



**AUTENTICACIÓN CON CERTIFICADOS DIGITALES
CUALIFICADOS, CONTROL DE ACCESO Y SUBASTAS
CONFIABLES PARA COMPRAS DE MEDICAMENTOS EN
[SERVICIO DE SALUD DE UN GOBIERNO DE EUROPA]**

Fecha de realización: 22/05/2024

Consultoría realizada por:

Juan Luis Ruano Muriedas

José Joaquín Rojas Romero

Antonio José Suárez García

Índice

Índice.....	2
Resumen ejecutivo.....	3
Metodología y resultados.....	4
Consulta 1.....	4
Consulta 2: Verificación Automática de la Política de Control de Acceso teniendo en cuenta conflictos de intereses, separación/segregación de deberes dinámica y binding de deberes en el Servicio de Salud.....	6
Consulta 3: Desarrollo de smart contracts seguros.....	9

Resumen ejecutivo

Para la **consulta 1** se han detallado los procesos de generación de **Certificate Signing Request** para los empleados y para los servidores Apache mediante el uso de **OpenSSL**.

Para la **consulta 2** se ha realizado un script de verificación automática para el **reparto de tareas** de los empleados del Servicio de Salud. Siguiendo las **restricciones** propuestas y tratando de hacer un **reparto equitativo** de las tareas se ha diseñado el script como un **problema de emparejamiento de grafos** y usando teoría de grafos para resolverlo. Creemos que este resultado hará que la solución quede **más legible**, el planteamiento **más sencillo** y adicionalmente **no será necesario librerías externas**.

Para la **consulta 3** relacionada con la generación de **smart contracts seguros**, se ha provisto de un script programado en el lenguaje **Solidity** a través del entorno **Remix IDE**. Dicho script, llamado **smart_contract.sol** se encarga de cumplir con todos y cada uno de los puntos descritos para que los proveedores de medicamentos puedan realizar las pujas hasta una fecha límite, matizando en los casos en los que, por ejemplo, dos proveedores pujen al mismo tiempo. En resumen, este **smart contract** implementa una subasta de medicamentos con las siguientes características: establece un **valor de mercado** y una **fecha de finalización** de la subasta, permite a los participantes **realizar pujas**, verificando que cumplan con ciertos requisitos, implementa un **bloqueo temporal** para evitar **pujas simultáneas exactamente al mismo tiempo**, determina al **ganador** o ganadores de la subasta al finalizar el **plazo establecido**, **devuelve las fianzas** a los participantes no ganadores y **transfiere el premio** a los ganadores, y **emite eventos** para notificar sobre **nuevas pujas** y la **finalización de la subasta**.

Metodología y resultados

Consulta 1

1. ¿Cómo se puede generar la solicitud de los certificados digitales a las Autoridades de Certificación cualificadas para las partes interesadas del Servicio de Salud?

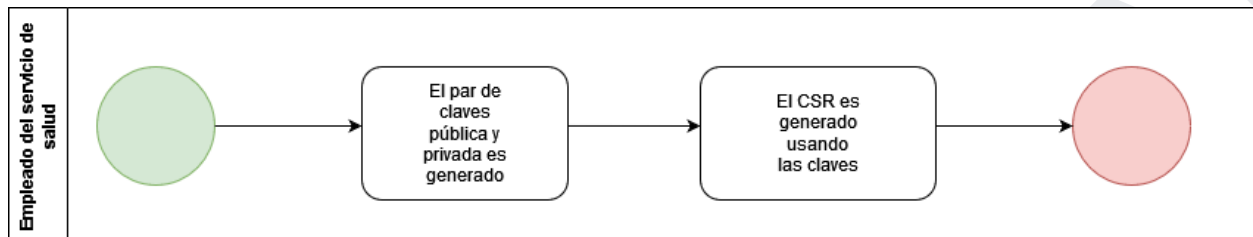


Figura 1. Proceso de generación de un certificado CSR

Para generar una **Certificate Signing Request(CSR)**, el empleado primero deberá generar un par de claves pública y privada. Para ello, recomendamos el uso de **OpenSSL**. Para usar OpenSSL lo más sencillo es que el empleado instale Git y abra Git Bash. Una vez dentro, ejecutar los siguientes comandos:

```
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt  
rsa_keygen_bits:4096  
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Una vez generado el par de claves, el empleado deberá generar el CSR desde el mismo Git Bash con el comando:

```
openssl req -new -key private_key.pem -out request.csr
```

Una vez ejecutado el comando, rellene con los datos que se le solicitan.

Con esto, el CSR ya ha sido generado(request.csr) y solo falta enviarlo a la Autoridad de Certificación.

2. ¿Cómo se puede generar la solicitud a las Autoridades de Certificación de los certificados digitales para los servidores Web de la Entidad?

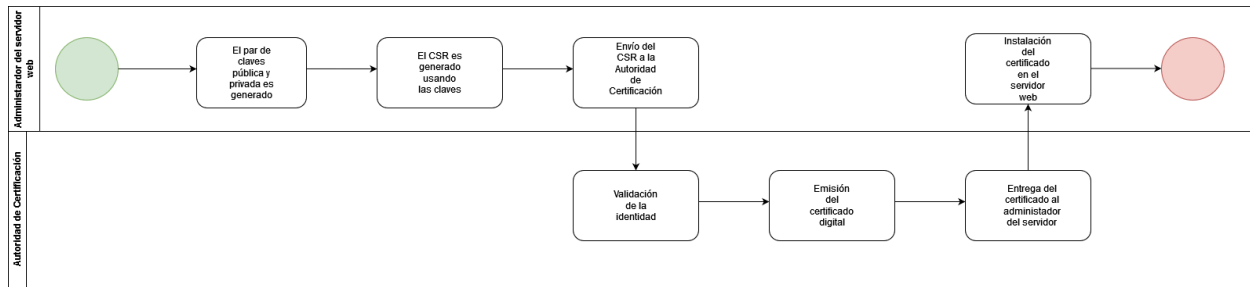


Figura 2. Proceso de obtención del certificado digital para un servidor web.

El proceso para generar una CSR para un servidor es prácticamente idéntico al proceso para generar una CSR para un empleado. El administrador del servidor deberá instalarse OpenSSL y ejecutar el comando:

```
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
rsa_keygen_bits:4096
```

Una vez generado el par de claves, ejecutar el comando:

```
openssl req -new -key private_key.pem -out request.csr
```

Al ejecutar el comando, OpenSSL le pedirá que ingrese la información requerida de manera interactiva, la cual contiene campos como el país, la provincia o un nombre común, entre otros. Tras esto, el CSR se envía a la CA para su validación. La CA puede validar el dominio de varias formas, como mediante un desafío HTTP (subiendo un archivo a un servidor web), un desafío DNS (añadiendo un registro DNS específico), o un correo electrónico enviado a una dirección administrada.

Consulta 2: Verificación Automática de la Política de Control de Acceso teniendo en cuenta conflictos de intereses, separación/segregación de deberes dinámica y binding de deberes en el Servicio de Salud

El problema propuesto trata de conseguir el **emparejamiento equitativo de tareas** con los empleados del servicio, y se puede presentar como un problema de programación lineal pero nosotros hemos querido escoger **otro punto de vista** y hacerlo como un problema de **emparejamiento máximo** al tratarlo como si fuera un **grafo bipartito**. Se ha decidido hacerlo así porque creemos que el **código quedará más legible, el planteamiento es más sencillo y no será necesario librerías externas**.

Vamos a pasar a describir las restricciones dando algunas **observaciones extra** que se han encontrado:

- R1: Las tareas T2.1 y T2.2 deben ser realizadas por usuarios diferentes (Separación de deberes). *Esta restricción está **marcada en el código**.*
- R2: T3 y T4 deben ser realizadas por usuarios distintos (Separación de deberes). *Esta restricción **nunca dará lugar** ya que según tenemos entendido nadie posee un rol que pueda realizar ambas tareas, exceptuando el DG pero está restringido por R4.*
- R3: Si GTR realiza T2.1 su compañero MDS se le debe asignar la tarea T2.2 (Binding de deberes). *Esta restricción también está **marcada en el código**.*
- R4: Si JVG participa en alguna instancia, solamente podrá participar en una única tarea que es la tarea T1. (Conflicto de Intereses). *JVG posee el rol de DG y por tanto tiene la habilidad de realizar cualquier tarea pero esta restricción **solo le permite realizar la T1**.*
- R5: La participación de los actores en las 20 instancias debe estar equilibrada (la carga laboral de dichas instancias debe estar repartida por igual entre todos los empleados) dentro de las posibilidades que determinan todas las anteriores restricciones. *Existen algunos empleados que son los únicos que pueden realizar tareas específicas lo que **condicionará a las instancias** de tal forma que será necesario que trabajen en todas.*

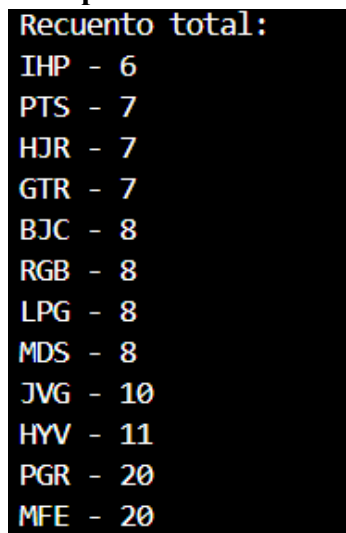
Una vez vistas las restricciones, vamos a pasar al **planteamiento del problema**. Si bien tenemos que cada tarea solo pueden realizarla ciertos roles (los asignados o los heredados) podremos **fusionar ambas tablas** dando lugar a una que **relacione las tareas con quien puede realizarlas**:

```
tareas_asignados = {  
    'T1': ['HYV', 'JVG'],  
    'T2.1': ['GTR', 'LPG', 'RGB', 'HYV', 'BJC'],  
    'T2.2': ['RGB', 'MDS', 'LPG', 'HYV'],  
    'T3': ['PGR'],  
    'T4.1': ['MFE'],  
    'T4.2': ['HJR', 'PTS', 'IHP']  
}
```

Figura 1. Tabla que relaciona las tareas con los actores que pueden realizarlas

Se habrá fijado que **he separado la tarea 4 en dos** y es que al necesitar dos personas, una de cada rol para completarla **es más sencillo plantearlo de esta manera**. Una vez tenemos esta tabla, el problema consiste en resolver el emparejamiento máximo del grafo bipartito donde las tareas es un grupo de vértices y el personal el otro. Por supuesto, **el script intentará equilibrar el número de tareas asignadas entre los miembros del servicio pero nunca podrá saltarse las restricciones**.

Para los resultados, **podrá ver todas las iteraciones ejecutando el script** el cual posee descripciones en el código y señalan dónde están las restricciones, sin embargo, nos gustaría hacer algunas **observaciones sobre el reparto total** de tareas:



```
Recuento total:  
IHP - 6  
PTS - 7  
HJR - 7  
GTR - 7  
BJC - 8  
RGB - 8  
LPG - 8  
MDS - 8  
JVG - 10  
HYV - 11  
PGR - 20  
MFE - 20
```

Figura 2. Reparto total de tareas del personal

Como habrá podido observar, PGR y MFE poseen el número máximo de asignaciones dado a que como puede observar en la Figura 1 **son los únicos que pueden desempeñar las tareas 3 y 4.1** y por tanto es necesario que estén en todas las iteraciones **o no se podría completar la instancia**. Después vemos que JVG y HYV al ser **los únicos que pueden desempeñar la tarea 1** se reparten el trabajo de esta en la mayoría de ocasiones aunque HYV al tener un rol más flexible se le ha requerido en algún puesto extra. En el caso de **los demás, las tareas se ha podido repartir con más equilibrio** ya que un mayor número de gente puede realizarlas.

Esperemos que este planteamiento y el script proporcionado haya sido de utilidad y de su interés, siéntase libre de contactar con nosotros para cualquier problema o si necesita algún otro servicio nuestros expertos podrán ayudarle con cualquier actividad de verificación automática.

CONFIDENTIAL

Consulta 3: Desarrollo de smart contracts seguros

Para realizar las posibles pujas por parte de los proveedores ante el Servicio de Salud Pública, nuestro equipo ha desarrollado un script en el lenguaje Solidity para gestionar los *smart contracts* empleando la red de *Ethereum*.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.7.0 <0.9.0;
3
4  contract Subasta {
5      address payable public owner;
6      uint256 public endDate;
7      uint256 public mercado;
8      uint256 public maxPujadores = 30;
9      bool public bloqueoPujas;
10 }
```

Figura 3.. Parámetros del contrato

En la **figura 3** se muestra cómo se ha modificado un parámetro del contrato de la subasta. En concreto se ha establecido un número máximo de pujadores igual a 30, tal y como especificaba el requisito. Este parámetro será usado en posteriores funciones.

```
15  struct Puja {
16      address payable participante;
17      uint256 monto;
18      bool activa;
19  }
20
21  Puja[] public pujasArray;
22  mapping(address => uint256) public fianzas;
23  Puja[] public ganadores;
24
25  event NuevaPuja(address indexed participante, uint256 monto);
26  event SubastaFinalizada(address[] ganadores, uint256[] montos);
27
```

Figura 4. Estructura de la puja.

En la **figura 4** se modifica la estructura Puja y se agrega un campo *activa* para controlar si una puja está activa o no. Además, se agrega un mapping *fianzas* para almacenar las fianzas de cada participante (línea 22).

Por otro lado se elimina la función *tasa()* y se modifica la función *pujar(uint256 monto)* para cumplir con los requisitos: verificar que la puja sea un valor entero positivo, verificar que la puja no supere el valor de mercado, verificar que no se haya alcanzado el límite de pujadores,

verificar que la subasta no haya finalizado, verificar que el participante no haya pujado anteriormente y retener el 10% de la puja como fianza y transfiere el 90% restante al participante. Dichos cambios se ven reflejados en la siguiente **figura 5**. Además, se modifica la función *pujar(uint256 monto)* para que los participantes envíen la cantidad exacta de la puja como valor de la transacción.

```
50 function pujar(uint256 monto) public payable {
51     require(monto > 0, "La puja debe ser un valor entero positivo");
52     require(monto < mercado, "La puja no debe superar el valor de mercado");
53     require(pujasArray.length < maxPujadores, "Se ha alcanzado el límite de pujadores");
54     require(block.timestamp < endDate, "La subasta ha finalizado");
55     require(msg.value == monto, "Debes enviar la cantidad exacta de la puja");
56
57     if (bloqueoPujas) {
58         require(block.timestamp % 2 == 0, "Espere un momento y vuelva a intentar la puja");
59     }
60
61     for (uint256 i = 0; i < pujasArray.length; i++) {
62         require(pujasArray[i].participante != msg.sender, "No se permiten pujas duplicadas");
63     }
64
65     pujasArray.push(Puja(payable(msg.sender), monto, true));
66     fianzas[msg.sender] = monto;
67
68     emit NuevaPuja(msg.sender, monto);
69
70     if (pujasArray.length == maxPujadores || block.timestamp >= endDate) {
71         finalizarSubasta();
72     }
73 }
```

Figura 5. Función de puja.

Se ha agregado una función privada *finalizarSubasta()* que se encarga de **determinar al ganador y devolver las fianzas** a los participantes no ganadores, de manera que: determina la puja ganadora y la **segunda puja** más baja (figura 4, líneas de la 81 a la 86 en la **figura 6**), transfiere al ganador la cantidad de la segunda puja más baja (línea 108 en la **figura 7**) y devuelve a los participantes no ganadores el 10% de su puja. Todos estos requisitos pueden verse reflejados en las siguientes capturas, **figuras 6 y 7**.

Además se han agregado nuevos eventos: *NuevaPuja* y *SubastaFinalizada* para notificar sobre nuevas pujas y el resultado de la subasta (línea 109 de la **figura 7**).

```

75     function finalizarSubasta() private {
76         require(block.timestamp >= endDate, "La subasta aún no ha finalizado");
77
78         uint256 pujaGanadora = type(uint256).max;
79         uint256 pujaSegunda = type(uint256).max;
80
81         for (uint256 i = 0; i < pujasArray.length; i++) {
82             if (pujasArray[i].monto < pujaGanadora && pujasArray[i].activa) {
83                 pujaSegunda = pujaGanadora;
84                 pujaGanadora = pujasArray[i].monto;
85             } else if (pujasArray[i].monto < pujaSegunda && pujasArray[i].activa && pujasArray[i].monto != pujaGanadora) {
86                 pujaSegunda = pujasArray[i].monto;
87             }
88         }
89
90         for (uint256 i = 0; i < pujasArray.length; i++) {
91             if (pujasArray[i].monto == pujaGanadora) {
92                 ganadores.push(pujasArray[i]);
93             } else if (pujasArray[i].activa) {
94                 pujasArray[i].participante.transfer(fianzas[pujasArray[i].participante]);
95                 fianzas[pujasArray[i].participante] = 0;
96             }
97         }

```

Figura 6. Función de fin de subasta

```

99         address[] memory ganadoresArray = new address[](ganadores.length);
100         uint256[] memory montosArray = new uint256[](ganadores.length);
101
102         for (uint256 i = 0; i < ganadores.length; i++) {
103             ganadoresArray[i] = ganadores[i].participante;
104             montosArray[i] = pujaSegunda;
105             fianzas[ganadores[i].participante] = 0;
106         }
107
108         owner.transfer(pujaSegunda * ganadores.length);
109         emit SubastaFinalizada(ganadoresArray, montosArray);
110     }

```

Figura 7. Continuación de la función de fin de subasta.

Se agrega una función *reclamarPremio()* que permite a los ganadores reclamar su premio después de que finalice la subasta. Verifica que la subasta haya finalizado y que el participante sea un ganador, y transfiere el premio (segunda puja más baja) al participante y establece su fianza en 0. Estos cambios pueden verse integrados en la siguiente figura 8.

```

112     function reclamarPremio() public {
113         require(block.timestamp >= endDate, "La subasta aún no ha finalizado");
114         require(fianzas[msg.sender] > 0, "No eres un ganador de la subasta");
115
116         uint256 premio = fianzas[msg.sender];
117         fianzas[msg.sender] = 0;
118         payable(msg.sender).transfer(premio);
119     }
120 }

```

Figura 8. Función de reclamar premio

Este código cumple con los requisitos adicionales, incluyendo el envío de la cantidad exacta de la puja como valor de la transacción, el almacenamiento de fianzas, la determinación de múltiples ganadores en caso de pujas iguales, y la función para reclamar el premio después de que finalice la subasta.

Para **mitigar problemas de pujas simultáneas** en el contrato inteligente de la subasta, se puede implementar un mecanismo de **bloqueo temporal** que evite que dos pujas se registren exactamente al mismo tiempo. A continuación se muestra una posible implementación de dichas funciones, trabajando y siendo llamadas dentro de la función **pujar()**.

```
40     function activarBloqueoPujas() public {
41         require(msg.sender == owner, "Solo el owner puede activar el bloqueo de pujas");
42         bloqueoPujas = true;
43     }
44
45     function desactivarBloqueoPujas() public {
46         require(msg.sender == owner, "Solo el owner puede desactivar el bloqueo de pujas");
47         bloqueoPujas = false;
48     }
```

Figura 9. Funciones para el bloqueo de pujas en caso de pujas realizadas al mismo tiempo.

En la **figura 5** (línea 57) puede contemplar cómo se han realizado las llamadas a los bloqueos de pujas desde la misma función de **pujar()**. Este código incluye la lógica para activar y desactivar un bloqueo temporal que evita pujas simultáneas exactamente al mismo tiempo. Cuando se activa el bloqueo, se verifica que el timestamp actual sea par antes de permitir que se registre una nueva puja. Una vez que se desactiva el bloqueo, las pujas pueden registrarse normalmente.

De esta manera, el modelo de smart contract diseñado, cuenta con el **cumplimiento de los siguientes requisitos**:

- Las pujas que se admiten deberían tener un valor positivo entero mayor de cero, toda puja que no cumpla esto no será considerada.
- También hay un valor máximo que determina el Servicio de Salud, por encima del cuál no se admiten pujas. Esto evita que se tenga que pagar un valor superior al que tiene estimado el Servicio de Salud que vale el suministro de medicamentos.
- Los pujantes no pueden pujar dos veces, por tanto hay que controlarlo.
- Puede haber hasta 30 pujadores en la subasta, una vez cubierto se cierra la subasta
- La subasta se abre con una fecha determinada al crear el Smart contract
- La subasta se cierra cuando hay 30 pujas o se llega al deadline o fecha límite. Cualquier apuesta después del deadline no está admitida.

- Gana la subasta el que presenta la puja más baja al Servicio de Salud, pero se le pagará lo que haya pujado el segundo más alto. En caso que no haya dos pujadores o más, se le pagará lo que él ha pujado.
- Se paga al hacer la puja al menos el 10% de lo pujado, en caso que sean ganadores de la subasta, se le devolverá sumando al costo del suministro dicha cantidad, en caso que sea ganador y no suministre el material, no se le devolverá. A todos los pujadores se le devolverá la cantidad entregada una vez finalizada la subasta.
- Es necesario que los pujadores puedan conocer al ganador cuando finalice la subasta.
- El código debe recoger que los participantes en la subasta deben entregar una cantidad de Ethereum cuando hacen una puja, y se le deben devolver a cada uno cuando finalice todo el proceso de subasta, excepto al ganador si no proporcionase la cantidad de medicamento estipulado.
- El precio de las cantidades que apuestan los proveedores debe ser siempre inferior al precio de mercado en el momento de comienzo de la subasta y aparecerá reflejado en el Smart contract. Igualmente debe aparecer cuando se finalizará la subasta de medicamentos y las cantidades aportadas por los proveedores no ganadores de la subasta se deben devolver una vez finalizada la subasta.

Puede comprobarse el **script completo** con más detalles sobre la implementación del *smart contract* en el fichero adjunto llamado *smart_contract.sol*. Cualquier duda acerca de la ejecución y/o implementación no dude en ponerse en contacto con nuestro equipo y le responderemos con la información que requiera en cualquier momento.