# Obligatorio: Aspectos de Seguridad de Sistemas Informáticos

# Período: 2020

## Entregable

Se deberá entregar la documentación asociada a la resolución de la propuesta, así como el código ejecutable, el código fuente, las evidencias de las corridas, y todo lo que sea requerido para que el sistema pueda ejecutar.

## La asignación – Parte 1

Implementar un sistema de seguridad simple, en Java, siguiendo las reglas de seguridad de Bell-LaPadula (BPL), seguridad simple, la propiedad * y la tranquilidad fuerte.

Un sistema simple no seguro.

Imagine un sistema con sujetos y objetos.
Los objetos en este sistema son variables enteras simples.
Cada objeto tiene un nombre y un valor (inicialmente 0).
Cada sujeto tiene un nombre y una variable entera TEMP que registra el valor que leyó más recientemente (también inicialmente 0).
Los sujetos pueden realizar operaciones READ o WRITE en los objetos.

Para un READ, el sujeto lee el valor actual del objeto y guarda ese valor en su variable TEMP (un READ posterior le pasará por arriba).

Cuando un sujeto hace un WRITE, el valor del objeto se actualiza.

Los objetos son administrados por la clase ObjectManager como un sistema de archivos muy simple que Lee y Escribe objetos por su nombre.

La entrada a su sistema es un archivo de comandos. Para la asignación actual, los comandos legales tienen la forma:

READ nombre_sujeto nombre_objeto
WRITE nombre_sujeto nombre_objeto valor

Todos los campos de la instrucción son caracteres excepto el valor, que es un número entero.

Se leerán líneas sucesivas del archivo de comandos y las mismas se analizarán en la clase InstructionObject.
Esta clase deberá analizar la línea ingresada y generar una instrucción READ, WRITE o BADINSTRUCTION.
Los comandos no son sensibles a mayúsculas, incluso los nombres de los sujetos y de los objetos.
Pueden existir más de un espacio en blanco entre los elementos de la sentencia, pero puede asumir que cada comando viene en una sola línea.

Deberá considerar la posibilidad de que las instrucciones vengan con errores, (que venga un verbo diferente a READ o WRITE, número equivocado de parámetros, tipos de argumentos, etc.),

Para las instrucciones con errores, genere un objeto constante BadInstruction.

Por ejemplo, supongamos que su archivo de entrada contiene las siguientes instrucciones. Los dos primeros y los últimos tres están mal y los otros son sintácticamente correctas.

write hal hobj
read hal
write lyle lobj 10
read hal lobj
write lyle hobj 20
write hal lobj 200
read hal hobj
read lyle lobj
read lyle hobj
foo lyle lobj
Hi lyle, This is hal
The missile launch code is 1234567

Tenga en cuenta que, en esta etapa, no está verificando los atributos de seguridad ni la definición de los sujetos u objetos (aún), simplemente verificará la correcta sintaxis de los comandos.

Suponiendo que “hal”, “moe” y “lyle” son sujetos y “lobj”, “mobj” y “hobj” son objetos, se requiere que escriba un intérprete para este sistema, que ejecutaría las instrucciones legales e ignoraría las demás.

Una implementación razonable podría terminar con un estado en el que los objetos tienen los valores: lobj 200, mobj 2 y hobj 20; las variables TEMP de los sujetos en el final contendrían: hal 20, moe 2 y lyle 20.

El sistema seguro.

Ahora, agreguemos seguridad a nuestro sistema.
Comience asignando etiquetas de seguridad asociadas a los sujetos y a los objetos (levels).

Estas etiquetas son mantenidas por la clase ReferenceMonitor y no se pueden cambiar después de que son creados (tranquilidad fuerte).
Esencialmente, el ReferenceMonitor administra las asignaciones de las etiquetas de seguridad a los nombres de los sujetos / objetos.

En la versión segura de nuestro sistema, cada vez que un sujeto solicita realizar una acción (READ o WRITE), el InstructionObject le envía la instrucción generada al ReferenceMonitor, que decide si realiza la acción o no, en base a las propiedades BLP (Seguridad Simple y Propiedad *).

Si la instrucción es a la vez sintácticamente correcta y permitida por las reglas de BLP, el ReferenceMonitor le dice al ObjetctManager que realice la acción apropiada sobre los objetos; de lo contrario, no se accede a ningún objeto.

En el sistema seguro, también ampliamos la noción de cuando una instrucción es sintácticamente incorrecta, en el InstructionObject una instrucción que referencia a un sujeto u objeto que no existe es una BadInstruction.

Suponiendo que una instrucción no es mala, el ReferenceMonitor siempre devuelve un valor entero al sujeto: el valor del objeto leído, si el comando era un READ correcto, y 0 en caso contrario.

Si el sujeto está realizando un READ, almacena este valor en su variable TEMP.
Piense en el ReferenceMonitor como un firewall alrededor del ObjectManager.
Tenga en cuenta que el ObjectManager en sí no conoce ni se preocupa por las etiquetas o la seguridad:
solo realiza accesos simples.

La clase superior, SecureSystem administra los sujetos, al InstructionObject y al ReferenceMonitor. El InstructionObject lee las instrucciones en forma sucesiva de la lista de instrucciones, las analiza y las envía al ReferenceMonitor, el cual le solicita al ObjectManager que los ejecute (o no).
El valor devuelto por el ReferenceMonitor se pasa al sujeto que ejecuta la instrucción.

Su tarea es implementar todo esto, sujeto a las siguientes restricciones:

1) El ObjectManager debe ser local al ReferenceMonitor. Eso asegura que no pueda acceder a objetos excepto a través de la interfaz del ReferenceMonitor.
2) SecurityLevel deber ser una clase con una relación definida de “dominación”. Puede asumir que los niveles son linealmente ordenados. Es decir, no necesita preocuparse por la categoría “necesidad de saber”. Dentro de esa clase se definen tres niveles constantes HIGH, MEDIUM y LOW tales que HIGH domina a MEDIUM y a LOW, y MEDIUM domina a LOW.
3) Cuando analiza una instrucción desde el archivo de entrada en el InstructionObject, cree una instrucción con los campos que representen el tipo de instrucción (READ, WRITE, BAD), el nombre del sujeto, el nombre del objeto, el valor si lo hay. Esta instrucción es el que se le pasa al ReferenceMonitor. Para una instrucción mal formada, usted debería pasar una instrucción constante BadInstruction. El ReferenceMonitor debe saber cómo tratar con estos objetos.

Para ver cómo se comporta su sistema, debe escribir un método de depuración printState en la clase SecureSystem que imprima los valores actuales del estado: el valor de los objetos y el valor TEMP para los sujetos. Ver el siguiente ejemplo para ver cómo debería ser el resultado de esto.

Idealmente, podría iterar sobre sus estructuras de datos para manejar una cantidad arbitraria de sujetos / objetos, pero para esta tarea puede asumir que hay solo tres de cada uno y usted conoce sus nombres. (Nota: Imprimir este tipo de información no es algo que un usuario típico del sistema debería ser capaz de hacer).

La función principal en la clase SecureSystem debe realizar las siguientes tareas:

1) Crear tres sujetos nuevos: lyle con nivel de seguridad LOW, moe con nivel de seguridad MEDIUM y hal con nivel de seguridad HIGH. Almacene estos sujetos en el estado e informe al ReferenceMonitor sobre ellos.

2) Creas tres objetos nuevos: lobj con nivel de seguridad LOW, mobj con nivel de seguridad MEDIUM y hobj con nivel de seguridad HIGH. Almacene estos objetos en el ObjectManager, diciéndole al ReferenceMonitor acerca de sus niveles. El valor inicial de cada uno debe ser 0.

3) Lea las instrucciones sucesivas del archivo de entrada y ejecútelas siguiendo las limitaciones de lectura y escritura de Bell-LaPadula. Usted debería tener los métodos executeRead y executeWrite dentro de su clase ReferenceMonitor que verifica las solicitudes de acceso y realiza la actualización apropiada (si la hay) en el estado, siguiendo la semántica de instrucciones descrita anteriormente.
4) Después de cada instrucción, llame a printState para mostrar el cambio de estado, si lo hay, desde la ejecución de la instrucción.

A continuación hay un fragmento de código de la función principal. No tiene que seguir este modelo, pero se le agradece si lo hace.

```
// LOW, MEDIUM and HIGH are constants defined in the SecurityLevel
// class, such that HIGH dominates MEDIUM and LOW, and MEDIUM dominates LOW.

SecurityLevel low = SecurityLevel.LOW;
SecurityLevel medium = SecurityLevel.MEDIUM;
SecurityLevel high = SecurityLevel.HIGH;

// We add three subjects, one high, one medium and one low.

sys.createSubject("lyle", low);
sys.createSubject(“moe”, medium);
sys.createSubject("hal", high);

// We add three objects, one high, one medium and one low.

sys.getReferenceMonitor().createNewObject("Lobj", low);
sys.getReferenceMonitor().createNewObject(“Mobj”, medium);
sys.getReferenceMonitor().createNewObject("Hobj", high);

...
```

Al diseñar su sistema, considere las siguientes preguntas:
¿Existen fallas de seguridad en este diseño? ¿Cuáles son? ¿Cómo se pueden arreglar?
Como ejemplo de ejecución, dada la lista de instrucciones de la Parte 1, mi implementación da el siguiente resultado.
Nota: ninguna de estas salidas sería visible para hal y lyle.

```
> java SecureSystem instructionList

Reading from file: instructionList
```

Bad Instruction
The current state is:
lobj has value: 0
mobj has value: 0
hobj has value: 0
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 0

Bad Instruction
The current state is:
lobj has value: 0
mobi has value: 0
hobj has value: 0
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 0

lyle writes value 10 to lobj
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 0
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 0

hal reads lobj
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 0
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 10

lyle writes value 20 to hobj
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 10

hal writes value 200 to lobj
The current state is:
lobj has value: 10
mobj has value:0
hobj has value: 20
lyle has recently read: 0
moe has value: 0

hal has recently read: 10

hal reads hobj
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 20

lyle reads lobj
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20
lyle has recently read: 10
moe has recently read: 0
hal has recently read: 20

lyle reads hobj
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 20

Bad Instruction
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 20

Bad Instruction
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20
lyle has recently read: 0
moe has recently read: 0
hal has recently read: 20

Bad Instruction
The current state is:
lobj has value: 10
mobj has value: 0
hobj has value: 20

lyle has recently read: 0
moe has recently read: 0
hal has recently read: 20

La asignación – Parte 2

Actualice su sistema de seguridad, agregando las siguientes tres instrucciones:

CREATE nombre_sujeto nombre_objeto
DESTROY nombre_sujeto nombre_objeto
RUN nombre_sujeto

La semántica de CREATE es tal que un nuevo objeto es adicionado al estado con SecurityLevel igual al nivel del sujeto que lo crea. Inicialmente tiene un valor de 0. Si ya existe un objeto con el mismo nombre, en cualquier nivel, la operación no es válida (no se ejecuta).

DESTROY eliminará el objeto indicado del estado, asumiendo que el objeto existe y que el sujeto tiene acceso de WRITE sobre el objeto, de acuerdo a la propiedad * de BLP. Si no, la operación no es válida (no se ejecuta).

RUN permite que el sujeto indicado, ejecute algún código privado arbitrario. No tiene acceso a ninguno de los objetos del estado. Modela lo que sea que procesa el sujeto y puede hacer lo que desee con el valor que tenga en la variable TEMP. Debe codificar esto como un método dentro de su clase SecureSubject.

Para este trabajo, el objetivo de RUN es permitirle a lyle hacer cualquier procesamiento que sea necesario para obtener el valor del bit que le envía hal, agregarlo a un byte que está creando, y si el byte está completo, escribirlo en la salida.

Tener en cuenta que las llamadas a RUN harán cosas diferentes, dependiendo del valor del estado local, particularmente el sujeto de ejecución. Es decir, RUN hace cosas muy diferentes para lyle que para hal. El código que RUN ejecuta puede hacer cualquier cosa con el estado local del sujeto ejecutando, pero no deben acceder a ninguno de los objetos del sistema. Puede tener variables locales y manipularlas como quiera, pero no puede acceder a ninguna de las SecureObjects.

Al igual que con la primera parte, definirá tres sujetos, hal, moe y lyle, pero no definirá objetos inicialmente. Va a generar las instrucciones necesarias para pasar información de hal a lyle usando el canal secreto que se indica más abajo. Sin embargo, genere las instrucciones para ejecutarse sobre la marcha en lugar de leerlas desde un archivo externo. Eso significa que su programa genera instrucciones y las ejecuta según sea necesario. No hay archivo de instrucciones.

La ejecución de las instrucciones generadas debe implementar un canal secreto que pase información, un bit por vez, de hal a lyle. Los sujetos existentes serán hal, moe y lyle y tienen esos nombres. También tenga en cuenta que puede evitar la mayoría de los controles sintácticos y de análisis, ya que debe generar solo instrucciones correctas.

Su programa debe registrar las instrucciones que genera, una por una, en un archivo llamado "log". Esto nos permitirá asegurarnos de que estás utilizando el canal secreto para transferir la información.

Implemente el siguiente canal encubierto:

Para enviar un bit en 0, hal ejecuta:

```
RUN     hal
CREATE  hal obj
```

Para enviar un bit en 1, hal solo ejecuta:

```
RUN     hal.
```

En el caso de moe, siempre ejecuta lo siguiente:

```
WRITE   moe obj 9
RUN     moe
```

lyle recibe el bit (mediante la presencia o ausencia del objeto), mediante la ejecución de:

```
CREATE  lyle obj
WRITE   lyle obj 1
READ    lyle obj
DESTROY lyle obj
RUN     lyle
```

Si lyle ve que devuelve un valor de 1 de la instrucción READ, entonces su CREAR tuvo éxito, y hal no ha creado el objeto previamente, (enviando un bit en 1)

Si lyle ve que devuelve un valor de 0 de la instrucción READ, entonces hal creó previamente el objeto y el CREATE de lyle ha fallado, y hal ha enviado un bit en 0. (Tenga en cuenta que el valor de 1 en el WRITE podría haber sido cualquier valor distingo de cero).

La instrucción RUN de lyle le permite hacer lo que tenga que hacer para registrar el bit en su estado interno, agregarlo al byte que acaba de crear, y sacar el byte si ha recibido el 8vo bit para ese byte.
Definitivamente necesitas RUN para lyle. No tienes permitido hacer esto en una rutina de nivel superior.

El objetivo es enviar un flujo de bits arbitrario (en realidad, un archivo ASCII completo) a través de este canal. Se recomienda hacer funcionar su programa enviando un pequeño número fijo de bits y asegurándose de que lleguen correctamente. Sin embargo, la versión que entregue deberá tener un parámetro de nombre de archivo en la línea de comando. Su programa leerá el contenido del archivo (probablemente un byte a la vez usando un ByteArrayInputStream), convertir cada byte en 8 bits, enviarlos a través del canal encubierto, reconstruir el byte en el lado de la recepción y escribirlo a un archivo. La idea es transferir los contenidos de un archivo ASCII arbitrario a través del canal.

Una alternativa al uso de ByteArrayInputStream, se puede leer una línea del archivo y ponerla en una cadena y acceder a los bits desde ahí. No lea todo el archivo hacia una cadena. Eso no es escalable.

Su programa puede funcionar para archivos de tipo no ASCII, pero no tiene que ser así. En un sistema realista donde los sujetos están ejecutando en hilos concurrentes, eso sería muy útil porque lyle tendría que saber en qué momento puede dejar de recibir. Si solo le preocupan los archivos tipo ASCII, puede asumir que el archivo solo tendrá caracteres ASCII. Eso significa que puede usar un carácter no ASCII para señalar el final de la entrada. Por ejemplo, si lyle recibe un byte nulo 00000000, entonces sabe que puede dejar de recibir. Pero para esta versión, su programa principal probablemente sabrá cuando hal está sin bits para enviar y simplemente deja de generar instrucciones para lyle.

Tenga presente que la secuencia de ejecución de hal, de moe y de lyle puede ser arbitraria, es decir, no siempre ejecutará primero hal y luego lyle, podría ejecutar moe en medio.
Si definimos H como una ejecución de hal, M una ejecución de moe y L una ejecución de lyle, una posible cadena de ejecuciones podría ser, por ejemplo:
HHMLMHLMLLMHLMHMHHMLHMLLMLHML.
Se suministrará un archivo de texto “secuencia.txt” con la secuencia en que ejecutan los diferentes procesos.

Su programa deberé considerar este hecho, a los efectos de verificar que no se pierda nada de información en la transmisión y que la misma sea completa.

Por lo general, la lectura de los bytes desde el archivo, abrirlos en bits, etc, debería ser realizado por los sujetos utilizando su operación RUN.

La cadena de ejecuciones “secuencia.txt” y el archivo de texto “mensaje.txt” a ser transferido por el canal oculto, estarán disponibles en Aulas dos semanas antes de la fecha de entrega.

Tenga presente que el carácter de fin de línea puede estar representado por CR/LF o CR dependiendo del sistema que utilice.

Deberá verificar que el archivo enviado es idéntico al archivo recibido.