UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Validación de la integridad del Software en la Integración Continua Iterativa mediante el uso del Módulo de Plataforma de Confianza (TPM)

Integrity validation of Software in Iterative Continuous Integration through the use of Trusted Platform Module (TPM)

Realizado por

Jordy Ryan Casas Correia

Tutorizado por

Antonio Muñoz Gallego

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2019

Fecha defensa: **___ de julio de 2019**

Fdo. El/la Secretario/a del Tribunal

Resumen

El desarrollo del software ha pasado de ser rígido y poco flexible, a ser ágil en constante cambio con la aparición de entornos de Integración y Despliegue continuo, con herramientas como *Jenkins*.

Sin embargo, los desarrolladores suelen confiar en dichos entornos debido a la gran cantidad de comodidades que ofrecen, centrando su atención respecto a la seguridad únicamente en la autenticación, sin tener en cuenta otras características de seguridad como la integridad de los datos, que para nuestro caso concreto se corresponde con el código fuente.

Es importante que el código fuente de un proyecto software no sea modificado de forma maliciosa, y en la actualidad, no existe ningún método que permita comprobar que la integridad del proyecto no ha sido violada. Para la implementación de este método se ha decidido utilizar un elemento seguro, en concreto el conocido como Módulo de Plataforma de Confianza (*Trusted Platform Module*, o TPM en Inglés). Este elemento consiste en un chip hardware que proporciona funcionalidades criptográficas, seguras e inmutables, que cada vez se encuentran con más frecuencia disponibles en la gran mayoría de computadores domésticos desde 2003.

Palabras clave: **Seguridad Informática, Integración Continua, TPM, Computación Confiable, Desarrollo del Software**

Abstract

Software development has gone from being rigid and not very flexible, to being agile with constant changes, thanks to the creation of continuous integration and delivery environments such as Jenkins.

However, developers often rely on such environments due to the large number of amenities they offer, focusing their attention at authentication only, without taking into account other aspects important in security as the integrity of the data, the source code in this case.

It is very important that the source code of a software project cannot be modified in a malicious way, and nowadays, there is no safe method to verify that the integrity of the project has not been violated. This secure method can be implemented through the use of the Trusted Platform Module, or TPM, a hardware chip that provides cryptographically secure and immutable functionalities, available in the vast majority of home computers since 2003.

Keywords: **IT security, Continuous Integration, TPM, Trusted Computing, Software Development**

Índice

1

Introducción

Motivaciones

La motivación principal para la realización de este TFG ha sido la identificación de un problema para mantener la integridad del código en el proceso CI-CD. Para ello se propone el uso de un elemento hardware con capacidades criptográficas que tenemos disponible en un gran número de máquinas (incluso las domésticas) y que en su mayoría tienen un potencial de uso totalmente desaprovechado.

Al conocer la existencia de dicho componente, sumado a las metodologías ágiles de desarrollo software que se han ido aprendiendo a lo largo de la carrera, se podía observar que **el modelo actual de integración continua tiene carencias respecto a** cómo **comprobar integridad del software**, que se acentúa más **en despliegues bajo premisa**, en máquinas controladas por la misma entidad que desarrolla (o gestiona el desarrollo) el software a desplegar.

Objetivos

Los objetivos de este trabajo de fin de grado son:

- **Conseguir una nueva aplicación basada en las funcionalidades del TPM**, que sea útil para el modelo actual de la integración continua, minimizando costes, más allá de confiar en las máquinas sobre las que se ejecuta el ciclo de CI-CD se lleva a cabo.
- **Obtener un componente software utilizando el TPM**, que sea capaz de detectar cuando la nueva versión a desplegar de cierto programa no corresponde con la versión desarrollada intencionalmente por los desarrolladores, evitando posibles agujeros de seguridad.

- **Demostrar los resultados mediante el desarrollo de un caso de uso** en el que el componente software ayudaría a resolver un problema de seguridad real.

Metodología

Para llevar a cabo este trabajo de fin de grado, se ha trabajado con una metodología ágil inspirada en las aprendidas durante la carrera, especialmente marcada por la metodología *Scrum*.

El proceso se basa en continuas iteraciones de tiempo variable, con un máximo de tres semanas, desarrollando cinco fases por ciclo iterativo:

1. **Análisis de requisitos**: para conocer hacia dónde se quiere avanzar en cada ciclo.
2. **Diseño**: para modelar qué se quiere implementar en cada ciclo.
3. **Implementación**: para desarrollar la funcionalidad objetivo del ciclo.
4. **Pruebas/Prototipo**: para probar la funcionalidad implementada y mostrarla al cliente (tutor).
5. **Valoración de resultados**: para documentar el resultado del ciclo, por parte del autor del proyecto y del cliente (tutor).

La comunicación entre cliente (tutor) y el autor del proyecto (estudiante) se ha realizado tanto en reuniones personales como mediante herramientas digitales. Además, por la tutorización, se aceptarán consultas entre autor y cliente en cualquier fase de los ciclos iterativos.

Estructura del Trabajo de Fin de Grado

Este trabajo de fin de grado está estructurado de la siguiente manera:

1. ***Metodologías ágiles y la importancia de la seguridad en la automatización***: breve recopilatorio del estado del arte sobre la integración continua, el desarrollo ágil.
2. ***Riesgos de seguridad en los servidores de ensamblaje y pruebas***: Posibles riesgos de seguridad presentes, prueba de concepto de un escenario en el que existen algunas de las amenazas de seguridad analizadas en el capítulo. Proposición del problema a resolver.
3. ***TPM***: Introducción al componente hardware que hará posible la solución al problema propuesto y definición de los conceptos asociados. Breves ejemplos de uso del componente hardware para facilitar la comprensión de posteriores capítulos.
4. ***Utilizando el TPM para asegurar la integridad***: análisis sobre la utilidad del TPM para resolver el problema propuesto. Resolución del problema con el componente software desarrollado, describiendo los detalles más importantes de la implementación.
5. ***Resultados y conclusiones***: uso del componente software desarrollado sobre un escenario demostrativo, conclusiones de la solución y posibles mejoras.

Convenciones

Este trabajo de fin de grado se atiene a las siguientes convenciones:

- El código fuente será mostrado en una **tipografía monoespaciada** para su fácil legibilidad y distinción de sus caracteres.
- Las referencias en el documento seguirán la normativa APA.

2

Metodologías ágiles y la importancia de la seguridad en la automatización

El desarrollo del software ha evolucionado radicalmente estos últimos años, pasando de metodologías clásicas y rígidas como el desarrollo en cascada, hacia otras más modernas y **ágiles**, con **menor acoplamiento entre** las funciones que desenvuelve **cada miembro del equipo**, y orientándolo cada vez más hacia la inminente automatización que demanda la cuarta revolución industrial, como *Scrum*, *Lean* o *Kanban*. Cada vez es más necesario un desarrollo de las características de las metodologías ágiles, con menores tiempos entre despliegue y despliegue de las aplicaciones, para mantener la competencia y relevancia dentro del mercado. Para ello se recurre a entornos CI-CD que facilitan la labor del desarrollo del software.

Definición de entorno CI-CD

Se considera un entorno CI-CD (*Continuous Integration - Continuous Delivery* en Inglés) a aquel entorno de trabajo que orqueste y se componga de:

- Servidor de Control de Código Fuente
- Servidor de Ensamblaje y Pruebas
- Servidor de Despliegue

Usualmente todos los componentes se encuentran aislados, en máquinas diferentes, y, pese a que todos los puntos son importantes, el que más destaca cuando se habla de entorno CI-CD es el servidor de ensamblaje y pruebas, pues fue el componente que cambió la forma en la que se entrega el software a día de hoy (ya existían repositorios para el control de código fuente, y el producto ya se desplegaba anteriormente, usualmente de forma manual). Por ello, cuando se referencia a entorno CI-CD, se entiende que se habla del servidor de ensamblaje y pruebas, y para referirnos al proceso completo, se utiliza el concepto *pipeline CI-CD*.

El riesgo de la inversión del modelo de amenazas

Sin embargo, de forma cada vez más habitual **se descuida la seguridad y privacidad a favor de la usabilidad** o de las mejoras inmediatas que se puedan obtener, sin pensar demasiado en las repercusiones futuras. Suceden a diario casos de filtraciones de bases de datos de compañías grandes, porque no se aseguraron correctamente los privilegios de acceso, o accesos no autorizados a recursos privados, porque algún miembro del equipo subió accidentalmente una clave SSH o un token de servicio a un repositorio público. Quizá lo más agravante para la seguridad de los entornos ágiles de hoy día es el cómo se ha invertido el modelo de amenazas.

Anteriormente a la existencia de esta solución, a la hora de desplegar un proyecto, **el modelo era desconfiar de la máquina en la que se ejecutaría el software**, no dando por hecho que fuese segura, pero confiando en que el software que se desplegaba haría lo correcto, es decir, **implementando las medidas de seguridad correspondientes para que operase de forma correcta** ante un número de posibles casos adversos, que estaban previstos en las máquinas de entonces. Existía un modelo de amenazas definido.

A día de hoy, se confía ciegamente en el proveedor de despliegue en la nube, porque se considera que el aislamiento que proporcionan las máquinas virtuales o contenedores es confiable, y que dichos proveedores habrán previsto e implementado medidas de seguridad para que todo funcione correctamente. E incluso cuando se trata de despliegues bajo premisa, se confía en que los contenedores o la virtualización impedirán que sucedan imprevistos desagradables.

De lo que sí se desconfía es del software que es desplegado, pues en la gran mayoría de casos no se conoce con certeza si es lo suficientemente seguro como para evitar cualquier tipo de problema de seguridad. Se tiende a aislar el despliegue lo máximo posible de la máquina anfitriona, restringiendo sus privilegios y el acceso al hardware lo máximo posible. Sin embargo, tan solo este aislamiento es cuestionable.

Al fin y al cabo, hay otro software que controla estas máquinas virtuales y se encarga de hacer de intermediario entre todas las máquinas virtuales que alberga una máquina y el hardware sobre el que se ejecutan: el hipervisor.

El hipervisor no es más que un sistema operativo básico dedicado a gestionar máquinas virtuales. Si alguien consigue acceder al hipervisor de forma directa, será capaz de gestionar y alterar las máquinas virtuales sin restricciones, y sin que sea posible detectar la intrusión desde ninguna de las máquinas virtuales. A esta técnica se le conoce como *hyperjacking*, que consiste en introducir un hipervisor malicioso que "gestione" el hipervisor original.

No obstante, no es necesario recurrir a ello para que un ciclo de despliegue continuo sea modificado de forma maliciosa.

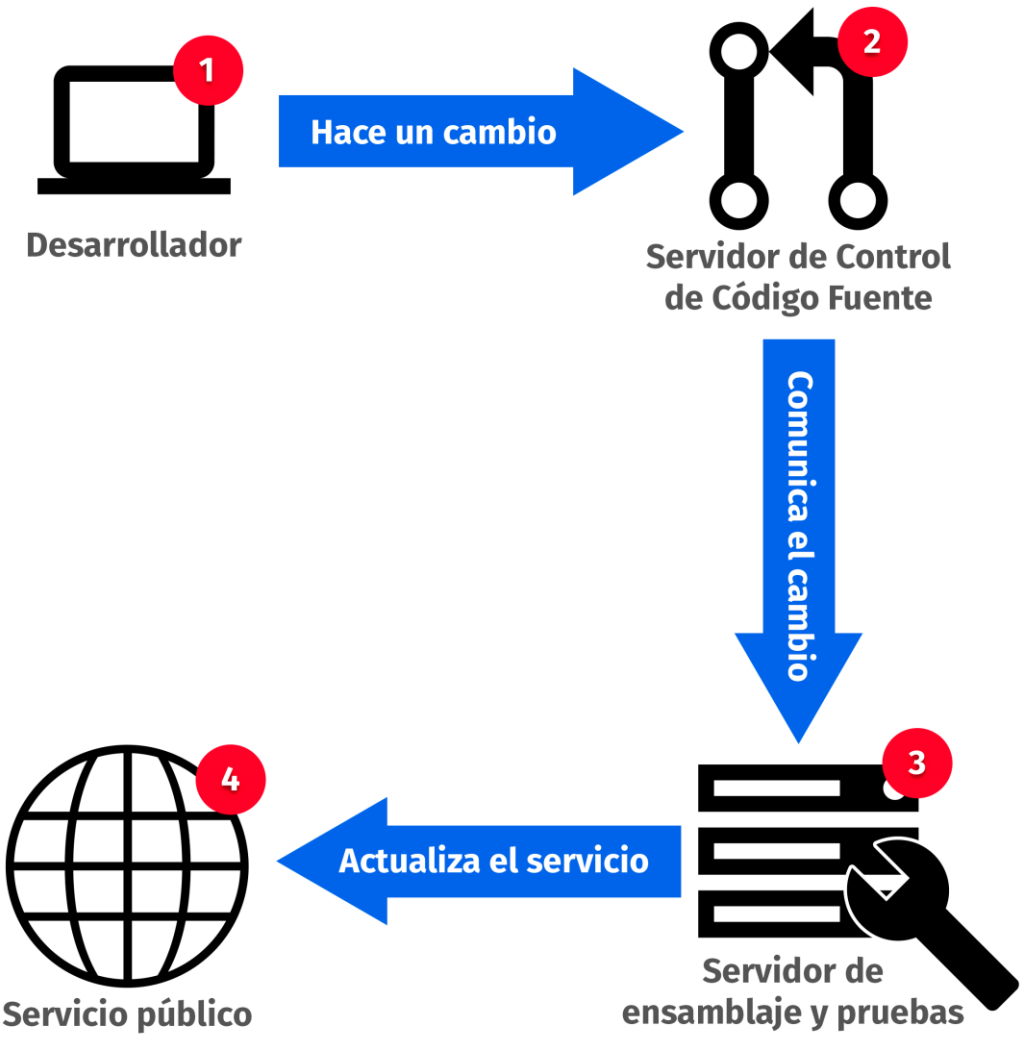

Figura 1: Flujo de integración continua utilizado en la actualidad.

Se puede resumir en cuatro pasos:

1. El desarrollador desarrolla una característica y la sube al servidor de control de código fuente (habitualmente basado en Git).
2. Los cambios realizados en el servidor de control de código fuente son transmitidos a un servidor de ensamblaje y pruebas.
3. En este servidor, se ensambla una nueva versión del programa y se le realizan las pruebas unitarias y de cohesión que se hayan preparado para él.
4. Una vez comprobado que el programa supera con éxito todas las pruebas, se publica la nueva versión (despliegue).

Se han identificado los **posibles puntos débiles que puede presentar el ciclo**, asumiendo que las comunicaciones entre todos los pasos descritos ya están aseguradas.

Se consideran puntos débiles los siguientes casos:

- **La máquina del desarrollador podría estar infectada**, de forma que se puede introducir código malicioso directamente en el fuente. Aunque bien es cierto que esta situación **es poco probable**, pues pese a que es factible, **sus contribuciones serían comprobadas por las revisiones de código** y el problema sería detectado rápidamente.
- El servidor de control del código fuente podría ser infectado, pero sería aún más complicado realizar cambios no controlados, debido a la naturaleza incremental del control de versiones, que se realizan comparando qué cambia de un fichero a otro, no reemplazando ficheros, y sería fácil comprobar dónde se ha producido un cambio indeseado.
- Por tanto, se considera que el servidor de ensamblaje y pruebas es el que presenta mayor vulnerabilidad. Es una máquina en la que suele confiar, porque tan sólo recoge el código fuente del paso anterior, y utilizando herramientas ya contrastadas compila la nueva versión del programa y ejecuta pruebas sobre la misma.

Un agente malicioso con acceso al servidor de ensamblaje y pruebas podría, por ejemplo, introducir una rutina encargada de detectar cada vez que se obtenga el código fuente y se modifiquen los archivos, y que entonces sustituyese algún archivo clave del código fuente para introducir una puerta trasera.

Este caso no sería tan sencillo de comprobar como los casos descritos anteriormente, pues **el código fuente ya se da por revisado, y por tanto, válido**. Se debe incorporar a nuestro flujo de trabajo alguna comprobación de integridad, para así tener la certeza de que lo que se está desarrollando es lo mismo que llega a los usuarios finales, sin modificaciones maliciosas que puedan afectar al proyecto.

Estudiar los riesgos asociados al servidor de ensamblaje y pruebas será de inconmensurable ayuda para tratar de prevenir brechas de seguridad importantes que afecten a todo el proyecto software.

3

Riesgos de seguridad en los servidores de ensamblaje y pruebas

Existen riesgos de seguridad una vez el servidor de ensamblaje se encuentra vulnerado. En este capítulo se expondrán las amenazas de seguridad con más riesgo a ser explotadas, y el estado en el que se encuentra la seguridad de los mismo.

Recomendaciones de seguridad para un entorno CI-CD a día de hoy

Según un artículo en *DigitalOcean*[1], una empresa dedicada al despliegue de servidores virtuales bajo premisa, la mejor forma de asegurar un entorno de CI-CD es aislarlo lo máximo posible del acceso externo. Como se detalla en el artículo "ya que el sistema de CI-CD tiene acceso completo a tu repositorio de código y las credenciales suficientes para desplegar en varios entornos, **es esencial asegurarlo para salvaguardar información interna y garantizar la integridad de tu producto**.". También se aconseja que el acceso a dichos recursos sea a través de VPN para minimizar los accesos no autorizados.

[1] Ellingwood, J. (3 de Abril de 2018). *An Introduction to CI/CD Best Practices.* Obtenido de DigitalOcean: https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best-practices

Sin embargo, proteger un servidor de CI-CD no es tan sencillo como proteger un servidor habitualmente accesible públicamente como un servidor web, pues se puede acceder a él de varias formas:

- **SSH**: la forma habitual de acceso directo a un servidor para los administradores de sistemas. Suele estar bien protegida, debido al uso ya extendido de las claves SSH.
- **APIs**: Por ejemplo, cuando se conectan servicios como *GitHub* o *GitLab* a nuestro entorno de CI-CD mediante su clave de API privada. Aquí se encuentra otro punto a controlar, se hace indispensable que la cuenta que posea dichas claves de API estén protegidas con una contraseña adecuada y se lleve a cabo un proceso de autenticación en dos pasos (*2 factor authentication*). Es complicado gestionar este último paso, pues *Grzegorz Milka*, ingeniero del software en *Google*, reveló que menos del 10% de los usuarios de Google utilizan la autenticación en dos pasos en la conferencia *USENIX Enigma 2018*[2]. No asegurar estas claves correctamente puede desembocar en filtraciones del código fuente de un proyecto, o de modificaciones indeseadas a través de la falsificación de identidad.
- **A través de una interfaz:** Algunas soluciones de CI-CD proporcionan una interfaz para poder gestionar el servidor de ensamblaje y pruebas, como Jenkins o GitLab, que proveen una interfaz web. En particular, Jenkins permite además el acceso con credenciales que gestiona él mismo, de tal modo que hay otro punto más a controlar: la seguridad del acceso a la interfaz.

Otro hecho destacable es el cómo grandes equipos de software (y principalmente los equipos de software abierto) hacen caso omiso a estas recomendaciones sobre el aislamiento de los servidores CI-CD. Según la página "*Who is Using Jenkins?*" de la Wiki

[2] Milka, G. (17 de Enero de 2018). *Anatomy of Account Takeover*. Obtenido de USENIX: https://www.usenix.org/conference/enigma2018/presentation/milka

de Jenkins[3], proyectos tan extendidos como KDE (Entorno de escritorio para Linux), Apache (servidor web), AngularJS (framework para desarrollo web) o Ubuntu son accesibles públicamente.

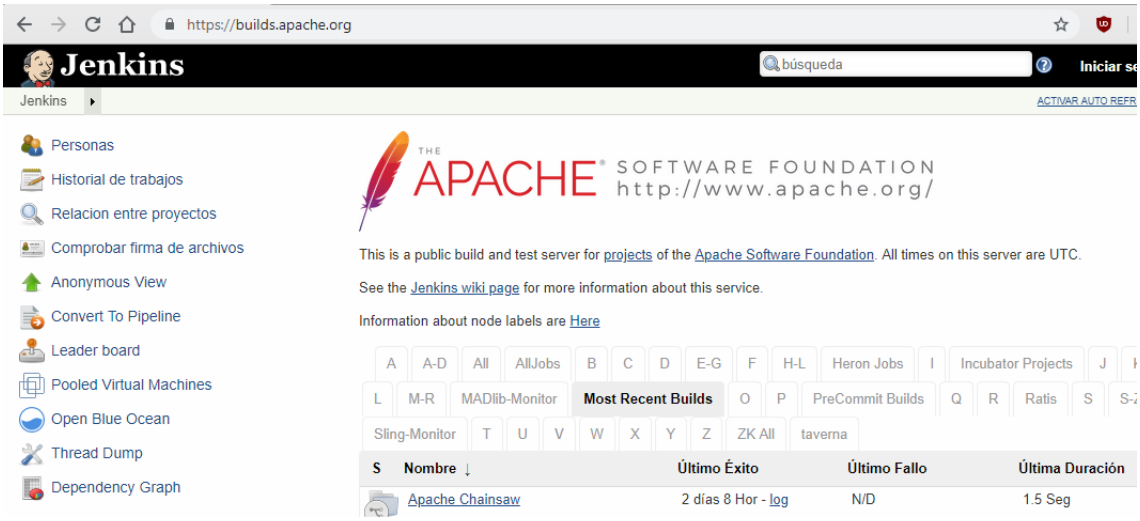

Figura 2: Uso de Jenkins de *Apache Software Foundation*, accesible públicamente.

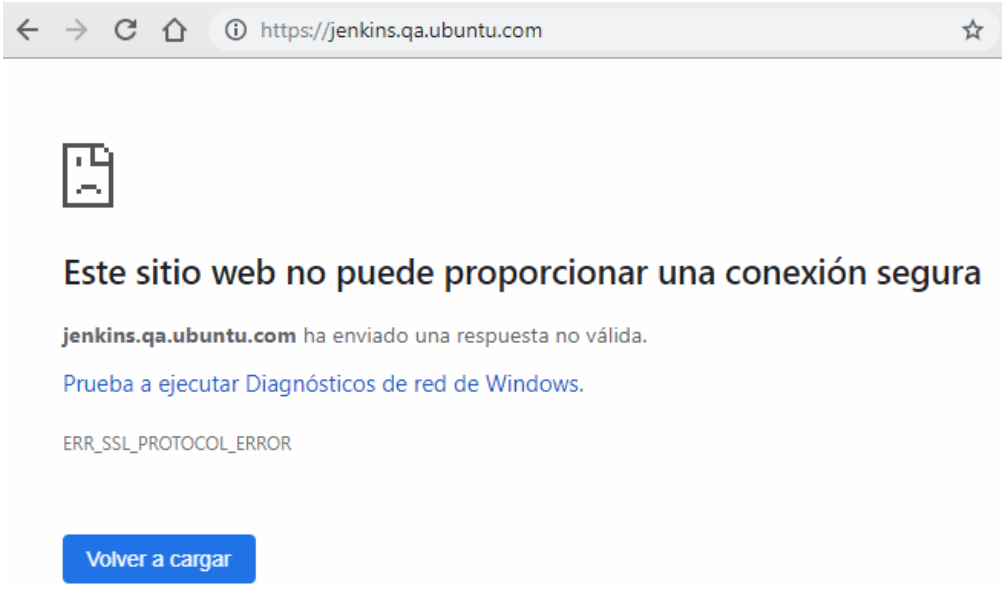

Figura 3: Servidor público de Jenkins de Ubuntu, que no tiene el certificado SSL al día, a Junio de 2019

Pese a que todo el software mencionado es de código abierto, sigue siendo un riesgo de seguridad. A día de hoy, muchos proyectos se basan en herramientas de código abierto para poder agilizar el desarrollo, ya que su confiabilidad reside en la libertad

[3] Prokop, M. (1 de Octubre de 2017). *Who is using Jenkins?*. Obtenido de Jenkins Wiki: https://wiki.jenkins.io/pages/viewpage.action?pageId=58001258

de poder observar, modificar y discutir públicamente cualquier parte de su código fuente. Es por ello que, por ejemplo, en 2003, se intentó un ataque directo al *kernel* de Linux[4], mediante la inserción de una puerta trasera en uno de los repositorios CVS que servía como copia espejo del repositorio principal. Según detalla el artículo, "alguien irrumpió (digitalmente) en el servidor e introdujo el cambio.". Se detalla que era "un cambio que nunca había sido aprobado, ni aparecía en el repositorio principal". El cambio detectado fue el siguiente, en la función *wait4*:

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

Aunque este cambio puede parecer inofensivo a simple vista, asigna al usuario que la ejecuta el identificador 0, dándole privilegios absolutos sobre la máquina. Por suerte, este cambio fue detectado a tiempo y nunca llegó al repositorio principal.

Por otro lado, el artículo de *DigitalOcean* describe: "Los fallos en una pipeline CI-CD **son inmediatamente visibles** y detiene el avance de los mismos a las etapas siguientes del ciclo. Es un mecanismo de control de acceso que **salvaguarda los entornos más importantes del código fuente no confiable**.". El hecho de que el entorno de ensamblaje y pruebas ejecute las pruebas que se hayan preparado para un proyecto software en concreto crea una falsa sensación de seguridad si la integridad del código fuente no está garantizada. Utilizar un entorno CI-CD no impide que sucedan brechas de seguridad importantes, como será expuesto a lo largo de este capítulo.

Qué puede hacer un atacante con acceso administrativo al servidor

La máquina que ejecuta el servidor de ensamblaje y pruebas suele ser un objetivo muy preciado para los atacantes externos por varios motivos.

[4] Felten, E. (9 de Octubre de 2013). *The Linux Backdoor Attempt of 2003*. Obtenido de Freedom to Tinker: https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/

Uso del hardware para un beneficio particular

Usualmente, el hardware empleado en estos servidores suele ser potente, capaz de compilar y ejecutar pruebas rápidamente para que el desarrollo y despliegue no se vea ralentizado. En la gran mayoría de los casos, las soluciones de CI-CD ofrecen la posibilidad de ser ejecutados en varios nodos a la vez, es decir, varios servidores dedicados explícitamente a esta tarea.

Para un atacante estos servidores son un objetivo muy goloso, pues puede utilizar dichos servidores para beneficio propio, como por ejemplo el minado de criptomonedas. No es la primera vez que sucede, como relata *Guardicore Labs*[5] en su artículo sobre una campaña de *cryptojacking* (introducir un programa para minar criptomonedas en una máquina ajena) que se está llevando a cabo desde China, con más de 50000 servidores infectados.

Otras posibilidades incluyen el uso de los servidores como el envío masivo de correos maliciosos[6], ejecución de ataques de denegación de servicio (DDoS) coordinados mediante *botnets* como sucedió en 2017 con *Mirai*[7], o la petición de un rescate económico de los datos albergados mediante *ransomware*, poco probable debido a que los datos que se usan en un servidor de este tipo deberían estar en una máquina separada que gestione únicamente el código fuente.

Hacking ético o vandalismo

Cabe la posibilidad de que el atacante tan solo quiera gastar una broma, vandalizar el software, o enviar un aviso sobre la falta de seguridad del servidor, siendo así considerado hacking ético, o de sombrero blanco. Este sería el mejor de los casos.

---

[5] Harpaz, O. & Goldberg, D. (29 de Mayo de 2019). *The Nansh0u Campaign – Hackers Arsenal Grows Stronger.* Obtenido de Guardicore Labs: https://www.guardicore.com/2019/05/nansh0u-campaign-hackers-arsenal-grows-stronger/

[6] Huckaby, J. (5 de Marzo de 2014). *Warning: Hackers are using SSH Tunnels to send SPAM.* Obtenido de RackAid: https://www.rackaid.com/blog/spam-ssh-tunnel/

[7] Servicio AntiBotNet. (20 de Junio de 2019). *Botnet: Mirai.* Obtenido de la Oficina de Seguridad del Internauta: https://www.osi.es/es/servicio-antibotnet/info/mirai

Perjuicio al equipo de desarrollo y su propiedad intelectual

Al acceder al servidor de ensamblaje y pruebas del software, el atacante es capaz de acceder a los repositorios de código fuente conectados al mismo servidor, siendo capaz de copiar y duplicar dicho código a su antojo. La gravedad de esta amenaza se incrementa cuando el código fuente todavía no ha sido materializado como producto final al público al que está destinado, especialmente en proyectos con licencia propietaria, donde lo más innovador del proyecto puede ser el uso de algoritmos nuevos creados específicamente para ese proyecto, como en el campo de la Inteligencia Artificial.

Perjuicio al usuario final

En el caso de que al atacante le interesen los datos de los clientes del software, acceder al servidor de ensamblaje y pruebas puede serle de utilidad, ya que podría introducir en el código fuente algún mecanismo de phishing, que le permita obtener los datos que necesite de los clientes al ser compilado el proyecto software.

Dicho acceso a datos podría ser a través del robo de credenciales, que además incurriría en otro riesgo de seguridad más amplio para el usuario final: la reutilización de contraseñas. En un estudio sobre más de 61 millones de contraseñas, realizado por parte del departamento de Ciencias de la Computación de la universidad *Virginia Tech*, el 52% de los usuarios reutilizan sus contraseñas[8], o modificaciones parciales de las mismas.

[8] Wang, C. & Wang, G. & Jan, S. & Hu, H. & Bossart, D. (Marzo de 2018). *The Next Domino To Fall: Empirical Analysis of User Passwords across Online Services.* Obtenido de Virginia Polytechnic Institute and State University: https://people.cs.vt.edu/gangwang/pass

Prueba del concepto en un sistema con Jenkins vulnerado

Es más sencillo ver el problema de seguridad con un ejemplo concreto. Para ello, se ha desarrollado una aplicación simple con un formulario de inicio de sesión.

Aplicación con inicio de sesión

Esta aplicación proporcionaría servicio a las personas que tengan perros, mediante un pequeño aparato recargable que se adhiere al collar, para permitir que los dueños conozcan en caso de emergencia la ubicación de su mascota. Para ello, los usuarios iniciarían sesión en el portal que les permitiese ver estos datos.

La aplicación que demuestra la debilidad identificada en el proceso está desarrollada en Nuxt.js para la parte del frontend, un marco de trabajo que se basa en Vue.js, otro marco de trabajo para crear interfaces de usuario y aplicaciones de una sola página. Para el backend, se ha preferido usar un simple script en PHP.

En cuanto al despliegue se hará a través del servidor web interno de PHP, y para la interfaz Nuxt.js se compila a través de *WebPack*, una utilidad que permite empaquetar proyectos con dependencias en JS y produce archivos HTML, JS, CSS y el resto de recursos que se utilizan en una web, creando un paquete que puede ser desplegado de forma estática. Se denomina despliegue de página estática a aquel despliegue en el que la página puede funcionar sin más procesamiento por parte del servidor que entregarle los ficheros al navegador cliente.

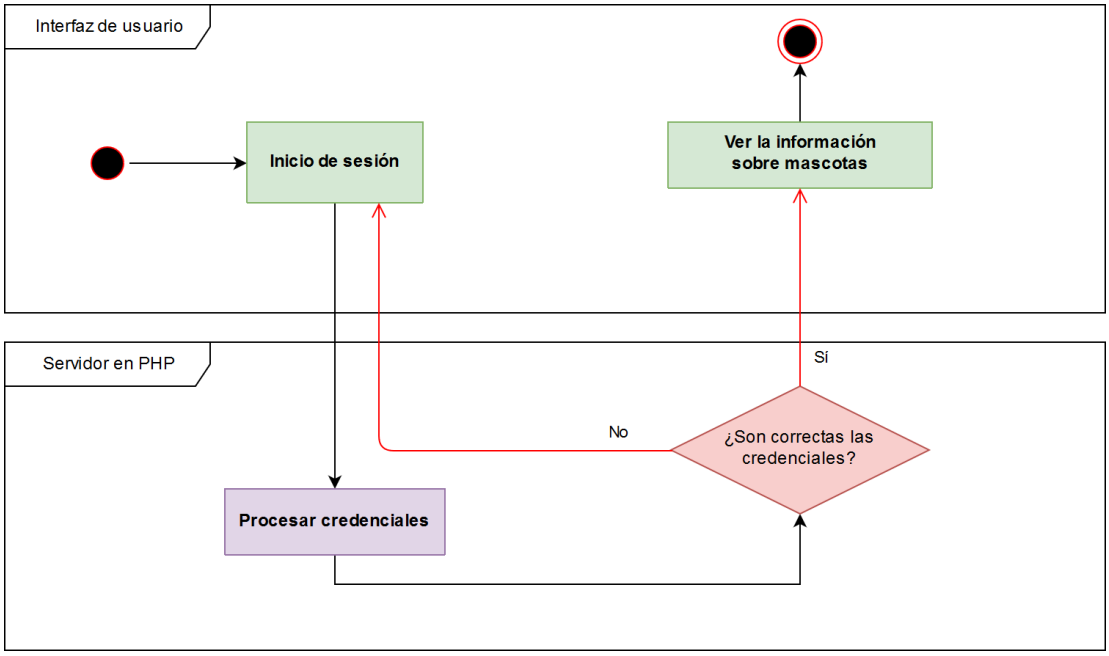

Figura 4: Diagrama de flujo en UML que representa la funcionalidad básica de la aplicación demostrativa

Figura 5: Captura del formulario de inicio de sesión la hipotética aplicación, *DogLand*

GitLab como servidor de control de código fuente

Se ha elegido GitLab como servidor de control del código fuente del proyecto, ya que ofrece de forma gratuita infinidad de repositorios privados. También podría desplegarse en una máquina propia bajo premisa, pues cuenta con una edición comunitaria de código abierto. A él subiremos nuestro proyecto como privado.

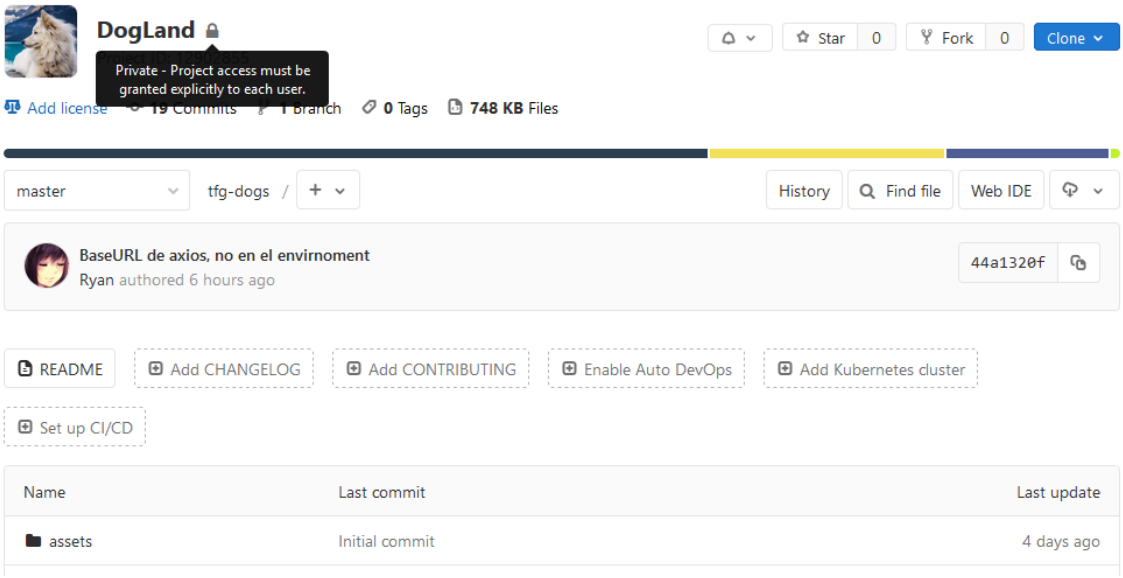

Figura 6: Proyecto subido a GitLab, marcado como proyecto privado.

GitLab permite el ingreso a su plataforma mediante redes sociales y mediante la combinación usuario/contraseña clásica. Además **ofrece de forma gratuita la autenticación en dos pasos, altamente recomendable**.

Jenkins como servidor de ensamblaje y pruebas

Jenkins es un proyecto de código abierto que permite automatizar el ensamblaje y las pruebas dentro de un proyecto software. Además, mediante complementos, es posible conectarlo con GitLab directamente, para que genere automáticamente una versión nueva a probar cada vez que se sube un cambio al repositorio. Es una solución de CI-CD completa y madura.

Figura 7: Proyecto *DogLand*, configurado para ser ensamblado y probado en Jenkins.

Configurar dicha plataforma queda fuera del alcance de este trabajo de fin de grado, sin embargo, es resaltable saber el cómo se ensambla el proyecto, para conocer qué proceso de ensamblaje sigue:

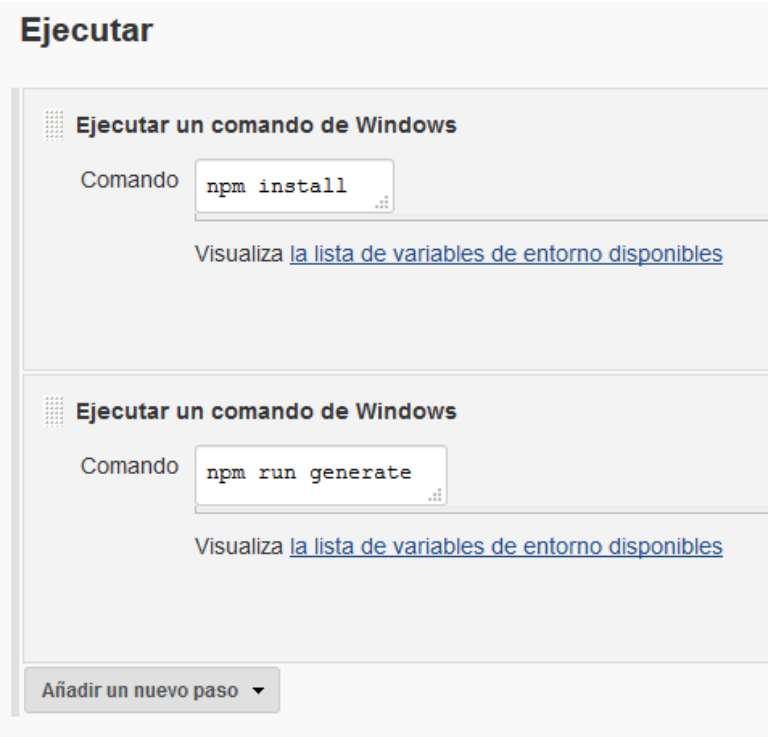

Figura 8: Comandos utilizados para ensamblar el proyecto.

Los comandos utilizados para ensamblar el proyecto son dos:

- **`npm install`**: descarga e instala todas las dependencias del proyecto en la carpeta de trabajo de Jenkins. Si ya están instaladas, el comando verifica que todas sean correctas.
- **`npm run generate`**: ensambla y genera el paquete desplegable de la aplicación web, que por defecto va a parar a una carpeta *dist* dentro de la carpeta de trabajo de Jenkins.

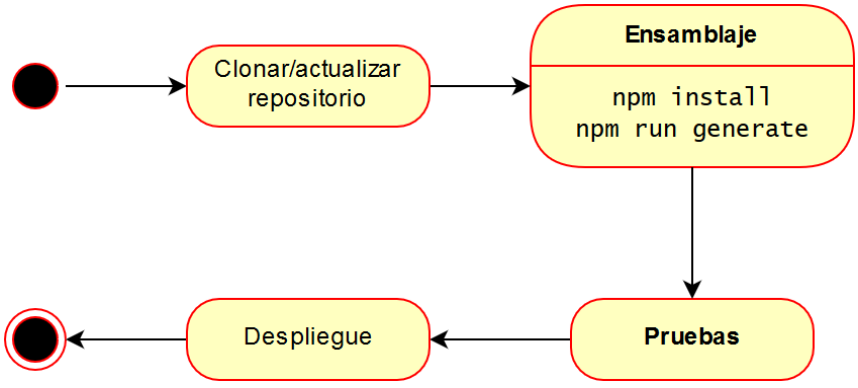

Figura 9: Diagrama de flujo UML simplificado del ciclo CI-CD de Jenkins.

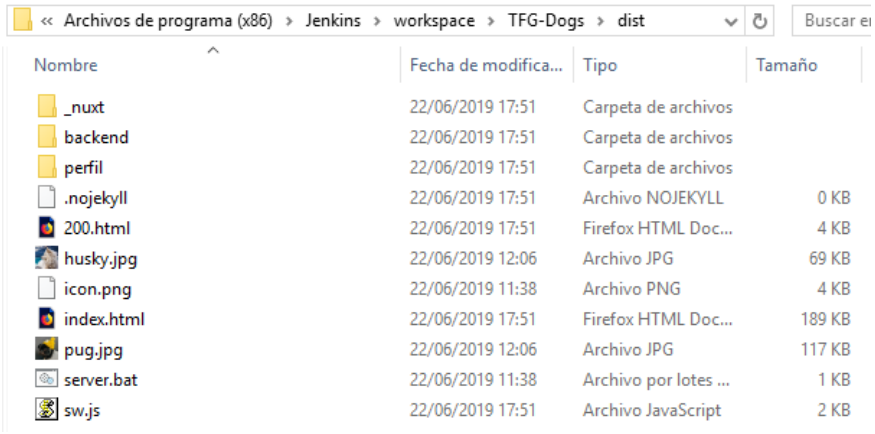

Figura 10: Contenido de la carpeta `dist` tras el ensamblaje hecho por Jenkins.

Servidor de ensamblaje vulnerado, acceso con privilegios administrativos

En este hipotético caso, el servidor de ensamblaje y pruebas ya ha sido vulnerado, y el atacante cuenta con acceso con privilegios administrativos.

Para simular un posible malware a instalar en el servidor vulnerado, se ha creado un pequeño programa en C#, que detecta cuando se escriben los ficheros del proyecto al hacer el clonado desde GitLab y reemplaza el contenido de los mismos. El programa se ejecuta en segundo plano, sin ocupar apenas memoria, sin utilizar apenas CPU gracias a que utilizar una espera pasiva mediante el uso de múltiples hilos, sin restricciones, y sin ser detectado por los antivirus del sistema.

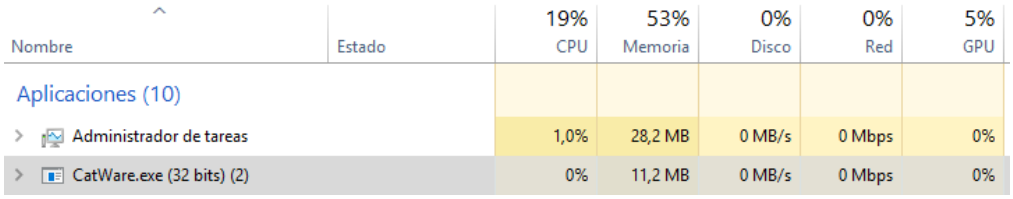

Figura 11: Captura del mínimo uso de recursos por parte del malware ficticio `CatWare.exe`.

Las fotos que aparecen en la aplicación las sustituye por fotos de gatos, y es capaz de manipular el texto del formulario de inicio de sesión. Es decir, es capaz de modificar, ya sea de forma inocua o maliciosa, los ficheros del proyecto.

Cabe pensar que este comportamiento sería detectable por Jenkins, ya que detectaría que los ficheros no son los mismos que se han subido a GitLab, o que al menos desharía los cambios al replicar el repositorio. Esta es la salida del registro de Jenkins con dicho programa ejecutándose en segundo plano:

Figura 12: Salida del registro de Jenkins, editada para resaltar los puntos de interés.

Al analizar la salida, es posible observar diferentes puntos de interés:

- Contacta con GitLab correctamente mediante la clave SSH proporcionada, visible en la parte indicada con una línea discontinua azul.
- Ejecuta `npm install` tras actualizar el directorio con los cambios subidos a GitLab, tal y como fue configurado. Al poseer las dependencias que necesita previamente descargadas, tan solo audita que todas sean correctas. Visible en la figura anterior en la parte señalada en verde.
- Ejecuta `npm run generate` para generar el paquete de página estática que podrá ser desplegado posteriormente. Visible en la figura anterior en la parte señalada en magenta.

- Termina con un mensaje de ***SUCCESS***, que indica que todo el ciclo CI-CD ha sucedido con éxito, visible en la figura anterior en rojo.

Al ser desplegado el proyecto mediante el servidor interno de PHP, se puede observar que pese a que todo indica que el ensamblaje ha sido correcto, el despliegue no ha ido como se esperaba.

Figura 13: La página final ha sido modificada por el malware, que está siendo ejecutado en segundo plano.

De haber sido este hipotético caso un entorno de CI-CD real, tan solo con pasar las pruebas de Jenkins el proyecto ya hubiese llegado al público de la aplicación, con las modificaciones malintencionadas realizadas sin ningún tipo de control.

Al revisar el repositorio remoto en el servidor de control de código fuente de GitLab, no es posible encontrar nada extraño, todos los archivos están cómo se subieron, y el último cambio está hecho por un desarrollador autorizado previamente, y no incluye nada.

Las modificaciones realizadas son puramente estéticas, pues son más fáciles de ver en este ejemplo.

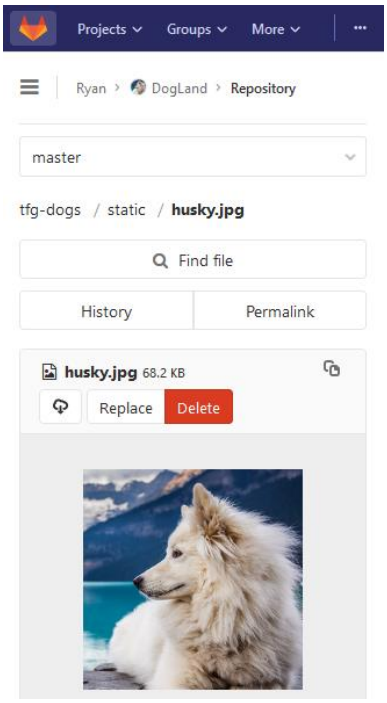

Figura 14: El repositorio está intacto.

No obstante, las modificaciones que realiza el malware diseñado para este caso de prueba pueden ir más allá de la estética. Es importante mencionar el hecho de que si no se alterase la estética de la aplicación web, esto podría pasar mucho más desapercibido, siendo sencillo para este malware colocar alguna puerta trasera para poder entrar al servidor de despliegue, o incluso registrar todo lo que se escriba en el formulario de inicio de sesión para conocer las credenciales de los clientes finales de la aplicación, estando así ante un ataque de intermediario difícilmente detectable en un ciclo CI-CD.

En estos ciclos CI-CD siempre se hacen las revisiones de código fuente al subir los cambios al repositorio de control de código fuente, y no se suele mirar el resultado del ensamblaje con demasiado detenimiento, pues lo genera un servidor que consideramos de confianza, el servidor de ensamblaje y pruebas.

El reto de seguridad: la integridad del software

El reto de seguridad que se plantea es el **cómo se puede hacer un control de la integridad de forma segura y confiable**, si la máquina que ensambla el software no es confiable por completo. Para ello, el TPM, será de gran utilidad para poder desarrollar un componente software capaz de comprobar la integridad del código fuente con ciertas garantías de seguridad, que serán proporcionadas por la naturaleza del propio TPM.

4

TPM: Módulo de Plataforma de Confianza

Qué es el TPM y por qué utilizarlo

El Módulo de Plataforma de Confianza (o *TPM*, *Trusted Platform Module* en inglés) es un **coprocesador criptográfico**, incluido en la gran parte de las computadoras domésticas y servidores desde el año 2003.

Se pueden destacar las **ventajas más importantes** que ofrece:

- Es **asequible**, e incluso **viene de serie** en multitud de plataformas actuales, por lo que no suele haber necesidad de adquirir hardware adicional.
- La **documentación** disponible al respecto es **extensa**, permitiendo su uso tanto por particulares como por equipos de trabajo.
- Está contrastado como **solución confiable**, gracias a su uso como componente clave a la hora de realizar el **arranque seguro** sistemas de computación modernos, así como en la **encriptación de ficheros**, presente en soluciones como *Microsoft BitLocker*.[9]

[9] Microsoft. (28 de Febrero de 2019). *Preguntas más frecuentes de información general de BitLocker y requisitos.* Obtenido de Microsoft Docs: https://docs.microsoft.com/es-es/windows/security/information-protection/bitlocker/bitlocker-overview-and-requirements-faq

Qué puede hacer el TPM.

- **Identificación de dispositivos e individuos**: antes de existir el TPM, la identificación de dispositivos electrónicos se hacía mediante su dirección *IP* o incluso su dirección *MAC*, la cual es fácilmente falsificable, y por tanto, una solución insegura. El TPM ataja este problema.
- **Generación segura de certificados y claves**: El TPM implementa por *hardware* un generador de números aleatorios seguro, lo cual ayuda a la generación segura de claves.
- **Almacenamiento seguro de claves**: al almacenarlos en un hardware especializado en ello, los ataques mediante software no tienen ningún efecto, proporcionando un medio seguro en el que llevar claves. Dicho almacenamiento es proporcionado en forma de **Memoria de Acceso Aleatorio No Volátil**, o *NVRAM* para abreviar. De esta forma, aunque se cambie de disco o se formatee, no se perderán ni las claves ni los certificados asociados al dispositivo.
- **Detección de modificaciones no autorizadas**: se puede verificar que un equipo funciona correctamente y no ha sido modificado sin autorización gracias al TPM.
- **Cifrado seguro:** el TPM puede ser utilizado para cifrar un archivo, una carpeta, un disco entero u otra información sensible como contraseñas maestras de un gestor de contraseñas.
- **Extensa variedad de algoritmos de cifrado**, como AES, RSA y ECC, todos ellos basados en estándares públicos y probados.
- **Generación de hashes criptográficamente seguros y únicos**, mediante algoritmos como SHA 256.

Interactuando con el TPM a través de TSS.NET

Se pueden hacer pruebas con el TPM directamente con el hardware, o fácilmente mediante software, con un emulador, con menos riesgos y con un manejo más sencillo y portable de la memoria permanente.

Este apartado está centrado en cómo hacer las pruebas con el simulador, que son fácilmente usables con un TPM en hardware, que más adelante serán útiles para el desarrollo de la solución que permita asegurar la integridad del software. El simulador lo ofrece Microsoft como miembro del *Trusted Computing Group*, grupo encargado del diseño y desarrollo del estándar en el que se basan todos los TPM del mercado, que a posteriori son fabricados por Infineon y AMD, entre otros. Se encuentra disponible en:

https://www.microsoft.com/en-us/download/details.aspx?id=52507

Una vez descargado, se obtiene un zip que contiene un único fichero ejecutable, que se debe descomprimir para poder ser utilizado mediante la consola de Windows o *PowerShell*, ya que es una aplicación para la línea de comandos, sin interfaz gráfica.

Para ejecutarlo, se escribe el nombre del ejecutable seguido del puerto en el que se desea que escuche. Por ejemplo:

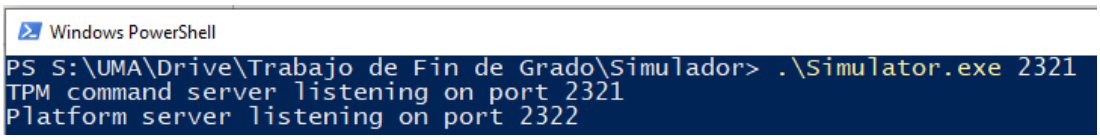

Figura 15: Ejecución del simulador en *PowerShell con el comando:* `.\Simulator.exe 2321`

Si no se indica de forma explícita un puerto, el simulador escuchará automáticamente por el puerto por defecto, el puerto 2321. Una vez iniciado, deberá permanecer en segundo plano mientras se quieran realizar pruebas sobre el simulador. El simulador escribe en disco, en el mismo directorio en el que se encuentra, un archivo denominado *NVChip*, que es la simulación de la memoria permanente.

Para hacer dichas pruebas se utiliza la librería TSS.Net, cuyo uso y compilación de ejemplos están descritos en detalle en el último apartado del Apéndice A. Este apartado describe las funciones básicas que se consideran más importantes para la realización de la solución software al problema propuesto en el capítulo anterior.

Conectar con el TPM

En todas las pruebas es necesario realizar la conexión al chip TPM. Es conveniente conocer las instrucciones comunes con las que conectar al TPM, o al simulador.

```
// Se selecciona el TPM o el simulador, acorde a lo que se quiera utilizar
Tpm2Device tpmDevice;
switch (tpmDeviceName){
	case DeviceWinTbs:
		tpmDevice = new TbsDevice();
	break;

	default:
		// Se inicializará el simulador por defecto.
		tpmDevice = new TcpTpmDevice("127.0.0.1", 2321);
		break;
}

// Conectar con el TPM o simulador
tpmDevice.Connect();

// Se inicializa el objeto del TPM con el dispositivo elegido
var tpm = new Tpm2(tpmDevice);

// Si se está usando el simulador, hay que encenderlo de forma artificial
if (tpmDevice is TcpTpmDevice){
	tpmDevice.PowerCycle();
	tpm.Startup(Su.Clear);
}
```

Una vez se haya terminado de usar el TPM, es recomendable realizar el proceso de desconexión con el método Dispose.

```
tpm.Dispose(); // No se usará el objeto TPM
```

Obtener capacidades del TPM

Mediante la librería es posible obtener una lista de las capacidades compatibles con el TPM que poseemos con el método *GetCapability*. Se puede conseguir una lista de los algoritmos que soporta el TPM, una lista de los comandos soportados por el TPM y los bancos PCR (*Platform Configuration Register*, o Registro de Configuración de la Plataforma) disponibles. Obtener, por ejemplo, la lista de algoritmos soportados es sencillo con la librería:

```
ICapabilitiesUnion capabilities;
tpm.GetCapability(Cap.Algs, 0, 1000, out capabilities);
```

Cap.Algs representa que, de todas las capacidades que se quieren obtener, se require una lista de algoritmos, que irá a parar a la variable de salida *capabilities*.

```
// Se hace un cast a un subtipo de array específico de la librería
var algorithms = (AlgPropertyArray)capabilities;

// Listar todos los algoritmos
Console.WriteLine("Algoritmos soportados:");
foreach (var algorithm in algorithms.algProperties){
      Console.WriteLine("  {0}", algorithm.alg.ToString());
}
```

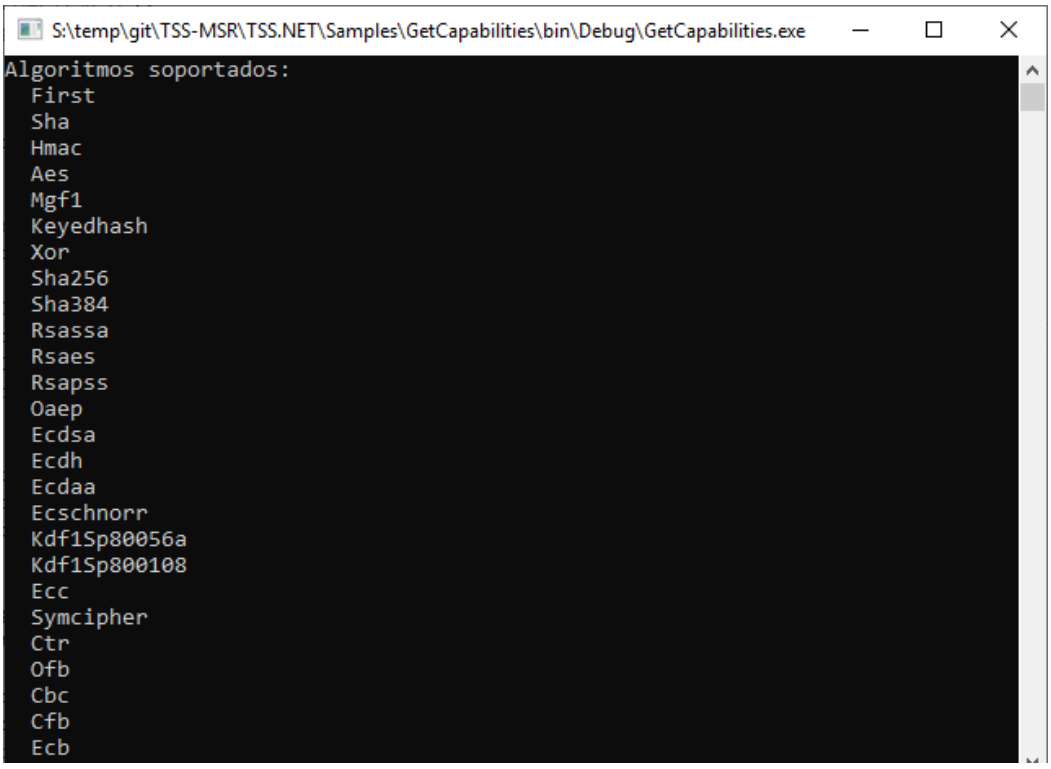

Figura 16: Resultado de obtener los algoritmos soportados por el simulador.

Utilizar el Generador de Números Aleatorios

Con la librería se puede utilizar de forma sencilla el Generador de Números Aleatorios, o RNG (*Random Number Generator*), para generar cadenas aleatorias de forma segura.

```
// Generar N bytes aleatorios
ushort N = 16;
byte[] randomBytes = tpm.GetRandom(N);
WriteBytes(randomBytes); // Método que usaremos para representar los bytes
```

Dicho código genera una salida por consola similar a:

```
Cadena generada aleatoriamente: 7046ad6ae799c464d635b79e97799164
```

Crear hashes con un algoritmo, a partir de datos concretos

Es posible utilizar el TPM para generar un hash con un algoritmo concreto, a partir de los datos de entrada que se le quieran proporcionar. En este caso de ejemplo, se creará un hash utilizando el algoritmo SHA-256, de los números naturales impares menores que 10:

```
TkHashcheck validation;                // Ticket (sin usar en este ejemplo)
byte[] hashData = tpm.Hash(
       new byte[] { 1, 3, 5, 7, 9 },    // Datos a hashear
       TpmAlgId.Sha256,                 // Algoritmo usado (SHA256 aquí)
       TpmRh.Owner,                     // Jerarquía del ticket (sin usar)
       out validation                   // Ticket (sin usar en este ejemplo)
);
Console.WriteLine("Hash: " + BitConverter.ToString(hashData));
```

```
Hash secuencial: 77-6F-02-43-A8-43-35-F0-DA-24-66-65-D2-9D-72-49-C8-5F-DF-41-
6E-85-2D-31-E4-C2-AC-17-26-F9-74-65
```

Computación de los valores obtenidos de aplicar las funciones hash a una secuencia de datos

De forma análoga a lo visto anteriormente, es posible crear un hash a partir de agregar datos a una secuencia, que puede llegar a ser más grande que el buffer que posee el TPM. De nuevo, como ejemplo, se creará un hash utilizando el algoritmo SHA-256, de los números naturales impares menores que 10:

```
// Crear un handle necesario para ir agregando datos a la secuencia hash
TpmHandle hashHandle = tpm.HashSequenceStart(
       AuthValue.FromRandom(10),    // Autorización aleatoria para esta sesión
       TpmAlgId.Sha256              // Usando SHA-256
);

// Se agregan números impares secuencialmente
tpm.SequenceUpdate(hashHandle, new byte[] { 1, 3 });
tpm.SequenceUpdate(hashHandle, new byte[] { 5, 7 });

TkHashcheck validation;
byte[] hashedData = tpm.SequenceComplete(
    hashHandle,          // El Handle que se creó anteriormente
    new byte[] { 9 },    // Último bloque
    TpmRh.Owner,
    out validation
);

Console.WriteLine("Hash secuencial: " + BitConverter.ToString(hashedData));
```

Se observa que se obtiene el mismo valor que en el subapartado anterior, pese a que los datos han sido añadidos de forma separada:

```
Hash secuencial: 77-6F-02-43-A8-43-35-F0-DA-24-66-65-D2-9D-72-49-C8-5F-DF-41-
6E-85-2D-31-E4-C2-AC-17-26-F9-74-65
```

5

Utilizando el TPM para conseguir la integridad del código fuente

El reto planteado consiste en poder asegurar la integridad del software que se quiere desplegar bajo premisa, de forma que sea el mismo que se desarrolló intencionalmente, sin puertas traseras ni modificaciones indeseadas. El TPM es, con las funcionalidades descritas en el capítulo anterior, el encargado de proporcionar una plataforma segura para generar la solución que nos permitirá alcanzar el objetivo del TFG.

Análisis de la solución desarrollada

La solución propuesta se ha denominado *Trusted Integrity Plaftform*, o **TIP** para abreviar. TIP consiste en utilizar un nuevo componente en el *pipeline* CI-CD, que llamaremos Servidor TIP, que será dotado de un TPM y contará con una pila de software de confianza, dedicada únicamente a comprobar la integridad de los proyectos software.

Repositorios en Git: fuente de integridad de confianza

Git provee integridad de datos, es decir, los repositorios en los que se almacenan el código fuente de un proyecto software ya tienen medidas de comprobación de integridad.

Según la web oficial de Git "Esta funcionalidad está implementada en los niveles más bajos en Git, y forma parte de su filosofía"[10].

Git lo consigue manteniendo siempre una lista de hash de todos los ficheros del proyecto, y creando un hash conjunto para cada vez que se envía un registro de cambio mediante el comando `git commit`.

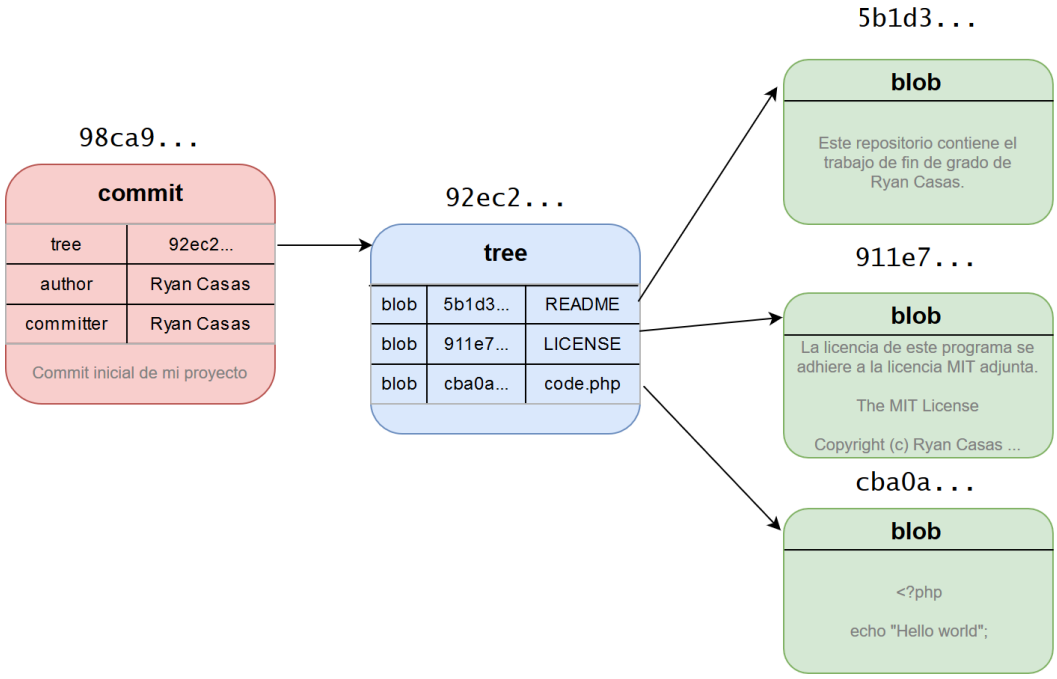

Figura 17: Descomposición de un *commit*.

Tal y como se ilustra en la figura anterior, cada *commit* (en rojo) tiene un hash asociado, generado a partir de los datos del árbol de archivos (representado en azul), cada uno con el hash del archivo que le corresponde (representados en verde). Al hacer un cambio en un archivo, dicho cambio se propaga por los hashes del árbol, y del *commit*, siendo imposible que haya un cambio en el repositorio sin que Git lo detecte. Por tanto, se puede concluir que aquello que esté presente en un repositorio controlado por Git es de confianza, ya que mantiene la integridad de los datos.

[10] Chacon S. & Straub B. (22 de Marzo de 2019). *Git Basics*. Obtenido de Git: https://git-scm.com/book/en/v1/Getting-Started-Git-Basics#Git-Has-Integrity

Medidas de seguridad explícitas para el Servidor TIP

El servidor TIP debe ser un servidor dedicado a la verificación de la integridad, dotado de un TPM, sin más software presente que el absolutamente necesario para obtener el código fuente del repositorio (un cliente Git), y realizar la propia verificación de integridad (parte de la solución aportada más adelante).

Puesto que el software que se ejecute en el servidor TIP es predecible, permite implementar el protocolo de arranque seguro que proporciona como funcionalidad el propio TPM (por ejemplo, desde Windows 8 se utiliza), para verificar que la pila de software no ha cambiado.

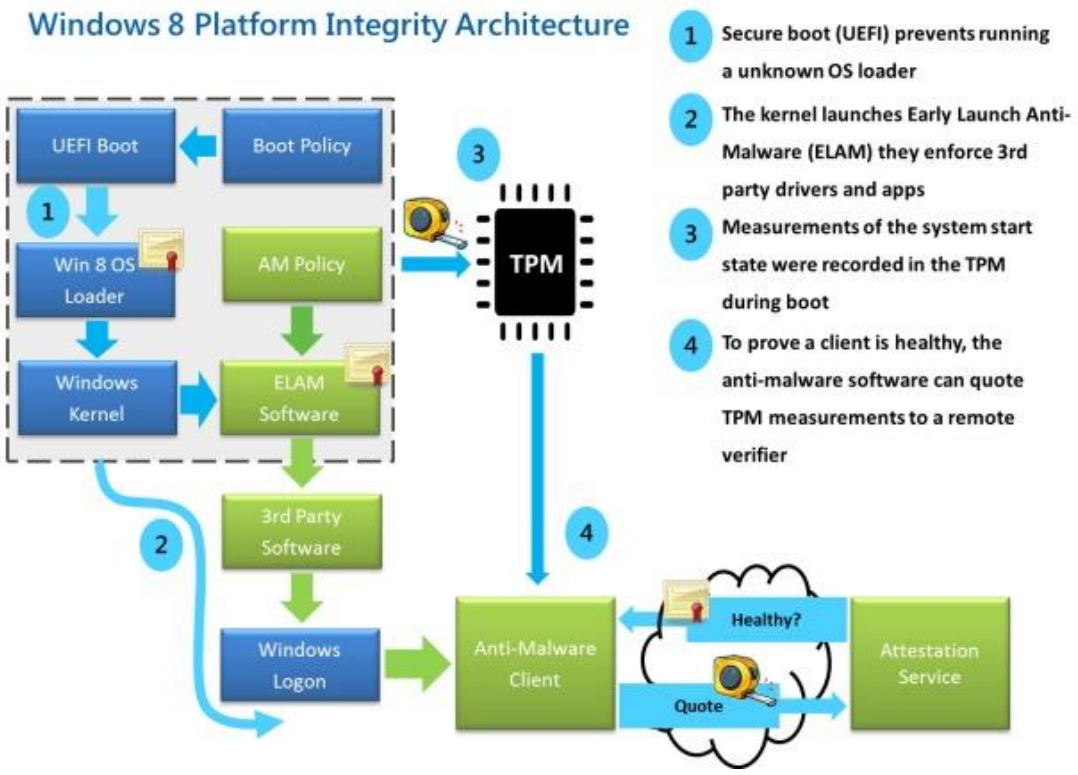

Figura 18: Diagrama que explica el arranque seguro implementado con ayuda del TPM a partir de Windows 8. Créditos: Microsoft.

Dicha medida de seguridad resulta útil para asegurar el servidor TIP con un paso más, haciendo uso del TPM, aunque no es indispensable, siendo más relevantes las medidas de seguridad que se aplican a un servidor web. Algunas de esas medidas:

- Utilizar sólo acceso mediante claves SSH para administrarlo.
- Capar todos los puertos que no estén sirviendo contenido web.
- Restringir y controlar el acceso al gestor de paquetes.

Triple verificación de integridad

La integridad del software se comprueba, como mínimo, tres veces a lo largo del ciclo CI-CD:

- **Antes de instalar las dependencias necesarias para el proyecto**: Garantiza que el código fuente que recibe el servidor de ensamblaje y pruebas sea idéntico al que existe en el servidor de control de código fuente.
- **Antes del ensamblaje**: Garantiza que el código fuente que se está ensamblando no haya sido modificado por ningún agente externo presente en el servidor de ensamblaje y pruebas**.**
- **Antes del despliegue:** Garantiza que el ciclo se ha cumplido sin modificaciones indeseadas tras haber ensamblado el proyecto.

Cada comprobación de integridad se ejecuta siguiendo unos pasos concretos, que se pueden resumir en tres fases principales:

1. **Recepción del código sospechoso**: El servidor de ensamblaje y pruebas envía al servidor TIP un archivo comprimido con el código fuente que posee, considerado "***sospechoso.zip***", que trata de descomprimir el servidor TIP. Si no es capaz de descomprimirlo, desechará el archivo y dará por inválida la verificación de integridad.
2. **Recepción del código de confianza**: El servidor TIP descarga, por otro lado, el código fuente desde el repositorio Git del proyecto, que se considera de confianza.
3. **Comprobación de los *bigHashes***: El servidor TIP comprueba, mediante el uso del TPM, que tanto el contenido de *sospechoso.zip* como el código fuente del repositorio sean iguales. Para ello, consultará con Git los hashes de cada fichero sin tener que recalcularlos, mediante los metadatos que Git registra, los concatenará en una cadena única que denominaremos *bigHash*, y ejecutará una función hash con el TPM sobre el *bigHash*. Si ambos *bigHash* (tanto el del proyecto, como el sospechoso) son iguales, la prueba de integridad resultará satisfactoria.

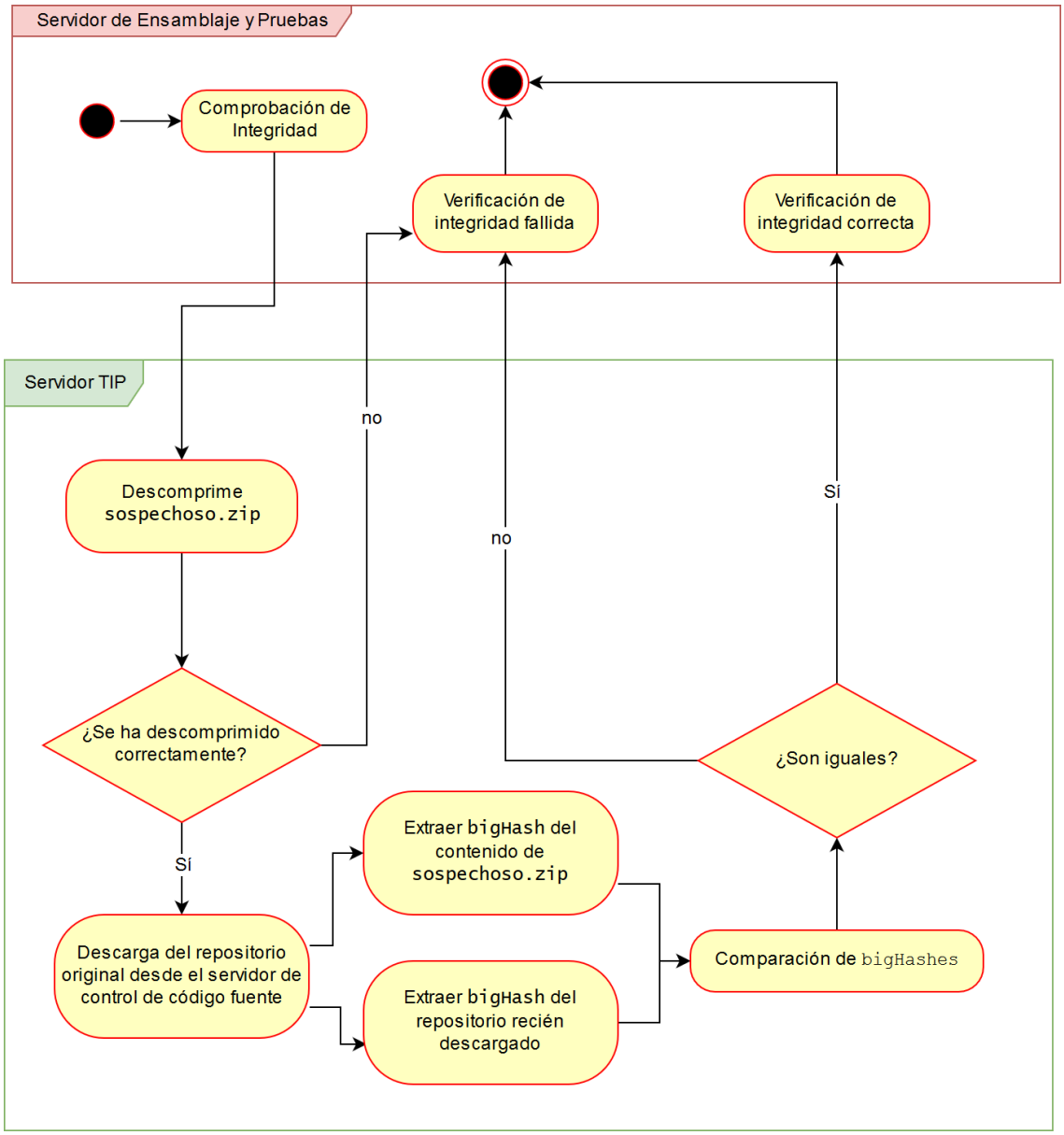

Figura 19: Diagrama de flujo UML sobre un ciclo de comprobación de integridad.

En la siguiente figura se muestra en un diagrama de secuencias del proceso de comunicación se sigue al implementar TIP en un ciclo de CI-CD. En ella, se puede observar la triple verificación descrita en este apartado, así como dónde se producen las comprobaciones de integridad. Se dice que la figura representa una visión aproximada del sistema, porque la obtención del código fuente por parte del servidor TIP se realiza en cada comprobación de integridad, pero se omiten del diagrama para facilitar su comprensión, así como se omiten todos los comandos de ensamblaje y pruebas, característicos de cada proyecto particular.

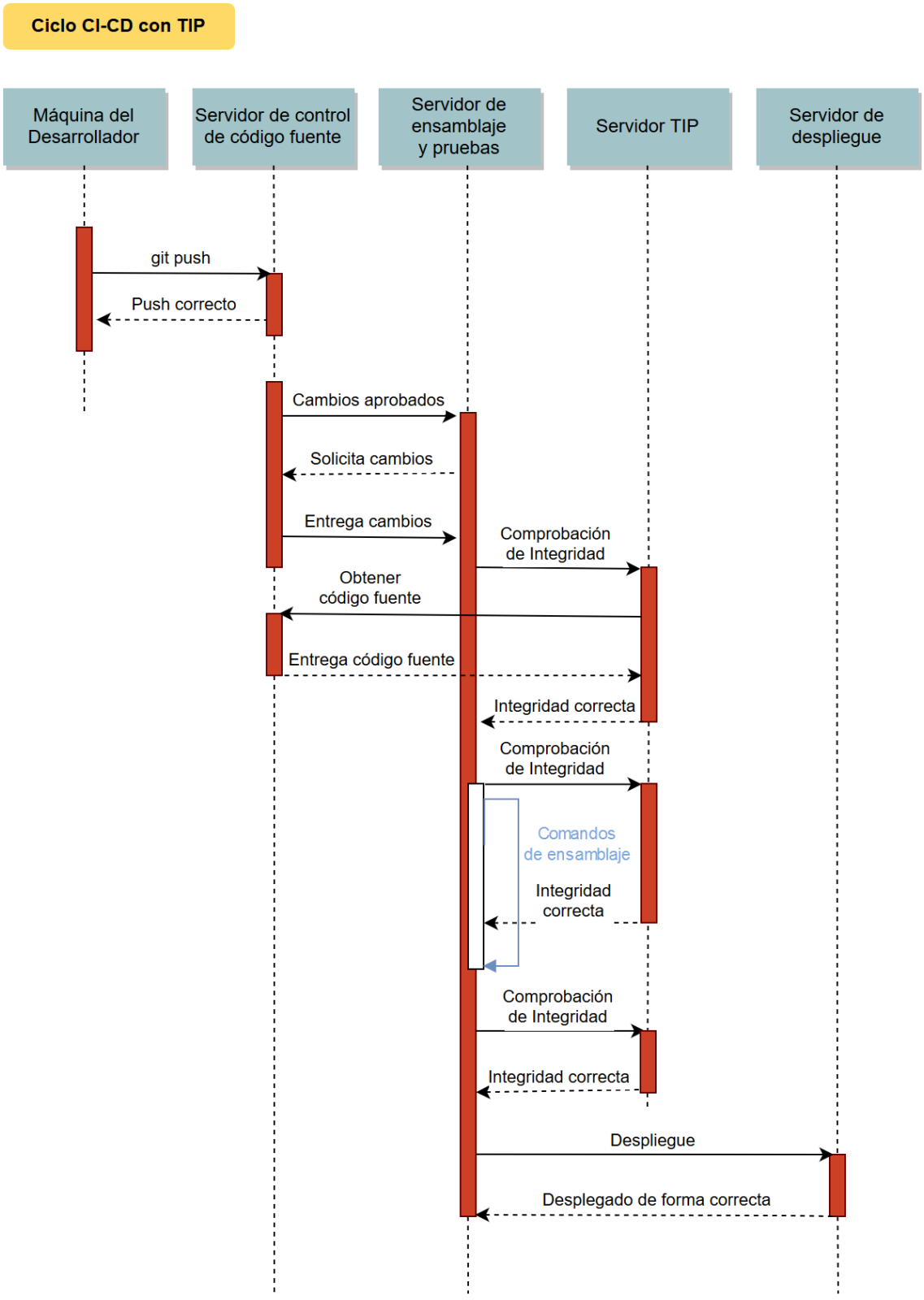

Figura 20: Diagrama de secuencias en UML que muestra de forma aproximada un ciclo CI-CD con TIP.

Análisis de una implementación concreta de un servidor TIP

En este apartado se analiza una implementación concreta de un servidor TIP, ya que la idea puede ser portada a varios lenguajes de programación, y la función hash a utilizar con el TPM pueden ser varias.

La implementación propuesta:

- Funciona sobre **Windows 10 1809** en adelante, y aunque no se descarta su retrocompatibilidad con versiones anteriores, tampoco se ha comprobado.
- La implementación de la funcionalidad con el TPM se ha implementado en C#. Por ello, se requiere tener como mínimo .NET Framework 4.6 instalado para funcionar. Por suerte, en Windows 10 viene instalado por defecto, y se actualiza a la par que el sistema operativo, de forma automática.
- La implementación del servidor TIP utiliza de scripts de *PowerShell* y un script PHP que hace de puerta de entrada. El servidor web que atenderá las peticiones del servidor de ensamblaje y pruebas será el servidor interno de PHP, pues se está realizando una prueba de concepto, pero puede ser ejecutado sobre cualquier servidor que acepte PHP-CGI.
- El script de comunicación con el servidor TIP para la comprobación de integridad, que se agrega en el ciclo CI-CD como parte de la batería de pruebas, está escrito en PowerShell, y probado con Jenkins. La elección de este último es debido a que es uno de los entornos CI-CD de código abierto más utilizados y extendidos.
- Sigue en la medida de lo posible la normativa AENOR ISO/IEC 25010:2011, con especial hincapié en las partes de funcionalidad, coexistencia, interoperabilidad, instalabilidad, reusabilidad, modularidad, y por supuesto, integridad.

Script para la comunicación con el servidor TIP

El script que se comunica con el servidor TIP desde Jenkins está escrito en PowerShell, y es sencillo. Primero configura los parámetros de subida:

```
$params = @{
    "repository"=$repository;    # URL completa del repositorio
    "realTPM"=$real;             # Si se está usando el TPM físico o el
                                 # simulador
}
```

`$repository` y `$real` son parámetros de entrada del script PowerShell.

También hay que configurar los parámetros para la compresión del archivo .zip. En este caso, los directorios y archivos a excluir.

```
$DirsToExclude=@(
    "dist",
    "node_modules",
    ".npm",
    ".nuxt",
    ".git"
)

$FilenamesToExclude=@(
    "sw.js"
)
```

Una vez configurado, se procede a copiar a una carpeta temporal todos los archivos del espacio de trabajo de Jenkins. Los archivos serán filtrados una vez copiados, eliminando aquellos que están en la lista de exclusión. De esta forma, se preserva el estado del espacio de trabajo.

```
$sourceRoot = "."
$destinationRoot = "${env:TEMP}\tip-$(New-Guid)"

Copy-Item -Path $sourceRoot -Filter $DirsToExclude -Recurse -Destination
$destinationRoot -Container
Set-Location $destinationRoot
foreach($file in GetFiles "./"){
    if(Test-Path $file -PathType Leaf){
        $File = Get-ChildItem $file
        if($File.Name -in $FilenamesToExclude){
            Remove-Item $File
        }
    }
}
```

Más adelante, se genera el fichero comprimido, con la copia de los ficheros del espacio de trabajo de Jenkins:

```
Get-ChildItem -Path "${destinationRoot}\*"           |
    where { $_.Name -notin $DirsToExclude}           |
    Compress-Archive -DestinationPath "${destinationRoot}.zip" -Force
```

En este comando, se recogen primero todos los archivos del directorio, se filtran aquellos directorios que no se desean comprimir y se comprime. Cabe destacar el uso de la opción `-Force`, que en el comando `Compress-Archive` sirve para sobreescribir el fichero comprimido en caso de que ya exista.

Una vez obtenido el fichero comprimido, procedemos a enviarlo al servidor TIP:

```
$result = Get-ChildItem "${destinationRoot}.zip" |
      Send-MultiPartForm "${tipServer}/gateway.php" $params
$result
exit $result
```

`$tipServer` es un parámetro de entrada del script, que por defecto es http://127.0.0.1:9119, es decir, un servidor local. Se sale del comando con la salida del resultado otorgado por el servidor TIP (0 si no hay errores, 1 si hay algún error).

Servidor TIP en PHP

Para el servidor TIP, hay un fichero gateway.php, que será el fichero principal encargado del servidor TIP. Este fichero, tiene algunas variables configurables:

```
#define("API_KEY", "lGnZ...");
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Credentials: true");
header("Access-Control-Allow-Headers: X-API-KEY, Origin, X-Requested-
With, Content-Type, Accept, Access-Control-Request-Method");
header("Access-Control-Allow-Methods: GET, POST, OPTIONS, PUT, DELETE");
```

Hay un comentario al principio, que define una API_KEY. Esta es una posible mejora, comentada en el capítulo siguiente. El resto son cabeceras de control HTTP, para permitir que el servidor TIP a desplegar pueda interactuarse desde fuera.

Una vez hechos los pasos de configuración necesarios en el fichero PHP, se procede a descomprimir el archivo ZIP comprimido, si es que ha sido subido.

```
// Move the uploaded suspicious ZIP
if(!isset($_FILES, $_FILES['files']))
    die("No suspicious ZIP uploaded");

$suspiciousZip = uniqid("tip-suspect-", 1).'.zip';
$suspiciousZipRoute = $_ENV['TEMP']."/$suspiciousZip";
move_uploaded_file($_FILES['files']['tmp_name'][0], $suspiciousZipRoute);
unset($_FILES);

// Unzip it
$zip = new ZipArchive;
if($zip->open($suspiciousZipRoute) === false){
    @unlink($suspiciousZipRoute);
    die("Invalid ZIP");
```

```
}
$zip->extractTo('suspects'.$path);
$zip->close();
unlink($suspiciousZipRoute);
```

El archivo comprimido se descomprime a una carpeta *suspects,* donde van a parar todos los archivos comprimidos sospechosos que se suben. Si ya se había descomprimido antes un archivo del mismo repositorio, se borra el directorio antiguo por completo. Una vez descomprimido, el `.zip` se elimina.

La constante ABSPATH que aparece en el código define la ruta al directorio de trabajo actual del servidor TIP.

Cuando ha terminado con el fichero subido, se procede a descargar el repositorio del proyecto desde el servidor de control de código fuente:

```
// Clone the trusted repository
exec("git clone ".escapeshellarg($_POST['repository'])."
".escapeshellarg(ABSPATH.$directory));
```

Se puede observar que se recoge por $_POST la variable `repository` del script para la comunicación con el servidor TIP, descrito en el apartado anterior.

Una vez clonado el repositorio, se procede a calcular los *`bigHash`* de los que se hablaba anteriormente en este capítulo. Para ello, se invoca un script en PowerShell aparte, que se definirá más adelante:

```
// Retrieve the trusted repository's TPM hash
$command = 'powershell.exe -executionpolicy bypass -File
'.ABSPATH.'/utils/tip-hashGitProject.ps1'.
            " -workspace ".escapeshellarg(ABSPATH.$directory).
            " -device ".escapeshellarg(DEVICE)
;

$projectBigHash = exec($command);


// Retrieve the suspected integrity repo hash
$command2 = 'powershell.exe -executionpolicy bypass -File
'.ABSPATH.'/utils/tip-hashGitProject.ps1'.
```

```
            " -workspace ".escapeshellarg(ABSPATH.'/suspects'.$path).
            " -device ".escapeshellarg(DEVICE).
            ' -suspicious'
;

$suspiciousBigHash = exec($command2);
```

Justo después, eliminamos las carpetas que hemos utilizado:

```
// Remove the folders created before
function rmdirr($dir){
    exec('powershell.exe -executionpolicy bypass -File
'.ABSPATH.'/utils/tip-removeDir.ps1 -workspace '.escapeshellarg($dir));
}

rmdirr(ABSPATH.'/suspects'.$path);
rmdirr(ABSPATH.$directory.'/');
```

Por último, se comparan ambos `bigHash`, y se llega a una conclusión. 0 si son iguales, 1 si no lo son:

```
echo ($suspiciousBigHash === $projectBigHash)?0:1;
```

Interacción con el TPM, generación del hash

Para realizar el hash de un repositorio Git entero, se recorren uno por uno los ficheros y se va almacenando en una variable cada uno de los hashes. Sin embargo, se ha decidido en esta implementación no calcular los hashes de cada fichero, en pro de reutilizar los hashes que ya ha calculado Git. Para ello, se ha implementado un script en PowerShell.

```
$bigHash = ""
Set-Location "${workspace}/"
foreach($file in GetFiles "./"){
    if(Test-Path $file -PathType Leaf){
        if($suspicious){
            $hashInfo = git hash-object $file
        }else{
            $hashInfo = git ls-files -s $file
```

```
            $hashInfo = ($hashInfo -split " ")[1]
        }
        $bigHash += $hashInfo
    }
}
```

Lo más destacable de esta porción es el comando `git ls-files -s $file`, con el que se extraen los metadatos concretos que tiene en caché Git de un fichero. Hay que manipular la salida, pues el comando no entrega directamente el hash.

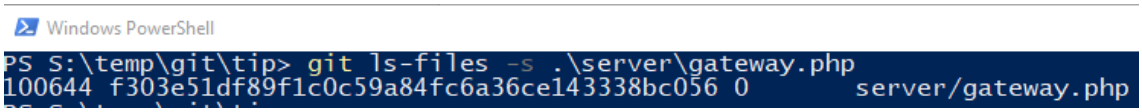

Figura 21: Ejemplo de salida del comando `git ls-files -s`, con el que se consiguen los metadatos de un archivo en un servidor Git.

Si el directorio que se está examinando es del que se sospecha, utilizaremos el comando `git hash-object $file` para obtener dinámicamente el hash del objeto, pues lo que Git tendría en caché no tiene por qué ser el hash del objeto en cuestión.

Habiendo obtenido la concatenación de los hashes, se procede a utilizar una función hash que haga uso el TPM. En esta implementación, se ha decidido utilizar SHA-256, ya que SHA-1 se considera un algoritmo de hash vulnerable a ciertos ataques[11], y lamentablemente el TPM no soporta SHA-512.

Lo más reseñable de la implementación de este hash de una cadena de tamaño variable, que además puede exceder el buffer de entrada del TPM, es cómo gestionarla.

```
string buffer = "";
foreach(char c in toHash){
	buffer += c;
	if(buffer.Length > 31){ // If the TPM buffer is filled up
		tpm.SequenceUpdate(hashHandle, Encoding.ASCII.GetBytes(buffer));
		buffer = "";
	}
}
```

[11] Stevens, M. & Bursztein E. & Karpman P. & Albertini A. & Markov Y. (23 de Febrero de 2017). *The first collision for full SHA-1*. Obtenido de SHAttered: https://shattered.io/static/shattered.pdf

Es importante conocer que el buffer del TPM tiene problemas al exceder los 32 bytes[12], por lo que el control manual es necesario. Además, es importante destacar el hecho de que pueden haber cadenas que no estén partidas de 32 en 32:

```
// If there's anything remaining
byte[] remain;
if (buffer.Length == 0){
    remain = new byte[] { };
}else{
    remain = Encoding.ASCII.GetBytes(buffer);
}

// Hash with the remaining part of the string
TkHashcheck validation;
byte[] hashedData = tpm.SequenceComplete(hashHandle, remain, TpmRh.Owner, out
validation);
```

[12] Así se comenta en el *bugtracker* de la implementación del TPM de IBM: https://sourceforge.net/p/ibmtpm20tss/tickets/6/

6

Resultados y conclusiones

Integración en el ciclo de Jenkins

Para probar que el sistema funcione, se ha decidido integrar en el ciclo CI-CD de la aplicación "ficticia" demostrada en el segundo capítulo, *DogLand*. Para ello, se debe copiar el script de comunicación con el servidor TIP en la raíz del proyecto.

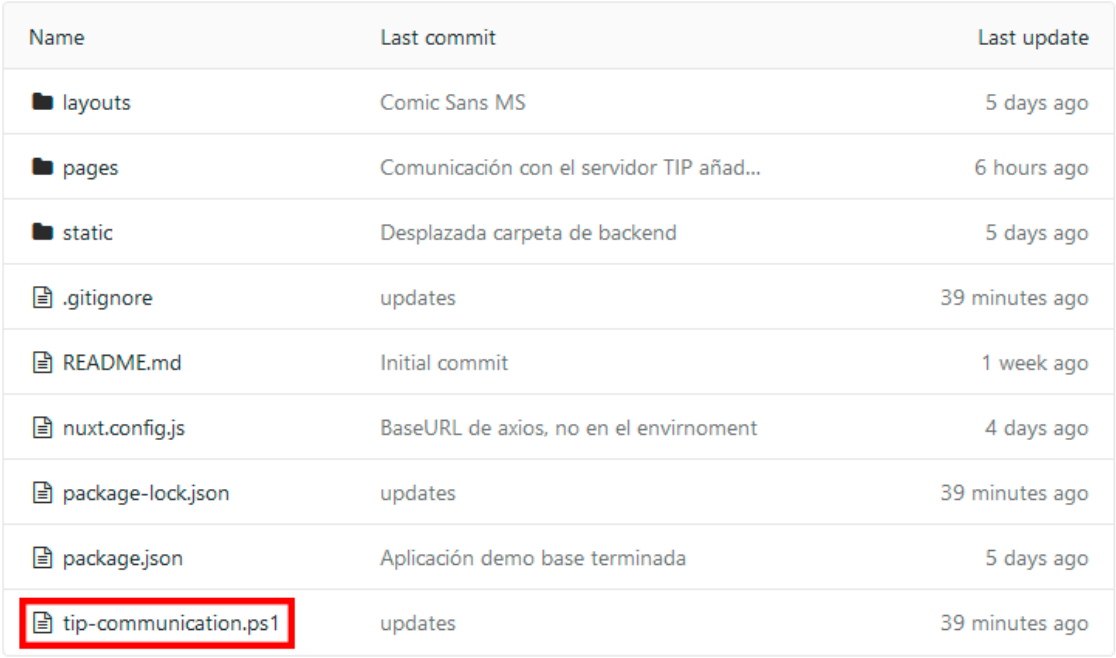

Figura 22: Script PowerShell para la comunicación con el servidor TIP, subido en la carpeta raíz del proyecto.

Una vez ubicada como parte del proyecto y se ha configurado como se especifica en el capítulo anterior, hay que reconfigurar el proyecto Jenkins para integrar la comunicación con el servidor TIP.

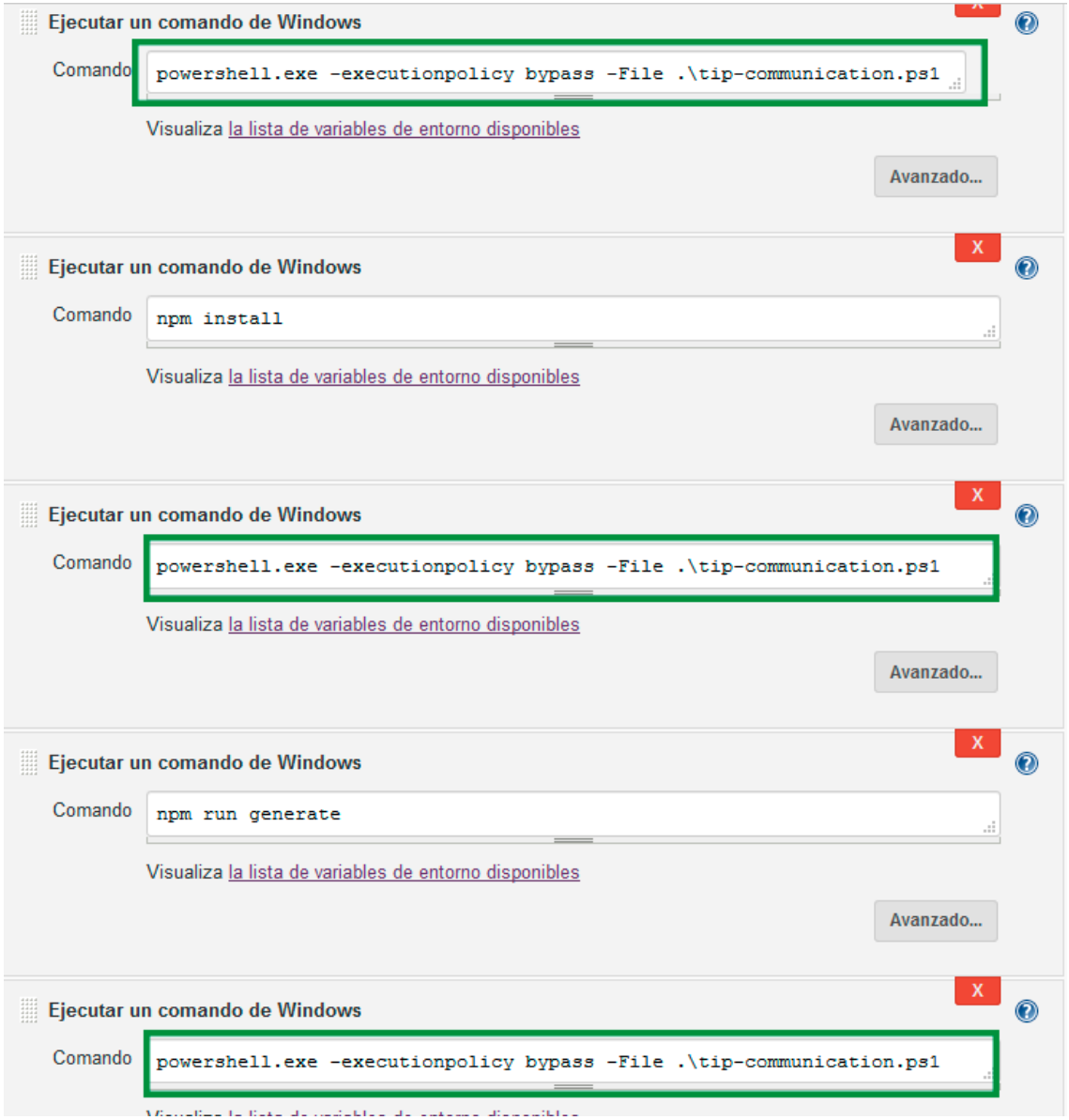

Figura 23: Comandos para la comunicación con el servidor TIP en el ciclo de Jenkins, señalados en verde.

Como se especifica en el capítulo anterior, se comprueba la integridad como mínimo tres veces, y por ello el comando aparece repetido tres veces.

Resultados de la integración con Jenkins

Al haberlo integrado con Jenkins con las instrucciones del apartado anterior, a partir de ahora todos los ciclos pasarán la triple verificación de integridad.

Para probarlo, se ejecuta manualmente un ciclo:

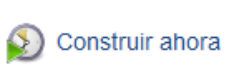

Figura 24: Botón para forzar a Jenkins la ejecución de un ciclo, pese a la ausencia de cambios en Git.

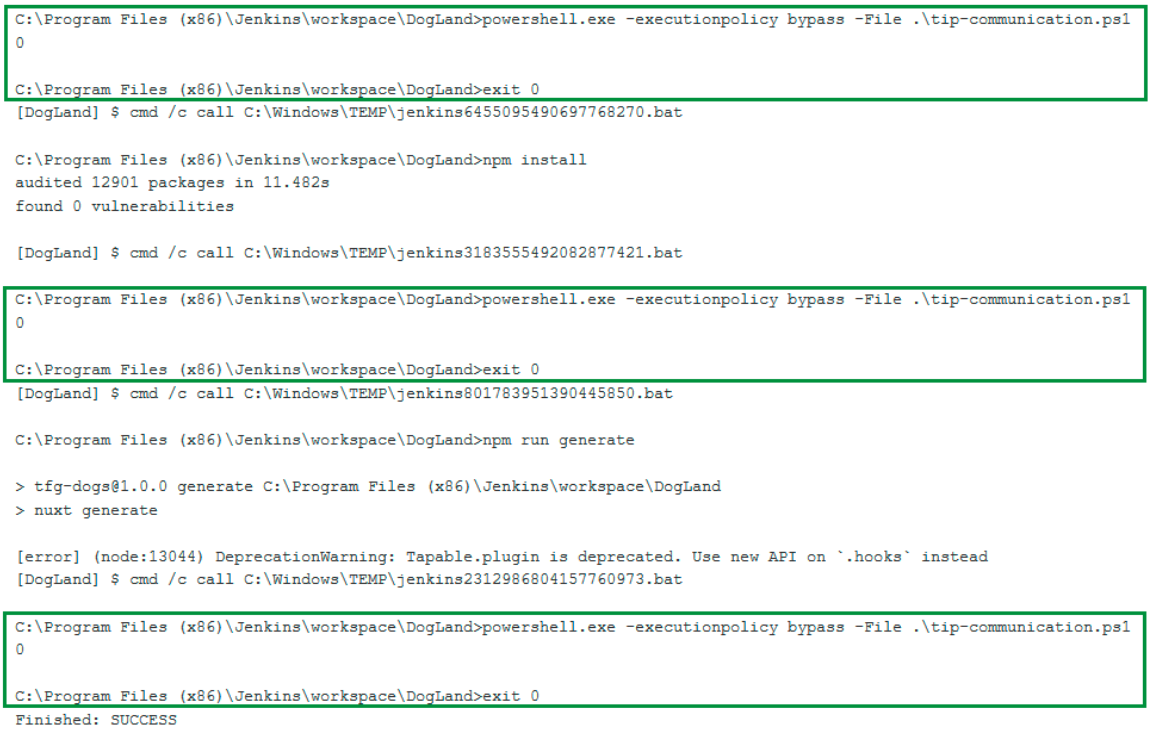

Figura 25: Salida del ciclo de Jenkins, con la comprobación de integridad correcta.

Se puede observar que el ciclo ocurre correctamente, tal y como se espera. Además, si el malware que se propuso en el capítulo 2 se está ejecutando, el ciclo falla por completo, al fallar la verificación de integridad:

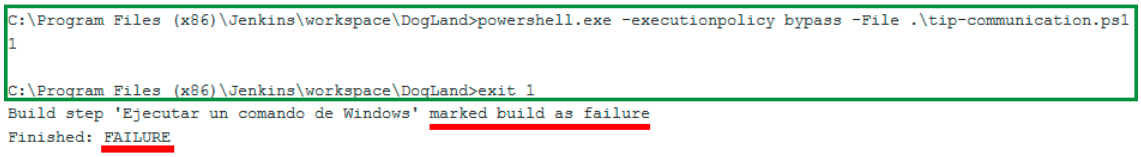

Figura 26: Salida de un ciclo de Jenkins al ejecutarse un malware que modifica los ficheros del espacio de trabajo.

Se produce un efecto colateral interesante con la verificación de integridad: si alguna dependencia del proyecto varía entre la versión que subió el desarrollador y la versión que Jenkins está tratando de compilar, también se producirá un error, justo en la verificación de integridad anterior al ensamblaje. Este suceso es útil, pues ayuda a conocer que la versión que se despliega de la aplicación contiene dependencias actualizadas, que potencialmente pueden alterar el funcionamiento de nuestra aplicación (como en el caso de una deprecación o eliminación de una característica de la que dependa el proyecto).

Posibilidades de mejora e ideas para ampliar

Tanto la solución propuesta como la implementación realizada es válida como prueba de concepto, aunque para integrarlo en proyectos de mayor escala es posible que se tengan que realizar algunos cambios, mejoras o ampliaciones.

Implementación multihilo del Servidor TIP

El servidor TIP realiza algunas acciones que se pueden paralelizar, como por ejemplo:

- Obtener el repositorio Git de confianza, mientras se descomprime el archivo `sospechoso.zip` recibido.
- La obtención de los *`bigHashes`*, que actualmente se realiza de forma secuencial.

Esta idea acarrea también tener en cuenta los problemas de concurrencia asociados, además de la posible reimplementación de la parte en PHP del servidor TIP, ya que PHP no es un lenguaje adecuado para realizar tareas multihilo, debido a que su estructura es fundamentalmente crear un hilo por petición HTTP.

Creación del archivo sospechoso.zip mediante "enlaces duros"

Se podría optimizar la creación del fichero comprimido sospechoso.zip si se utilizasen enlaces duros, o *hardlinks* (*JUNCTIONS* en Windows), en vez de copiar el espacio de trabajo a una carpeta temporal, ahorrando así espacio en disco y el tiempo de copiar los archivos. La idea no se ha llegado a implementar, pues requiere el estudio más detenidamente del comportamiento de los comandos de PowerShell con dichos enlaces duros, ya que pueden resultar en riesgos de seguridad no previstos con programas de terceros[13].

[13] Como la vulnerabilidad descubierta en el Servicio de Mantenimiento de Mozilla por el equipo de Project Zero en Google: https://googleprojectzero.blogspot.com/2015/12/between-rock-and-hard-link.html

Integración con otros entornos de CI-CD

Existen más entornos CI-CD bajo premisa, como *PHP-CI*, *GitLab Community Edition* o *Abstruse*. Sería de especial interés ver cómo se comporta en otros entornos de CI-CD, pese a que la solución propuesta no debería dar ningún problema, ya que es portable, independiente y fácilmente integrable.

Implementación para GNU/Linux

Se podría implementar todo el código del servidor TIP para una distribución GNU/Linux, teniendo en cuenta los siguientes puntos:

- La funcionalidad con el TPM, escrita en C#, puede ser compilada directamente para .NET Core, disponible en varias distribuciones de Linux, o ejecutar los .exe utilizando *Mono*. También podría implementarse la misma funcionalidad en C++.
- Los scripts de PowerShell no ofrecen ninguna dificultad, pues PowerShell es multiplataforma.[14]
- PHP es fácilmente instalable en casi cualquier distribución GNU/Linux. La implementación del archivo `gateway.php` es perfectamente reutilizable, tan solo sería necesario cambiar la forma en la que se invocan los script de PowerShell.

Por tanto, una versión para Linux del servidor TIP no sería algo tan descabellado. Sin embargo, se ha evitado entrar en la compatibilidad multiplataforma de forma explícita, pues excedería con creces el tiempo dedicado al TFG.

Uso de claves público/privada mediante el TPM en el servidor TIP

En un principio, se trató de implementar una medida extra de seguridad sobre el servidor TIP, que consistía en agregar mediante una tercera fuente de integridad: el desarrollador.

---

[14] Compatible con todos los sistemas operativos expuestos en https://docs.microsoft.com/es-es/powershell/scripting/install/installing-powershell-core-on-linux?view=powershell-6

Cada vez que el desarrollador enviase un cambio al servidor Git, enviaría además una copia del proyecto firmada con la clave privada del desarrollador al servidor TIP, que sería considerada de confianza. Al llegar al servidor TIP, dicha copia del proyecto sería almacenada. Al ejecutarse una verificación de integridad, la copia del desarrollador sería leída con su clave pública (que previamente habría enviado por un medio seguro al servidor TIP). Con las tres copias presentes, la del desarrollador, la copia de Git y la copia sospechosa directa del servidor de ensamblaje, se ejecutaría la comprobación de integridad con tres copias diferentes, que deberían ser iguales entre sí, otorgando más rigidez al servidor TIP a la hora de evaluar la integridad.

Si bien esto parecía una buena idea en principio, surgieron varios inconvenientes que impidieron que se llevase a cabo.

El primero de ellos es que, pese a que es sencillo generar una clave privada dentro del TPM, extraer una clave pública al disco duro no es una tarea trivial. Antes de 2016 existía un método para extraer en formato XML la clave pública, funcionalidad que Microsoft deprecó sin indicar ninguna alternativa. Se intentó recrear la funcionalidad perdida, calculando la clave pública RSA mediante su módulo y exponente, datos que sí se proporcionan en la librería:

```
var rsaParams = (RsaParms)keyPublic.parameters;
var exponent = rsaParams.exponent != 0
     ? Globs.HostToNet(rsaParams.exponent)
     : RsaParms.DefaultExponent;

var modulus = (keyPublic.unique as Tpm2bPublicKeyRsa).buffer;
```

Con dichos datos, se intenta recrear el formato de una clave pública, siguiendo el estándar RFC 4716[15]:

```
// RFC 4716
var pemKey = Combine(
    new byte[] {0x00, 0x00, 0x00, 0x07},
    new byte[] {0x73, 0x73, 0x68, 0x2d, 0x72, 0x73, 0x61},
    new byte[] {0x00, 0x00, 0x00, 0x03},
    exponent,
    new byte[] {0x00, 0x00, 0x00, 0x80},
    modulus
);

Console.WriteLine("---- BEGIN SSH2 PUBLIC KEY ----");
uint i = 0;
foreach(char c in System.Convert.ToBase64String(pemKey)){
    Console.Write(c);
    if(++i == 72){
        Console.Write("\n");
        i = 0;
    }
}
Console.WriteLine("---- END SSH2 PUBLIC KEY ----");
```

De esta porción de código se obtuvo lo que parecía una clave pública:

```
---- BEGIN SSH2 PUBLIC KEY ----
AAAAB3NzaC1yc2EAAAADAQABAAAAgACcAAEAAAAUy/kXGEqcyS3HoaC9TbHOs4bvPn4AgNms
FI4MNyRdUeJD4XPOXDxp2kq2uV6JvsI8P5DsArG5KQsZpPV9+HrxXT/iV4h1LChUzYtWfYOg
qRI2uNgqlqaAJ2qaaiF68XZ1fn8t9dSFat9ZFEAcPjyNnNrSItRAzX2R/xVjTp7VB6BRh42F
3pNRVU6PGTQHZ5YbMkG4qs3f
---- END SSH2 PUBLIC KEY ----
```

Figura 27: Salida de la porción de código anterior.

Sin embargo, al comprobar si la clave era válida con la utilidad que se incluye en Git, no se obtuvo el resultado que se esperaba:

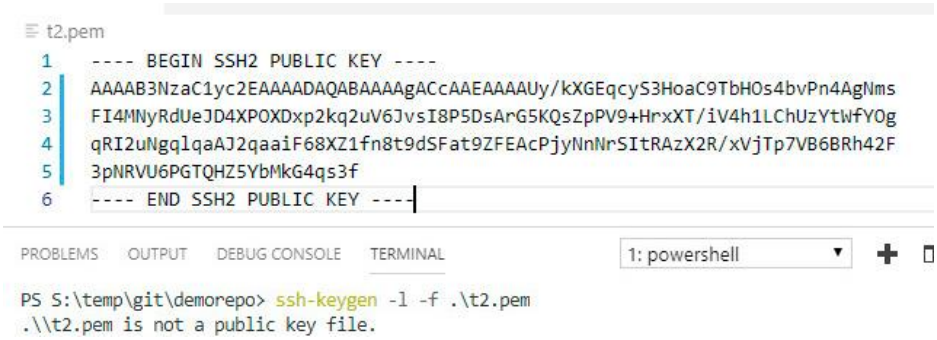

Figura 28: Salida de la comprobación de la estructura de la clave pública, fallida.

[15] Estandar para claves públicas SSH, según la IETF (2006): https://tools.ietf.org/html/rfc4716

A este inconveniente se le suma la insuficiente documentación proporcionada por parte de Microsoft para el uso de la librería TSS.NET en C#, la cual es una recopilación de los ejemplos presentes en el repositorio en un documento de Word.

Si en un futuro la documentación es mejor, o existe algún método directo para interactuar con las claves públicas, tal y como existían anteriormente, sería interesante implementar esta funcionalidad añadida.

Autenticación adicional mediante el uso de clave(s) API

Se podría implementar, como medida adicional de seguridad, una clave API que restringiese el acceso al servidor TIP a aquellas peticiones que tuviesen dicha clave.

Sin embargo, se ha optado por no implementarlo directamente sobre el servidor TIP, pues existen proyectos de código abierto que ya implementan esta funcionalidad, como *API Umbrella* o *Kong*, y sería más interesante integrarlo con estas herramientas, pudiendo conocer así métricas sobre el uso del servidor TIP, o limitarlo por uso.

Referencias

Ansetti, M., A. Ardagna, C., Damiani, E., & Saonara, F. (2013). A test-based security certification scheme for web services. *ACM Transactions on the Web (TWEB)*, Artículo nº 5.

Arthur, W., Challener, D., & Goldman, K. (2015). *A Practical Guide to TPM 2.0: Using the Trusted Platfrom Module in the New Age of Security.* Apress Open.

Muñoz, A., & Maña, A. (2013). Bridging the GAP between Software Certification and Trusted Computing for Securing Cloud Computing. *IEEE International Workshop on Security and Privacy Engineering, Assurance, and Certification (SPEAC 2013)*, (págs. 103-110).

Pearson, S., & Balacheff, B. (2003). *Trusted Computing Platforms: TCPA Technology in Context.* Prentice Hall.

Apéndice A

Dificultades encontradas a la hora de experimentar con el TPM

Virtualización mediante máquinas virtuales

Parece una buena idea utilizar máquinas virtuales desechables para hacer pruebas experimentales con el TPM. Idealmente, estarían aisladas del sistema operativo anfitrión, e incluso se podría virtualizar el TPM para evitar acceder al hardware directamente, evitando así riesgos como eliminar accidentalmente las claves almacenadas en él.

Al principio se sopesó la posibilidad de utilizar *Oracle VirtualBox*, una implementación gratuita de un hipervisor doméstico completamente abierto a cambios debido a su licencia *GPLv2*. Se daba por hecho que algo tan extendido como el TPM (recordando que está presente en la gran mayoría de computadoras de consumo) ya estaría implementado, pero no es así. En la actualidad, *VirtualBox* (versión 6) no ofrece ninguna vía para virtualizar el TPM, ni siquiera utilizando de forma directa el TPM de la máquina anfitriona. Pese a que la licencia GPLv2 hubiese permitido contribuir a que VirtualBox obtuviese soporte para virtualizar el TPM, se sale del contexto de este Trabajo de Fin de Grado y aumentaría ampliamente las horas dedicadas al mismo.

Por ello, se recurrió a la versión gratuita, aunque propietaria, de VMware: *VMware Workstation Player*. Tras realizar la instalación de la máquina virtual, se procedió a encriptar el disco virtual al completo, tal y como propone la documentación, descubriendo así que era necesario adquirir una licencia de *VMWare Workstation Pro*

como mínimo para activar el TPM virtual, con un coste de 275€, algo que rompe completamente con los objetivos de este Trabajo de Fin de Grado: **utilizar el TPM para minimizar costes y aumentar la seguridad de los proyectos software**.

Por último, se decidió probar con *Hyper-V* de *Microsoft* sobre Windows 10. También existía la opción de habilitar el TPM virtual. Pero para utilizar *Hyper-V* es necesario una licencia de Windows 10 Pro, que asciende hasta los 279€.

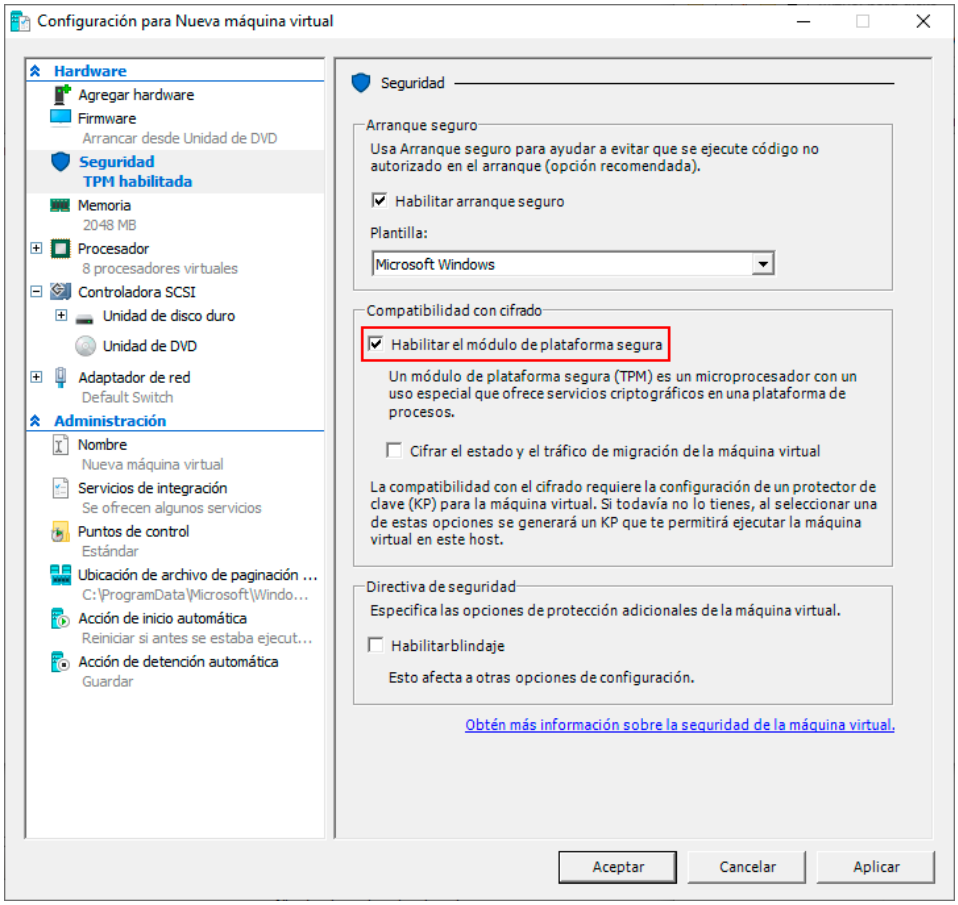

Figura 29: Captura de la configuración de una máquina virtual en *Hyper-V*, con la opción de TPM habilitada.

Así que pese a parecer una magnífica idea el hecho de utilizar máquinas virtuales para disminuir el riesgo de la experimentación, el coste económico requerido es demasiado elevado.

Trabajar con el simulador de Microsoft de forma directa

Al fallar la idea de utilizar máquinas virtuales para virtualizar sistemas completos con un TPM, se busca una posible forma de trabajar de con fiabilidad con un entorno desechable para hacer pruebas. En *A Practical Guide to TPM 2.0: Using the Trusted Platfrom Module in the New Age of Security* se menciona la existencia de un simulador, desarrollado por Microsoft, que permite emular todas las funcionalidades del TPM, y permite una comunicación mediante un modelo Cliente/Servidor a través de sockets.

En dicha guía, se menciona un pequeño script para comprobar a bajo nivel que el simulador funciona correctamente:

```
#!/usr/bin/python
import os
import sys
import socket
from socket import socket, AF_INET, SOCK_STREAM
platformSock = socket(AF_INET, SOCK_STREAM)
platformSock.connect(('localhost', 2322))
# Power on the TPM
platformSock.send('\0\0\0\1')
tpmSock = socket(AF_INET, SOCK_STREAM)
tpmSock.connect(('localhost', 2321))
# Send TPM_SEND_COMMAND
tpmSock.send('\x00\x00\x00\x08')
# Send locality
tpmSock.send('\x03')
# Send # of bytes
tpmSock.send('\x00\x00\x00\x0c')
# Send tag
tpmSock.send('\x80\x01')
# Send command size
tpmSock.send('\x00\x00\x00\x0c')
# Send command code: TPMStartup
tpmSock.send('\x00\x00\x01\x44')
# Send TPM SU
tpmSock.send('\x00\x00')
# Receive the size of the response, the response, and 4 bytes of 0's
reply=tpmSock.recv(18)
for c in reply:
    print("%#x " % ord(c))
```

Sin embargo, al tratar de ejecutarlo con la última versión de Python (3.7) suceden varios errores. Al principio uno de sintaxis, fácilmente resoluble, pero aparecen otros errores, relacionados con los tipos de datos, dejando evidencia que el script no está hecho para funcionar con esta versión de Python:

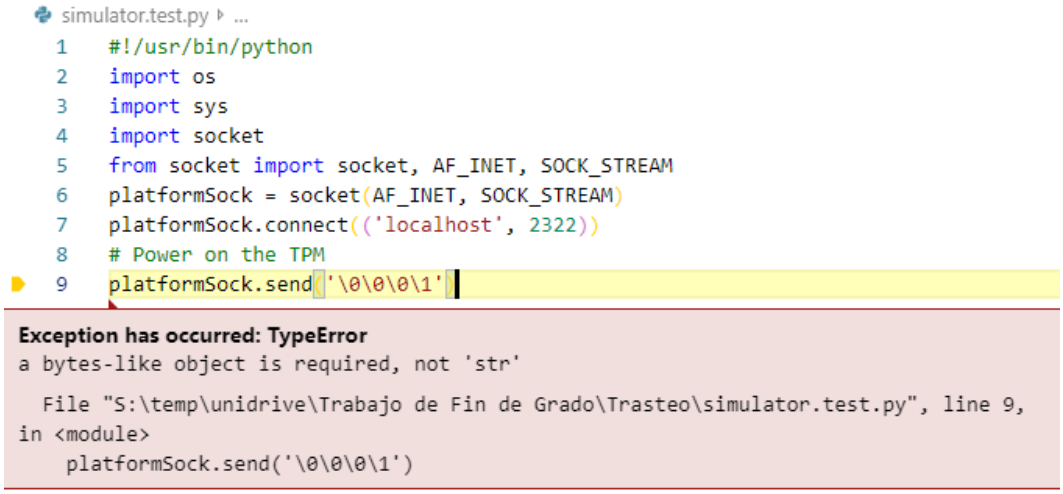

Figura 30: Errores al interpretar el script en Python 3.7

Al tener más conocimientos en PHP que en Python, se decide pasar ese pequeño script de Python a PHP, para saber si el script es correcto:

```php
<?php
/* Power ON TPM */
$platformSocket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
socket_connect($platformSocket, '127.0.0.1', 2322);
socket_write($platformSocket, '\0\0\0\1');

$sock = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
socket_connect($sock, '127.0.0.1', 2321);

/* Send TPM_SEND_COMMAND */
socket_write($sock, '\x00\x00\x00\x08');
/* Send locality */
socket_write($sock, '\x03');
# Send # of bytes
socket_write($sock, '\x00\x00\x00\x0c');
# Send tag
socket_write($sock, '\x80\x01');
# Send command size
socket_write($sock, '\x00\x00\x00\x0c');
# Send command code: TPMStartup
socket_write($sock, '\x00\x00\x01\x44');
# Send TPM SU
socket_write($sock, '\x00\x00');
socket_close($platformSocket);
socket_close($sock);
```

El código ejecuta exactamente la misma tarea que su versión en Python, y esta vez, se interpreta sin problema alguno mediante el servidor interno que posee PHP.

Al ejecutar el script PHP con el simulador del TPM iniciado en el puerto 2321 (puerto para el TPM) y 2322 (puerto para la plataforma del TPM), pero enviar los comandos no se envían correctamente como se indica en la guía, y el simulador arroja algunos errores:

```
PS S:\temp\unidrive\Trabajo de Fin de Grado\Simulador> .\Simulator.exe
TPM command server listening on port 2321
Platform server listening on port 2322
Client accepted
Unrecognized platform interface command 1546673200
Platform server listening on port 2322
Client accepted
Unrecognized platform interface command 1546673200
Client accepted
Platform server listening on port 2322
Unrecognized TPM interface command 1551380528
```

Figura 31: Errores arrojados por el simulador al intentar comunicarnos con él mediante el script en PHP

Es decir, los comandos enviados están llegando adecuadamente a través del socket al simulador del TPM, pero no los reconoce. Considerando de nuevo la posiblidad de ejecutarlo en Python, pero con una versión más antigua, surge la cuestión de qué versión es la adecuada. En un encuesta, lanzada un año antes de la publicación del libro, la versión más utilizada era la 2.7. Se decide probar dicha versión.

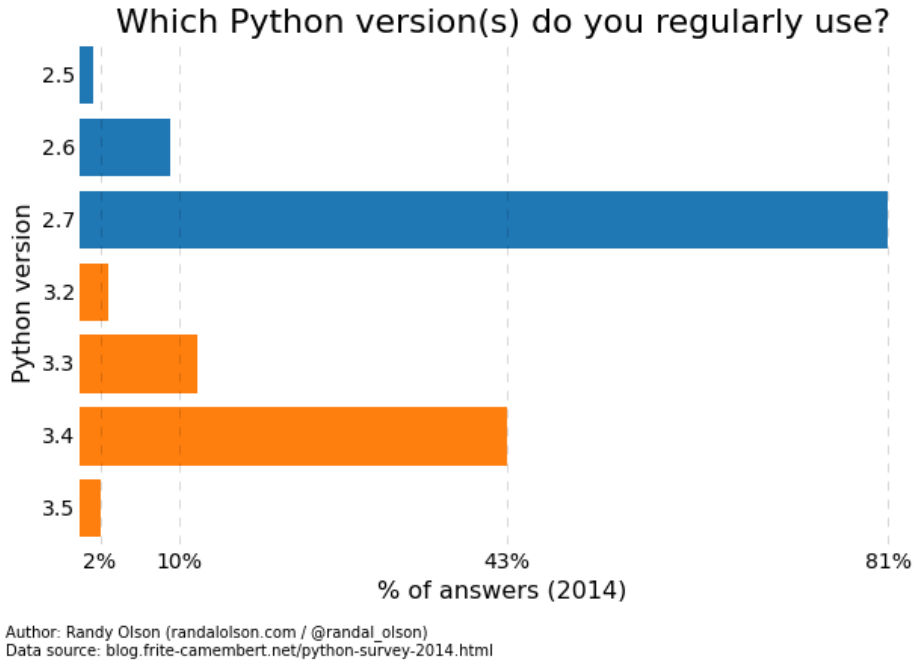

Figura 32: Versiones más populares de Python según una encuesta en 2014.

Al ejecutar el script propuesto con la versión 2.7 de Python, se consigue con éxito. El simulador ya está funcionando.

Trabajar con el simulador de Microsoft a través de una librería

Trabajar con el simulador de Microsoft directamente a través de sockets es un buen ejercicio para practicar el acceso a bajo nivel que permite el TPM. Pero a la hora de crear una aplicación con seguridad basada en el TPM, es lógico pensar en automatizar las rutinas de abrir una conexión con el TPM, enviar datos, recibir datos, etc. Sobre todo, abstraer a alto nivel funciones como generar un hash, o un número aleatorio.

Podría hacerse desde cero para un lenguaje como PHP o Python, utilizando las instrucciones a bajo nivel como se ha visto en el apartado anterior. Por suerte, Microsoft ya ha implementado en C#, C++ y .NET lo que denomina ***TPM Software Stack*** (o **TSS** para abreviar). Esta "pila de Software" permite probar rápidamente las funciones del TPM a través del simulador, y escribir programas basados en ellos.

También incluye pequeños scripts de demostración. Por ejemplo el programa *getRandom* permite obtener una cadena aleatoria generada de forma segura con el TPM o el simulador, en hexadecimal:

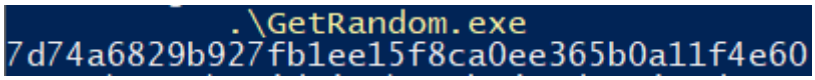

Figura 33: Salida de un programa demostrativo al ser compilado y ejecutado en *PowerShell*

Utilizar la TSS es la opción con la que finalmente se ha decidido elaborar el código de los programas auxiliares del TFG.

El único inconveniente encontrado es que no hay ninguna indicación de cómo compilar los ejemplos. Para ello, lo más sencillo es descargar la versión comunitaria de *Visual Studio 2019*, de la web oficial proporcionada por Microsoft.

Una vez instalado con las extensiones necesarias (que el propio IDE irá proponiendo si es menester), se podrá clonar el repositorio oficial de la TSS de Microsoft, para poder ejecutar los programas de demostración.

Con el repositorio clonado, dentro de la carpeta *TSS.NET* se encuentra un archivo *TSS.NET.sln*, que con abrir directamente con Visual Studio será suficiente para cargar todas las demostraciones.

Al cargar el fichero *.sln*, se abrirá un panel como este:

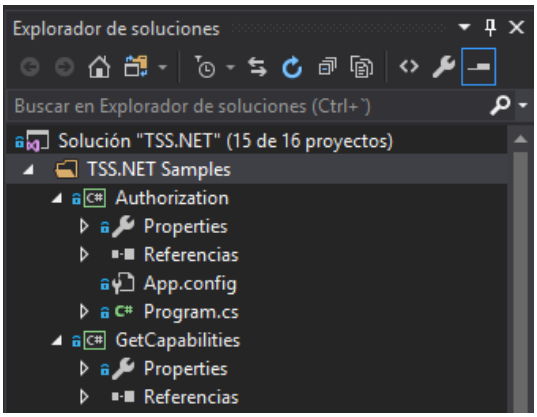

Figura 34: Explorador de soluciones de Visual Studio

Es posible examinar el programa de cada una de las demostraciones haciendo doble clic en cada uno de los *Program.cs* que se presentan. Para poder compilar una en concreto, se selecciona la demostración deseada, se hace clic derecho, y se selecciona la opción de **Compilar**.

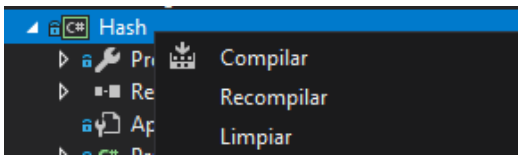

Figura 35: Menú sobre un proyecto en Visual Studio

Cuando hayamos terminado de compilar el programa, es posible ejecutarlo utilizando la barra superior, desplegando la misma solución que se quiere probar, y pulsando el botón Iniciar.

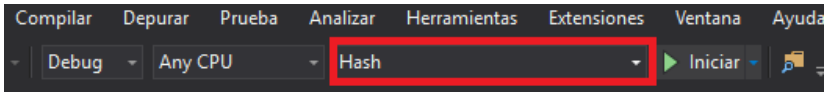

Figura 36: Barra de acceso rápido a la ejecución de un programa en Visual Studio

Es importante que el simulador se esté ejecutando en segundo plano en el puerto por defecto, o estas demostraciones fallarán si no se modifican previamente.

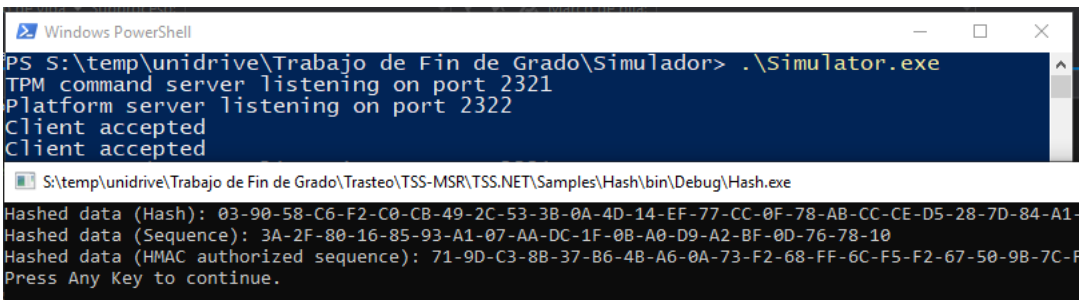

Figura 37: Ejemplo con el simulador abierto en segundo plano del programa de demostración de hashes.