Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Sistema de monitorización remota de equipos

Remote systems monitoring tool

Álvaro Fernández Rodríguez

La Laguna, 9 de junio de 2021

D. **Pino Caballero Gil**, con N.I.F. 45534310Z; Catedrática de Universidad adscrita al Departamento de Ingeniería Informática y Sistemas de la Universidad de La Laguna, como tutora

D. **Igor Lukic**, con N.I.F. 16819087S; Redteam Leader de OneCyber y fundador de Hackron, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

”*Sistema de monitorización remota de equipos*”

ha sido realizada bajo su dirección por D. **Álvaro Fernández Rodríguez**, con N.I.F. 79063770M.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de junio de 2021

Agradecimientos

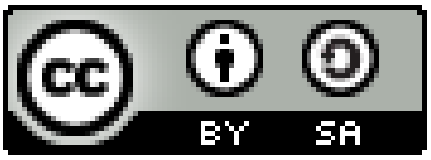
A Pino e Igor, por su esfuerzo y apoyo desinteresado durante este año tan duro.

A Adriana, por ayudarme a seguir adelante.

A mis amigos, por acompañarme en todo momento.

A mi familia, por darme la motivación que necesitaba para terminar la carrera.

Licencia



Resumen

En el mundo de los datacenters, sistemas de alto rendimiento o infraestructuras críticas, es innegable la necesidad de herramientas que permitan monitorizar los equipos informáticos, tanto a nivel de software como hardware. Es por ello que la mayoría están pensadas para ser usadas por personas con grandes conocimientos de informática, que buscan gran profundidad y capacidad. Cuando hablamos de redes domésticas, empresas pequeñas o proyectos personales, es complicado encontrar una alternativa sencilla para asegurar el estado de nuestros equipos. Por esta razón, este proyecto presenta una alternativa cuyo objetivo es conseguir una monitorización simple pero efectiva, a la vez que una presentación atractiva de los datos recogidos.

Palabras clave: Monitorización, ingeniería de sistemas, sistemas operativos, C++, Grafana

Abstract

In the world of data centers, high-performance systems or critical infrastructures, there is no denying the need for tools to monitor computer equipment, both at the software and hardware level. That is why most of them are designed to be used by highly qualified computer scientists, who are looking for great depth and capacity. When we talk about home networks, small companies or personal projects, it is difficult to find a simple alternative to ensure the state of our equipment. For this reason, this project presents an alternative whose objective is to achieve a simple but effective monitoring, as well as an attractive presentation of the collected data.

Keywords: Monitoring, systems engineering, operating systems, C ++, Grafana

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Motivación

Debido a la naturaleza de los entornos en los que se necesitan sistemas de monitorización [1], estos han evolucionado hacia la excesiva dificultad y la sobrecarga de funciones. Por ello, para controlar el estado de un sistema de pocos equipos no críticos, hay que preparar una infraestructura desproporcionada en complejidad y acabar teniendo demasiados datos y opciones innecesarias.
Debido a esto y el auge de los sistemas de computación en la nube, están surgiendo nuevas soluciones (Ver Figura 1.1) con mayor facilidad de integración y atractivo, como Grafana y su sistema de monitorización Prometheus.

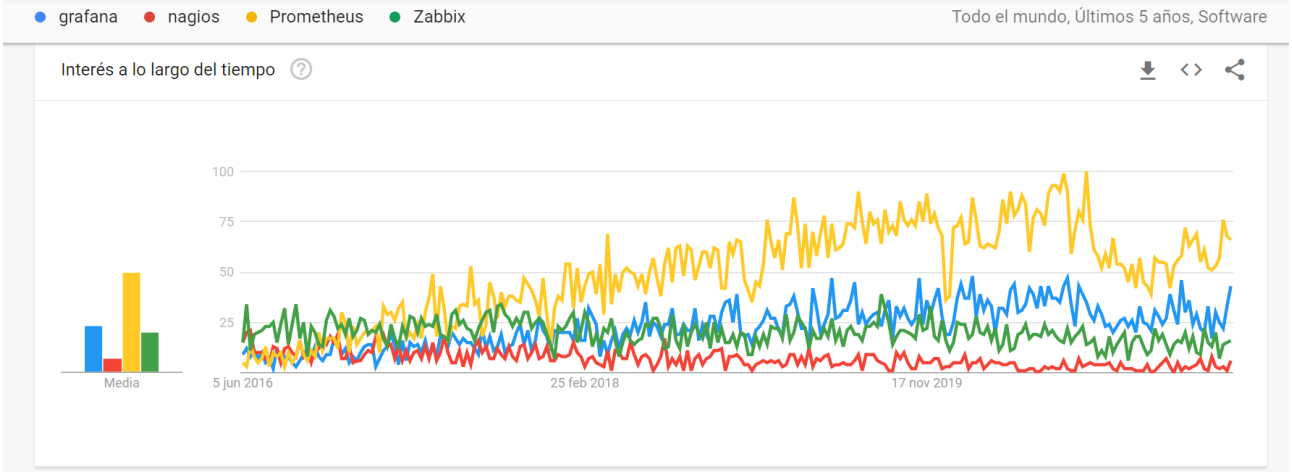


Figura 1.1: Gráfica de búsquedas en Google desde 2016

1.2. Objetivos

Este proyecto proporciona una alternativa de monitorización básica viable para sistemas domésticos, o entornos empresariales reducidos. Para ello, ha sido necesario diseñar una herramienta con poca complejidad de uso, pero flexible. Además, ha sido vital incluir un sistema de representación de datos a la vez sencillo y atractivo. Por otro lado, no deja de tratarse de un sistema de monitorización en el que necesitamos confiar, es decir, que es necesario garantizar el transporte seguro de la información.

1.3. Fases

Fase del desarrollo	Tiempo
Investigación inicial	1 semana
Aplicación cliente	1 semana
Aplicación servidor	1 semana
Implementación de base de datos	1 semana
Diseño de Grafana y acceso a MySQL	2 semanas
Comunicación segura	3 semanas
Soporte para múltiples clientes	2 semanas
Sistema de alertas	2 semanas

1.4. Estructura de la memoria

En el primer capítulo se aborda la idea general, las fases del proyecto y el estado actual del tema. En el segundo, se introduce el diseño del sistema y la elección de tecnologías, para en el tercero explorar la aplicación final. A continuación, en el cuarto se revisarán los aspectos más importantes de la codificación, y en el quinto se incluyen las conclusiones y los trabajos futuros. Por último, en el sexto se elabora un presupuesto tanto del desarrollo, como de su despliegue en un entorno SaaS.

1.5. Descripción del problema

Debido a la naturaleza crítica de los sistemas de monitorización [2], la mayoría de soluciones buscan atraer al público corporativo. Para lograrlo, normalmente sobrecargan sus aplicaciones de opciones para soportar todo tipo de sistemas y *software*, y crean esquemas de monetización restrictivos evitando dar soporte a usuarios que no compren licencias.

1.6. Aplicaciones similares

1.6.1. Nagios

Como podemos observar en la Figura 1.2, el esquema de Nagios no representa una idea pensada para usuarios domésticos, ya que tampoco es su objetivo. Programado en Perl con un sistema basado en plugins, necesita utilizar *forks* del proyecto para que la interfaz sea más asequible. Tanta es su complejidad, que ofrecen certificaciones de pago con validez de 2 años [3]. Ya que su *target* son entornos empresariales, su modelo de negocio hace que tengan que ocultar características y soporte detrás de un muro de pago.

1.6.2. Pandora

Al igual que Nagios, Pandora (ver Figura 1.3) está programado en Perl, aunque en este caso se trata de un desarrollo libre español. Igualmente, su modelo de negocio esconde una librería de módulos mucho más completa [4] y el soporte detrás de paquetes de pago. Aunque la interfaz es más amigable, tiene menos opciones de personalización que otros programas y una comunidad más limitada.

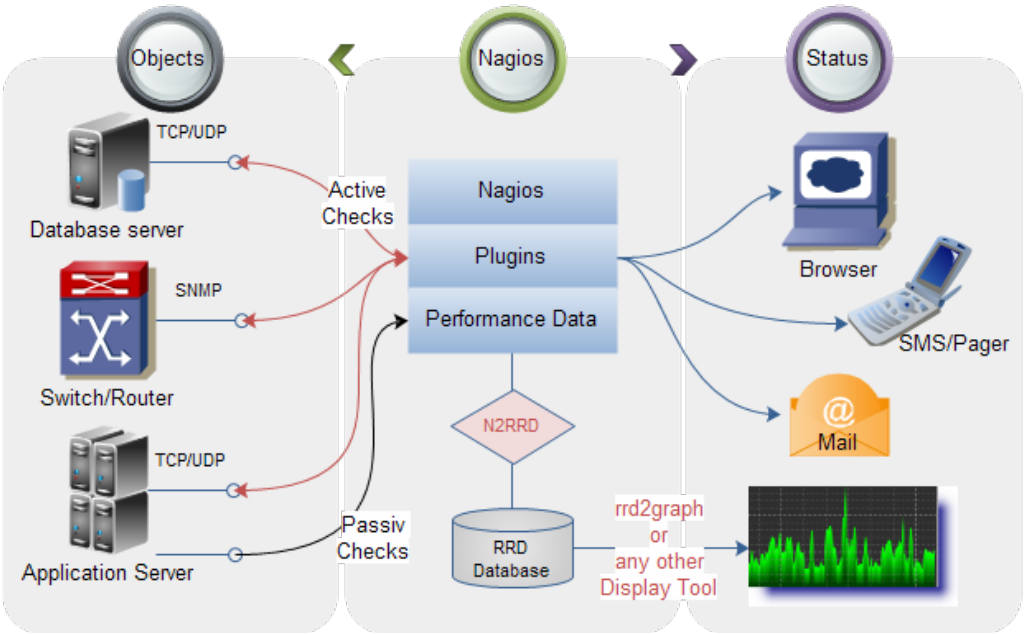


Figura 1.2: Esquema de Nagios

Figura 1.3: Dashboard de Pandora

1.7. Conceptualización de la propuesta

Una solución posible al problema anteriormente descrito es un software sin ánimo de lucro, personalizable y sencillo de utilizar, y ese es precisamente el objetivo de este proyecto. Si permitimos a usuarios y empresas utilizar el mismo *software* libre y gratuito, el soporte comunitario será más efectivo y eliminará en gran parte la necesidad de atención personalizada [5].
Están surgiendo alternativas más sostenibles y respetuosas con la esencia del software libre, como ofrecer *SaaS*, permitiendo a las empresas contratar los servicios *on demand*, en vez de tener que alojarlo en sus servidores.

Capítulo 2

Diseño

2.1. Esquema general

El modelo cliente-servidor es una elección obvia para este caso de uso, ya que el objetivo de todo el sistema es tener la información de todos los clientes centralizada, para tener un fácil acceso y poder responder con rapidez a problemas futuros y presentes. Además, gran parte de la utilidad radica en la base de datos y el sistema de visualización, el cual es la clave de la aplicación (Ver Figura 2.1).

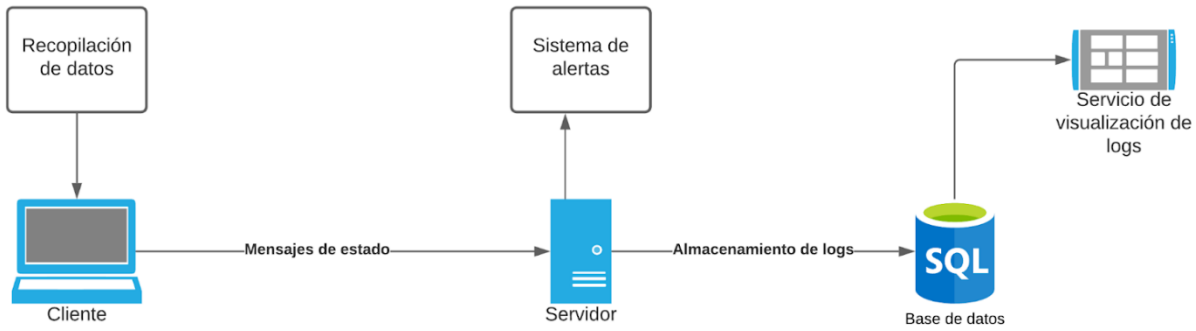


Figura 2.1: Esquema general del sistema

2.2. Cliente

El primer paso es conocer el estado del equipo. Para ello, se ejecutan periódicamente comandos de sistema definidos de antemano, y mediante expresiones regulares se obtienen valores normalmente numéricos. Entonces, el cliente los introduce en una estructura, los encripta con la clave pública del servidor, los firma con su clave privada, y los envía mediante comunicación TCP al servidor. Para ello, en la configuración hay que especificar la dirección y el puerto del servidor.

2.3. Servidor

El servidor se encarga de recibir las actualizaciones, para lo que se utiliza un puerto TCP al cual los clientes envían sus mensajes. Una vez recibidos, se comprueba que la firma corresponde con la del cliente y se desencriptan. También se registran en un hilo de ejecución especial, para controlar el tiempo entre comunicaciones y así detectar caídas. Cuando ya los datos se encuentran en su formato original, se envían al agente de base de datos.

2.4. Base de datos

Una vez el agente de gestión de base de datos recibe los datos a introducir, realiza una conexión con el servidor de MySQL y configura la tabla con los campos necesarios si no está creada. A continuación, prepara una sentencia de tipo INSERT con los valores recibidos, y se manda al servidor para su ejecución.

2.5. Visualización de logs

El servicio de visualización de logs se encarga de recoger los datos del servidor MySQL y representarlos gráficamente, además de proporcionar una interfaz web que puede ser accesible desde internet.
Es vital que la información sea fácilmente accesible y proporcione una idea rápida y clara del estado de los equipos. Para ello, tener múltiples opciones de personalización resulta claramente conveniente, ya que cada sistema tiene su sus estructura y peculiaridades, y necesitará ajustes propios.

2.6. Elección de tecnologías

Una vez realizado el esquema general, hay que encontrar las tecnologías adecuadas para satisfacer nuestros requisitos.

2.6.1. C++

Uno de los lenguajes más famosos y utilizados del mundo, es ideal cuando buscamos la optimización, como es el caso cuando ejecutamos la aplicación en varios sistemas a la vez y de forma permanente [6].
Además, al ser el lenguaje principal del grado, la experiencia ha sido un factor muy importante a la hora de realizar la elección. Su popularidad además asegura la existencia de implementaciones a la hora de integrar otros sistemas, como bases de datos o esquemas de encriptación.

2.6.2. Botan

El objetivo de Botan es ser la mejor librería de criptografía para C++ [7]. Implementa numerosos sistemas como TLS, certificados X.509, *hashing* o algoritmos post-cuánticos. Numerosas organizaciones usan y participan en Botan [8], como Qt, IBM, Bosch, Cisco o VMware. Esto es vital a la hora de confiar la seguridad de nuestro proyecto en una librería *open source*, ya que cualquier fallo será rápidamente detectado y no llegará a la rama principal.

2.6.3. CMake

CMake es un sistema de generación automática de código, utilizada para gestionar la automatización de la construcción de *software*, en este caso C++. Concretamente, nos abstrae del sistema de compilación Make, además de asegurarse de que las dependencias requeridas se encuentren instaladas y localizadas.

2.6.4. MySQL

MySQL es un sistema de gestión de bases de datos relacional, y está considerada la base de datos de código abierto más popular del mundo. Esto nos asegura facilidad a la hora de encontrar librerías de integración con los demás sistemas del proyecto, que es el punto de mayor interés a la hora de la elección de servidor de bases de datos, ya que simplemente se usará como almacén de datos.

2.6.5. TravisCI

TravisCI es un sistema de integración continua, cuyo objetivo es realizar tareas automatizadas en proyectos alojados en GitHub. Utiliza un fichero .yaml para establecer las acciones que deben ejecutarse cada vez que se hace un nuevo *commit*:

```yaml
language: cpp
os: linux

jobs:
    include:
        os: linux
        dist: focal
        addons:
            apt:
                update: true
                packages:
                    - g++
                    - cmake
                    - libmysqlcppconn-dev
                    - botan
```

```
                - libbotan-2-dev

script:
    - mkdir -p build
    - cd build
    - cmake ..
    - make
```

Nos permite establecer pruebas para numerosos sistemas operativos y versiones, creando un entorno virtual para cada uno [9]. Gracias a ello, nos aseguramos de que todos los cambios al código siguen manteniendo la estabilidad del proyecto, sin tener que ejecutar los tests manualmente. Como se observa en la Figura 2.2 se puede crear una imagen dinámica para añadirla a nuestro repositorio y comprobar en ella el resultado de la última ejecución automática.

Figura 2.2: Fragmento de README.md en GitHub

2.6.6. Grafana

Grafana es un visualizador de eventos especializado en herramientas de monitorización. Gracias a su extensa comunidad y soporte de grandes organizaciones [10], disfruta de múltiples integraciones con diferentes sistemas de fuentes de datos, entre ellos MySQL. Además, permite una personalización inigualada por otros *frameworks* y compartir e implementar rápidamente nuevos *dashboards*.

Capítulo 3

Sistema SimpleMon

3.1. Definición

SimpleMon (ver Figura 3.1) es un sistema de monitorización orientado a redes pequeñas y proyectos personales. Utiliza un sistema push [11] de actualizaciones, donde los clientes mandan mensajes periódicos a un servidor central. Este se encarga de coordinar las alertas y almacenar los datos en un sistema MySQL. Una vez recogidos, se utiliza un servicio de Grafana para representarlos según un dashboard personalizable y accesible.

Figura 3.1: Logo SimpleMon

3.2. Ejemplo

En la figura 3.2 podemos apreciar un ejemplo de representación de datos de SimpleMon para 2 equipos monitorizados. Podemos observar que se pueden modificar los tiempos atendidos, los *thresholds* de las variables individuales, y la situación y tamaño de los paneles, lo cual permite revisar rápidamente eventos complejos y revisar el estado concreto de los sistemas en un momento concreto.

Adicionalmente, como vemos en la Figura 3.3 podemos seleccionar de forma dinámica la lista de equipos que queremos monitorizar, y la vista se adaptará automáticamente, añadiendo nuevos paneles para las diferentes fuentes de información.

Figura 3.2: Dashboard de ejemplo para 2 equipos

Figura 3.3: Dashboard de ejemplo para 1 equipo

3.3. Instalación y configuración

Para compilar el proyecto primero hay que instalar las dependencias, las cuales se encuentran disponibles en el repositorio de GitHub del programa. Una vez disponibles, en el proyecto se incluyen 2 scripts, encargados de compilar e instalar tanto el cliente como el servidor.

3.3.1. Cliente

El script de instalación del cliente tiene 2 funciones, compilar el ejecutable e instalarlo en el sistema, configurando el servicio asociado:

```
# Compile
mkdir build
cd build
cmake ..
make client

# Check build exit code
if [ $? -ne 0 ]
then
   echo "Build failed, exiting" >&2
   exit
fi
```

Después se instala el ejecutable generado con sus ficheros de configuración y se crea el servicio:

```
# User creation
sudo useradd -r -s /bin/nologin simplemond

serviceText="[Unit]\n"
serviceText+="Description=SimpleMon Client\n"
serviceText+="After=network.target\n"
serviceText+="[Service]\n"
serviceText+="Restart=always\n"
serviceText+="RestartSec=3\n"
serviceText+="User=simplemond\n"
serviceText+="ExecStart=/usr/sbin/simplemon-client\n"
serviceText+="[Install]\n"
serviceText+="WantedBy=multi-user.target\n"
```

```
# Copy program to /usr/sbin
sudo cp client /usr/sbin/simplemon-client

# Copy config to /etc/simplemon-client
sudo mkdir /etc/simplemon-client
sudo mkdir /etc/simplemon-client/config
sudo mkdir /etc/simplemon-client/keys
sudo cp ../config/client.conf /etc/simplemon-client/config/
sudo chown simplemond
 ↪  /etc/simplemon-client/config/client.conf
sudo chmod 600 /etc/simplemon-client/config/client.conf

# Copy service
sudo rm /etc/systemd/system/simplemon-client.service
echo -e $serviceText | sudo tee -a
 ↪  /etc/systemd/system/simplemon-client.service >/dev/null

systemctl start simplemon-client
systemctl enable simplemon-client
```

3.3.2. Servidor

Muy similar al script para el cliente, el del servidor compila e instala el programa y posteriormente configura el servicio para su ejecución automática:

```
# Compile
mkdir build
cd build
cmake ..
make server

# Check build exit code
if [ $? -ne 0 ]
then
   echo "Build failed, exiting" >&2
   exit
fi
```

Después se instala el ejecutable generado con sus ficheros de configuración y se crea el servicio:

```
# User creation
sudo useradd -r -s /bin/nologin simplemond

# Copy program to /usr/sbin
sudo cp server /usr/sbin/simplemon-server

# Copy config to /etc/simplemon-server
sudo mkdir /etc/simplemon-server
sudo mkdir /etc/simplemon-server/config
sudo mkdir /etc/simplemon-server/keys
sudo cp ../config/server.conf /etc/simplemon-server/config/
sudo chown simplemond
 ↪  /etc/simplemon-server/config/server.conf
sudo chmod 600 /etc/simplemon-server/config/server.conf

sudo cp ../config/sql.conf /etc/simplemon-server/config/
sudo chown simplemond /etc/simplemon-server/config/sql.conf
sudo chmod 600 /etc/simplemon-server/config/sql.conf

# Copy service
sudo rm /etc/systemd/system/simplemon-server.service
echo -e $serviceText | sudo tee -a
 ↪  /etc/systemd/system/simplemon-server.service >/dev/null

systemctl start simplemon-server
systemctl enable simplemon-server
```

3.4. Seguridad

La aplicación tiene 2 focos importantes en los que centrarnos a la hora de analizar la seguridad: la comunicación y la configuración de sistema. Primero se hace uso de cmake para generar el programa:

3.4.1. Comunicación

Para asegurar la comunicación decidimos utilizar una de las librerías de criptografía más potentes actualmente disponible en C++, Botan. Con su módulo de PKI, se generó una clave privada para el servidor y una para

los clientes, con sus correspondientes claves públicas. De esta manera, el cliente puede cifrar los mensajes con la clave pública del servidor y firmarlos con su privada, asegurando la confidencialidad, integridad y autenticación de la comunicación.

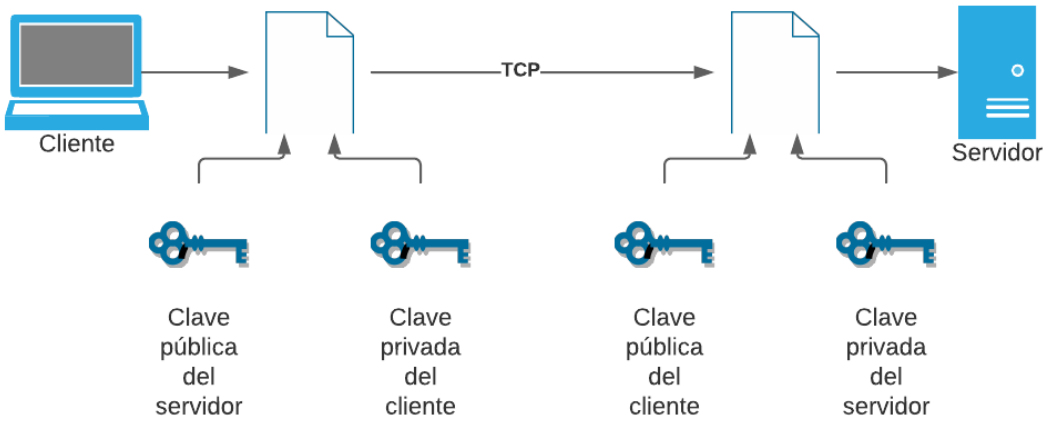


Figura 3.4: Esquema de encriptación de clave pública

3.4.2. Configuración de sistema

Es vital asegurar una correcta securización de los ficheros de configuración y claves de comunicación, ya que de no ser así, un atacante que accediera a nuestro sistema podría falsificar la identidad del equipo, y por ejemplo, seguir reportando falsamente mientras el equipo original se encuentra apagado.
Para ello, los archivos deben ser propiedad del usuario que ejecuta el servicio, además de deshabilitar la lectura y modificación a otros usuarios. En permisos de sistemas Linux correspondería con 600, como se muestra en el listado de archivos de configuración del programa:

```
ls -la config keys
config:
drwxr-xr-x 2 root       root 4096 May 10 17:15 .
-rw------- 1 simplemond root   90 May 10 17:15 server.conf
-rw------- 1 simplemond root   63 Mar 16 22:32 sql.conf

keys:
drwxr-xr-x 2 root       root 4096 Mar 16 19:38 .
-rw------- 1 simplemond root  422 Mar 16 19:34 client.pub
-rw------- 1 simplemond root 1913 Mar 16 19:29 server.priv
```

Capítulo 4

Implementación

4.1. Utilidades generales

4.1.1. Mensajes

El cliente y el servidor comparten una definición de mensaje, para así poder comunicarse mediante una estructura de datos concreta. Cabe destacar el uso del tipo de datos *short* en vez de *int*, ya que podemos ahorrar 2 bytes por campo, permitiéndonos ajustarnos más fácilmente al límite de tamaño que nos impone la solución criptográfica [12]:

```
struct StatusMessage
{
  public:
    std::array<char, 32> hostname;
    short uid = 0;
    short free_mem = 0;
    short free_disk = 0;
    short used_cpu = 0;
};
```

4.2. Cliente

4.2.1. Ejecución de comandos

Para obtener los datos del estado de la máquina, es necesario elaborar una función encargada de ejecutar los comandos en el sistema operativo y recoger las salidas:

```
std::string exec(const char *cmd)
{
    std::array<char, 128> buffer;
```

```
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd,
     ↪  "r"), pclose);
    if (!pipe)
    {
        throw std::runtime_error("popen() failed!");
    }
    while (fgets(buffer.data(), buffer.size(), pipe.get()) !=
     ↪  nullptr)
    {
        result += buffer.data();
    }
    return result;
}
```

4.2.2. Especificación de comandos

Para llenar la estructura de datos, hay que definir de antemano los comandos a ejecutar. Hacemos uso de la función de ejecución de comandos del sistema para realizar consultas sobre los diferentes valores que nos interesan:

```
StatusMessage fillMsg()
{
    StatusMessage msg;
    try
    {
        msg.uid = stoi(exec("id | awk '{ print $1 }' | cut -c
         ↪  5-8"));
        auto hostname = exec("hostname");
        strncpy(msg.hostname.begin(), hostname.data(), 32);
        hostname[31] = '\0';
        msg.free_mem = stoi(exec("cat /proc/meminfo | grep
         ↪  MemFree | awk '{ print $2 }'")) / 1000;
        msg.free_disk = stoi(exec("df -h | grep /dev/sda1 |
         ↪  awk '{ print $4 } ' | "
                                  "sed 's/.$//'"));
        msg.used_cpu = 100 - stod(exec("top -b -i -n 1 | grep
         ↪  '%Cpu(s):' | awk "
```

```
                                                "-F',' '{print $4}' |
                                                 ↪  awk '{print
                                                 ↪  $1}'"));
    }
    catch (const std::invalid_argument &ex)
    {
        std::cerr << ex.what() << std::endl;
    }
    return msg;
}
```

4.2.3. Socket cliente

Para simplificar su uso, se realizó una implementación de abstracción de la librería *socket*. Además, permite una implementación futura de SSL más sencilla:

```
Socket_Client::Socket_Client(std::string addr, int port)
{
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port);

    // Convert IPv4 and IPv6 addresses from text to binary
     ↪  form
    if (inet_pton(AF_INET, addr.c_str(), &serv_addr.sin_addr)
     ↪  <= 0)
    {
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }

    if (connect(sock, (struct sockaddr *)&serv_addr,
     ↪  sizeof(serv_addr)) < 0)
    {
```

```
        std::cout << addr << " " << port << std::endl;
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }
}

Socket_Client::~Socket_Client()
{
    close(sock);
}

void Socket_Client::send(const char *buf, size_t length)
{
    ::send(sock, buf, length, 0);
}
```

4.2.4. Encriptación, firma y envío

El cliente es el encargado de cifrar el mensaje con la clave pública del servidor y firmarlo con su privada. Además, necesitamos el fichero de configuración para obtener la contraseña y otros valores personalizables para modificar la ejecución de nuestro programa. Una vez realizadas las ejecuciones de sistema requeridas para completar el mensaje, el cifrado, y computada la firma, se juntan en una estructura para ser mandadas en un paquete TCP:

```
int main(int argc, char const *argv[])
{
    while (1)
    {
        Config conf;
        conf = parse_config("/etc/simplemon-client/config/cli
         ↪  ent.conf");
        StatusMessage msg = fillMsg();
        char buffer[768] = {0};
        std::unique_ptr<Socket_Client> s =
         ↪  std::make_unique<Socket_Client>(conf.ip_address,
         ↪  conf.port);
```

```
std::unique_ptr<Botan::RandomNumberGenerator> rng(new
 ↪  Botan::AutoSeeded_RNG);

// Load keys
Botan::DataSource_Stream
 ↪  in("/etc/simplemon-client/keys/client.priv");
std::unique_ptr<Botan::Private_Key>
 ↪  priv(Botan::PKCS8::load_key(in,
 ↪  conf.key_password));
std::unique_ptr<Botan::Public_Key>
 ↪  pub(Botan::X509::load_key("/etc/simplemon-client/
 ↪  keys/server.pub"));

Botan::PK_Encryptor_EME enc(*pub, *rng.get(),
 ↪  "EME1(SHA-256)");
Botan::PK_Signer signer(*priv, *rng.get(),
 ↪  "EMSA1(SHA-256)");

std::vector<uint8_t> ct = enc.encrypt((const unsigned
 ↪  char *)&msg, sizeof(msg), *rng.get());

std::cout << "Sending encrypted data: " <<
 ↪  Botan::hex_encode(ct) << std::endl;
signer.update(ct);
std::vector<uint8_t> signature =
 ↪  signer.signature(*rng.get());

if (conf.logs != "none")
{
    std::cout << "Encrypted msg size = " << ct.size()
     ↪  << std::endl;
    std::cout << "Signature size = " <<
     ↪  signature.size() << std::endl;
    std::cout << "Msg + Signature size = " <<
     ↪  ct.size() + signature.size() << std::endl;
}
memcpy(&buffer, ct.data(), ct.size());
memcpy(&buffer[384], signature.data(),
 ↪  signature.size());
```

```
        // Send encrypted and signed msg
        s->send((const char *)&buffer, sizeof(buffer));

        sleep(conf.resend_period);
    }
}
```

4.3. Servidor

4.3.1. Recepción del mensaje

La tarea principal del servidor es recoger los mensajes, verificar su autenticidad y desencriptarlos. Una vez tenemos la estructura de datos original, se envían a los diferentes agentes accesorios, como la base de datos de almacenamiento y el sistema de alertas:

```
int main(int argc, char const *argv[])
{
    Config conf;
    conf = parse_config("/etc/simplemon-server/config/server.
     ↪  conf");

    std::unique_ptr<Socket_Server> s =
     ↪  std::make_unique<Socket_Server>(conf.port);

    std::unique_ptr<Botan::RandomNumberGenerator> rng(new
     ↪  Botan::AutoSeeded_RNG);

    Botan::DataSource_Stream
     ↪  in("/etc/simplemon-server/keys/server.priv");
    std::unique_ptr<Botan::Private_Key>
     ↪  priv(Botan::PKCS8::load_key(in, conf.key_password));
    std::unique_ptr<Botan::Public_Key> pub(Botan::X509::load_
     ↪  key("/etc/simplemon-server/keys/client.pub"));

    Botan::PK_Decryptor_EME dec(*priv, *rng.get(),
     ↪  "EME1(SHA-256)");
    Botan::PK_Verifier verifier(*pub, "EMSA1(SHA-256)");
```

```
    AlertManager alertMng(1);

    std::thread(&AlertManager::CheckingLoop,
     ↪  &alertMng).detach();

    while (1)
    {
        short buffer[1024] = {};
        StatusMessage msg;
        s->read((char *)buffer, 768);
        bool verified =
            verifier.verify_message((const unsigned char
             ↪  *)&buffer, 384, (const unsigned char
             ↪  *)&buffer + 384, 384);
        if (conf.logs != "none")
        {
            std::cout << "Msg is " << (verified ? "valid" :
             ↪  "invalid") << std::endl;
        }
        if (!verified)
        {
            continue;
        }

        memcpy(&msg, dec.decrypt((const unsigned char
         ↪  *)&buffer, 384).data(), sizeof(msg));

        alertMng.HostReport(std::string(msg.hostname.data()))
         ↪  ;

        if (conf.sql != "none")
        {
            ingestToSql(msgToSql(msg));
        }
    }
}
```

4.3.2. Socket servidor

Similar a la implementación del cliente, se realizó una clase análoga para el servidor:

```
Socket_Server::Socket_Server(int port)
{
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        // perror("socket failed");
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }

    // Forcefully attaching socket to the port
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR |
     ↪  SO_REUSEPORT, &opt, sizeof(opt)))
    {
        // perror("setsockopt");
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    // Forcefully attaching socket to the port
    if (bind(server_fd, (struct sockaddr *)&address,
     ↪  sizeof(address)) < 0)
    {
        // perror("bind failed");
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }
}

Socket_Server::~Socket_Server()
{
    close(server_fd);
```

```
}

void Socket_Server::read(char *buf, size_t length)
{
    if (listen(server_fd, 3) < 0)
    {
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }
    int addrlen = sizeof(address);
    if ((sock = accept(server_fd, (struct sockaddr *)&address,
     ↪  (socklen_t *)&addrlen)) < 0)
    {
        throw std::system_error(EAGAIN,
         ↪  std::generic_category(), strerror(errno));
    }

    ::read(sock, buf, length);
}
```

4.3.3. Inserción a MySQL

Para introducir los datos en MySQL hay que hacer uso de una librería de conexión específica. Los ejemplos de la documentación [13] ofrecen una muestra perfecta para nuestro caso de uso, donde debemos conectarnos a la base de datos, preparar una tabla determinada, y elaborar una inserción:

```
void ingestToSql(StatusVector msg)
{
    try
    {
        SQL_Config conf = parse_sql_config("/etc/simplemon-se
         ↪  rver/config/sql.conf");

        sql::Driver *driver;
        sql::Connection *con;
        sql::Statement *stmt;
        sql::PreparedStatement *pstmt;

        /* Create a connection */
```

```
        driver = get_driver_instance();
        con = driver->connect("tcp://127.0.0.1:3306",
         ↪  "simplemond", conf.password);
        /* Connect to the MySQL test database */
        con->setSchema("SimpleMon");

        stmt = con->createStatement();
        stmt->execute("CREATE TABLE IF NOT EXISTS
         ↪  monitor(hostname CHAR(32),uid INT, free_mem INT,
         ↪  free_disk INT, "
                      "used_cpu INT, date DATE, time
                       ↪  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                       ↪  ON UPDATE CURRENT_TIMESTAMP, "
                      "PRIMARY KEY (uid, time))");
        delete stmt;

        /* '?' is the supported placeholder syntax */
        pstmt = con->prepareStatement(
            "INSERT INTO monitor(hostname, uid, free_mem,
             ↪  free_disk, used_cpu) VALUES (?, ?, ?, ?,
             ↪  ?)");
        pstmt->setString(1, msg.at(0).first);
        for (int i = 1; i < msg.size(); i++)
        {
            pstmt->setInt(i + 1, msg.at(i).second);
        }
        pstmt->executeUpdate();
        delete pstmt;
        delete con;
    }
    catch (sql::SQLException &e)
    {
        std::cerr << "# ERR: SQLException in " << __FILE__;
        std::cerr << "(" << __FUNCTION__ << ") on line " <<
         ↪  __LINE__ << std::endl;
        std::cerr << "# ERR: " << e.what();
        std::cerr << " (MySQL error code: " <<
         ↪  e.getErrorCode();
```

```
            std::cerr << ", SQLState: " << e.getSQLState() << "
             ↪  )" << std::endl;
        }
    }
```

4.3.4. Sistema de alertas

Para gestionar las alertas hay que tener un hilo constantemente ejecutando comprobaciones para ver si alguno de los *hosts* ha pasado más tiempo del establecido en reportar, ya que de ser así habría que enviar una notificación. Para ello, utiliza una lista de reportes guardada por un *mutex*, ya que tanto el hilo principal también ha de modificarlo cuando se recibe un mensaje, y de no tener una estructura de control de concurrencia podría perderse información crítica:

```
void AlertManager::HostReport(const std::string &host)
{

    const std::scoped_lock<std::mutex> lock(reportsMutex);

    std::list<Report>::iterator it =
        std::find_if(reports.begin(), reports.end(),
         ↪  [host](const Report &rep) { return rep.first ==
         ↪  host; });

    if (it == reports.end())
    {
        reports.push_back({host,
         ↪  std::chrono::steady_clock::now()});
        return;
    }

    it->second = std::chrono::steady_clock::now();
}
void AlertManager::CheckingLoop()
{

    using namespace std;

    while (1)
```

```
    {
        std::this_thread::sleep_for(chrono::seconds(5));
        const std::scoped_lock<std::mutex>
         ↪  lock(reportsMutex);
        for (std::list<Report>::iterator it =
         ↪  reports.begin(); it != reports.end();)
        {

            auto delay = chrono::duration_cast<chrono::second
             ↪  s>(chrono::steady_clock::now() -
             ↪  it->second);
            if (delay >= allowedDownTime)
            {
                SystemNotify(it->first);
                it = reports.erase(it);
            }
            else
            {
                ++it;
            }
        }

        reports.remove_if([this](Report &rep) {
            return allowedDownTime < chrono::duration_cast<ch
             ↪  rono::seconds>(chrono::steady_clock::now() -
             ↪  rep.second);
        });
    }
}
```

4.3.5. Sistema de configuraciones

Para lograr los objetivos del proyecto es vital realizar un sistema de configuración. Para ello se realizó un *parser* que fuera capaz de leer archivos y reconocer opciones preestablecidas para cada componente del programa, y posteriormente usar las alternativas a desde el *struct* de configuración:

```
struct Config
{
```

```
    int port = 6540;
    std::string logs = "none";
    std::string sql = "true";
    std::string monitor = "";
    int downtime = 1;

    std::string ip_address = "127.0.0.1";
    int resend_period = 20;
    std::string key_password = "";
};

Config parse_config(std::string filename)
{
    Config conf;
    std::ifstream cFile(filename);
    if (cFile.is_open())
    {
        std::string line;
        while (getline(cFile, line))
        {
            line.erase(std::remove_if(line.begin(),
             ↪  line.end(), isspace), line.end());
            if (line[0] == '#' || line.empty())
                continue;
            auto delimiterPos = line.find("=");
            auto name = line.substr(0, delimiterPos);
            auto value = line.substr(delimiterPos + 1);

            // Parse options
            if (name == "ip_address")
            {
                conf.ip_address = value;
            }
            else if (name == "port")
            {
                conf.port = stoi(value);
            }
            else if (name == "resend_period")
```

```
            {
                conf.resend_period = stoi(value);
            }
            else if (name == "logs")
            {
                conf.logs = value;
            }
            else if (name == "sql")
            {
                conf.sql = value;
            }
            else if (name == "monitor")
            {
                conf.monitor = value;
            }
            else if (name == "key_password")
            {
                conf.key_password = value;
            }
        }
    }
    else
    {
        std::cerr << "Couldn't open " + filename + " for
        ↪  reading.\n";
    }
    return conf;
}
```

Además, ya que la estructura principal es básica, y el grueso del código de configuración lo suponen las opciones, valía la pena copiar la función dentro del archivo de SQL, para así tener las opciones separadas:

```
struct SQL_Config
{
    int port = 6540;
    std::string logs = "none";
    std::string ip_address = "127.0.0.1";
    std::string password = "";
};
```

```
SQL_Config parse_sql_config(std::string filename)
{
    SQL_Config conf;
    std::ifstream cFile(filename);
    if (cFile.is_open())
    {
        std::string line;
        while (getline(cFile, line))
        {
            line.erase(std::remove_if(line.begin(),
             ↪  line.end(), isspace), line.end());
            if (line[0] == '#' || line.empty())
                continue;
            auto delimiterPos = line.find("=");
            auto name = line.substr(0, delimiterPos);
            auto value = line.substr(delimiterPos + 1);

            // Parse options
            if (name == "ip_address")
            {
                conf.ip_address = value;
            }

            else if (name == "port")
            {
                conf.port = stoi(value);
            }
            else if (name == "logs")
            {
                conf.logs = value;
            }
            else if (name == "password")
            {
                conf.password = value;
            }
        }
    }
```

```
    else
    {
        std::cerr << "Couldn't open " + filename + " for
        ↪  reading.\n";
    }
    return conf;
}
```

4.4. Análisis en Hackron

El 31 de Marzo de 2021 se celebró en Santa Cruz de Tenerife el congreso de ciberseguridad Hackron (ver Figura 4.2), uno de los eventos de mayor repercusión de habla hispana. Debido a la situación sanitaria de este año, se retransmitió por *streaming* [14], y aprovechando el formato se preparó una competición CTF abierta a todo el mundo, con múltiples retos y desafíos.
Una de las pruebas consistía en intentar romper la seguridad de SimpleMon, para ello se procuró un acceso sencillo a la máquina, y gracias a los *logs* de la aplicación cliente se podían observar los mensajes cifrados (ver Figura 4.1).
Debido a problemas técnicos, hubo que reducir el evento de CTF [15], pero preparar el reto supuso una oportunidad para reexaminar no solo la seguridad de la aplicación, sino el esquema de despliegue, ya que era la primera vez que se instalaba en un entorno diferente al de desarrollo.

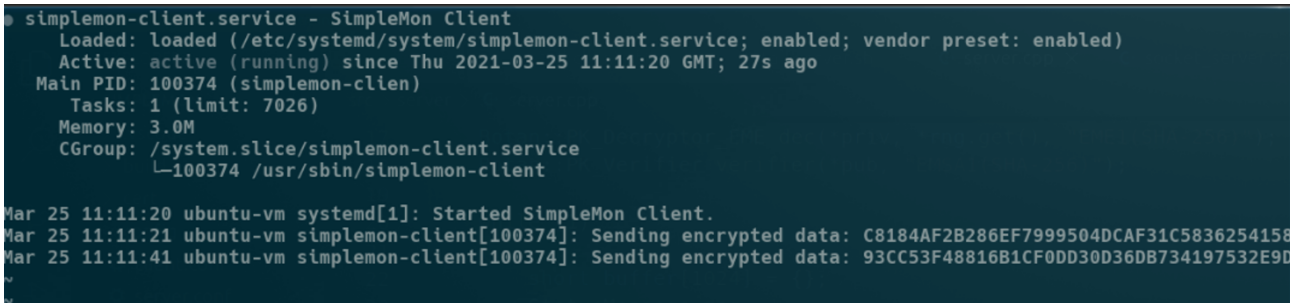


Figura 4.1: Logs de aplicación en CTF de Hackron

Figura 4.2: Logo Hackron

Capítulo 5

Conclusiones y líneas futuras

En este proyecto se ha desarrollado una nueva aplicación que permite monitorizar equipos informáticos de una manera sencilla y efectiva, cumpliendo así los requisitos propuestos como objetivo. Cada día que pasa, existen *front-ends* más potentes, y la seguridad toma un papel más relevante. De ahí, que utilizar tecnologías punteras en este trabajo haya sido una necesidad, y a la vez la clave del éxito del desarrollo.
Por ejemplo, para lograr una presentación atractiva de los datos recogidos hemos usado Grafana, un *software* de análisis y visualización de métricas muy potente y con muchas posibilidades. De hecho, sus características lo convierten en prácticamente un requisito en los entornos de monitorización actuales.
Como futuras mejoras, se pretende lograr que el programa sea más personalizable y accesible, mediante un mayor uso de ficheros de configuración para, por ejemplo, poder definir los parámetros a controlar en los diferentes sistemas.

Capítulo 6

Conclusions and future work

In this project, a new application has been developed that allows to monitor computer equipment in a simple and effective way, thus fulfilling the requirements proposed as the objective. Every day that passes, there are more powerful front-ends, and security takes a more relevant role. Hence, using top technologies in this work has been a necessity, and at the same time the key to the success of the development.
For example, to achieve an attractive presentation of the collected data we have used Grafana, a very powerful analysis and metrics visualization software with many possibilities. In fact, its characteristics make it practically a requirement in current monitoring environments.
As future improvements, it is intended to make the program more customizable and accessible, through greater use of configuration files to, for example, be able to define the parameters to be controlled in the different systems.

Capítulo 7

Presupuesto

Al tratarse de un proyecto de desarrollo de *software*, en el presupuesto se reflejarán los gastos de personal asociados a la elaboración del programa. Adicionalmente, se incluye un plan de gastos para el despliegue del sistema en varias plataformas de en la nube como SaaS.

7.1. Desarrollo

Concepto	Coste/Hora	Horas	Total
Investigación inicial	12€	20	240€
Aplicación cliente	12€	20	240€
Aplicación servidor	12€	20	240€
Implementación de base de datos	12€	20	240€
Diseño de Grafana y acceso a MySQL	12€	40	480€
Comunicación segura	12€	40	480€
Soporte para múltiples clientes	12€	40	480€
Sistema de alertas	12€	40	480€
Total	**12€**	**240**	**2880€**

Tabla 7.1: Presupuesto de desarrollo

7.2. Despliegue SaaS

Gracias al diseño de la aplicación, una máquina con 1 núcleo de CPU y 1 Gb de RAM es capaz de proporcionarnos todo el soporte necesario para mantener el servidor para menos de 30 máquinas. Una de las mayores ventajas de este sistema reside en la disponibilidad asegurada del *host*, lo que lo elimina como punto de fallo en la tarea de monitorización.

Concepto	Coste
Configuración inicial	200€
Amazon EC2 T2	5.22€/Mes
Google Cloud N1 Standard	7.30€/Mes
Kamatera Cloud Server	4€/Mes

Tabla 7.2: Presupuesto de despliegue SaaS

Capítulo 8

Apéndice: Código e instalación

El código desarrollado y una guía de despliegue pueden ser encontrados en el siguiente repositorio de GitHub: `https://github.com/da3m0nsec/SimpleMon`

Bibliografía

[1] “State of it monitoring: A report roundup.” [Online]. Available: https://www.bmc.com/blogs/state-of-monitoring/

[2] *Effective Monitoring and Alerting*. Reading, Massachusetts: O’Reilly Media, Inc., 2012.

[3] “Become a nagios certified professional.” [Online]. Available: https://www.nagios.com/services/certification/

[4] “Pandora fms enterprise.” [Online]. Available: https://pandorafms.com/library/category/enterprise/

[5] “What is open source, and why is it important?” [Online]. Available: https://www.bigcommerce.com/ecommerce-answers/what-open-source-and-why-it-important/

[6] “Is c++ still relevant in 2021?” [Online]. Available: https://exyte.com/blog/is-c++-still-relevant

[7] “Botan: Crypto and tls for modern c++.” [Online]. Available: https://botan.randombit.net/

[8] “Botan users.” [Online]. Available: https://github.com/randombit/botan/wiki/Users

[9] “Travis: Core concepts for beginners.” [Online]. Available: https://docs.travis-ci.com/user/for-beginners/

[10] “Everything you need to know about grafana.” [Online]. Available: https://www.skedler.com/what-is-grafana-why-use-it-everything-you-should-know/

[11] “Push vs. pull monitoring configs.” [Online]. Available: https://steve-mushero.medium.com/push-vs-pull-configs-for-monitoring-c541eaf9e927

[12] “How much data can you encrypt with rsa keys.” [Online]. Available: https://info.townsendsecurity.com/bid/29195/how-much-data-can-you-encrypt-with-rsa-keys

[13] *MySQL Connector/C++: Usage Examples*. [Online]. Available: https://dev.mysql.com/doc/connector-cpp/1.1/en/connector-cpp-getting-started-examples.html

[14] “La casa del carnaval de santa cruz acoge la octava edición del congreso de ciberseguridad.” [Online]. Available: https://www.eldia.es/santa-cruz-de-tenerife/2021/03/29/casa-carnaval-santa-cruz-acoge-45980790.html

[15] “Hackr0n 2021: Doble entrevista y el ctf entre «bambalinas».” [Online]. Available: https://derechodelared.com/hackr0n-2021/