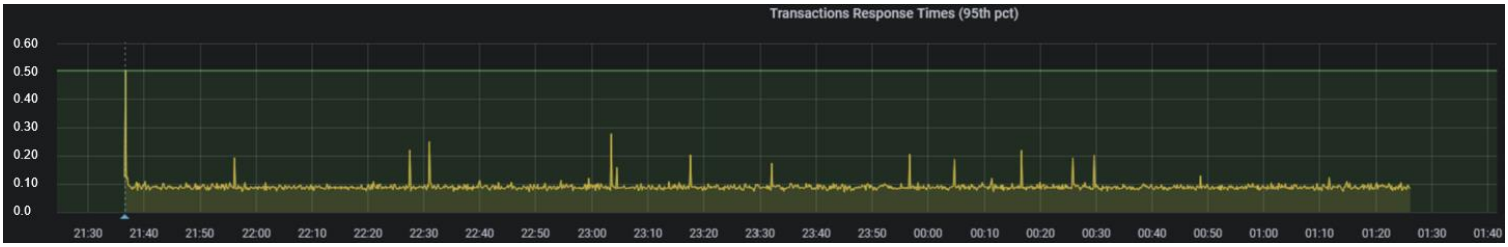
Note: The values below in this graph have been normalized in respect to the yellow response time graph above

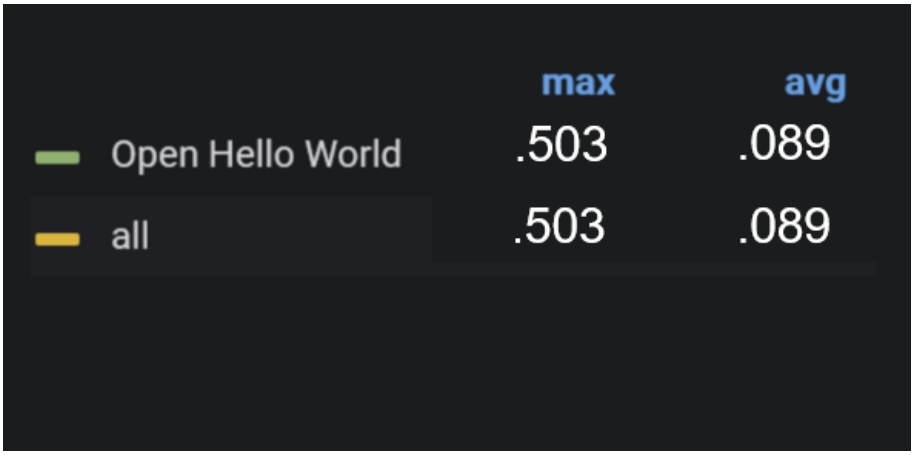


PKCS11 Disabled Mechanisms Test Results

Now when performing encryption and using the PKCS11 configuration file containing a list of disabled encryption mechanisms, the JMETER workload once again yielded great performance metrics. This time performance improved, the normalized average throughput increased to 2 and our normalized average response time went down to .089. We were surprised to see this large of a performance increase. Although this performance improvement could have been coincidental and a result of reduced system stress in my environment during this particular test run, the performance metrics only slightly differed and did not change enough to raise eyebrows.







ICASTATS Counters (Disabled Mechanisms)

function	hardware			software		
	ENC	CRYPT	DEC	ENC	CRYPT	DEC
SHA-1		0			0	
SHA-224		0			0	
SHA-256		**34129224**			**0**	
SHA-384		0			0	
SHA-512		0			0	
SHA-512/224		0			0	
SHA-512/256		0			0	
SHA3-224		0			0	
SHA3-256		0			0	
SHA3-384		0			0	
SHA3-512		0			0	
SHAKE-128		0			0	
SHAKE-256		0			0	
GHASH		0			0	
P_RNG		0			0	
DRBG-SHA-512		**22752793**			**0**	
ECDH		0			0	
ECDSA Sign		0			0	
ECDSA Verify		0			0	
EC Keygen		0			0	
Ed25519 Keygen		0			0	

Ed25519 Sign	0		0	
Ed25519 Verify	0		0	
Ed448 Keygen	0		0	
Ed448 Sign	0		0	
Ed448 Verify	0		0	
X25519 Keygen	0		0	
X25519 Derive	0		0	
X448 Keygen	0		0	
X448 Derive	0		0	
RSA-ME	0		0	
RSA-CRT	**11376408**		**0**	
DES ECB	0	0	0	0
DES CBC	0	0	0	0
DES OFB	0	0	0	0
DES CFB	0	0	0	0
DES CTR	0	0	0	0
DES CMAC	0	0	0	0
3DES ECB	0	0	0	0
3DES CBC	0	0	0	0
3DES OFB	0	0	0	0
3DES CFB	0	0	0	0
3DES CTR	0	0	0	0
3DES CMAC	0	0	0	0
AES ECB	0	0	0	0
AES CBC	0	0	0	0
AES OFB	0	0	0	0
AES CFB	0	0	0	0
AES CTR	0	0	0	0
AES CMAC	0	0	0	0
AES XTS	0	0	0	0
AES GCM	0	0	0	0

Native LPAR Symmetric Counters

My current PKCS11 configuration disables symmetric encryption mechanisms, telling the `IBMPKCS11Impl` security provider to avoid processing symmetric encryption requests. As a result, the `icastats` command will not display symmetric encryption statistics. The only way to monitor and verify that CPACF is accelerating the symmetric encryption requests is to use the `cpacfstats` command. This command is only available at the LPAR level, any virtual machines running on the LPAR host cannot run this command (More info).

Running the same test above natively on an LPAR yielded the same performance metrics and allowed for me to verify that the symmetric encryption requests were being accelerated:

```
[root@ltickvmf ~]# cpacfstats
 des counter: 0
 aes counter: 765217315
 sha counter: 978682749
 rng counter: 35667176
 ecc counter: unsupported
```

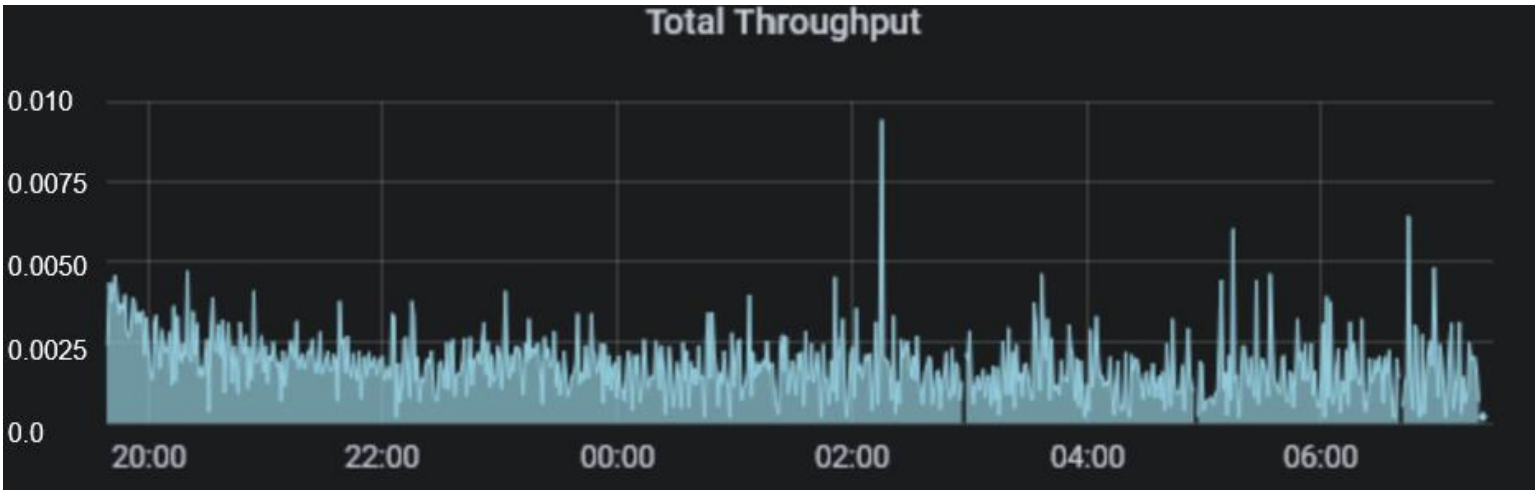
PKCS11 No Disabled Mechanisms Test Results

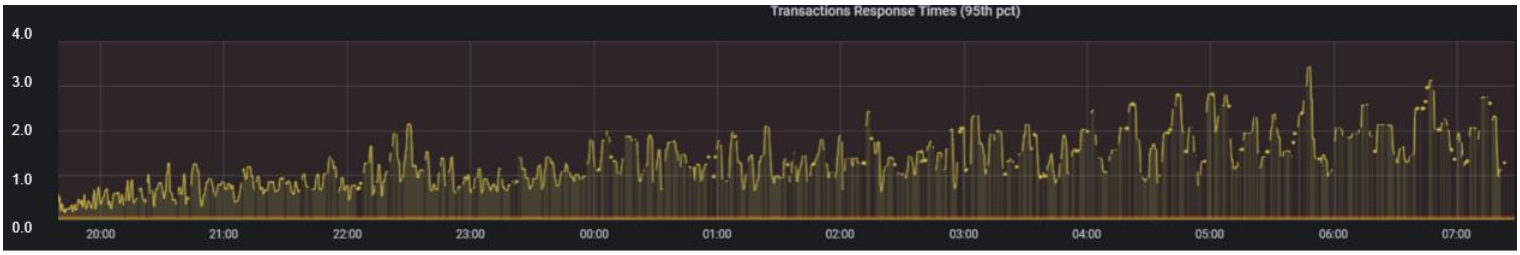
For this test, I decided to enable all encryption mechanisms in my PKCS11 configuration file. This will tell my PKCS11 security provider, `IBMPKCS11Impl`, to try and handle processing all supported encryption requests driven from my test application.

Modified configuration file (*Added `attributes` statements*):

```
name = crypto_hw
description = config for IBM Crypto Express HW (icatok)
library = /usr/lib64/opencryptoki/PKCS11_API.so
tokenlabel = icatok
attributes (*, CKO_PRIVATE_KEY, *) = {
                          CKA_SENSITIVE = true
                          CKA_SIGN=true
                          CKA_DECRYPT=true
                          CKA_DERIVE=true
                        }
attributes (*, CKO_PUBLIC_KEY, *) = {
                          CKA_VERIFY=true
                          CKA_ENCRYPT=true
                          CKA_DERIVE=true
                        }
```

When enabling all encryption mechanisms, you can notice an extremely significant degradation. The normalized throughput only yields an average of .018 with an alarming, normalized response time average of 2.16. In some cases, the application's response time would become so degraded that it would no longer be reachable from a web browser and Java out of memory (OOM) errors would appear on my system.





ICASTATS Counters (No Disabled Mechanisms)

Note: With no disabled mechanisms configured, the `icastats` command will display the symmetric encryption requests being handled by the `IBMPKCS11Impl` provider:

```
function       |             hardware          |              software
---------------+-------------------------------+-------------------------------
               |        ENC     CRYPT      DEC  |        ENC      CRYPT     DEC
---------------+-------------------------------+-------------------------------
          SHA-1 |                  0            |                  0
        SHA-224 |                  0            |                  0
        SHA-256 |              17013            |                  0
        SHA-384 |              79204            |                  0
        SHA-512 |                  0            |                  0
    SHA-512/224 |                  0            |                  0
    SHA-512/256 |                  0            |                  0
       SHA3-224 |                  0            |                  0
       SHA3-256 |                  0            |                  0
       SHA3-384 |                  0            |                  0
       SHA3-512 |                  0            |                  0
      SHAKE-128 |                  0            |                  0
```

SHAKE-256	0		0	
GHASH	0		0	
P_RNG	0		0	
DRBG-SHA-512	**176281**		**0**	
ECDH	**5644**		**0**	
ECDSA Sign	0		0	
ECDSA Verify	0		0	
EC Keygen	**5671**		**0**	
Ed25519 Keygen	0		0	
Ed25519 Sign	0		0	
Ed25519 Verify	0		0	
Ed448 Keygen	0		0	
Ed448 Sign	0		0	
Ed448 Verify	0		0	
X25519 Keygen	0		0	
X25519 Derive	0		0	
X448 Keygen	0		0	
X448 Derive	0		0	
RSA-ME	0		0	
RSA-CRT	**5671**		**0**	
DES ECB	0	0	0	0
DES CBC	0	0	0	0
DES OFB	0	0	0	0
DES CFB	0	0	0	0
DES CTR	0	0	0	0
DES CMAC	0	0	0	0
3DES ECB	0	0	0	0
3DES CBC	0	0	0	0
3DES OFB	0	0	0	0
3DES CFB	0	0	0	0
3DES CTR	0	0	0	0
3DES CMAC	0	0	0	0
AES ECB	**170576**	**0**	**0**	**0**
AES CBC	0	0	0	0
AES OFB	0	0	0	0
AES CFB	0	0	0	0
AES CTR	0	0	0	0
AES CMAC	0	0	0	0
AES XTS	0	0	0	0
AES GCM	**73899**	**11389**	**0**	**0**

Conclusion

Looking at the test results gathered above in my team's test environment, I believe it is evident when performing clear key encryption with IBM Java 8 users should configure the `IBMPKCS11IMPL` provider to only use the mechanisms required to support RSA. In my experience, when leaving all the cryptographic mechanisms enabled by default, problems persist across each test. Clear key encryption can be an effective approach for users looking to accelerate RSA ciphers, any workload performing TLS/SSL based processing would be a candidate to reap the benefits of clear key encryption, however neglecting to configure the PKCS11 security provider to only perform the encryption ciphers necessary can lead to degraded application performance and exposure to existing defects.

Known Issues

Defect 1:

This defect impacts RHEL8 and SLES15. If a user chooses to leave all encryption mechanisms enabled by default, the following `attributes` statements must be added to your PKCS11 configuration file to circumvent the following bug. At the time of writing this report, a permanent solution has not been provided for this defect.

```
attributes (*, CKO_PRIVATE_KEY, *) = {
                          CKA_SENSITIVE = true
                          CKA_SIGN=true
                          CKA_DECRYPT=true
                          CKA_DERIVE=true
                        }
attributes (*, CKO_PUBLIC_KEY, *) = {
                          CKA_VERIFY=true
                          CKA_ENCRYPT=true
                          CKA_DERIVE=true
                        }
```

Defect 2:

This defect only impacts SLES12. When running with the `IBMPKCS11Impl` provider on IBM Java 8, WAS and Liberty applications are not reachable over their respective TLS/SSL connections.